Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Comparison of Compiler's Intermediate Representations and Input/Output Access Patterns with String Kernels

Dissertation

zur Erlangung des Doktorgrades
an der Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
der Universität Hamburg

eingereicht beim Fach-Promotionsausschus Informatik von
Raul Ernesto Torres Carvajal
aus
Pasto (Kolumbien)

Hamburg, Juni 2018

Prüfungskommission:
Prof. Dr. Thomas Ludwig, einfaches Mitglied / Schriftführung
Prof. Dr.-Ing. Stephan Olbrich, stellv. Vorsitz
Prof. Dr. Matthias Rarey, Vorsitz

Gutachter:
Prof. Dr. Thomas Ludwig
Prof. Dr. rer. nat. Martin Schulz

Datum der Disputation:
09.11.2018

Abstract

Kernel methods aim for the detection of stable patterns robustly and efficiently from a finite data sample by embedding data items into a space of higher dimensionality where data points have linear relations. Strings kernels apply this methodology to find relationships between string objects by checking for the number of shared substrings and using this measure as a similarity score. Due to the low number of studies conducted in the area of code comparison and Input/Output (I/O) access pattern recognition using string kernels, the goal of this thesis is to propose a suitable, general representation, as well as the corresponding strategies of comparison based on kernel methods, such that they can be used successfully to determine a reliable similarity measure among a collection of programs. Therefore, we propose different conversion strategies from these original sources to a weighted string representation; the defined representation is a collection of tokens whose weights allow the modulation of the contribution of each token in the calculation of the overall similarity.

The resulting strings are compared with a new family of kernel functions, which correspond to the major contribution of this thesis: the *kastx spectrum kernel* family. In order to create a similarity measure among two strings, these kernels are based on the longest common substrings; the idea behind this approach is to give more relevance to the largest common pieces of code rather than to small and disperse code instructions. The size of the valid matching substrings is controlled by the cut weight, a parameter given by the user, that specifies the minimum weight that those substrings should have.

We tested our methodology in two scenarios: *i)* pattern recognition in I/O traces, and *ii)* comparison of intermediate representations of a compiler. In the first scenario, the clustering analysis showed that the proposed kernels managed to conform clusters that reflected the similarity of patterns taken from two popular I/O benchmarks. For the second scenario, a set of C functions was organized in four different classes, according to their purpose; clustering analysis here also showed a cluster organization that reflected the affinity among functions of the same class. These new kernels obtained similar, and in some cases, better results when compared to the *blended spectrum kernel* [SC04], a string kernel with widespread use in cheminformatics problems. The provided kernels will enrich the spectra of available string kernel functions on the literature and might be used in the future in similarity studies, not only in the field of computer science, but also in other areas like cheminformatics, bioinformatics or natural language processing.

Zusammenfassung

Kernel-Methoden zielen auf die robuste und effiziente Erkennung stabiler Muster aus einem endlichen Datenmuster, durch Einbettung der Datenelemente in einen höherdimensionalen Raum, in den dir Datenelemente lineare Beziehungen haben. String-Kernels wenden diese Methode an, um Beziehungen zwischen String-Objekten zu finden, indem sie die Anzahl der geteilten Teilstrings prüfen und dieses Maß als Ähnlichkeits-Score verwenden. Aufgrund der geringen Anzahl von Studien, die im Bereich des Code-Vergleichs und der Eingabe/Ausgabe(E/A)-Mustererkennung unter Verwendung von String-Kernels durchgeführt wurden, ist das Ziel dieser Arbeit eine geeignete, allgemeine Darstellung sowie entsprechende Vergleichsstrategien basierend auf Kernel-Methoden vorzuschlagen, so dass diese erfolgreich zur Bestimmung einer zuverlässigen Vergleichsmetrik zwischen einer Sammlung von Programmen genutzt werden können. Daher schlagen wir verschiedene Umwandlungsstrategien ausgehend von diesen Ursprungsquellen zu einer gewichteten String-Darstellung vor. Die definierte Darstellung ist eine Sammlung gewichteter Tokens, bei der die Gewichtungen die Modulation des Beitrags jedes Tokens in der Berechnung der Ãhnlichkeit ermöglichen.

Die resultierenden Strings werden mit Hilfe einer neuen Familie von Kernel-Funktionen verglichen, welche den Hauptbeitrag dieser Thesis darstellt: Die *kastx spectrum kernel*-Familie. Um eine Ähnlichkeitsmetrik zwischen zwei Strings zu berechnen, werden Kernel-Funktionen, die auf den längsten gemeinsamen Substrings basieren, verwendet, mit der Intention, dass die längsten gemeinsamen Codestücke mehr Relevanz erhalten als kleine und verstreute Teile des Codes. Die Größe der gültigen übereinstimmenden Teile zwischen zwei Strings wird begrenzt durch das Schnittgewicht, ein durch den Nutzer vorgegebener Parameter, das die minimale erwünschte Gewichtung der Substrings festlegt.

Wir testen die Methode in zwei Szenarien: *i)* Mustererkennung in E/A-Spuren und *ii)* Vergleich von intermediären Darstellungen eines Compilers. Im ersten Szenario zeigte die Clusteranalyse, dass die vorgeschlagenen Kernels Cluster finden konnten, die in zwei beliebten E/A-Benchmarks eine große Ähnlichkeit der Muster zeigten. Im zweiten Szenario wurde eine Menge von C Funktionen in vier unterschiedliche Klassen eingeteilt, basierend auf ihrem Zweck. Eine Clusteranalyse zeigte, dass die Funktionen der einzelnen Cluster untereinander Ähnlichkeit aufwiesen. Die neuen Kernels erzielten ähnliche und in manchen Fällen bessere Ergebnisse im Vergleich mit dem textitblended spectrum kernel [SC04], einem String Kernel der im Feld der Chemieinformatik große Verbreitung findet. Die bereitgestellten Kernel erweitern das Spektrum der in der Fachliteratur veröffentlichen String-Kernel und können in der Zukunft für Ähnlichkeitsstudien verwendet werden, nicht ausschließlich in der reinen Informatik, sondern auch in verwandten Feldern wie der Chemieinformatik, der Bioinformatik und der Verarbeitung natürlicher Sprache.

Acknowledgments

I would like to infinitely thank my supervisor, Prof. Dr. Thomas Ludwig; his assertive guide, support and supervision were key factors for the success of this work. I also express here my gratitude to the other supervisors, those who encourage me to never give up in this project: Dr. Julian M. Kunkel and Dr. Manuel F. Dolz. Many thanks to the reviewers for taking the time to evaluate this work.

I make here a special mention for the following persons, who took the time to help me to review the literature and also to build a consistent text: Dr. Ruslan Krenzler (Leuphana University, Germany); Dr. Diego Moreno (Universidad Autónoma de Yucatán, Mexico); Dr. Julio Maza (Universidad de Cartagena, Colombia); MSc. Neftali Forero (CTO TodosEn4, Colombia); Viviana Torres (Universidad de Nariño, Colombia); Maximilian Hopf (Kreditech Holding SSL GmbH, Germany).

Friends and family were always there. Thanks for believing in me.

Finally, I would like to acknowledge the financial support from the Colombian Administrative Department of Science, Technology and Innovation (Colciencias).

Contents

1 Introduction

In this first chapter, we present to the reader the problem of program similarity as the main theme that motivates this work. Among the variety of available forms of assessing a similarity score, the comparison of programs can be based on the code itself, the intermediate representations inside a compiler, or the traces left in storage access monitoring. In that sense, we start, in Section 1.1, with the review of the basic background on the topics of code similarity and the intermediate representations. We also explore the relationship of Input/Output (I/O) access patterns and kernel methods with the research problem. Kernel methods are able to detect patterns from a finite data sample by embedding data items into a space of higher dimensionality where data points have linear relations. Throughout this chapter we show how comparing programs brings benefits to a variety of tasks like performance evaluation, code sharing and plagiarism detection. The reader is progressively introduced into the use of string kernel functions for comparing programs, a relatively new area in which this thesis contributes with three novel implementations. Strings kernels are functions well suited to find relationships between string objects. Section 1.2 contains an overview of the contributions of this work, while Section 1.3 presents the general structure of the document.

1.1 Motivation

Numerous studies have been conducted in the problems of code similarity and I/O pattern recognition. However, the usage of string kernels in these domains is an area not widely explored. This section introduces the reader into the research problem and the suitability of the application of kernel methods as a solution for it.

1.1.1 Code Similarity

Computer programs exhibit similarities that can be detected before, during and after execution time. *Being similar* means sharing syntactical or semantical structure in a significant proportion. Programs that are similar tend to behave in a similar manner too, a fact that can be harnessed, for example, for the analysis and improvement of the overall performance of a set of programs by focusing on finding patterns that behave similar but have a different performance. The detection of program similarities has been identified as an emerging topic in software engineering areas [CX12].

From the perspective of code sharing, finding similar code can, for example, assist the programmer in finding code that is already implemented in a library and hinting users to utilize the library instead of recoding. Developers might be even able to find syntactically dissimilar, yet more efficient, versions of their algorithms with similar semantics. The optimizations implemented in an algorithm might benefit similar programs, once the similarity relation has been established.

Furthermore, the search could be specialized on finding common mistakes at designing or writing programs, the so-called *code smells* [MVL03]. A possible way to do this could be by comparing our own code against a collection of code excerpts already recognized as containers of code smells, and subsequently applying machine learning to extract knowledge from the similarity scores.

On the counter direction of code sharing, we have code plagiarism, a phenomena that has increasingly motivated research in program similarity [MKZ06]. Starting from computer science lectures where teachers need to verify the originality of their students' work, till the protection of copyrighted code that might have been illegally used, plagiarism detection is a very active research field.

Code Clones and Detection Approaches

In the work of Beth [Bet14], the general strategies used to obscure plagiarism are listed. They range from simple changes like comment alteration, whitespace padding and identifier renaming, to more complex procedures like code reordering and modification of algebraic expressions. These modifications provoke the occurrence of code clones, which are defined by Dang et al. [DW15] as portions of code with high similarity in syntax or semantics. The occurrence of code clones are normally used to measure the level of plagiarism in a program and are believed to difficult the maintenance of software [KKI02].

The severity of the alterations determines the type of the clone. To illustrate this point, let us consider a code sample written in C, like in Listing 1.1; the function performs the simple calculation of the expression: $b \times c + d \div e$, and stores it into a variable. The following is a list of possible clones that can be obtained by modifying the given code:

- A *Type-1* clone would present simple changes in code layout, spacing and comments, like in Listing 1.2; in the given example, the major change corresponds to the absence of comments.

- A *Type-2* clone would also include identifier renaming and alteration in data types, like in Listing 1.3; the example shows how `float` and `int` data types are changed to `double` and `long` respectively.

- A *Type-3* clone would additionally contain insertion, deletion or modification of instructions, like in Listing 1.4; the main instruction in the example makes use of brackets to explicitly express the operation precedence.

- A *Type-4* clone would have a different implementation to the original piece of code but serve the same purpose, like in Listing 1.5; in the given example, the main instruction is broken down into three parts.

```c
void example()
{
  // variable declaration
  float a = 0.0 ;
  int b = 1 ;
  int c = 2 ;
  int d = 3 ;
  int e = 4 ;

  // calculation
  a = b * c + d / e ;
}
```

Listing 1.1: C code segment enclosing the instruction $a = b \times c + d \div e$.

```c
void example(){
float a=0.0;
int b=1;
int c=2;
int d=3;
int e=4;
a=b*c+d/e;
}
```

Listing 1.2: *Type-1* Clone.

```c
void function()
{
  double v = 0;
  long w = 1;
  long x = 2;
  long y = 3;
  long z = 4;
  v = w * x + y / z;
}
```

Listing 1.3: *Type-2* Clone.

```c
void function()
{
  long w, x, y, z;
  w = 1;
  x = w + 1;
  y = x + 1;
  z = y + 1;
  double v = (w * x) + (y / z);
}
```

Listing 1.4: *Type-3* Clone.

```c
void function()
{
  long y = 3;
  long z = 4;
  long w = 1;
  long x = 2;
  double l = y / z;
  double m = w * x;
  double v = l + m;
}
```

Listing 1.5: Type-4 Clone.

We believe that not all code clones can be taken as evidence of plagiarism, especially *Type-4* clones, whose implementation would differ radically from a presumed original version; this belief is not new. For example, Al-Ekram et al. [AlE+05] described how clones can be created by accident, due to the precise protocols that a piece of code must follow, when using a particular API from a library. Other works [God09] have instead approached to code clones as an effect of software evolution.

In our research, we aim for contributing with methods suitable for complex comparison tasks. Hence, we have discarded approaching to code as mere text or lexemes, which would additionally bind the proposed solution to a particular language. Instead, we have targeted two intermediate representations delivered by the popular LLVM compiler infrastructure [LA04], which placed our contribution into the category of the syntactical approaches, which are ideal for detecting *Type-1*, *Type-2* and *Type-3* clones, and automatically promoted the application of our methods to the extensive variety of programming languages that LLVM can compile.

Compilers' Intermediate Representations

Before compilation, code comparison can be performed upon the typed code itself or on source code metrics. Upon compilation, however, the already mentioned intermediate representations (IRs) come to the picture and bring deeper information about the program's characteristics, as they are meant to facilitate the analysis and optimization tasks of a compiler. Researchers have also utilized them to track the evolution of software projects [NFH05]. The additional knowledge they implicitly carry make them a better option in comparison to raw code when it comes to the selection of a source of information for a similarity study.

Complex compiler infrastructures like LLVM [LA04], GCC [Gri02] and Open64 [Dev01], might work with different types of interconnected IRs, some of them closer to the source code, others closer to the machine instruction level. Due to its wide spread in the compiler infrastructures, we were interested mainly in two of them: *Abstract Syntax Trees (ASTs)* and *Three-address Codes*.

- ***Abstract Syntax Trees (ASTs):*** Graphical intermediate representations store the program's information in a graph-like data structure. Among them, abstract syntax trees (ASTs) are widely used in compiler infrastructures. ASTs are defined as contractions of parse trees where most non-terminal symbols are ignored while the precedence and the meaning of the expressions are preserved, thus saving space. Their level of abstraction is not far from the original source code.

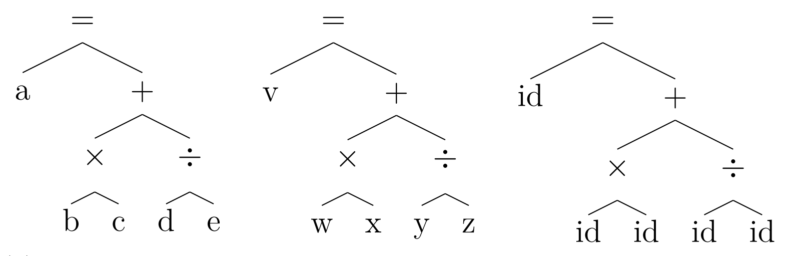
  It is exactly the abstraction level the feature that makes researchers focus on ASTs as a source of information for code comparison tools, e.g. DECKARD [Jia+07]. ASTs provide high level information that is not evident at simple view in source code. One simple example is the scenario where one looks up into an AST node that represents a variable and ask for the data type instead of its name, which delivers more meaningful insights of the program than the arbitrary selection of variable names made by the programmer. A more complex scenario is the possibility to connect a node that represents a function call with the parent node of its implementation.

  To illustrate how ASTs can be used to find similarities, let us consider the ASTs of both the original code (Figure 1.1a) and its *Type-3* clone (Figure 1.1b) from the previous section. Both trees are structurally the same; only the identifiers differ. A typical normalization step would replace all individual identifiers with a common keyword, which would result in the same AST for both examples (Figure 1.1c).

- ***Three-address Codes:*** Linear IRs are simple sequences of operations, similar to machine code. Among the popular linear IR models, there exists the three-address code model. In this model, most operations have at most an operator and three addresses: two addresses for the operands and one for the result. Three-address codes exhibit compactness without forcing destructive operations, which gives room for further code optimization. Plus, many modern processor architectures are already based on three-address operations, which makes the translation more intuitive.

One of the advantages of this representation in comparison with the ASTs, is that it is obtained once the compiler has performed the optimization passes, hence eliminating redundant or dead portions of code.

Let us again consider both the original code and its *Type-3* clone, but this time using their three-address codes (Listings 1.6 and 1.7 respectively). Here also, a normalization step would replace all individual identifiers with a common keyword and produce the same three-address code for both examples (Listing 1.8). An additional compression step would collapse repeated consecutive line occurrences into a single one (Listing 1.9).



(a) $a = b \times c + d \div e$ (b) $v = (w \times x) + (y \div z)$ (c) $id = id \times id + id \div id$

Figure 1.1: Abstract Syntax Trees for a) Original and b) *Type-3* Clone c) Normalized version.

$$\begin{array}{l} v1 \leftarrow d \\ v2 \leftarrow e \\ v3 \leftarrow v1 \div v2 \\ v4 \leftarrow b \\ v5 \leftarrow c \\ v6 \leftarrow v4 \times v5 \\ v7 \leftarrow v3 + v6 \end{array}$$

Listing 1.6: Three-Address Code for Original Code.

$$\begin{array}{l} v1 \leftarrow y \\ v2 \leftarrow z \\ v3 \leftarrow v1 \div v2 \\ v4 \leftarrow w \\ v5 \leftarrow x \\ v6 \leftarrow v4 \times v5 \\ v7 \leftarrow v3 + v6 \end{array}$$

Listing 1.7: Three-Address Code for *Type-3* Clone.

$$\begin{array}{l} id \leftarrow id \\ id \leftarrow id \\ id \leftarrow id \div id \\ id \leftarrow id \\ id \leftarrow id \\ id \leftarrow id \times id \\ id \leftarrow id + id \end{array}$$

Listing 1.8: Normalized Three-Address Code.

$$\begin{array}{l} [id \leftarrow id]_2 \\ [id \leftarrow id \div id]_1 \\ [id \leftarrow id]_2 \\ [id \leftarrow id \times id]_1 \\ [id \leftarrow id + id]_1 \end{array}$$

Listing 1.9: Compacted Three-Address Code.

The Intermediate Representations of the LLVM Compiler

LLVM [LA04] is a complete compiler framework that uses different IRs to perform program analysis, transformation and code generation. Among the variety of tools available under this infrastructure, there exists Clang [Inf17], a frontend for C/C++/Objective C programs. Upon compilation, Clang first captures the syntactical structure of the program in an acyclic graph-like structure, called the Clang AST. It has to be clarified that the Clang AST is in practice not meant to be a tree, but it can be traversed as such, once we have access to the particular dependencies of the internal nodes.

Afterwards, the Clang AST is traversed to generate a linear IR, the place where most of the transformations and optimizations are performed, before the final stage of specific machine code generation. The LLVM Linear IR is the backbone that connects the frontends and backends of the whole compiler infrastructure [LA14]. LLVM provides a library that allows to handle this linear representation as a low level AST, which is very useful when it comes to do program analysis.

Consider the abbreviated versions of the Clang AST (Figure 1.2) and the LLVM IR (Listing 1.10) for the instruction: $a = b \times c + d \div e$, and its *Type-3* clone: $v = (w \times x) + (y \div z)$. As identifiers and operation names in the AST are replaced by generic names, both instructions have the same tree. In the case of the LLVM IR, variable names are replaced automatically by the compiler with incremental register names, resulting in the same code for both examples.

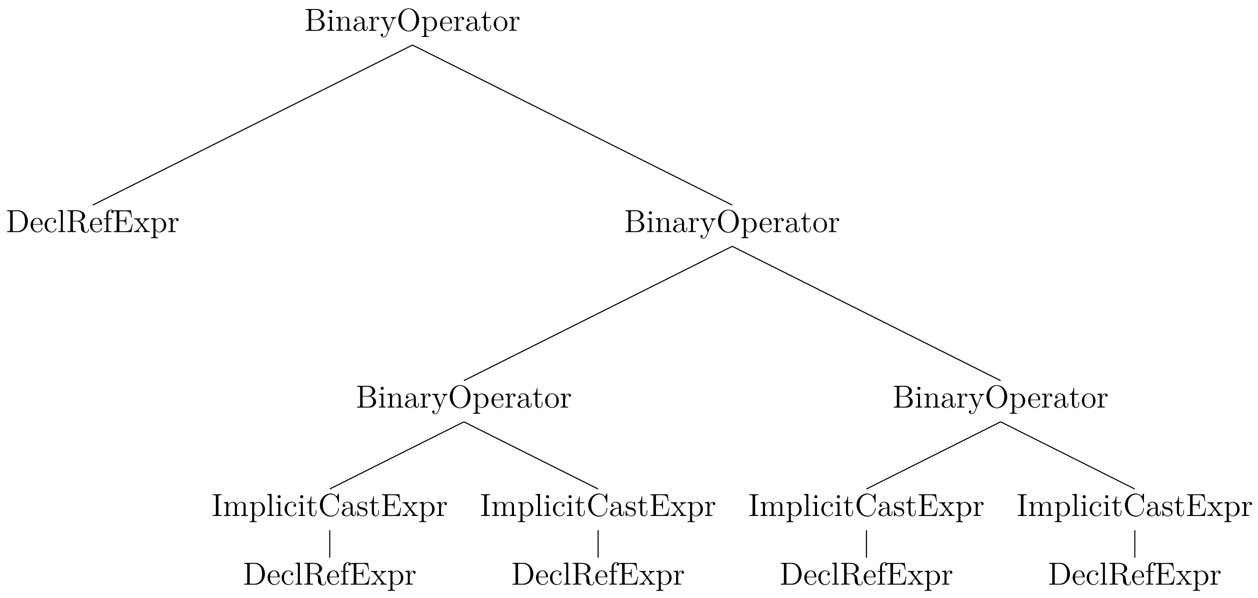


Figure 1.2: Abbreviated Clang AST for both Original ($a = b \times c + d \div e$) and *Type-3* Clone ($v = (w \times x) + (y \div z)$).

```
%5 = load %1
%6 = load %2
%7 = mul %5 %6
%8 = load %3
%9 = load %4
%10 = sdiv %8 %9
%11 = add %7 %10
```

Listing 1.10: Simplified LLVM Linear IR code segment for both Original ($a = b \times c + d \div e$) and *Type-3* Clone ($v = (w \times x) + (y \div z)$).

The configuration is more than ideal: a high level representation closer to the source code, and a low level representation closer to the machine code. In addition to that, the versatility of both, the Clang AST and the linear representation of LLVM, made this compiler platform the best candidate to perform our study. Previous efforts where done during this research with other program analysis infrastructures like the Rose Compiler Project [J Q00] and the ANTLR Parser [Par15]. However, the difficulty to handle their ASTs and the relatively reduced community giving support in large scale projects, made us decline on them.

As it has been seen, comparing the code of a set of programs before their execution has a clear application in plagiarism detection and code sharing. In the exact case of code sharing, a program can benefit from optimizations already present in a similar piece of code. However, before execution, any improvements on the performance are only theoretical.

1.1.2 Similarity of Input/Output Access Patterns

Computer programs can also be characterized upon and after execution time, which provides an experimental proof of the performance of a program. This characterization can be done in a variety of ways. One could use real time monitoring of hardware performance counters, like in the PAPI project [Bro+00] or the LIKWID tool suite [THW10]. Another approach is to focus on the communication and synchronization bottlenecks, like the Scalasca tool-set does [Gei+08].

Projects like the SIOX architecture [Kun+14] focus instead on the analysis of I/O operations. The organization of I/O operations is a critical factor that impacts directly the performance of a real world application. This performance can be studied after execution by looking at access patterns, which can be seen as the fingerprints of a program. The identification and analysis of these patterns is important in performance evaluation, because it helps, not only to understand the impact factors of the underlying file system, but also to design better ways of organizing I/O operations.

Performance analysis and optimization becomes more complex when it comes to parallel applications; this need is more evident in high performance computing (HPC) applications, which usually relay on parallel file systems. Parallel file systems are minded for accessing files in a simultaneous, concurrent and efficient way [Kun13]. The contents of a file in a system like this are usually scattered among different I/O subsystems, in order to take advantage of the locality. These systems should provide, among other capabilities, persistence, consistence, performance, and manageability. Other desired features might include: scalability, fault-tolerance and availability. However, a bad design in the I/O operations of a program might cause a suboptimal utilization of the parallel design.

I/O Traces

By looking at the I/O traces of a parallel program one can seek for bottlenecks in I/O operations; these traces may contain timestamps, operation names and information about the number of bytes involved, which constitute patterns that depict the behavior of storage access over a period of time. Each pattern can be characterized by the following properties: access granularity, randomness, concurrency, load balance, access type and predictability. Liu et al. [Liu+14] mentioned three additional features seen on supercomputing I/O patterns: burstiness, periodicity and repeatability. Traces sharing similar patterns are more likely to manifest the same weaknesses or strengths in their I/O design, an indication of great utility for the performance analysis of parallel programs. A straightforward application of a comparison study could be, for example, the determination of the efficiency of a program by comparing its traces against a database of patterns previously labeled as suboptimal.

Let us consider two apparently different I/O traces (Listings 1.11 and 1.12). Normalization in this case can be done by ordering the operations according to the file handle, and by replacing the byte number with a generic identifier, which results into the same pattern for both traces (Listing 1.13). Compression would generate a more succinct version of the pattern (1.14).

```
read file 1 bytes 8
write file 2 bytes 8
read file 1 bytes 8
write file 2 bytes 8
```

Listing 1.11: I/O pattern with interlaced operations.

```
read file 3 bytes 16
read file 3 bytes 16
write file 4 bytes 16
write file 4 bytes 16
```

Listing 1.12: I/O pattern with ordered operations.

```
read file x bytes n
read file x bytes n
write file y bytes n
write file y bytes n
```

Listing 1.13: Normalized I/O pattern.

```
[read file x bytes n]₂
[write file y bytes n]₂
```

Listing 1.14: Compacted I/O pattern.

1.1.3 String Kernel Functions for Similarity Search

We have described how intermediate representations and I/O traces of similar programs present common patterns in their internal structure that can be used as a measure of similarity. The natural question that arises after it is related on what to use to compare those data structures. The amount of metrics or algorithms for performing this task is large.

Graph-based program characterization has been proved to be an effective approach when it comes to find similarities among computer programs [PCA12]. They have been successfully adopted for compiler optimization, like in the work of Nobre et al. [NMC16].

Cesare et al. [CX12] stated that kernel methods have been scarcely used in software similarity problems, which represents a great opportunity for new research efforts. Following this hint, it called our attention the fact that there were not so many available works related to the use of string kernel functions to compare data structures coming from the particular domain of compilers or I/O access patterns; the available studies worked mostly with tree kernel functions.

Kernel Methods

There is a group of algorithms that have been successfully applied in problems involving structured data like trees and strings [BHS07]: they are called *kernel methods*. Kernel methods make part of the large constellation of machine learning techniques and are well documented in the book of Shawe-Taylor and Cristianini [SC04]. This group of algorithms are claimed to be strong enough for detecting stable patterns robustly and efficiently from a finite data sample, based on the idea that original data items can be embedded into a space where linear relations manifest as patterns. Kernel methods follow the modular design that characterizes a modern machine learning system:

- ***Feature Extraction Subsytem:*** The stage where data is transformed to a meaningful representation which can be mined.
- ***Clustering/Classification Subsystem:*** This stage corresponds to the application of a learning algorithm.

Usually, in machine learning, data is delivered as a collection of attribute-value tuples, but, as we have seen, ASTs, linear representations and I/O traces do not comply with this model, and any comparison strategies based of the attribute-value model cannot be adopted straight away. The need for an appropriate mapping of these data structures motivated our contribution, and placed our work in the category of the first stage of the kernel methods strategy: the feature extraction subsystem.

String Kernels

Kernels designed for dealing with strings are denominated *string kernels*. Strings kernels are explained in a comprehensive way in [VS03]. They check for the number of shared substrings among a collection of strings. One of the advantages of string kernels is that they can easily deal with the comparison of strings of variable size. To illustrate the notion of string kernels, let us consider the strings *aabcc* and *abccc*:

- The *bag-of-characters kernel*, only takes into account single-character matching. In this case, the similarity score is given in Table 1.1. The summation of all the scores in the table is 9, and corresponds to the kernel value among the example strings.

- On its turn, the *k-spectrum kernel* searches for shared substrings of size k. For $k = 2$, the similarity score is given in Table 1.2. The kernel value in this case is 4.
- The *k-blended spectrum kernel* sums up all the scores of all the spectrum's evaluation below and including k. For $k = 2$, the kernel value corresponds to the summation of the previous scores, in other words, 13.

Unfortunately, there are not so many applications of string kernels in the field of program comparison. Motivated on this lack, we found out there was room for some new string kernel functions that might give more relevance to the largest common substrings than to the smaller ones. This is one of the contributions of this thesis: it was important for our study to follow the modular design of kernel methods, hence, we have designed these new kernel functions in such way they can be further utilized for comparing anything that might be represented as a string or a tree. Proof of that is that we could successfully use the same kernel functions for performing three different comparison tasks whose data was coming from distinct origins: ASTs, linear IRs and I/O traces.

	a	**a**	**b**	**c**	**c**
a	1	1	0	0	0
b	0	0	1	0	0
c	0	0	0	1	1
c	0	0	0	1	1
c	0	0	0	1	1

Table 1.1: bag-of-characters (1-spectrum) kernel score for *aabcc* and *abccc*.

	aa	**ab**	**bc**	**cc**
ab	0	1	0	0
bc	0	0	1	0
cc	0	0	0	1
cc	0	0	0	1

Table 1.2: 2-spectrum kernel score for *aabcc* and *abccc*.

1.2 Goals

As it was seen, in order to identify common patterns inside a collection of computer programs, they should be represented in a form that is capable to abstract their relevant features; additionally, an appropriate strategy has to be used to find similarities or dissimilarities using the new representation as input. Following this scheme, we have aimed for proposing a modular design, based on kernel methods, which corresponds to the main contribution of this thesis:

1. As the name suggest, string kernels work upon strings. Therefore, taking into account the origin of our data (ASTs, IRs and I/O traces), we have designed a particular string representation compound by weighted tokens. At the same time, we have aimed for a design with an acceptable level of generality that allows that data coming from other domains could also be easily expressed. The token literal part, for example, can be built with the relevant pieces of information taken from the original data, like data type names, keywords or function names. The use of weights gives room for further compression, because the weight can be used to express the number of repetitions of a token.

2. Original data must be converted into the proposed string representation. In our contribution, we have focused on the conversion of the following data sources:

   - *Data available before execution:* Clang ASTs as a high level representation on the syntactical level, and LLVM linear representations as a low level representation closer to the machine code.
   - *Data available after execution:* I/O traces as an evidence of the performance of an application.

   We describe in detail these conversion procedures in Chapter 3. They are intended to be a guide on how to translate data coming from other domains.

3. The use of string kernels in the comparison of computer programs is relatively new. Therefore, we have not limited this research only to the study of the comparison performance of some string kernels from the literature. Instead, from the study of the particularities of the compiler's intermediate representations and the I/O traces we have discovered that some novel strategies of comparison could be created inspired on them. These are three new string kernel functions that are focused on finding the largest common substrings, inspired on the Greedy String Tiling algorithm of Wise [Wis93] and the Blended Spectrum Kernel [SC04].

   The rule of thumb for them is that the matching substrings must be at least independent in one of the original strings. Being independent means not being a substring of a previous segment occupied by another matching substring. This way, starting from the largest matching substring, the search for more matches narrows down progressively. The idea is to give more importance to the big code portions that are shared and minimize the effects of isolated repetitive tokens or instructions.

   Consider the example strings on Figure 1.3. Both strings share 3 substrings, which correspond to the largest common substrings, and are, at least, independent in one of the original strings.

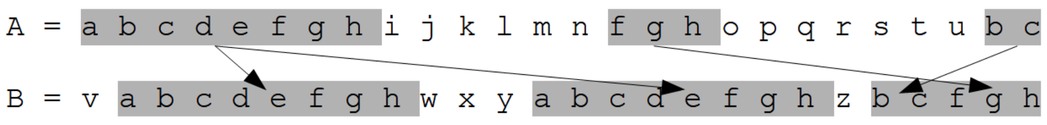

A = a b c d e f g h i j k l m n f g h o p q r s t u b c
B = v a b c d e f g h w x y a b c d e f g h z b c f g h

Figure 1.3: Two Strings and their Longest Matching Substrings.

Our kernels will enrich the spectra of available string kernel functions on the literature and might be used in the future in similarity studies, not only in the field of Computer Science, but also in Cheminformatics, Bioinformatics and Natural Language Processing (NLP).

4. To provide an experimental proof of our ideas, we have designed an evaluation methodology that used one string kernel from the literature as a baseline algorithm for comparison, the *blended spectrum kernel* [SC04].

1.3 Structure of the Thesis

This thesis is organized as follows: in Chapter 2, in addition to a more in-depth revision of the foundations of Compiler's Intermediate Representations, I/O Access Patterns and Kernel Methods, the related work and the state of the art are presented. Chapter 3 is dedicated to describe the contributions of this research, firstly, by unveiling the rational behind the process of converting Intermediate Representations and I/O traces into the proposed string representation, and secondly, by detailedly explaining the proposed kernel functions that will compare such strings. In Chapter 4 we present a proof of concept of the string kernels with a synthetic example, which would help to understand better the reach of our research. The experimental evaluation of our approach to find patterns in I/O Traces is conducted in Chapter 5, whereas Chapter 6 does the same with the Intermediate Representations of a popular compiler. Finally, Chapter 7 summarizes the results and details possible future paths for the current research efforts.

Summary

This introductory chapter has collected the basic problematic present in Program Similarity. When programs are represented as linear strings, it is possible to find patterns using string comparison algorithms. The need for novel comparison methods has motivated us to propose three new string kernel functions able to work with the Intermediate Representations of a well known compiler as well with I/O traces. However, they could also be used for any other problems involving data delivered in the form of trees or strings. The following chapters depict in detail the impact of our contribution.

2 Background

The similarity among a set of programs can be determined in many different ways. Among other sources of information for performing this comparison, a program's code, its intermediate representations or its patterns in storage access can be used. In order to understand better how program similarity works, we have made a survey on its fundamental topics. We start by reviewing in Section 2.1 the definition of code similarity and code clones, and continue with the intrinsics of the intermediate representations of a compiler in Section 2.2. Same is done with the Input/Output (I/O) aspects of applications, covered in Section 2.3, which gives the reader a basic understanding on how I/O is important for parallel evaluation. Next, in Section 2.4, Kernel Methods are introduced to the reader as one of the possible and unexplored approaches to perform the comparison task. Finally, the compendiums of related work, corresponding to Section 2.5, and the state of the art in code similarity, in Section 2.6, are reviewed, in order to depict what is the current state of this fascinating area.

2.1 Code Similarity

When a set of code pieces share syntactical or semantical structure in a significant proportion, they are said to be similar. The detection of this similarity is important, as programs that are similar tend to behave in similar manner. For example, similar code pieces are created as part of cloning process; plagiarism can be detected by finding the similar structures that two programs share.

When a source code snippet is copied, it requires some changes in order to make it work on the new location. The severity of these modifications are commonly an indication of the purpose of the copy. According to this, we can distinguish between three basic categories of code changes: *boiler-plate code*, *pervasive modifications* and *code obfuscation.*

2.1.1 Boiler-Plate Code

These modifications of a program are minimal and they are usually contained in the scope of a function definition. They are commonly generated when developers need to adapt a code template for a determined task or to adjust an existing code portion for a similar purpose. They are not cataloged as plagiarism acts.

A classical example of *boiler-plate code* are the *get* and *set* methods in Java classes (see Listing 2.1), which are necessary, as direct access to the private members of a class is not permitted. Manual creation of these methods is usually performed by copying and pasting already implemented code, which is an error-prone process, as programmers might, for example, forget to update the identifier names (see Line 16 of Listing 2.1).

```
1  ...
2  public String getName()
3  {
4    return name;
5  }
6  public String getAddress()
7  {
8    return address;
9  }
10 public void setName(String newName)
11 {
12   name = newName;
13 }
14 public void setAddress(String newAddress)
15 {
16   name = newAddress;    // Semantical error: programmer did not update the identifier
17 }
18 ...
```

Listing 2.1: Boiler-Plate Get and Set Methods in Java.

2.1.2 Pervasive Modifications

On the contrary, pervasive modifications are not restrained to a single function block. Instead, they are spread along a whole file. The modifications might be a combination of lexical, syntactical or semantical alterations, which are indicators of an effort to hide plagiarism. However, it might also be related with code adaptation, for example, when several functions with boiler-plate code are stored under the same file.

Common Code Modifications

Beth presented on a research note [Bet14], some of the most common program modifications:

Comment Alteration: It consists in the modification of the comments of the source code, which are usually ignored by the compiler but might contain useful information for understanding the structure of a program. Because the edition of comments has no effect on the program's flow and output, it is the easiest way to manipulate the aesthetics of the program.

Whitespace Padding: Compilers are commonly insensitive to the number of white spaces that can separate one token from another. This also includes the layout of the line breaks. A plagiarist might alter this to make his program look different to the original.

Identifier Renaming: Variable names are scoped conventions that help programmers and compilers to keep track of values that need to transcend on their programs. Though they must follow a minimal set of naming rules, which are defined by the language, they are not fixed inside the parsing structure of it. Hence, renaming variables offers a great opportunity for plagiarism, as a simple comparison tool will determine that two programs are different if their identifiers are changed.

Code Reordering: In the majority of programming languages, the order in which functions are defined is not important, as long as the function declaration (the function's "signature") is known before the function is called. Functional units in source code might be rearranged by the plagiarist to decrease the possibility of detection.

Modification of Algebraic Expressions: Programming languages can express mathematical operations and its properties in a straightforward way. The commutative and associative properties of an operation can be used by the attacker to achieve the same result by *i)* modifying the order of the operands, *ii)* adding redundant operations or *iii)* taking advantage of operation precedence.

Code Clones

The mentioned alterations lead to the occurrence of code clones. As stated by Dang et al. [DW15], code clones are portions of code with high similarity in syntax or semantics. They make the maintenance of software a difficult task [KKI02] by spreading bugs contained in weakly-tested code portions that are copied from one location to another in the same project. It is also common that the developers forget to document the copy operation, which at least would have given a hint of where else bug fixes should be applied. Generally, the level of plagiarism in a program can be estimated by the amount of code clones it possesses.

A straightforward classification of clone detection approaches can be found in the work of Vislavski et al. [VBR16], where we can learn that the class of an approach is highly determined by the type of clones it can detect:

Type-1 clones: They are the easiest to detect, due to the simplicity of the differences they present: changes in the code layout, the number of spaces or line breaks and/or the comments content (see Listing 1.2). *Textual approaches* are said to treat code merely as text, omitting the syntactical and semantical structures of a particular language; they are ideal for finding these clones.

Type-2 clones: They include, in addition to the alterations of Type-1 clones, lexical changes manifested as differences on data types or variable names (see Listing 1.3). Being the tokens of a language the unit of comparison for *Lexical approaches*, they are more suited to detect the differences in lexemes introduced by Type-2 clones.

Type-3 clones: Difficulty increases when it comes to detect these clones, due to the fact that they include also syntactical alterations in the form of instruction modification, deletion and insertion, which obscure even more the plagiarism act (see Listing 1.4). *Syntactical approaches* are characterized by the usage of more complex data structures, like compilers' intermediate representations (IRs) or source code metrics, which are able to characterize a program beyond the lexical level. In fact, intermediate representations are the central data structure of compilers where almost all the transformations and analysis are done [TC11], and hence, provide more useful information before execution time than plain text code does.

Type-4 clones: Detecting these clones is even more difficult, because the code here was practically rewritten as a new implementation, keeping only a few syntactical similarities with the original code (see Listing 1.5). *Semantic approaches* must use more advanced tools in order to detect them (e.g. control and data flow analysis) and are not as efficient as syntactical approaches.

2.1.3 Code Obfuscation

Code obfuscation is defined in [BS05] as the intentional practice of making code as much unintelligible as possible. It can be performed automatically in the compiler side by applying transformations that drastically change the structure of the code but preserve the program semantics. One motivation of obfuscating code is related to copyright protection. Another one is the concealment of malicious software: malware and viruses, for example, make use of obfuscation techniques to bypass detection from their counterpart applications. For a taxonomy of these techniques, the work of Collberg et al. [CTL97] can be consulted.

For example, Listing 2.2 shows a code segment in JavaScript, while Listing 2.3 shows the same code with some basic obfuscation techniques: lack of indentation, string encoding, creation of an array of strings, and identifier renaming. [1]

The comparison of obfuscated code pieces might be useful to test the robustness of a particular obfuscation technique. However, in this research, we were mainly interested in the detection of program similarity in a broader way. For this reason, we considered that the detection of strong obfuscated code pieces was out of the scope of the present work.

```
function NewObject(prefix)
{
  var count=0;
  this.SayHello=function(msg)
  {
    count++;
    alert(prefix+msg);
  }
  this.GetCount=function()
  {
    return count;
  }
}
var obj=new NewObject("Message : ");
obj.SayHello("You are welcome.");
```

Listing 2.2: JavaScript Original Code.

[1]Code samples where taken from https://javascriptobfuscator.com.

```
var _0x58a2=["\x53\x61\x79\x48\x65\x6C\x6C\x6F",
"\x47\x65\x74\x43\x6F\x75\x6E\x74",
"\x4D\x65\x73\x73\x61\x67\x65\x20\x3A\x20",
"\x59\x6F\x75\x20\x61\x72\x65\x20\x77\x65\x6C\x63\x6F\x6D\x65\x2E"];
function NewObject(_0x34c6x2)
{
var _0x34c6x3=0;
this[_0x58a2[0]]= function(_0x34c6x4)
{
_0x34c6x3++;alert(_0x34c6x2+ _0x34c6x4)
}
;this[_0x58a2[1]]= function()
{
return _0x34c6x3
}
}
var obj= new NewObject(_0x58a2[2]);
obj.SayHello(_0x58a2[3])
```

Listing 2.3: JavaScript Obfuscated Code.

2.2 Compiler's Intermediate Representations (IRs)

IRs are interesting for the problem of program comparison, due to the fact that they can deliver new and useful information about a program that might not be easily accessed by looking only at the raw code. The intermediate representation of a program is for the compiler the central data structure where most of the optimizations and analyses are made [TC11]. It acts as the middle man between the compiler's frontend and backend in the process of target code generation (see Figure 2.1). Once a program is converted to this representation, the compiler does not depend anymore on the written code to perform its passes. The passes of a compiler are progressive processes that act over the intermediate representation and transform it at each pass, in order to generate correct and optimized code according to the target language.

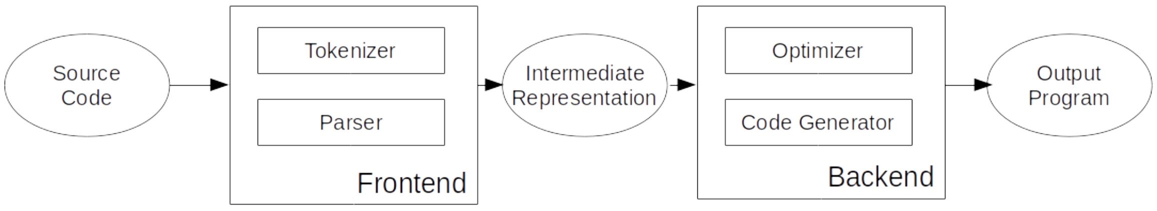


Figure 2.1: Typical Compiler Infrastructure.

For this reason, IRs need to represent the information of a program in a detailed way that is simply not possible with raw code. The level of detail is highly determined by the target language. High level IRs are closer to the source code and are preferred for source-to-source translation tasks, while lower level IRs are the indicated choice to generate machine code. In practice, compiler infrastructures rely on several interconnected IRs, which can be graphical or linear. For a deeper insight, refer to the book of Torczon and Cooper [TC11], from which we have summarize the following classification.

2.2.1 Graphical Intermediate Representations

Graphical intermediate representations store the program's information in a graph-like data structure. Their abstraction level is usually not far from the source code. In some cases, the original code can be recreated with minor modifications. The following data structures are among the most relevant graphical IRs:

Parse Trees

They are built up following the derivation rules of the language grammar. Hence, they are large, as each derived symbol receives a node on the tree. For example, Figure 2.2 shows a parse tree for the expression $a = b \times c + d \div e$.

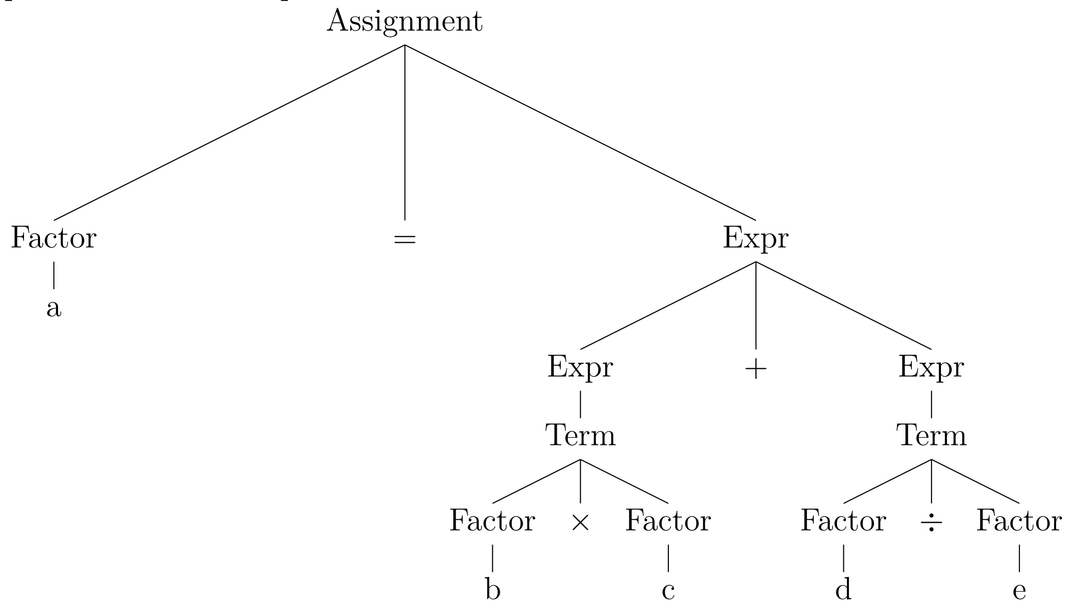


Figure 2.2: Parse Tree for $a = b \times c + d \div e$.

Abstract Syntax Trees (ASTs)

ASTs are defined as contractions of parse trees where most non-terminal symbols are ignored while the precedence and the meaning of the expressions are preserved, thus saving space. Usually, their level of abstraction is still not far from the original source code. For example, Figure 2.3 shows the AST for the same expression ($a = b \times c + d \div e$).

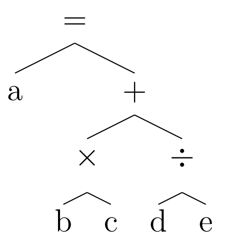


Figure 2.3: Abstract Syntax Tree for $a = b \times c + d \div e$.

They offer a balanced trade-off between size and abstraction. That is why it is one of the most common representations used inside compiler infrastructures. However, in practice, ASTs can be enriched with more detailed information that adds more nodes and attributes, hence increasing the complexity of the data structure, but with the ultimate purpose of reducing the number of passes needed to generate the target code.

Directed Acyclic Graphs (DAGs)

DAG are used in compilers as compact versions of ASTs, where there is only one subtree for an expression, no matter how many times it is repeated on the program. The tree structure is broken, as a node can be a child of multiple parent nodes. Special considerations have to be taken when the involved values in the expression are subjects to modifying operations, considerations that are not necessary when working with ASTs. DAGs are mainly used to reduce the overhead in systems with memory restrictions or to perform analysis that intend to expose redundancies in the source code.

Control Flow Graphs (CFGs)

CFGs are high level IRs used to represent the relationships between the basic blocks of a program. A basic block is defined as a set of instructions that are executed on its totality once the first instruction is reached. Control structures introduce disruptions in the program flow, as some instructions might not be reached depending on the conditional information. This creates new basic blocks that need to be interconnected. In a CFG, nodes represent basic blocks, while the edges represent the flow from one block to another. To be able to represent loops in a concise way, edges are allowed to point back to its originator node, which converts the graph into a cyclic data structure.

Program Dependence Graphs

They are auxiliary data structures used to associate a variable definition with each use of it. They can be very effective to help to detect the paths that have no dependency among each other, information of great utility for performing optimizations, like loop parallelization or reordering. The traditional PDGs are directed attributed graphs whose vertices represent the assignment statements and control predicates that occur in a program. Some of the vertices have an attribute that marks them as entry vertices, which represent the entry of procedures. The edges represent the dependences between the components of the program.

Call Graphs

These are also auxiliary data structures which keep track of the interprocedural interactions. Call graphs facilitate optimization passes that take into account the instructions of a linked function.

2.2.2 Linear Intermediate Representations

Linear IRs are low level intermediate representations consisting in compact sequences of instructions similar to assembly code. They are designed in this way to ease the generation of machine code. At this level of detail, it is more difficult to regenerate the original code without the help of other high level data structures. The following models are among the most relevant linear IRs: stack-machine code and three-address code.

Stack-Machine Code

As the name suggests it, they are based on a simple mechanism, where operands are push into the top of a stack, operations pull them up from it, and finally push the result back into the top. They belong to the category of one-address codes, characterized by instructions with one operation and an optional operand. The order of the instructions reflects the dependency of the operands and the operations. Data that is not transfered to memory will be lost, as the stack size is usually limited. This creates an implicit name space that reduces the size of the program considerably. For example, the corresponding stack-machine segment for the instruction $a = b \times c + d \div e$, would look like in Listing 2.4:

```
push d
push e
divide
push c
push b
multiply
add
```

Listing 2.4: Stack-Machine Code for $a = b \times c + d \div e$.

Three-Address Code

Among the popular linear IR models, there exist the three-address code. In this model, operations have at most an operator and three addresses: two operands and a result. Three-address codes exhibit compactness without forcing destructive operations, which gives room for further code optimization. Plus, many modern processor architectures are already based on three-address operations, which makes the translation more intuitive. For example, for the same instruction ($a = b \times c + d \div e$), the three-address code segment would look like in Listing 2.5:

```
$v1 \leftarrow d$
$v2 \leftarrow e$
$v3 \leftarrow v1 \div v2$
$v4 \leftarrow b$
$v5 \leftarrow c$
$v6 \leftarrow v4 \times v5$
$v7 \leftarrow v3 + v6$
```

Listing 2.5: Three-Address Code for $a = b \times c + d \div e$.

Three-address codes often use *static single-assignment* (SSA) as a naming convention. In SSA, each name definition corresponds to an operation and it is never rewritten or deleted. When control flow requires the selection between different variables, SSA uses ϕ functions to decide which value to select.

2.2.3 Hybrid Intermediate Representations

Hybrid IRs take advantage of the features of graph and linear representations. A typical configuration inside a compiler is to use a CFG to represent the basic block structure of the program, while, inside the basic blocks, the instructions are represented by their syntactical structure with and AST or a linear IR. Complex compiler infrastructures like LLVM [LA04], GCC [Gri02] and Open64 [Dev01] are examples of the usage of a design based on interconnected intermediate representations.

2.2.4 The LLVM Compiler Infrastructure

LLVM [LA04] is a complete compiler framework that uses different IRs to perform program analysis, transformation and code generation [LA14]:

- Abstract syntax trees (ASTs) are used as the first instance to represent C or C++ code parsed by Clang, a frontend for these languages.
- A linear representation, known as the LLVM IR, is the central data structure of the compiler.
- Direct acyclic graphs (DAGs) are used as auxiliary structures when translating to a machine-specific assembly language.
- Another data structure is used to implement assemblers and linkers.

LLVM is maintained as a collection of libraries, a design that facilitates its reuse.

The Clang Abstract Syntax Tree

Among the variety of tools available under this infrastructure, there exist Clang [Inf17], a frontend for C/C++/Objective C programs. Upon compilation, Clang first captures the syntactical structure of the program in an AST. Afterwards, the AST is traversed to generate the linear IR that is used by LLVM to perform transformation and optimization passes, and finally generate machine specific code. There are three core classes of AST nodes that represent the respective language constructs in C/C++ directly: *Declarations*, *Statements* and *Types*. All the class structure derives directly or indirectly from them.

Using the previous example ($a = b \times c + d \div e$), the abbreviated Clang AST would look similar to the one in Figure 2.4.

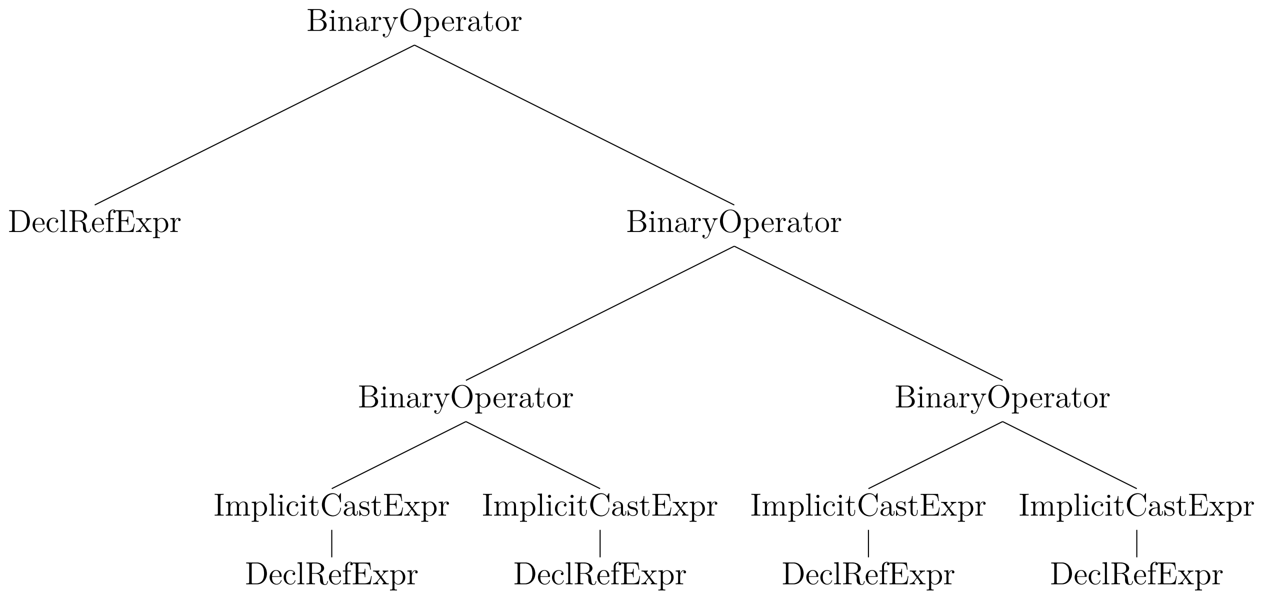


Figure 2.4: Abbreviated Clang AST for $a = b \times c + d \div e$.

The Linear Intermediate Representation of LLVM

According to Lopes and Auler [LA14], the LLVM Intermediate Representation is the backbone that connects the frontends and backends of LLVM, as well as its other IRs. It is designed as a three-address code and it uses the SSA naming convention. As a variable is never rewritten, the IR offers an unlimited number of virtual registers. The LLVM IR can be delivered in three equivalent forms: a) in-memory representation, b) on-disk bitcode representation or c) on-disk human-readable representation.

To illustrate some characteristics of this IR, we have placed the same instruction ($a = b \times c + d \div e$) inside a simple C function (see Listing 2.6), and generated its respective LLVM IR. Listing 2.7 shows how the same instruction ($a = b \times c + d \div e$) would look in human-readable LLVM IR. Notice that:

- Data types in the example are specified with the keywords **i32** and **float**.
- Local variables start with **’%’**, while global variables and functions start with **’@’**. The use of these special characters helps to avoid conflicts with reserved words.
- The contents of a function are enclosed by brackets.
- The **alloca** operation reserves space for a variable in the stack.
- Their addresses are later referenced with C style pointers using the character **’*’** after the data type keyword.
- The **store** operation saves a given value into an address of the stack.
- The **load** operation pulls a value from a memory location and puts it into a virtual register.

- The **align** keyword specifies the size of the address, which must be a multiple of 2.
- The **ret** keyword serves as the exit point for the function.
- Arithmetic operations **mul**, **sdiv** and **add** use a maximum number of operands of three, complying with the three-address code model.

```c
float a = 0;
void example()
{
  int b = 1;
  int c = 2;
  int d = 3;
  int e = 4;
  a = b * c + d / e;
}
```

Listing 2.6: C code segment enclosing the instruction $a = b \times c + d \div e$.

```llvm
;global variable
@a = global float 0.000000e+00, align 4

;global function
define void @example() #0
{
  ;stack allocation of local variables
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4

  ;local variable value initialization
  store i32 1, i32* %1, align 4
  store i32 2, i32* %2, align 4
  store i32 3, i32* %3, align 4
  store i32 4, i32* %4, align 4

  ; a = b * c + d / e
  %5 = load i32, i32* %1, align 4
  %6 = load i32, i32* %2, align 4
  %7 = mul nsw i32 %5, %6
  %8 = load i32, i32* %3, align 4
  %9 = load i32, i32* %4, align 4
  %10 = sdiv i32 %8, %9
  %11 = add nsw i32 %7, %10
  %12 = sitofp i32 %11 to float
  store float %12, float* @a, align 4

  ;return
  ret void
}
```

Listing 2.7: LLVM Linear IR code segment for $a = b \times c + d \div e$.

2.3 Parallel I/O

Parallel File Systems are minded for accessing files in a simultaneous, concurrent and efficient way. A comprehensive description can be found on the Ph.D thesis of Kunkel [Kun13]. The contents of a file are usually scattered among different I/O subsystems in order to take advantage of the highest local performance. These systems should provide, among other capabilities:

- *Persistence:* meaning data should be available for access any time later.
- *Consistence:* which means that accessed data should correspond to the original stored data.
- *Performance:* related to the efficient use of the underlying subsystems.
- *Manageability:* or the availability of tools to mount, check or repair the file system.
- *Scalability:* defined as the capability to work with an increasing numbers of clients.
- *Fault-tolerance:* the systems must be able to tolerate errors without compromising the correctness of the data.
- *Availability:* is the ability to continue operation (maybe with degraded performance) in the presence of those errors.

2.3.1 Performance

Several factors influence I/O performance in a parallel environment. Kunkel [Kun13] made a survey of the ones with the most impact:

- *Caching:* Improvement in the performance is gained when data is efficiently buffered from a slow access storage into main memory. Common strategies are the combination of separate write operations into a single write block, and the load of data ahead of its use.
- *Replication:* In a distributed environment, the creation of local copies of a single file can improve the performance, due to the reduction of network traffic.
- *High Availability:* Replication can also provide High Availability; if a network node is down, there still a copy of the needed file in another node. A trade-off in the number of replicas has to be found to avoid affecting the performance of the system.
- *I/O Forwarding:* In certain networks, requests from client nodes are not taken directly by server nodes. Instead, intermediate nodes analyze and forward those request appropriately, to avoid bottlenecks. However, a bad design can decrease performance by adding unnecessary network traffic.
- *Aggregation:* Independent operations coming from different nodes can be collected in a buffer before being written in the physical I/O, thus improving performance.
- *Scheduling:* A good scheduling algorithm is necessary to manage the queue processing. Without it, the system is susceptible of load imbalance and bottlenecks.

- *Parallel Design and Resource Consumption:* A good design should be able to use all the components (network, severs, and I/O subsystems) at its most efficient extent.
- *Software Implementation:* Computation affects I/O indirectly. Software components should be design taking into account the characteristics of I/O. Data structures like trees and hashes are optimal for this task.

Different approaches can be used to analyze the performance of a parallel file system. Investigating the mere performance, however, is not sufficient as optimal performance is a function of the exhibited access pattern. This makes it difficult for an observer to assess any measured performance. A typical strategy is to relate performance of an application with a similarly behaving application for which we know how well it behaves – like benchmarks, for example. Likewise, we may be interested to identify well-behaving or ill-behaving applications. Therefore, finding patterns via fingerprints inside I/O traces is an important use case.

2.3.2 I/O Access Patterns

I/O traces collected during the monitoring of data-intensive applications may contain timestamps, operation names and information about the number of bytes involved, which constitute access patterns that depict the behavior of storage usage over a period of time. These patterns can be used to determine the overall performance of an I/O system, e.g. one can seek for bottlenecks in I/O operations. They can be characterized with the following properties:

- *Access Granularity:* In other words, it is the amount of data accessed per I/O call or request. To avoid latency, this amount should be as big as possible.
- *Randomness:* Refering to the closeness among the accessed bytes. Physical devices perform better if the accessed data is contiguous.
- *Concurrency:* It is related to the number of concurrently issued I/O requests. The I/O scheduler should interleave requests in an efficient way.
- *Load Balance:* It makes reference to the distribution of the workload among the servers/devices. A good parallel access pattern would distribute data among all servers accordingly to their capabilities.
- *Access Type:* They can essentially be read or write.
- *Predictability:* It refers to the presence of a more or less regular pattern over time. Repetitive patterns are easier to optimize. Liu et al. [Liu+14] mentioned three predictable features seen on supercomputing I/O patterns:

- *Burstiness:* I/O in scientific applications presents different phases with a different intensity on the number of performed operations. The intensity of the access to the media on each phase is the so-called burstiness.
- *Periodicity:* Usually the I/O bursts do not show randomly. Instead they present a periodic behavior. For example, checkpointing is made periodically.
- *Repeatability:* Large scale applications tend to generate similar I/O traces with different data, which makes possible to think on I/O access patterns as a signature of the application.

2.4 Kernel Methods for Similarity Search

Shawe-Taylor and Cristianini have documented on their book [SC04] all the relevant aspects of Kernel Methods[2]. These methods are strong enough to detect stable patterns robustly and efficiently from a finite data sample by embedding it into a space of higher dimensionality where data points have linear relations (see Figure 2.5). Robustness is related to the capacity of a system to cope with erroneous data.

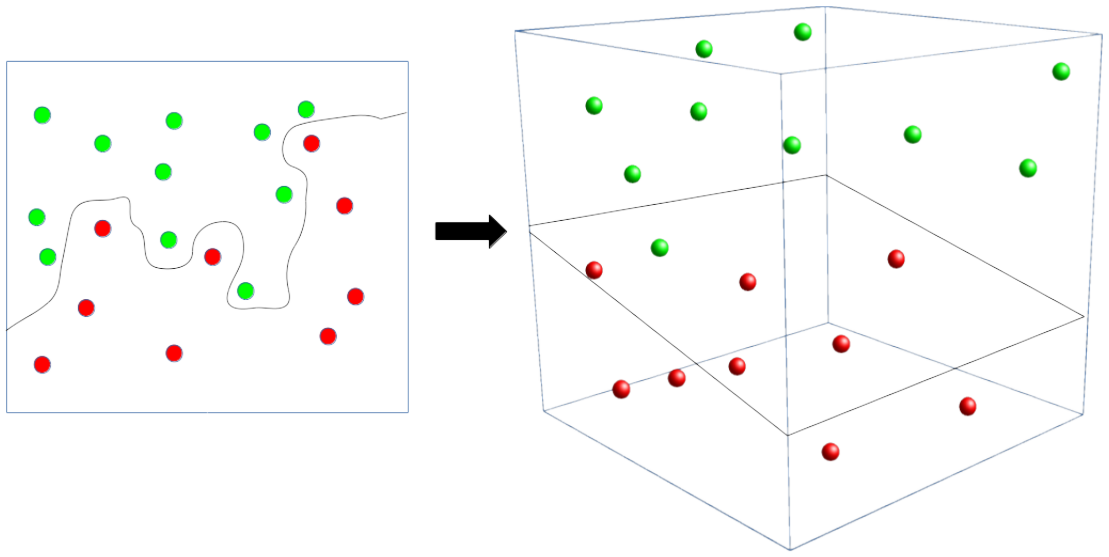


Figure 2.5: Mapping from a 2D input space with non-linear patterns into a 3D feature space with linear relations.

However, the explicit calculation of the new points is not necessary, if there is a function that can infer those linear relationships using only the original data. That function is called the *kernel function*.

2.4.1 Generalities

The construction of a kernel function is governed by the following premises:

[2]Some of the following paragraphs are based on the Master's thesis of Torres [Tor11].

1. Original data items should be embedded into a vector space called the *feature space*.
2. The images of the data in this new space present linear relations.
3. The learning algorithm does not need to know the coordinates of the data in the feature space; the pairwise inner products are enough.
4. These inner products can be calculated in an efficient way using a kernel function. This is the *kernel trick*.

To illustrate how this works, consider an embedding mapping such:

$$\phi : x \in \mathbb{R}^n \longmapsto \phi(x) \in F \subseteq \mathbb{R}^N.$$

This mapping will recode the original dataset S as:

$$S' = (\phi(x_1), y_1), ..., (\phi(x_l), y_l).$$

The kernel matrix $G = XX'$ is conformed by the inner products between data points $\langle \phi(x), \phi(z) \rangle$ in the feature space. However, a direct calculation of these points may be infeasible because the induced space might have several or infinite dimensions. Instead, a kernel function calculates those inner products efficiently using only the original data:

$$k(x, z) = \langle \phi(x), \phi(z) \rangle \, where \; \phi : x \longmapsto \phi(x) \in F.$$

2.4.2 Kernel Methods as Machine Learning Systems

A typical machine learning system consists of two subsystems [Kun14]: the *Feature Extraction Subsystem* and the *Learning Subsystem*. Kernel methods follow this design in a modular way:

Feature Extraction Subsystem

In the case of Kernel Methods, the *kernel function* builds the *kernel matrix*, which is the only information needed for the next subsystem. The kernel matrix can be seen as a similarity score among the original examples. The design of this function depends highly on the nature of the original data. Attribute-value tuples can be easily handled with polynomial or Gaussian kernels [Gen01]. Instead of using the original attribute to characterize a data item, polynomial kernels use a polynomial function that intuitively captures the interactions between those features. Gaussian kernel does the same but using instead a radial basis function that emits a similarity value between 0 and 1. String and tree objects require string and tree kernels respectively [VS03].

Learning Subsystem

The learning subsystem requires only the kernel matrix as the source of information. Examples are no longer characterized with a set of attributes related to some properties inherent to it. Instead, the new attributes correspond to the similarity score among examples. Learning algorithms are agnostic to that, so in theory, any of them able to deal with matrices might work. In practice, some algorithms are specially designed to deal with data coming from a kernelized space, like *kernel principal component analysis* (Kernel PCA) [SSM97] or *support vector machines* (SVM) [Gun+98].

Kernel Functions for Structured Data

Structured data like trees and strings present a more difficult challenge at designing an appropriate kernel function that can efficiently work with them.

Convolution Kernels: These kernels are detailed by Gaertner, in [GLF04], and Haussler, in [Hau99]. They are claimed to be the best approach for representing spaces that are not mere attribute-value tuples. These kernels exploit the composition relationship between objects.

Suppose $x, y \in X$, and $\vec{x} = x_1, x_2, x_3, ..., x_D$ are parts of x, and $\vec{y} = y_1, y_2, y_3, ..., y_D$ are parts of y, where $1 \leq d \leq D$. A kernel between the corresponding parts of both objects is $K_d(x_d, y_d)$, and gives a similarity measure between those parts. Therefore, the kernel evaluation between the objects is:

$$K_{conv}(x, y) = \sum_{\vec{x},\vec{y}} \prod_{d=1}^{D} K_d(x_d, y_d).$$

String Kernels

Strings kernels [VS03] are a convolution kernel subset that checks for the number of shared substrings among a collection of strings. Consider the following:

- A is a finite set of characters conforming the alphabet.
- A string is any $x \in A^k$ for k=0,1,2,...
- A^* are all the non-empty strings.
- $s, s', x, y \in A^*$ are strings.
- $num_s(x)$ corresponds to the number of occurrences of s in x as a substring.
- $\delta_{s,s'}$ corresponds to the Kronecker delta, emitting 1 as a value when s and s' are equal, or zero otherwise.
- w_s is a weighting factor that modulates the contributions of certain substrings.

A generic string kernel is defined as follows:

$$k(x,y) = \sum_{s\subseteq x, s'\subseteq y} w_s \delta_{s,s'} = \sum_{s\in A^*} num_s(x)\cdot num_s(y)\cdot w_s.$$

As mentioned, the weighting factor can modulate which substrings have an effect on the calculation. By changing the weighting factor, different kernels can be obtained:

***k*-spectrum kernel [LEN02]:** The k-spectrum kernel only counts sub-strings of length k:

$$w_s = 0\ \forall\ |s| \neq k.$$

Tables 2.1, 2.2 and 2.3 show examples of this kernel for different values of k.
It was originated from the protein classification problem. In the specific problem of remote homology detection and in combination with an support vector machine, they were able to achieve comparable performance to some state-of-the-art methods for the same problem.

***k*-blended spectrum kernel [SC04]:** The k-blended spectrum kernel is an improved version of the previous kernel. It only counts sub-strings whose length is smaller or equal to a given number k:

$$w_s = 0\ \forall\ |s| > k.$$

The summation of all scores (15) from Tables 2.1, 2.2 and 2.3 corresponds to the 3-blended spectrum kernel for the given strings.

bag-of-characters kernel: The bag-of-characters kernel only takes into account single-character matching:

$$w_s = 0\ \forall\ |s| > 1.$$

It corresponds to the 1-spectrum kernel (See Table 2.1).

bag-of-words kernel: The bag-of-words kernel searches for shared words among strings. Words are separated by a delimiter, usually a white space (see Table 2.4).

	a	**a**	**b**	**c**	**c**
a	1	1	0	0	0
b	0	0	1	0	0
c	0	0	0	1	1
c	0	0	0	1	1
c	0	0	0	1	1

Table 2.1: bag-of-characters (1-spectrum) kernel score for *aabcc* and *abccc*.

	aa	**ab**	**bc**	**cc**
ab	0	1	0	0
bc	0	0	1	0
cc	0	0	0	1
cc	0	0	0	1

Table 2.2: 2-spectrum kernel score for *aabcc* and *abccc*.

	aab	**abc**	**bcc**
abc	0	1	0
bcc	0	0	1
ccc	0	0	0

Table 2.3: 3-spectrum kernel score for *aabcc* and *abccc*

	this	**is**	**not**	**a**	**house**
a	0	0	0	1	0
house	0	0	0	0	1
this	1	0	0	0	0
might	0	0	0	0	0
be	0	0	0	0	0

Table 2.4: bag-of-words kernel calculation for "*this is not a house*" and "*a house this might be* ".

The Learning Algorithm

For this study we selected two algorithms with a wide spread on the scientific community, due to fact that they facilitate data visualization:

Hierarchical Clustering (HC): Hierarchical Clustering is an unsupervised learning algorithm for visualization of relationship between observations. Observations are fed to the algorithm in the form of a similarity or dissimilarity matrix [HTF03]. The algorithm builds up high-level clusters by merging low-level clusters. Visualization is made through dendrograms, a highly human-interpretable image, one of the reasons of its popularity.

On the one hand, HC can start from the top assuming a unique cluster, and then it continues separating into smaller groups. On the other hand, it can start from the bottom, taking each observation as as cluster and pairing clusters with less dissimilarity; at the end one cluster gathers all the sub-clusters [JMF99].

The distance between two clusters A and B, can be calculated as follows:

- *Single link:* It corresponds to the minimum distance between any observation in A and any observation in B.
- *Complete link:* This one, on the contrary, is the maximum distance between any observation in A and any observation in B.

- *Average link:* It calculates the average of the distance of all the examples in different clusters.

Kernel Principal Component Analysis (Kernel PCA): Kernel principal components analysis [SSM97] reduces the dimensionality of data for the sake of visualization. Kernel PCA performs *principal components analysis* (PCA) in a kernel-defined feature space [SC04] represented by a kernel matrix. Normally, PCA precises the eigenvectors and eigenvalues of the covariance matrix of the observations to project new data into the principal components. For the feature space $X = \phi(x_1), ..., \phi(x_l)$, the covariance matrix is given by:

$$lC = XX'$$

This information is unknown. But notice that the kernel matrix is given by a similar operation:

$$K = X'X$$

There is a clear relationship between both matrices; a u eigenvector of lC is derived from the corresponding eigenvector v and eigenvalue λ of K:

$$u = \lambda^{-1/2} X'v$$

For a specific $u_c, c = 1, ..., l$:

$$u_c = \lambda_c^{-1/2} \sum_{i=1}^{l} (v_c)_i \phi(x_i) = \sum_{i=1}^{l} \alpha_i^c \phi(x_i)$$

At this point it is not possible to calculate u_c due to the fact that it depends on the explicit calculation of $\phi(x_i)$. However, the projections of new points onto the direction of u_c can be calculated using the kernel matrix values:

$$P_{u_c}(\phi(x)) = u_c' \phi(x) = \left\langle \sum_{i=1}^{l} \alpha_i^c \phi(x_i), \phi(x) \right\rangle = \sum_{i=1}^{l} \alpha_i^c \left\langle \phi(x_i), \phi(x) \right\rangle$$

$$P_{u_c}(\phi(x)) = \sum_{i=1}^{l} \alpha_i^c k(x_i, x)$$

These projections in a reduced number of dimensions (first, second and third component) facilitate data visualization.

2.5 Related Work

This section explores the related work in three important areas: I/O patterns and program similarity are the target fields of this research. However, we also explored the application of kernel methods in other fields like protein analysis and natural language processing (NLP).

2.5.1 Program Similarity and Plagiarism

The work of Fu et al. [Fu+16] proposed a weighted kernel method for source code plagiarism detection based on ASTs. The weights of each AST node are here determined by a technique called TF-IDF (Term Frequency-Inverse Document Frequency). The authors claimed that their method resulted on improved detection in comparison with two other plagiarism detection tools, namely JPlag [JPl17] and Sim [GT99].

A tree kernel-based approach for clone detection can be found in the work of Corazza et al. [Cor+10]. They proposed a kernel based method to detect code clones using syntactic and lexical information. In the method, the nodes of the AST are characterized with a set of four attributes: *Instruction class*, *Instruction*, *Context* and *Lexemes*. The similarity score of two nodes is discretized into six values that are chosen by the authors after an adjustment process, and depends on the matching of the above mentioned attributes.

Bandara et al. [BW13] used an unsupervised technique called *sparse auto-encoder* to extract the features from a piece of code. Logistic regression was used to predict the author of a code segment.

In the direction of code smells detection, Danphitsanuphan et al. [DS12] proposed an approach to find a relationship between code smells and software structure bugs. Code smells are pieces of software that create difficulty to understand and improve programs. The authors focused on the following code smells, which can be easily measured via software metrics: *large class*, *large method*, *long parameter* and *lazy class*. Software structure bugs are defects originated on wrong commands at writing code. The authors focused on the following ones, which are frequent in Java programs: *bad field*, *ignored exceptions*, *field not initialized in constructor*, *dead local store*, *general exception catching* and *integer value casting*.

We can read in the paper of Park at al. [HP14] that they studied a combined approach of parse trees and function-call graphs. They created a composite kernel, with the parse tree kernel and the graph kernel. It is interesting how they modulated the weight of the kernels according to the cyclomatic complexity of the code. They used ANTLR to generate the parse tree. In their experiment, they used a set of programs already labeled as plagiarized. Compared against CCFinder, JPlag, and the standalone kernels they merged, they achieved the highest F1-score.

The motivation of the work of Sharma et al. [SPP07] was intrusion detection. They attempted not only to achieve high accuracy but also minimize the false positive rate in the detection. They used a bag-of-words approach to characterize intrusion; the words corresponded to the system calls that the program performed. In this way, a process can be characterized as a set of unordered system calls. They made an experiment to compare the performance of several scores based on a binary cosine similarity measure proposed in the paper of Liao et al. [LV02], and a radial basis function kernel (RBF). Though all the distances achieved 100% detection rate, the RBF kernel based techniques achieved the lowest false positive rate.

Automatic file type identification is also a field where similarity plays an important role. In the work of Gopal et al. [Gop+11], the authors tried to identify file types using machine learning. They used a feature-based approach based on “N-gram” bytes, which has a high relationship with the *k-spectrum kernel*. The value of “N” was empirically selected using cross validation. Support vector machines and multi-class k-nearest neighbors were employed as classification algorithms. Labels were taken from the file extension when the files were intact, and from a commercial tool when they were damaged.

In the work of Wang et al. [Wan+15], a new methodology was developed to detect platform-specific code smells (PSCSs) in HPC applications using abstract syntax trees and XML. A code smell is a code associated with a design problem which makes the application code hard to evolve and maintain. The detection method proposed by the authors takes the source code and parses it into AST with a clear hierarchy. After this the AST is converted to an XML document and finally XPath is used for identifying PSCSs patterns, showing this data to the user before the application execution. For example, working with the OpenACC programming language for GPUs, the principal types of PSCSs are : *triangular loop*, *live-out scalars*, *once-used array data*, *computed index*, *variable length loop*, *common subexpression* and *loop invariant*. This class of codes have a regular structure that can be identify into an XML file that represent the general AST. The pattern matching process is made using Xpath, a W3C standard language for expressing traversal and navigation in XML trees. This standard allows the detection of PSCSs patterns in a systematic way and is broadly descriptive.

The major difference of the solution proposed in this work with respect to the listed works is the flattening of the intermediate representation into a weighted string and the usage of string kernels as the comparison method.

2.5.2 I/O Patterns for Parallel Evaluation

Kluge [Klu11] proposed an intermediate representation of I/O events from HPC applications as a directed acyclic graph (DAG). In this DAG vertices are used to represent events, while edges are used to depict the chronological order of the events: the event represented by the vertex, where the edge has been originated, happened before the one where the edge points to. Two types of vertices are recognized:

- *I/O event vertex:* It represents an operation happening in a single process. It has only one incoming vertex and one outgoing vertex.
- *Synchronization vertex:* It involves several processes. It has as many incoming edges as the number of synchronized processes and as many outgoing edges as the number of spawned processes.

Kluge also proposed a redundancy elimination step where adjacent synchronization vertices can be merged into a single one.

Madhyastha et al. [MR02] applied two supervised learning algorithms to classify parallel I/O access patterns: a feed forward artificial neural network (ANN) and a hidden Markov model (HMM). Both strategies required training with previously labeled examples.

The artificial neuronal network was configured with 13 input and 10 output nodes with a hidden layer of 12 units. The number of units in the hidden layer was subject of a reduction to the smallest number that could offer the best precision. Standard back-propagation was used as the training algorithm. A hidden Markov model was fed with I/O access patterns and used to calculate the probability of accessing a given portion of a file in the future. HMMs were ideal to learn the hidden behavior of the application.

Behzad et al. [Beh+15] proposed an I/O auto-tuning framework that extracts the patterns from an application and searches for a match on a database of previously known pattern models. If there was a match, the associated model was adopted on the fly during the execution of the application. In order to perform the matching, the patterns must be abstracted using an array distribution notation based on High Performance Fortran syntax.

A different abstraction approach was made by Liu et al. [Liu+14]. They used the I/O bursts registered on noisy server-side logs of an application as a signature to find similarities between I/O samples. These logs were processed, their granularity was refined and the noise was reduced. The final signature was a 2D grid called CLIQUE [Agr+98] that related a coefficient with time. A CLIQUE performs multidimensional data clustering by identifying high density grids. Because the signature extraction was made over log files there was zero overhead in the application performance. Their experimentation showed that this signature can be effectively extracted regardless the significant noise in the server logs.

Koller and Rangaswami [KR10] used disk static similarity and workload static similarity at the block level to analyze the performance of concurrent applications of the same file system.

In the work of Luo et al. [Luo+15], an I/O tracing framework with elastics traces was developed. Their method is compound by four steps:

1. Gathering of a set of lossless and scalable I/O trace files.
2. Analysis of the set of trace files and extrapolation into large trace files.
3. Calculation of extrapolated data and creation of a single trace file.
4. I/O replay and verification.

To make full analysis over I/O processes, three tools were developed for Single Program Multiple Data (SPMD) programs:

- *ScalaIOTrace:* It allows lossless tracking and recording of the delta time between events and I/O calls with all parameters. For doing this, MPI-IO is intercepted at the MPI profiling layer. POSIX I/O is captured at link time interposed with domain-specific parameter compression.
- *ScalaIOExtrap:* It has the goal of obtaining I/O behavior by exploiting different methods: *high-level extrapolation*, *elastic string extrapolation*, *elastics data element extrapolation* and *handless and time extrapolation.*
- *ScalaIOReplay:* It provides a parallel trace replay of all events across tasks, preserving order of events.

Their experiments demonstrated that structural trace comparison, I/O size and execution time remains sufficiently accurate. The method also preserved event ordering and time accuracy in these large traces.

In another work, Feng et al. [Fen+15] studied the I/O performance of Hadoop. Hadoop is a MapReduce framework highly used for data-intensive work. The overall performance of Hadoop applications is greatly affected by their I/O performance. To study it, they injected byte codes into Hadoop systems and got related I/O traces. However, acquiring the I/O behaviors without an API interface is difficult. To solve this, they built a Java-based program that automatically collected Hadoop traces: the IOSIG+ Java tool. This tool has the capacity of tracing with low overhead on a Hadoop system. Its architecture consists of four major components:

- *KVTracer:* Used for tracing intermediate key-value pairs in map-reduce tasks.
- *Stream-Traces:* Used to watch inside of each data node.
- *CollectionsServer:* It saves all I/O signatures in ramdisc and transfers compact trace files.
- *TraceAnalyzer:* Offline tool to extract and analyze I/O behaviors.

IOSIG+ is claimed by the authors to be capable of capturing comprehensive details of I/O behaviors of Hadoop applications with a low overhead.

Byna et al. [Byn+08] used parallel I/O prefetching to study I/O systems. I/O prefetching is a technique used to improve file access for future consults based in regular patterns process of access to data. Their method detects the pattern of I/O accesses of an application, stores the pattern information as a signature representation, and reuses this signature for future consults to the data.

Their method functions as follows: Traces of a running application are collected and analyzed in order to find a I/O pattern, which is stored as an I/O signature. That signature contains information like repetitions of data access and size of data. After that, all signatures of an application are read, verified and used to prefetch data when a pattern is found.

The authors identified two major classes of patterns: *global patterns*, show how multiple process access a file, while *local patterns* depict a file access which is controlled by only one process. Local patterns can be organized into I/O signatures of five dimensions: spatiality, request size, representative behavior, temporal intervals and type of I/O operation.

As it has been shown, none of the mentioned works have made use of string kernels to perform a similarity study on the access patterns, which corresponds to the major line of investigation of our work.

2.5.3 Kernels in Other Fields

In the work of Zhang et al. [Zha+08], we can see the use of an improved version of the convolution kernel for semantic role labeling. Semantic role labeling attempts to assign labels to the elements that compound a parse tree of a natural language sentence. They attempted to incorporate linguistic knowledge on the convolution kernel.

Tikk [Tik+10] evaluated several kernels for the protein-to-protein interaction (PPI) problem and conclude that the *shallow linguistic kernel* outperformed the others. Additionally, they found out that Dependency trees are better than Syntax trees to extract features for PPI.

2.6 State of the Art

This section covers two important topics: On the one hand, the common algorithms used for detecting similarity in code. On the other hand, the most effective code clone detectors at the moment.

2.6.1 Algorithms for Similarity Detection

Beth [Bet14] and Cesare [CX12] made separated surveys on the strategies to find similarities among code pieces:

Levenshtein Distance [Lev66]

This algorithm is also known as the edit distance. The distance between two strings is defined as the minimum number of single character editions (deletions, insertions or replacements) that have to be made to convert one string into the other. As strings are usually of different sizes, a normalized version was proposed in [YB07].

Tree Edit Distance [Bil05]

The tree edit distance is a problem where a cost function is defined on each edit operation of a tree. An *edit script* S between two trees, namely T_1 and T_2 is a sequence of edit operations that turn T_1 into T_2. The total cost is the sum of the cost of each operation in

S. The minimum possible cost for this operation is the *tree edit distance* $\gamma(T_1, T_2)$. The general edit distance problem is difficult to solve, then different authors have introduced restricted versions for the problem.

To compute an ordered edit distance, two algorithms can be used.

Klein algorithm: Let F_1 and F_2 be ordered forest and γ be a metric cost function defined on labels. Let ν and ω be the rightmost root of the trees in F_1 and F_2 respectively. We have:

$$\begin{aligned}
\delta(\theta,\theta) &= 0\\
\delta(F_1,\theta) &= \delta(F_1-\nu,\theta)+\gamma(\nu\to\lambda)\\
\delta(\theta,F_2) &= \delta(\theta,F_2-\omega)+\gamma(\lambda\to\omega)\\
\delta(F_1,F_2) &= min\begin{cases}\delta(F_1-\nu,F_2)+\gamma(\nu\to\lambda)\\ \delta(F_1,F_2-\omega)+\gamma(\lambda\to\omega)\\ \delta(F_1(\nu),F_2(\omega))+\delta(F_1-T_1(\nu),F_2-T_2(\omega))+\gamma(\nu\to\omega)\end{cases}
\end{aligned}$$

Where $F-\nu$ denote the forest by deleting ν from F. $F-T(\nu)$ is the forest obtained by deleting ν and all descendants of ν. This equation suggests a dynamic program because the value $\delta(F_1, F_2)$ depends on a constant number of subproblems of smaller size.

Zhang and Shasha's algorithm: This algorithm defines the *keyroots* of a rooted, ordered tree T as follow:

$$keyroots(T) = \Big\{root(T)\Big\} \cup \Big\{\nu \in V(T) \,|\, \nu \,\text{has a left sibling}\Big\} \tag{2.1}$$

The relevant subproblems of T with respect to the keyroots are the prefixes of all special subforest $F(\nu)$.

The graph edit distance [SF83] is an specialization of the tree edit distance, where the edition operations are related not only to nodes but also to branches.

Winnowing [SWA03]

This algorithm is an efficient method to obtain fingerprints, useful to detect partial copies in documents. The detection of partial copies implies the partition of the document into small units called "k-grams". A "k-gram" is defined as a contiguous substring of a given size. Afterwards, a hash function is applied over the "k-grams", thus generating a subset of hashes that become the fingerprints of the document. When two documents share the same fingerprint, it is very probably that they share some "k-grams". It is recommended to select a large window size, in order to detect large enough matches. The algorithm warranties that at least one "k-gram" is shared in every match. The algorithm is also insensitive to white spaces, letter case, or punctuation. For the correct performance of the algorithm two limits have to be established: the upper limit controls the desired extension of the match, while the lower limit helps to avoid noise.

A key factor of this algorithm is that the selection of the hash depends only in the window content (the position of the window in the outside context is irrelevant). The authors found out that for a better performance of the algorithm, it is better to work with 64 bits to avoid accidental collisions.

Greedy String Tiling [Wis93]

Greedy string tiling (GST) is a method to determinate the degree similarity between two strings based on one-to-one matching.

GST aims to make comparisons of tiles (associations of a substring from P with a matching substring from T) assuming there is a current maximum match length which is the length of the largest maximal matches remaining obtainable from P and T (see listing 2.8).

```
length_of_tokens_tiled :=0
Repeat
  maxmatch := minimum-match-length
  starting at the first unmarked token of $P$, for each $P_p$ do
    starting  ath teh first unmarked token of $T$, for each $T_t$ do
  j := 0
  while $P_{p+j}$ = $T_{t+j}$ AND unmarked($P_{p+j}$) AND unmarked($T_{t+j}$) do
    j := j + 1
  if j=maxmatch then add match(p,t,j) to list of matches of length j
  else if j > maxmatch then start new list with match(p,t,j) and maxmatch :=j
  for each match(p,t, maxmatch) in list
    if not occluded then
  for j:=0 to maxmatch - 1 do
    mark_token($P_{p+j}$)
    mark_token($T_{t+j}$)
  length_of_tokens_tiled := length_of_tokens_tiled + maxmatch;
Until maxmatch = minimum-match-length
```

Listing 2.8: Greedy String Algorithm.

GST initially assumes that *maxmatch* is $minimum - match - length$. If a maximal match of that length is found, it is added to the list. Otherwise if a longer maximal match is found a new list is started. During the second phase, with each iteration the value of *maxmatch* will decrease monotonically until it becomes the global $minimum - match - length$. The algorithm above is optimal, in terms of maximizing the coverage of the strings, and computes a metric, provided that strings down to length 1 are allowed. Moreover it has worst-case complexity of $O(n^3)$.

To reduce this complexity, standard techniques are applied: the Karp-Rabin algorithm is based on the notion that if a hash value exists for a string of length s starting at t, the hash-value for a string of length s starting at $t + 1$ can be calculated using a simple recurrence relation. If two hash values are identical, the pattern and text substrings are compared item-by-item. The resulting technique has an average complexity that is close to linear, becoming a very good option in plagiarism detection and in computational biology to study genetic sequences.

Radcliff/Obershelp algorithm [Rat88]

Ratcliff and Obershelp developed an algorithm capable of deciding how similar two unidimensional patterns are; the output is given in the form of a confidence factor or percentage.

Initially, the algorithm examines two strings and locates the largest common substring. This match is used as an anchor among the strings. The algorithm keeps searching to the left and to the right of this anchor and repeats the procedure until all substrings are analyzed.

Consider two strings S and T. $|c|$ corresponds to the amount of characters in common found by the algorithm. The confidence factor is calculated as follows:

$$P(S,T) = \frac{2 \times |c|}{|S| + |T|}.$$

The worse-case scenario happens when there are no character coincidences, as $|S| \times |T|$ comparisons should be made. To reduce this number, the algorithm keeps track of the largest substring size: in case that the rest of the string is smaller than this size, the comparison skips this part.

The Normalized Compression Distance [CV05]

In order to understand the normalized compression distance, let us review its fundamental background.

Metric: It is a function (D) on a non-empty set mapped in R^+ satisfying:

1. $D(x,y) = 0$ if $x = y$.
2. $D(x,y) = D(y,x)$.
3. $D(x,y) \leq D(x,z) + D(z,y)$.

Metrics can be seen as measures of similarity between two objects. For example, if $D(x,y) \simeq 0$ then y and y are considered very similar.

Compressor: A compressor C is a function that reduces the size of data without losing information. C is normal if it satisfies the following:

- $C(xx) = C(x)$, and $C(\lambda) = 0$, where λ is the empty string.
- $C(xy) = C(yx)$.
- $C(xy) + C(z) \leq C(xz) + C(yz)$.

Normalized Information Distance: It is based on the Kolmogorov complexity, which is the length of the shortest binary program, for the reference universal prefix Turing machine, that on input of a string y, it converts it into x; it is denoted as $K(x|y)$. Essentially, the Kolmogorov complexity of a file is the length of the ultimate compressed version of the file. Thus the *Normalized Information Distance* is defined as:

$$NID(x,y) = \frac{max\Big\{K(x|y), K(y|x)\Big\}}{max\Big\{K(x), K(y)\Big\}}.$$

Compression Distance: It is defined based on a normal compressor and shows it is an admissible distance. A compressor C approximates the information distance $E(x,y)$, based on Kolmogorov complexity, by the compression distance $EC(x,y)$ defined as:

$$E_c(x,y) = C(xy) - min\Big\{C(x), C(y)\Big\},$$

where $C(xy)$ denote the compressed size of the concatenation of x and y, $C(x)$ and $C(y)$ denotes the compressed size of x and y respectively.

Normalized Compression Distance: It is the normalized version of the admissible distance $EC(x,y)$:

$$NCD(x,y) = \frac{C(xy) - min\Big\{C(x), C(y)\Big\}}{max\Big\{C(x), C(y)\Big\}}.$$

In practice, the NCD is a non-negative number $0 \leq r \leq 1 + \varepsilon$ representing how different the two files are. Smaller numbers represent more similar files.

2.6.2 Clone Detection Tools

Several surveys about the latest clone detection techniques have been performed in recent years. Ragkhitwetsagul et al. have recently published a comparison study of 30 code similarity detection techniques [RKC17]. Another recent study from Svajlenko et al. analyzed other tools [SR14]. Former studies include the work of Hage [HRV10], Roy et al. [RCK09] and Bellon et al. [Bel+07].

CCFinder [KKI02]

CCFinder is clone detection technique, which consist of the transformation of input source text and a token-by-token comparison. The token representation enables to detect clones with different line structure, which cannot be detected by using a line-by-line algorithm. They used the *suffix-tree matching algorithm* [Wei73]. CCFinder automatically identifies and separates each function definition, while initialization of values is removed. Identifiers are also normalized in order to treat each complex name as an equivalent simple name. The clone detection process consists of four steps:

Lexical Analysis: Each code line is divided into tokens, all tokens are concatenated, and the white spaces are removed.

Transformation: The token sequence is transformed using rules aiming at normalization of identifiers and identification of structures. For example, `std::ios_base::hex` $\rightarrow$ `hex` would be normalized as `(Name"::")+Name2` $\rightarrow$ `Name2`. Each identifier related to types, variables and constants is replaced with a special token.

Match Detection: All equivalent pairs of tokens are detected as clone pairs, each clone pair is represented as a quadruplet $(LeftBegin, LeftEnd, RightBegin, RightEnd)$ where $LeftBegin$ and $LeftEnd$ are the beginning and termination position for a token, and $RightBegin$ and $RightEnd$ belongs to cloned token.

Formating: Each location of a clone pair is connected to the line numbers on the original source files.

After this process, a matrix d_{xy} is created, where x represents a position of token in code 1 (t_x) and y represent a position of token in code 2 (t_y), thus $d_{xy} = 1$ if $t_x \equiv t_y$ otherwise $d_{xy} = 0$. Since it always holds that $d_{xy} = d_{yx}$ and $d_{xx} \equiv 1$, a clone pair found as a line segment of "1"s is parallel to the main diagonal of the matrix.
CCFinder provides some metrics to evaluate the clones founded. *Length* captures the apparent size of clones in the source code. **Population by clone class** is the number of elements of a given clone class (a clone class is a maximal set of code portions in which any pair of elements are a clone). **Deflation by clone class** gives an estimation of the amount of code which would be replaced by a shared code. **Coverage of clone code** is the percentage of code that include any portion of a clone. **Radius** measures the extent of influence of a clone class (if a clone class has a large radius, the code portions are widely spread over a software system)

JPlag [PMP02]

JPlag is web service that finds pairs of similar programs written in Java, Scheme, C or C++. It operates in two phases:

Converting the programs into token strings: The selected tokens should characterize the essence of a program's structure. Hence, whitespaces, comments and names of identifiers are ignored. JPlag puts semantic information into tokens to reduce false matches that can occur by pure chance. An example of code conversion in token strings is presented in Listing 2.9.

```
// JAVA SOURCE CODE                           GENERATED TOKENS
public class Count {                          //BEGIN_CLASS
  public static void main(String[] args)      //VAR_DEF, BEGIN_METHOD
  throws java.io.IOException {                //
    int count = 0;                            //VAR_DEF, ASSIGN
                                              //
    while (System.in.read() != 1)             //APPLY, BEGIN_WHILE
      count++;                                //ASSIGN, END_WHILE
    System.out.printIn(count+" chars.");      //APPLY
  }                                           //END_METHOD
}                                             //END_CLASS
```

Listing 2.9: JPlag token conversion example.

The conversion of code pieces into tokens is an approach that we used in this present work.

Comparison two token strings: The algorithm used is essentially the greedy string tiling [Wis93], where the goal is to find a maximal set of contiguous substrings that are as long as possible. For optimize the comparison process JPlag implement Karp-Rabin algorithm [KR87] where a hash function is used to verify a match. If the match is verified, then the algorithm tries to extend the match as far as possible.

DECKARD [Jia+07]

DECKARD converts the AST of a program into a feature vector, with the purpose of detecting cloned code fragments with syntactic similarity. At first, the feature vectors from the subtrees are generated by summing up the vectors of the children with their corresponding parent node. Next, the vectors of adjacent subtrees are generated by fusioning the vectors of the independent subtrees; this step allows the detection of larger clones. Due to the large amount of vectors generated in this step, it is necessary to group them with a hashing technique called Locality Sensitive Hashing (LSH). When the a group only has a vector, this is an indication of the absence of clones in the vicinity of this fragment. Hence, the fragment is discarded, as no clones associated were found. Finally, groups and subgroups are built from the obtained clones, in order to improve the scalability of the algorithm.

NiCad [RC08]

The strategy used by NiCad is based on three principles:

1. The format is standardized and the code is divided into blocks in which the changes can be easily detected.

2. Island grammars (isolated yet functional parts of a general grammar) are used as an efficient way to find potential clones.

3. Transformation rules (normalization) filter out uninteresting parts of the code.

NiCad uses the TXL parsing infrastructure [Dea+03] premises of: parse, transform and unparse. The *parse* phase loads the code according to the grammar rules. The *transform* phase applies normalization rules in the code, like as anonymizing of if blocks or normalizing of identifiers. The *unparse* phase transforms again the blocks into code with a spacing standard. An example is given in Listings 2.10 and 2.11.

```
define if_stament
     'if ( [expr] )            [IN] [NL]
     [statement]         [EX]
     [opt else_statement]
end define

define else_statement
     'else                     [IN] [NL]
     [statement]         [EX]
end define
```

Listing 2.10: TXL Parsing Rules.

```
redefine if_statement
     'if ( [expr] ) [statement] [opt
         ↪ else_statement]
end redefine

redefine else_statement
     'else [statement]
end redefine
```

Listing 2.11: Transformation Rules.

The removal of format and layout between codes allows the formation of lines, which are used as the unit of measure for finding clones. Edited differences between statements of code can be normalized to achieve generalization; for example, the expression `if(x<(n+y))` can be normalized either as `if(id<(id+id))` or as `if(x<(id+id))`, depending on the desired level of generalization. Uninteresting code segments can also be filtered by using TXL rules; for example, declaration and initialization statements can be removed because they do not affect the logic of a code piece.

The detection of potential clones is done using longest common subsequence (LCS) algorithm to compare the text lines after applying the mentioned pretty-printing and normalization techniques. When comparing two pieces of code, a score, called the *unique percentage of items* (UPI) is calculated as follows:

$$UPI = \frac{No\_of\_Unique\_Items \times 100}{Total\_No\_of\_Items}.$$

If the *UPI* for both pieces of code are under a certain threshold (e.g. 30%), they are assumed to be clones.

In oder to reduce the number of comparisons, only potential clones of exact or similar size are compared to each other.

SimCad [Udd+11]

SimCad employs a data clustering algorithm with a multi-level index-based searching that enables fast detection of clones. In a pre-processing phase comparison units are extracted and normalized in a per-block basis (e.g. function blocks), and a hash value with a SimHash function [Cha02] is calculated for the sake of indexing; The index is the key information piece for clone detection.

SimCad takes two parameters: the folder where the source code resides, and the programming language name. The language parameter is required by SimCad to use appropriate TXL [Dea+03] scripts for extraction and normalization of code fragments from the original source. Currently it supports four popular programming languages namely: C, Java, Python and C#.

SimCad can be used as a source code search engine: A third optional parameter takes a file or a folder that contains source code to be searched in a target project. This opens up a number of interesting possibilities in code and clone search. For example, a user might want to see if some arbitrary code exists in a target project. In such case, user needs to provide the arbitrary code location as input to the parameter item to search.

One of the potential aspects of SimCad is that its clone detection function is made more portable by packaging it into a library called SimLib. Thus, SimLib now can be used as an off-the-shelf clone detection library that can be easily integrated into other applications that are designed to work based on detected clones. The modular architecture makes SimLib a highly configurable and extensible API that can be tailored with minimal effort to build a fully customized clone detection tool for target data, or can be integrated to IDE of third party source code analysis system.

Krinke [Kri01]

The authors presented an approach to identify similar code in programs, based on finding similar subgraphs in attributed directed graphs. This approach is based on fine-grained program dependence graphs (PDGs). The idea is to identify similar subgraph structures which are stemming from duplicated code. Identified similar subgraphs can be directly mapped back onto the program code and presented to the user.

The PDG of Krinke is a specialization of the traditional PDG and is similar to an AST as well. On the level of statements and expressions, the AST vertices are almost mapped one-to-one onto PDG vertices. The definitions of variables and procedures have special vertices. The vertices may be attributed with a class, an operator and a value. The class specifies the kind of vertex: statement, expression, procedure call etc. The operator further specifies the kind, e.g. binary expression, constant, etc. The value carries the exact operator, like "+" or "-", constant values or identifier names.

The PDG has also specialized edges. Between the components of an expression, the PDG has a special control dependence which the authors call: the immediate control dependence, whose targets are evaluated before the source is evaluated. The data flow between the expression components is represented by another specialized edge: the value dependence edge, which is like a data dependence edge between expression components. Another specialized edge represents the assignments of values to variables: the reference dependence edges, which are similar to the value dependence edges, except that they show that a computed value is stored into a variable.

Graph isomorphism was the method used by the authors to find similarities among graphs. Two graphs are isomorphic if every edge is bijectively matched to an edge in the other graph and the attributes of the edges and the incident vertices are the same. The question whether two given graphs are isomorphic is NP-complete in general. But if only a subset of the vertices are be considered as “starting” vertices, the complexity can be reduced.

Summary

In this chapter the reader was introduced into the fundamental topics that supported our research. Kernel Methods arose as a plausible option for performing program comparison. The modular approach that characterize them, makes them appropriate to target the comparison of structured data, like is the case of I/O traces or the intermediate representations of a compiler. In the following chapter, we detail the comparison strategies we propose, which are based on string kernels, and correspond to the main contribution of this thesis.

3 Design

We have seen in the previous chapter how program similarity has motivated the scientific community to propose different methods of comparison. However, the utilization of string kernel functions for solving this problem is still an unexplored area. Due to the fact that data coming from I/O tracing activities or from a compiler, can be expressed as a set of strings, we have found there is an opportunity to contribute with new research in this area. We have developed novel strategies for converting trees, Clang abstract syntax trees, LLVM Intermediate Representations and I/O traces into weighted strings. Additionally, three novel kernel functions to find similarities between the resulting strings are introduced, which correspond to the main contribution of this work. The string representation acts as the middle man between the domain and the similarity functions, a design that preserves the modular approach that characterizes kernel methods. As long as a data structure, coming from any domain, can be expressed as a series of weighted tokens, the kernel functions will give a similarity score. In Section 3.1, we present a general description of the used methodology. Section 3.2 is dedicated to the explanation of the proposed string conversion process. Finally, in Section 3.3, we find the development of the mentioned string kernels.

3.1 Methodology

In the previous chapter, we saw that there are some syntactical approaches in the literature that resolve the problem of comparing programs, by *i)* reducing the data structures of a compiler into a sequence of tokens, and *ii)* using afterwards regular string comparison algorithms to emit a similarity measure among them. The approaches are syntactical because they are based not on the code itself but on an intermediate representation of a program. In this chapter we will see that, although I/O traces are data structures completely different to those intermediate representations of a compiler, they have some hierarchical structure in common with them; this fact motivated us to apply the same strategy on the domain of I/O access pattern identification.

3.1.1 In Search of a Representation for Diverse Domains

Resembling the idea of intermediate representations, we aimed for the creation of a representation that could be used as a common ground for all the domains of our interest. In order to achieve this, that representation should not depend on the domain. It happens that strings comply with such requirements, and have been widely and successfully used in different disciplines. It is important though, that the strings reflect the inherent structure present on the domain; for this reason, we thought it was better to work at first with tree-like data structures. In the case of abstract syntax trees, the problem was already solved. However, I/O traces had to be converted first to a tree hierarchy. In this sense, the proposed string representation arose as a flattened version of a tree. The utilization of strings as a common ground achieved also the language independence capability for the program comparison process.

When we were analyzing data from I/O traces and intermediate representations, we realized that some tokens occurred in a repetitive way. This repetitions could be expressed in a concise way if we added a weighting attribute to each token. The proposed string representation was then implemented with this additional numeric attribute that can be used to modulate the contributions of each token of the string. As it was said, the punctual application of this weighting feature in this study was the capability of abstracting repetitive tokens and hence compressing the string. However, such value can be used for other purposes: for example, certain nodes (`for` loops or `if` sentences) in an AST could be considered to have more weight than others (variable declarations or cast operations), as their impact in the overall code structure is higher.

3.1.2 Finding a Proper Comparison Strategy

In the literature, the comparison of strings has been made with different algorithms, like the *greedy string tiling*. However, there is not so much work in the application of string kernel functions in the problem of code similarity. But string kernels have been applied successfully in the comparison of data coming from domains like bioinformatics; this fact motivated us to contribute with a study of the application of these functions in the mentioned domains. Studies like ours can generate more interest in the utilization of string kernels on the program similarity problem, and hence help to bring progress in this unexplored area.

3.1.3 Proposed Infrastructure

In this thesis we propose a feature extraction system that, at the first stage, aims for the conversion of source code and I/O traces into weighted strings, and in a second stage, uses three novel string kernels to obtain a similarity score among the analyzed examples. Figure 3.1 shows how the flow of information works. The **highlighted parts**, essentially the conversion procedures and the novel string kernels, represent the contributions of this work, while the parts with a dotted line suggest possibilities of extension, like new kernels or new conversion methods.

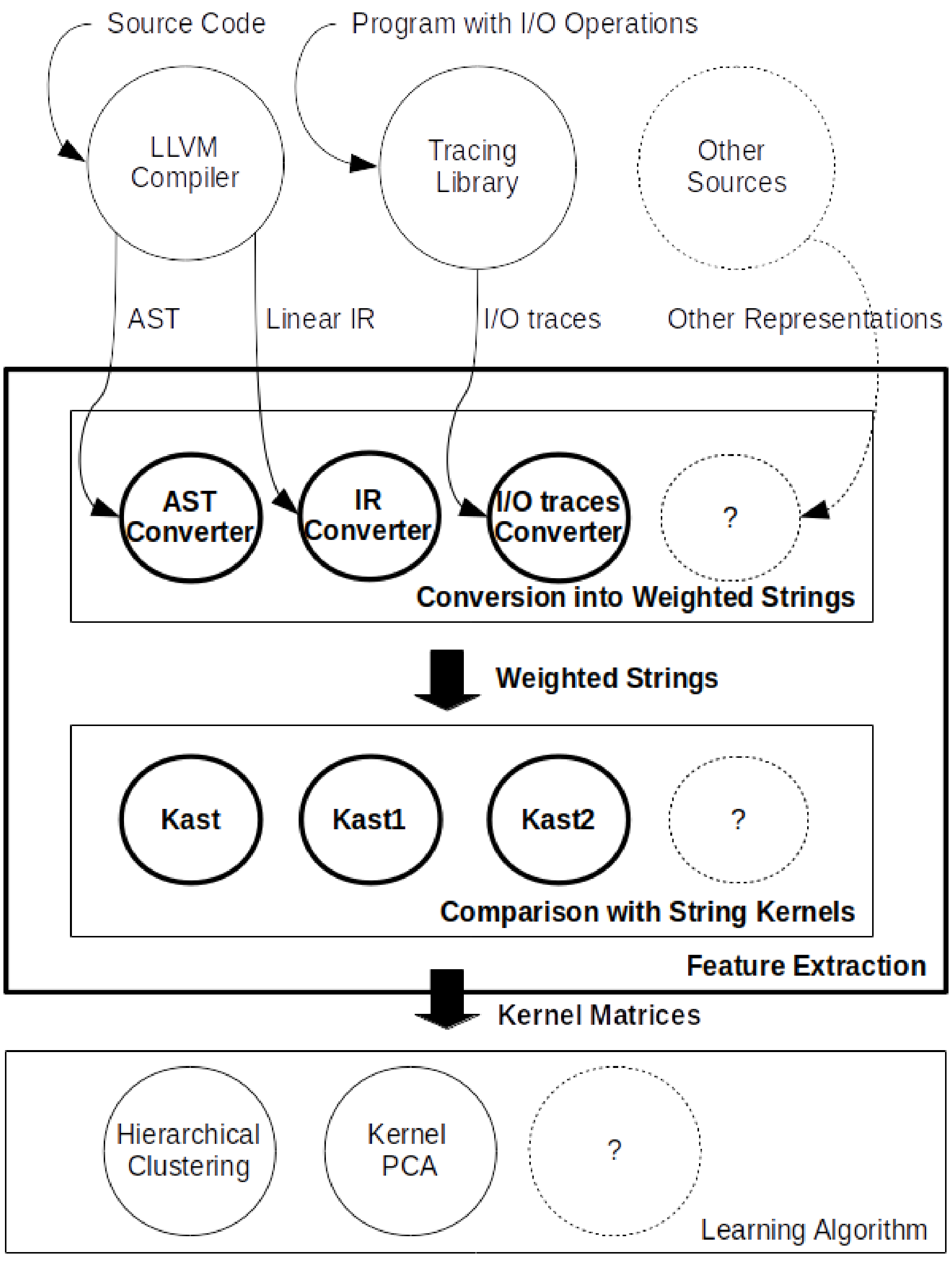


Figure 3.1: Diagram of the Proposed Modular Design for the Comparison of Intermediate Representation and I/O Traces.

The top of the figure represents possible domains from which data can be originated. Notice that, usually, data from the domain is not delivered in its original form; for example, source code can be delivered as an AST trough a compiling step, while the I/O behavior of a program can be obtained via a tracing library. Ideally, they should be tree-like data structures.

In the middle of the figure we find the **feature extraction subsystem**, the place where our contributions have focused. Our first contribution corresponds to the string converters for Clang ASTs, I/O traces and LLVM linear IRs. After the conversion, the resulting strings are available for the proposed kernel functions, which correspond to the second and main contribution of this work. The output of these kernels are a collection of matrices whose values can be interpreted as similarity score among the analyzed examples. The feature extraction subsystem can be extended by adding more comparison strategies (string kernels); an example of this is the addition of the *blended spectrum kernel*, which was selected from the literature as the baseline kernel for the evaluation section.

Finally, the learning algorithms are represented at the bottom of the information flow; any algorithm based on similarity scores can be used (E.g. hierarchical clustering, kernel principal component analysis).

In the following sections of this chapter, we depict in depth the particularities of the weighted string representation and the novel kernels.

3.2 The String Representation

In this work, we have developed a string representation for tree-like data structures that deals with them as sets of consecutive weighted tokens. The strategy is not new: as seen on the previous chapter, other authors have used tokens as unit of comparison as well. In the case of ASTs, the data structures are already in tree form, but LLVM intermediate representations and I/O traces are not. These ones have to be converted first into trees.

3.2.1 Definitions

Let us at first delimit the notion of tokens and strings for this study. A weighted token t is a consecutive set of symbols of variable size enclosed by the special characters `[` and `]`, with an associated attribute $weight(t)$ that corresponds to an integer value greater than zero. E.g.: $\texttt{[token\_literal]}_1$. A weighted string S is a set of n consecutive weighted tokens t (from here on referred simply as strings and tokens):

$$S = t_1 t_2 t_3 ... t_n. \tag{3.1}$$

Let T be a non-empty string, and let A and B be non-empty or empty strings. T is a substring of S, if it is fully contained in S:

$$S = ATB \tag{3.2}$$

The weight of a string corresponds to the summation of the weights of its tokens:

$$weight(S) = weight(t_1) + weight(t_2) + weight(t_3) + ... + weight(t_n). \tag{3.3}$$

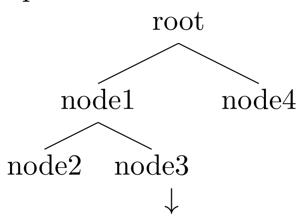
3.2.2 Conversion of Trees into Strings

Let us start considering first the simple conversion of a generic tree into a succession of weighted strings. To generate a string, a tree is traversed in a pre-order fashion. Each node of the tree is translated into a token of the string. Nodes usually possess some attributes that represent a specific property; the literal part of a token can be constructed from any of those attributes. In the case of I/O traces, for example, the token literal might be paired with the operation name. For ASTs, it might be paired with the class node name. For IRs, it might correspond to the instruction name. The default weight for these tokens is 1. In a further compression step, this value is used to express the number of consecutive repetitions of a token.

The [LEVEL_UP] Token

A few token literals correspond to abstract nodes added during the tree conversion process. For example, the special token `[LEVEL_UP]` is used to represent the distance between two tree nodes that in the pre-order dump happen to be consecutive. The weight of this token corresponds to the number of levels that the pre-order algorithm had to climb up to reach the next node. This distance is zero when jumping from a parent node to its first child, hence, in this case, there is no need for a token that indicates going to a lower level. As an example, consider the case of a code segment with an instruction inside a loop and an immediate instruction after it. Without the `[LEVEL_UP]` token, both instructions will be written one after the other in the string conversion, and there will not be a difference between this segment and another one that has no loop involved.

Figure 3.2 shows an example of a simple tree and its corresponding string representation, according to the previous conversion steps. The weights of the tokens and the weight of the resulting string are written as subscripts.



S_9 = [root]$_1$[node1]$_1$[node2]$_1$[LEVEL_UP]$_1$[node3]$_1$[LEVEL_UP]$_2$[node4]$_1$[LEVEL_UP]$_1$

Figure 3.2: Conversion of a Tree into a String.

3.2.3 Conversion of I/O Access Patterns into Strings

The contents of an I/O access pattern file correspond to a sequence of operations. In our approach, a few operations are considered negligible, as they don't make part of the effective I/O workload of the analyzed programs, and are hence ignored (e.g. `fileno`, `nmap` and `fscanf`).

Among the relevant operations, there are some of them that keep information about the number of bytes involved (e.g. `read` and `write`); this information can be used or ignored, which leads to the possibility to produce two different types of strings from a single I/O access pattern:

- Strings with associated byte information
- Strings without byte information

Looking into the I/O trace files, it is noticeable that operations are registered chronologically; with several file handles being traced at the same time, it is not always possible that all the operations belonging to the same file handle could have been written contiguously. Therefore, it is necessary to group each handle's actions without losing information. For that reason the traces are first converted into trees. Trees are ideal data structures for representing containment relationships among objects.

From I/O Access Patterns to Trees

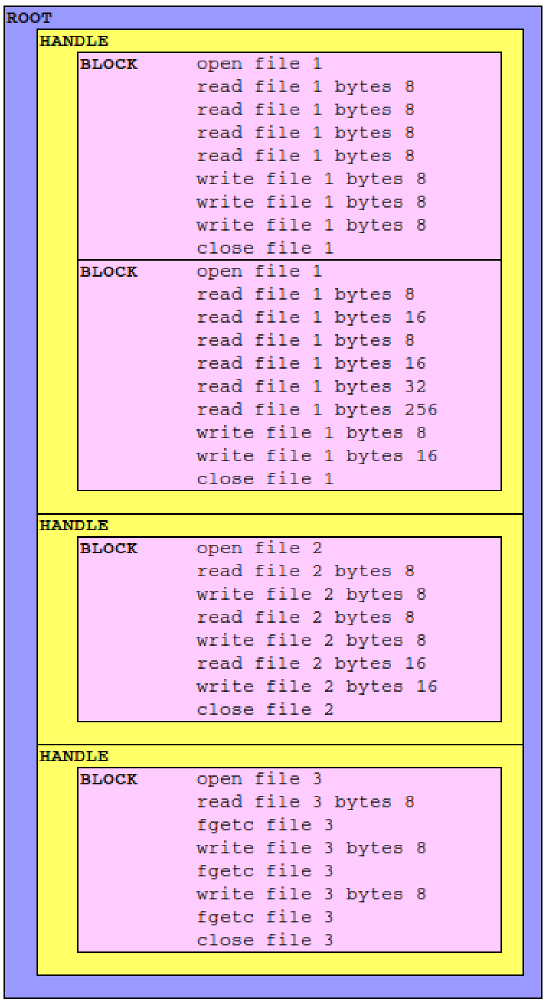
The trees that represent I/O traces have at most four levels: The `ROOT` level, the `HANDLE` level, the `BLOCK` level and the operation level. These levels reflect the intuitive relationships of the operations. Figure 3.3 shows how I/O traces are converted into strings following the next steps:

- At the highest level, an abstract root node groups all the operations of a single I/O access pattern file. Such node is represented as `ROOT`.
- At the second level, abstract nodes group all the operations belonging to the same file handle. Such nodes are represented as `HANDLE`.
- At the third level, abstract nodes group all the operations found between an `open` operation and its corresponding `close` operation. Such nodes are represented as `BLOCK`.
- At the deepest level, operations are given nodes, except for `open` and `close`, because the `BLOCK` node already plays the role of a delimiter.

Compression of the Tree

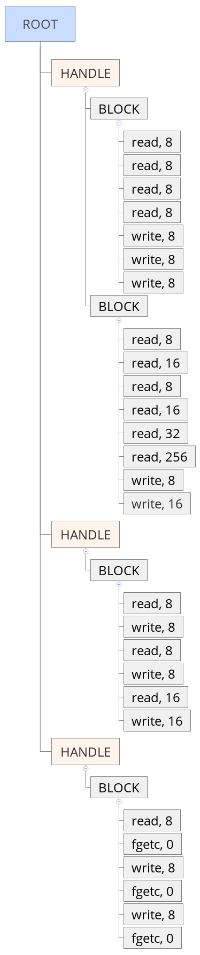
In order to save space, a set of consecutive operation nodes on the same block can be expressed as a single node when they present a repetitive pattern. A similar approach was applied by Kluge [Klu11]. The resulting node will have an additional field that stores the number of repetitions. This compression step is based on the following transformations, which are performed in the given order:

- Consecutive operations with the same name and number of bytes are simplified to a single operation with the same information (see Figure 3.4). For example, a `read` operation inside a loop reading a file *n* bytes per iteration. Compressing these nodes into a single one simplifies the representation, which has later a positive effect in the performance of the comparison process.
- Consecutive operations with the same name but different number of bytes are simplified to a single operation with the same name (see Figure 3.5). The new byte value is a combination of both previous byte numbers. For example, initializing in a loop an array of C structures compound of a 2-bytes integer and a 4-bytes integer will need a `read` operation extracting two bytes first and another `read` extracting four bytes afterwards.
- Consecutive operations with different names but same number of bytes are simplified to a single operation with the same number of bytes (see Figure 3.6). The new operation name is a combination of both previous names. For example, a series of interlaced `read` and `write` operations with the same number of bytes might indicate a tacit copy operation from one file to another.



```
ROOT
HANDLE
BLOCK      open file 1
           read file 1 bytes 8
           read file 1 bytes 8
           read file 1 bytes 8
           read file 1 bytes 8
           write file 1 bytes 8
           write file 1 bytes 8
           write file 1 bytes 8
           close file 1
BLOCK      open file 1
           read file 1 bytes 8
           read file 1 bytes 16
           read file 1 bytes 8
           read file 1 bytes 16
           read file 1 bytes 32
           read file 1 bytes 256
           write file 1 bytes 8
           write file 1 bytes 16
           close file 1
HANDLE
BLOCK      open file 2
           read file 2 bytes 8
           write file 2 bytes 8
           read file 2 bytes 8
           write file 2 bytes 8
           read file 2 bytes 16
           write file 2 bytes 16
           close file 2
HANDLE
BLOCK      open file 3
           read file 3 bytes 8
           fgetc file 3
           write file 3 bytes 8
           fgetc file 3
           write file 3 bytes 8
           fgetc file 3
           close file 3
```

(a) Access pattern.

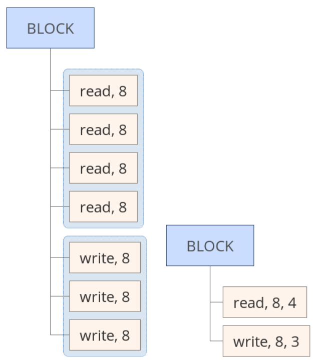


(b) Resulting tree.

Figure 3.3: Conversion of an I/O Access Pattern into a Tree.

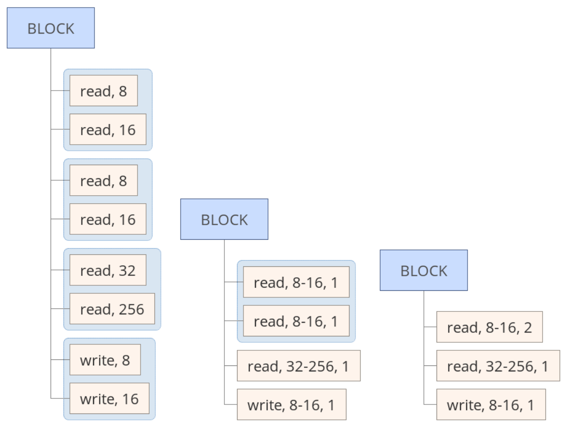
- Consecutive operations with different names and different number of bytes but with one operation having 0 as number of bytes are simplified to a single operation with the non-zero value as the number of bytes (see Figure 3.7). The new operation name is a combination of both previous names. For example. inside a loop, an `lseek` operation (which has no relationship with the bytes that have to be written or read) moves the pointer in the file descriptor; after that, a `write` operation stores some bytes there. In this case, the resulting operation name will be `write-lseek`.

The previous steps are repeated a second time to capture higher level patterns. It has to be mentioned that some operations (e.g. `read`, `write`) have memory addresses associated to them. If these addresses would be taken into account, the algorithm would be more precise to detect related operations, e.g, a copy operation compound by interlaced reads and writes. However, the degree of compression would be reduced. We have ignored these memory addresses, because the main focus of this research is the usage of kernels for determining, in an efficient way, how similar the patterns of a collection are. Hence, it is not in the scope the understanding of the underlying details of the pattern found. However, this task stands as one of the possibilities of extension for future work.



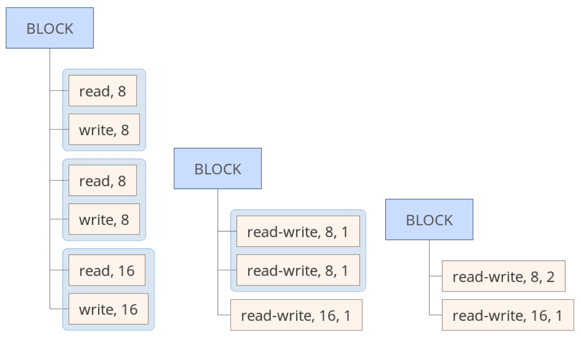
(a) Original pattern. (b) Compact pattern.

Figure 3.4: Compression of nodes with the same name and the same number of bytes.



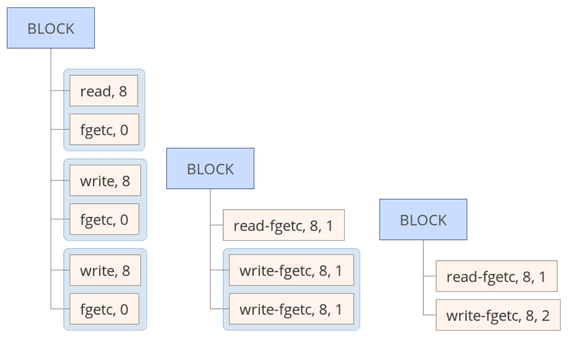
(a) Original pattern. (b) Intermediate pattern. (c) Compact pattern.

Figure 3.5: Compression of nodes with the same name but different number of bytes.



(a) Original pattern. (b) Intermediate pattern. (c) Compact pattern.

Figure 3.6: Compression of nodes with different name but same number of bytes.



(a) Original pattern. (b) Intermediate pattern. (c) Compact pattern.

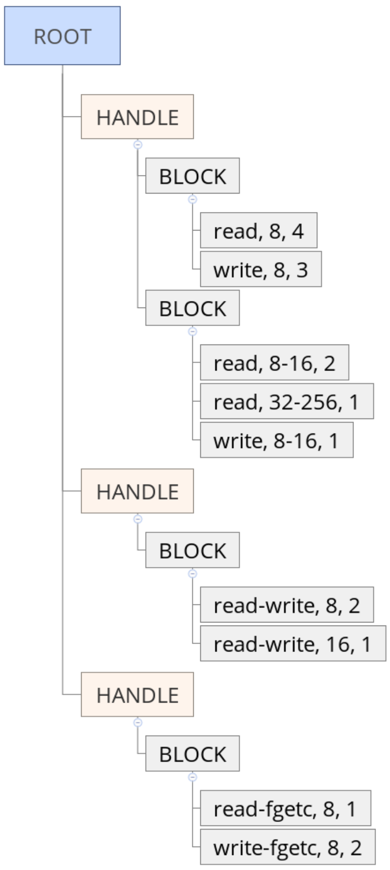
Figure 3.7: Compression of nodes with different name and one of them with zero as number of bytes.

From Trees to Strings

Once the tree is compacted, the string representation can be built (see Figure 3.8):

- The tree is traversed in a pre-order fashion; each node of the tree corresponds to a token.

- For leaf nodes, the literal part of a token is conformed by the name of the operation and the number of bytes enclosed by `[]`, while their weight corresponds to the number of repetitions.
- `ROOT`, `HANDLE` and `BLOCK` nodes are translated as `[ROOT]`, `[HANDLE]` and `[BLOCK]` respectively; their weights are initialized with 1.
- The `[LEVEL_UP]` token represents the change to an upper level when doing the pre-order traversal.



(a) Compacted tree.

Tokens	Repetitions
[ROOT]	1
[HANDLE]	1
[BLOCK]	1
[read_8]	4
[LEVEL_UP]	1
[write_8]	3
[LEVEL_UP]	2
[read_8-16]	2
[LEVEL_UP]	1
[read_32-256]	1
[LEVEL_UP]	1
[write_8-16]	1
[LEVEL_UP]	3
[HANDLE]	1
[BLOCK]	1
[read-write_8]	2
[LEVEL_UP]	1
[read-write_16]	1
[LEVEL_UP]	3
[HANDLE]	1
[BLOCK]	1
[read-fgetc_8]	1
[LEVEL_UP]	1
[write-fgetc_8]	2
[LEVEL_UP]	4

(b) Extracted tokens.

[ROOT]$_1$[HANDLE]$_1$[BLOCK]$_1$[read_8]$_4$ ... [LEVEL_UP]$_4$

(c) Final string.

Figure 3.8: Creation of a string of tokens from an I/O trace tree.

3.2.4 Conversion of Clang Abstract Syntax Trees into Strings

Clearly, Clang ASTs are already in a tree form. Hence, there is no need for a pre-processing step. These trees are complex in-memory data structures that store, not only the relationships between programming constructs, but also additional properties of these constructs, for example, the location of source code, which is useful for debugging purposes [PS15]. For the proposed conversion method, only the construct name and the hierarchy are important. All ASTs start with the root node `TranslationUnitDecl`, which represents a whole compilation unit (see Figure 3.9). Starting from it, all the tree hierarchy comes off, from functions till basic operators (see Figure 3.10).

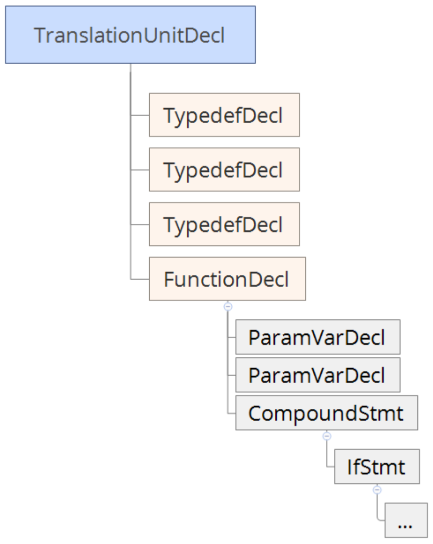


Figure 3.9: Simplified Clang AST example.

From Trees to Strings

The conversion rules are similar to the ones for I/O traces:

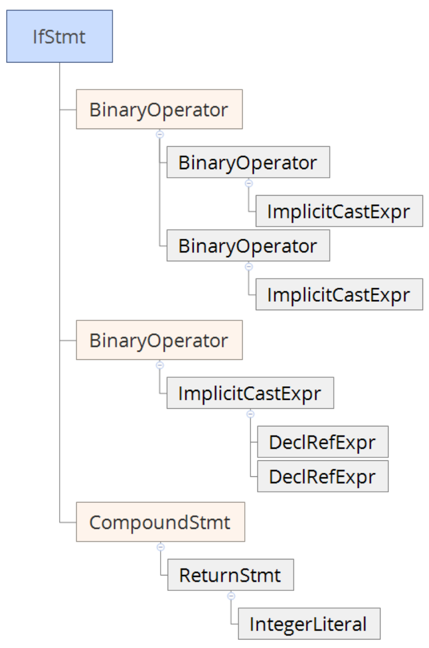
- ASTs are traversed in pre-order (see Figure 3.10). Each node of the tree corresponds to a token in the string, except for comment nodes and their children, which are ignored.
- The literal part of a token is the class name of its corresponding node, while the weight is the number of consecutive occurrences of the token, which is initialized with 1.
- In order to achieve generality, nodes representing loop statements, like `ForStmt` and `WhileStmt`, are represented with the new token `[LoopedStmt]`:

$$\boxed{\texttt{ForStmt}} \rightarrow \texttt{[LoopedStmt]}_1$$
$$\boxed{\texttt{WhileStmt}} \rightarrow \texttt{[LoopedStmt]}_1$$

- The `[LEVEL_UP]` token represents the change to an upper level when doing the pre-order traversal.

Compression of the String

In the case of ASTs, the compression step is performed on the strings. Space can be saved when a set of consecutive tokens follows a pattern that can be expressed as a single token. The weight of this new token corresponds to the summation of weights of all involved tokens, in order to preserve the original weight of the string. The following transformations are performed on the string when processing it from left to right:



(a) Abstract Syntax Tree.

Tokens	Repetitions
[IfStmt]	1
[BinaryOperator]	1
[BinaryOperator]	1
[ImplicitCastExpr]	1
[LEVEL_UP]	2
[BinaryOperator]	1
[ImplicitCastExpr]	1
[LEVEL_UP]	3
[BinaryOperator]	1
[ImplicitCastExpr]	1
[DeclRefExpr]	1
[LEVEL_UP]	1
[DeclRefExpr]	1
[LEVEL_UP]	3
[CompoundStmt]	1
[ReturnStmt]	1
[IntegerLiteral]	1
[LEVEL_UP]	4

(b) Extracted tokens.

`[IfStmt]`$_1$`[BinaryOperator]`$_1$`[BinaryOperator]`$_1$`[ImplicitCastExpr]`$_1$ `...` `[LEVEL_UP]`$_4$

(c) Final string.

Figure 3.10: Creation of a string of tokens from an Abstract Syntax Tree.

1. Consecutive tokens with the same literal part are represented as a single token and their weights are summed up. For example:

   `[BinaryOperator]`$_1$ `[BinaryOperator]`$_1$ `[BinaryOperator]`$_1$ `[BinaryOperator]`$_1$

   $\downarrow$

   `[BinaryOperator]`$_4$

   Rationale: The motivation here is merely space saving.

2. Tokens representing cast expressions, parent expressions and function calls are deleted but their weights are added up to the weight of the next subsequent token. For example:

   `[CStyleCastExpr]`$_1$ `[CallExpr]`$_1$ `[ImplicitCastExpr]`$_1$ **`[DeclRefExpr]`**$_1$

   $\downarrow$

   `[DeclRefExpr]`$_4$

   Rationale: Cast expressions are used to assist the compiler in the conversion of values between similar data types; although this is important at compilation time, it was proved to be cumbersome in the similarity study, because it introduces different tokens in two strings that otherwise would be semantically identical, which obstructs the achievement of a suitable abstraction level, necessary to establish a similarity measure among code examples. Likewise, parent expressions are internal constructions of the Clang AST, whose omission improves abstraction. The same is the case for function calls; keeping a token for a function call also breaks the structure of string segment, which in its case affects the similarity score among strings. The idea behind these omissions is to obtain the largest possible matching substrings. Experimentation showed that ignoring these tokens helped to increase the identification of Type-2 clones.

3. All tokens between a Declaration Statement token (`[DeclStmt]`) and a `[LEVEL_UP]` token are deleted but their weights are summed up to the weight of the former. For example:

$$\texttt{\textbf{[DeclStmt]}}_1\ \texttt{[VarDecl]}_1\texttt{[DeclRefExpr]}_1\texttt{[LEVEL\_UP]}_4$$
$$\downarrow$$
$$\texttt{\textbf{[DeclStmt]}}_3\ \texttt{[LEVEL\_UP]}_4$$

   Rationale: Similar to cast expressions, the tokens of a declaration statement introduce a level of detail that make the abstraction difficult. Omitting them facilitates the detection of similar declaration blocks.

4. If two pairs of tokens have the same literal part, they are collapsed as one pair and their weights are added up to the corresponding token. For example:

$$\texttt{[IntegerLiteral]}_1\texttt{[LEVEL\_UP]}_5\texttt{[IntegerLiteral]}_1\texttt{[LEVEL\_UP]}_2$$
$$\downarrow$$
$$\texttt{\textbf{[IntegerLiteral]}}_2\texttt{\textbf{[LEVEL\_UP]}}_7$$

   Rationale: The motivation here is also space saving.

The resulting compressed string will still comply with the original design.

3.2.5 Conversion of LLVM Intermediate Representations into Strings

In order to understand the organization of an LLVM IR, let us consider the human-readable version of the IR of Listing 3.1. The first line of the code makes reference to the module, which is the denomination for the top-level compilation unit. A module encloses all the functions, variables and other entries belonging to a compilation unit. For the purposes of this project, only the function blocks and their contents are taken into account. All constructions outside a function block are ignored.

From LLVM IRs to Trees

LLVM IRs are converted into trees following the usual rules. These trees have the following levels: The `Module` level, the `Function` level, the `BasicBlock` level, the instruction level and the operand level (see Figure 3.11):

- At the highest level, a root node groups all the constructions belonging to a module. Such node is represented as `Module`.
- The second level is compound by nodes that group all the instructions belonging to the same function. Such nodes are represented as `Function`. The function names are ignored.

- The third level corresponds to the nodes that group all the instructions belonging to the same basic block. A *basic block* is defined as a set of instructions where all of them are executed once the first instruction has been reached; these blocks are usually created by introducing conditional instructions, (e.g. *if* statements) that break the normal flow of a program. Such nodes are represented as `BasicBlock`.

- The fourth level belongs to the nodes grouping all the operands that shape an instruction. In this case, the nodes are represented by the name of the instruction.

- At the deepest level, operands are assigned nodes. Such nodes are represented as `NamedOperand` or `PointerOperand`, depending of the nature of the operand.

It can be seen that most of the construction names are replaced by a generic descriptor, except from the instruction names. This helps to focus on the similarities in the instruction patterns rather than in personalized identifier names that would make the abstraction less effective.

```
; ModuleID = 'group02_function03_version00.c'

; Function Attrs: nounwind uwtable
define void @group02_function03_version00(i32 %n, i8** %stringValues) #0 {
%1 = alloca i32, align 4
%2 = alloca i8**, align 8
br label %8

; <label>:8
%9 = load i32, i32* %i, align 4
br i1 %11, label %12, label %25

; <label>:12
%13 = load i32, i32* %i, align 4
br label %22

; <label>:93
ret void
}
```

Listing 3.1: LLVM Linear IR snippet.

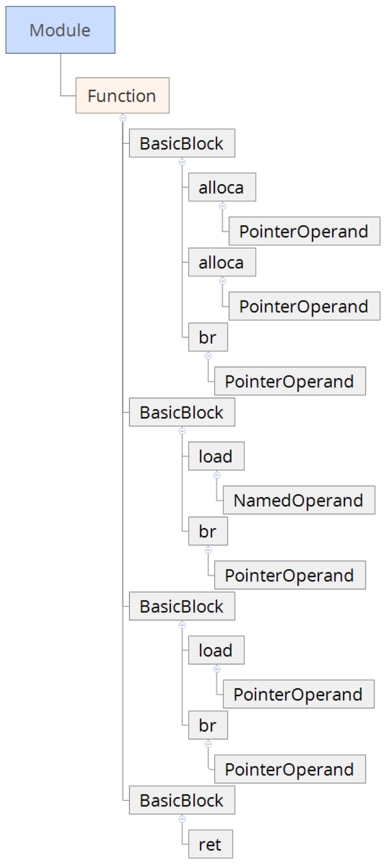
From Trees to Strings

The process is the same like for previous representations (see Figure 3.11):

- The tree is traversed in pre-order and each node properties are extracted; each node of the tree corresponds to a token in the string.

- The literal part corresponds to the name of node enclosed by `[ ]` while the weight corresponds to the number of repetitions.

- The `[LEVEL_UP]` token represents the change to an upper level when doing the pre-order traversal.



(a) LLVM IR tree.

Tokens	Repetitions
[Module]	1
[Function]	1
[BasicBlock]	1
[alloca]	1
[PointerOperand]	1
[LEVEL_UP]	2
[alloca]	1
[PointerOperand]	1
[LEVEL_UP]	2
[br]	1
[PointerOperand]	1
[LEVEL_UP]	3
[BasicBlock]	1
[load]	1
[NamedOperand]	1
[LEVEL_UP]	2
[br]	1
[PointerOperand]	1
[LEVEL_UP]	3
[BasicBlock]	1
[load]	1
[PointerOperand]	1
[LEVEL_UP]	2
[br]	1
[PointerOperand]	1
[LEVEL_UP]	3
[BasicBlock]	1
[ret]	1
[LEVEL_UP]	4

(b) Extracted tokens.

`[Module]`$_1$`[Function]`$_1$`[BasicBlock]`$_1$`[alloca]`$_1$ ... `[LEVEL_UP]`$_4$

(c) Final string.

Figure 3.11: Creation of a string of tokens from a LLVM IR tree.

3.3 A New Family of String Kernel Functions

Once data structures are converted into weighted strings, they can be easily compared using a string kernel function. In this study, three novel string kernel functions are proposed, all of them sharing the same strategy for string matching, but with differences in the way weights are summarized. For a better reference, we have named this family of kernels, the **kastx kernel family**.

In theory, the number of different tokens that can compound a string is infinite. In practice, this number is limited to the namespace of a particular domain. For the case of I/O traces, for example, it is limited by the I/O operations names and the number of bytes related to each operation. For the case of ASTs, the namespace is limited by the class node names. For LLVM IRs, the limitation comes from the available instruction list. Still, the number of tokens can be very high. In an hypothetical feature space, where every string is characterized by the presence or absence of each possible token with each possible weight, the number of features is still infinite. However, in practice, for a single string, most of the features of this hypothetical space are zero-valued. This is a fact that eases the creation of a feasible kernel function. This way, the new embedding space has a finite and small number of features, that corresponds to the actual tokens that exist in a set of samples.

Our approach is similar to the greedy string tiling algorithm of Wise [Wis93], whose goal is to find the longest common subsequence. However, our solution is focused on finding the longest common substrings, even if they do not conform the longest common subsequence. The idea behind is that we want to give more relevance to the longest pieces of code without considering gaps.

3.3.1 Definitions

Recall from Equation 3.1, that a weighted string S is a set of n consecutive weighted tokens t. For two weighted strings A and B, the kernels here proposed must follow the conditions given below:

1. The user must specify a minimum weight value as parameter (from here on referred simply as **cut weight**).
2. The aim is to find the longest matching substrings of A and B, whose weights are greater than or equal to the cut weight. They are called *valid matching substrings*. Invalid matching substrings have a weight value that is smaller than the cut weight, and are hence ignored.
3. A *valid matching substring* can appear more than once in each string.
4. A *valid matching substring* must not be a substring of another *valid matching substring* in at least one of the original strings.

In order to find the longest *valid matching substrings* efficiently, the algorithm starts searching for matches of maximum size. The maximum size is the number of tokens of the shortest string of the comparison (A or B). This size is reduced progressively until 1, but always checking that the substrings' weights are equal or above the cut weight. A copy of each one of the original strings is used to mark down the already found *valid matching substrings*. Potential *valid matching substrings* have to be checked against those copies to assure that condition 4 is met.

The Cut Weight

The cut weight selection has an effect on the computation cost, as the algorithm takes into account all the substrings whose weights are equal or greater than the cut weight. If the cut weight is 1, all substrings have to be compared. An increase on the cut weight allows the filtering of substrings with small weights. If the cut weight is closer to the weight of the strings (A or B), the number of substrings having a valid weight is reduced considerably. Hence, the higher the cut weight, the cheaper the computation.

Additionally, the cut weight controls the size of the consecutive parts that are shared. For example, if the cut weight is set to be as big as the weight of either A or B, we are only accepting that either A or B is contained fully on the other string. This might be useful to perform code search: specific segments of code can be survey in library or a project. A decrease on the cut weight permits that segments of the strings can be considered as *valid matching* *substrings*. If the cut weight is closer to 1, short sequences of tokens will contribute on the similarity score. A trade-off has to be found. Experimentation (see Chapters 5 and 6) showed that cut weight values up to 64 were optimal for the construction of a good similarity matrix.

The kernels presented here are asymmetric kernels, which means that the kernel value depends on the order in which two string are compared. Because the substrings conforming the first string are always the base for the search, there might be a difference when the strings are swapped. Given the kernels start always searching for the longest matching substrings, these changes happen with smaller matching substrings only, so the difference between the kernel values is not significantly high. This also shows that the smaller is the cut weight, the highest is the probability of this difference to appear.

Example:

Consider the strings A and B of Figure 3.12, and a cut weight value of 4:

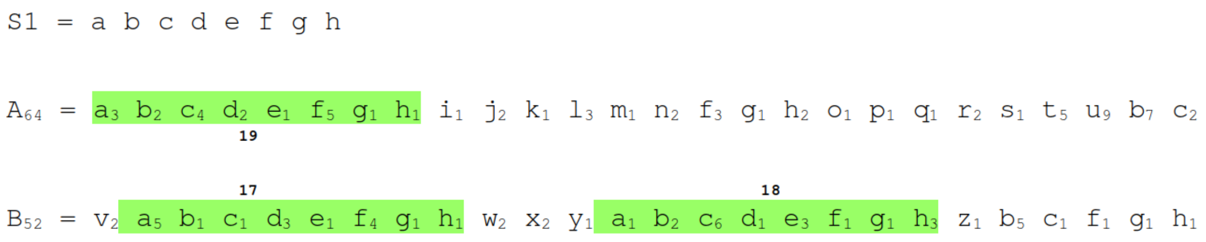
Figure 3.12: Examples of Strings

$A_{64} = a_3\ b_2\ c_4\ d_2\ e_1\ f_5\ g_1\ h_1\ i_1\ j_2\ k_1\ l_3\ m_1\ n_2\ f_3\ g_1\ h_2\ o_1\ p_1\ q_1\ r_2\ s_1\ t_5\ u_9\ b_7\ c_2$

$B_{52} = v_2\ a_5\ b_1\ c_1\ d_3\ e_1\ f_4\ g_1\ h_1\ w_2\ x_2\ y_1\ a_1\ b_2\ c_6\ d_1\ e_3\ f_1\ g_1\ h_3\ z_1\ b_5\ c_1\ f_1\ g_1\ h_1$

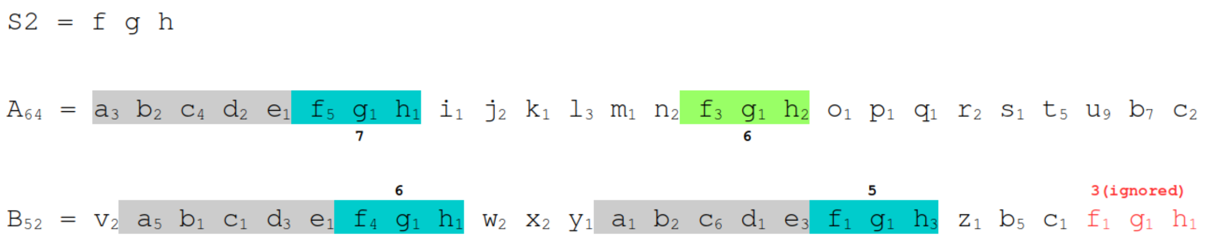
The first *valid matching substring* with the longest size ($S1$) is found once in A and twice in B (see Figure 3.13):

Figure 3.13: $S1$ as the Longest Matching Substring


S1 = a b c d e f g h
A64 = a3 b2 c4 d2 e1 f5 g1 h1 i1 j2 k1 l3 m1 n2 f3 g1 h2 o1 p1 q1 r2 s1 t5 u9 b7 c2
19
17
18
B52 = v2 a5 b1 c1 d3 e1 f4 g1 h1 w2 x2 y1 a1 b2 c6 d1 e3 f1 g1 h3 z1 b5 c1 f1 g1 h1

The second longest *valid matching substring* ($S2$) is found twice in A and twice in B (see Figure 3.14). This substring appears at least once as an independent substring in one of the strings, hence complying with condition 4. Notice that an extra occurrence is ignored because its weight is smaller than 4:

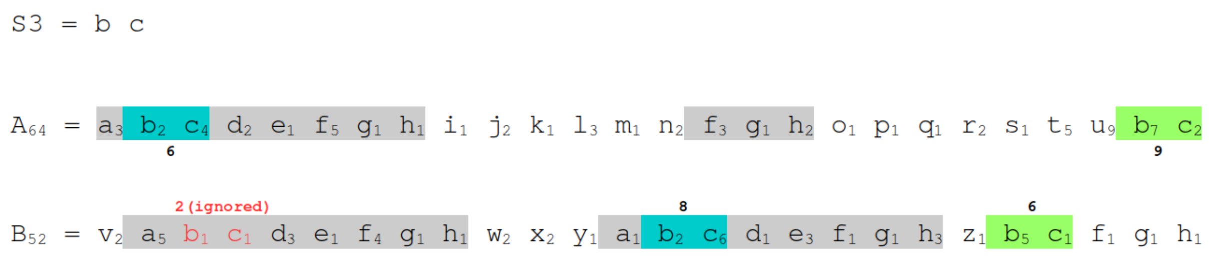
Figure 3.14: $S2$ is another Matching Substring


S2 = f g h
A64 = a3 b2 c4 d2 e1 f5 g1 h1 i1 j2 k1 l3 m1 n2 f3 g1 h2 o1 p1 q1 r2 s1 t5 u9 b7 c2
7
6
6
5
3(ignored)
B52 = v2 a5 b1 c1 d3 e1 f4 g1 h1 w2 x2 y1 a1 b2 c6 d1 e3 f1 g1 h3 z1 b5 c1 f1 g1 h1

The last and shortest *valid matching substring* ($S3$) is found twice in A and twice in B (see

Figure 3.15). As the substring appears as an independent case in both strings, it complies with condition 4. Here also an extra occurrence is ignored due to a smaller weight:

Figure 3.15: $S3$ is the last Matching Substring


S3 = b c
A₆₄ = a₃ b₂ c₄ d₂ e₁ f₅ g₁ h₁ i₁ j₂ k₁ l₃ m₁ n₂ f₃ g₁ h₂ o₁ p₁ q₁ r₂ s₁ t₅ u₉ b₇ c₂
6
9
2 (ignored)
8
6
B₅₂ = v₂ a₅ b₁ c₁ d₃ e₁ f₄ g₁ h₁ w₂ x₂ y₁ a₁ b₂ c₆ d₁ e₃ f₁ g₁ h₃ z₁ b₅ c₁ f₁ g₁ h₁

3.3.2 Kast Spectrum Kernel

When comparing two strings, this kernel aims to build a feature vector for each string, where each feature corresponds to a *valid matching substring*. Having two strings A and B, the first kernel here presented is the *kast spectrum kernel*, defined by the following rules:

- Each *valid matching substring* embeds a new feature for A and B. Hence, the size of the new embedding vector for both strings is equal to the number of *valid matching substrings*.
- The feature value is the summation of the weights of all the occurrences of the *valid matching substring* in the string.
- The kernel value is evaluated as the the inner product of the new feature vectors of A and B.

Rationale:

The *kast spectrum kernel* was the first kernel designed in this work and was conceived as an improved version of the *blended spectrum kernel*, where long string matching is favored against short string matching. That is the reason why our kernels take the cut weight as an inferior limit, while the baseline kernel uses it as the superior limit. Segments of the strings that have been already matched can only be used for further matching with unmatched segments. This strategy reduces the similarity score.

Example:

Let A and B be the same strings of the previous example (Figure 3.12). The function $weight_{w \geq n}(A)$ returns the summation of the weights of all the tokens of A whose weight is greater than or equal to n. For a cut weight of 4 ($n = 4$), the respective weights of A and B are:

$$weight_{w \geq 4}(A) = 64 \tag{3.4}$$

$$weight_{w \geq 4}(B) = 52 \tag{3.5}$$

Let S be a *valid matching substring* according to the general kernel definition. The function $weight\_k0_{w\geq n}(S)_A$ returns the summation of the weights of all the matching instances of S in A whose weight is greater than or equal to n. Three matching substrings are obtained: $S1$, $S2$ and $S3$ (see Figures 3.13, 3.14 and 3.15). For a cut weight of 4 ($n = 4$),the respective weights of each feature in A are calculated with:

$$weight\_k0_{w\geq 4}(S1)_A = 19 \tag{3.6}$$

$$weight\_k0_{w\geq 4}(S2)_A = 7 + 6 = 13 \tag{3.7}$$

$$weight\_k0_{w\geq 4}(S3)_A = 6 + 9 = 15 \tag{3.8}$$

The new embedding feature vector for A is then:

$$f0_{w\geq 4}(A) = \{19, 13, 15\} \tag{3.9}$$

Now, the respective weights of each feature in B are calculated with:

$$weight\_k0_{w\geq 4}(S1)_B = 17 + 18 = 35 \tag{3.10}$$

$$weight\_k0_{w\geq 4}(S2)_B = 6 + 5 = 11 \tag{3.11}$$

$$weight\_k0_{w\geq 4}(S3)_B = 8 + 6 = 14 \tag{3.12}$$

The new embedding feature vector for B is:

$$f0_{w\geq 4}(B) = \{35, 11, 14\} \tag{3.13}$$

The function $k0_{w\geq n}(A, B)$ returns the evaluation of the kernel value between A and B; this is no more than the inner product of the new feature vectors:

$$k0_{w\geq 4}(A, B) = \langle f0_{w\geq 4}(A), f0_{w\geq 4}(B)\rangle = 1018 \tag{3.14}$$

The function $\bar{k}0_{w\geq n}(A, B)$ is the normalized version of the kernel. A normalization step will use the weights of each string:

$$\bar{k}0_{w\geq 4}(A, B) = \frac{k0_{w\geq 4}(A, B)}{\sqrt{k0_{w\geq 4}(A, A) \times k0_{w\geq 4}(B, B)}} = \frac{k0_{w\geq 4}(A, B)}{weight\_k0_{w\geq 4}(A) \times weight\_k0_{w\geq 4}(B)} \tag{3.15}$$

$$\bar{k}0_{w\geq 4}(A, B) = \frac{1018}{64 \times 52} = \frac{1018}{3328} \approx 0.3059 \tag{3.16}$$

In other words, according to this kernel, the strings are 30.59% similar.

3.3.3 Kast1 Spectrum Kernel

This kernel also intends to build a feature vector for each string based on the *valid matching substrings.* Having two strings A and B, the second kernel here presented is the *kast1 spectrum kernel*, which has the following definition:

- Each *valid matching substring* embeds a new feature for A and B. Hence, the size of the new embedding vector for both strings is equal to the number of *valid matching substrings.*
- Only the weights of the independent *valid matching substrings* are taken into account to build the feature value, which corresponds to the summation of these weights.
- If the string does not present an independent occurrence of a particular *valid matching substring*, the feature value is set to 1, to avoid zero values when calculating the inner product.
- The kernel value corresponds to the inner product of the new feature vectors of A and B.

Rationale:

During the experimentation stage, the results of the original kernel, namely, the *kast spectrum kernel*, did not show a remarkable difference with the baseline kernel (see Sections 5 and 6). With the first kernel, though the similarity score was mainly extracted from the largest common substrings, the gaps between the found segments were still considered *valid matching substrings.* This means that for small cut weight values, the gaps could be, in the worse case, of the size of a token, which, if it is spread all over the strings, could introduce a high weight value for such a small match. For that reason, we introduced an improvement in the weight calculation: in the *kast1 spectrum kernel*, if there is a match, only the weights of the independent valid substrings are taken into account. This strategy reduces the noise introduced by the small matches.

Example:

Let A and B be the same strings of previous examples (see Figure 3.12). Recall the weight calculation for A and B (see Equations 3.4 and 3.5). The function $weight\_k1_{w \geq n}(S)_A$ returns, either:

- the summation of the weights of all the independent matching instances of S in A whose weight is greater than or equal to n,
- or 1 if there are no independent substrings.

Though, the matching substrings are the same (see Figures 3.13, 3.14 and 3.15), the weight function has changed. For a cut weight of 4 ($n = 4$),the respective weights of each feature in A are calculated with:

$$weight\_k1_{w \geq 4}(S1)_A = 19 \tag{3.17}$$

$$weight\_k1_{w \geq 4}(S2)_A = 6 \tag{3.18}$$

$$weight\_k1_{w\geq4}(S3)_A = 9 \tag{3.19}$$

The new embedding feature vector for A is:

$$f1_{w\geq4}(A) = \{19, 6, 9\} \tag{3.20}$$

Notice in Figure 3.14 that $S2$ does not appear as an independent *valid matching substring* in B. Hence, the partial feature value is set to 1 (see Equation 3.22). The respective weights of each feature in B are calculated with:

$$weight\_k1_{w\geq4}(S1)_B = 17 + 18 = 35 \tag{3.21}$$

$$weight\_k1_{w\geq4}(S2)_B = 1 \tag{3.22}$$

$$weight\_k1_{w\geq4}(S3)_B = 6 \tag{3.23}$$

The new embedding feature vector for B is:

$$f1_{w\geq4}(B) = \{35, 1, 6\} \tag{3.24}$$

The function $k1_{w\geq n}(A, B)$ returns the evaluation of the kernel value between A and B; this is no more than the inner product of the new feature vectors:

$$k1_{w\geq4}(A, B) = \langle f1_{w\geq4}(A), f1_{w\geq4}(B)\rangle = 725 \tag{3.25}$$

The function $\bar{k1}_{w\geq n}(A, B)$ is the normalized version of the kernel. A normalization step will use the weights of each string:

$$\bar{k1}_{w\geq4}(A, B) = \frac{k1_{w\geq4}(A, B)}{\sqrt{k1_{w\geq4}(A, A) \times k1_{w\geq4}(B, B)}} = \frac{k1_{w\geq4}(A, B)}{weight\_k1_{w\geq4}(A) \times weight\_k1_{w\geq4}(B)} \tag{3.26}$$

$$\bar{k1}_{w\geq4}(A, B) = \frac{725}{64 \times 52} = \frac{725}{3328} \approx 0.2178 \tag{3.27}$$

It is possible to say that these two strings are 21.78% similar.

3.3.4 Kast2 Spectrum Kernel

One of the differences of this kernel with respect to the others, is that it does not built a vector of features. It only extracts a single feature from each string. Having two strings A and B, the third kernel we propose is the *kast2 spectrum kernel*, which has the following definition:

- Only the weights of the independent *valid matching substrings* are taken into account to build a partial feature value, which is the summation of these weights.
- Unlike the previous kernel, if the string does not present an independent occurrence of a particular *valid matching substring*, no weight value is taken into account.

- The major difference with the previous kernels is that only a single feature value is created for each string A and B, which corresponds to the summation of all partial feature values.
- A penalization value is introduced and it corresponds to the number of effective segments reduced by one; an effective segment is an instance of an independent *valid matching substring*. this value is subtracted from the feature value; the reduction by one prevents exact matching strings from being penalized.
- The kernel value corresponds to the product of the feature values after penalization.

Rationale:

Finally, the *kast2 spectrum kernel* was created to reflect more naturally the way string matching is done. Instead of performing the inner product immediately after finding a valid matching substring, this kernel sums up all the their weights an creates a single feature. In other words, the feature value of a string is the weight of all matching segments of a string. The penalization value automatically favors long string matches: the more segmented is a string, the major the penalization.

Example:

Let A and B be the same strings from previous examples (see Figure 3.12). Recall the weight calculation for A and B (see Equations 3.4 and 3.5). The function $weight\_k2_{w \geq n}(S)_A$ returns, either:

- the summation of the weights of all the independent matching instances of S in A whose weight is greater than or equal to n,
- or 0 if there are no independent substrings.

Though, the matching substrings are the same (see Figures 3.13, 3.14 and 3.15), the weight function has been changed again. For a cut weight of 4 ($n = 4$),the respective weights of each partial feature in A are calculated with:

$$weight\_k2_{w \geq 4}(S1)_A = 19. \tag{3.28}$$

$$weight\_k2_{w \geq 4}(S2)_A = 6. \tag{3.29}$$

$$weight\_k2_{w \geq 4}(S3)_A = 9. \tag{3.30}$$

The only feature of A is the summation of the previous weights:

$$f2_{w \geq 4}(A) = 19 + 6 + 9 = 34. \tag{3.31}$$

The penalization value in this case is 2:

$$p2_{w \geq 4}(A) = 34 - 2 = 32. \tag{3.32}$$

Notice in Figure 3.14 that $S2$ does not appear as an independent *valid matching substring* in B. Hence, the feature value is set to 0 (see Equation 3.34). The respective weights of each substring in B are:

$$weight\_k2_{w\geq 4}(S1)_B = 17 + 18 = 35. \tag{3.33}$$

$$weight\_k2_{w\geq 4}(S2)_B = 0. \tag{3.34}$$

$$weight\_k2_{w\geq 4}(S3)_B = 6. \tag{3.35}$$

Here too, the only feature of B is the summation of the previous weights:

$$f2_{w\geq 4}(B) = 35 + 6 = 41. \tag{3.36}$$

For this case, the penalization value is also 2:

$$p2_{w\geq 4}(B) = 41 - 2 = 39. \tag{3.37}$$

The function $k2_{w\geq n}(A, B)$ returns the evaluation of the kernel value between A and B; this is no more than the product of these two values:

$$k2_{w\geq 4}(A, B) = \langle p2_{w\geq 4}(A), p2_{w\geq 4}(B)\rangle = 1248. \tag{3.38}$$

The function $\bar{k2}_{w\geq n}(A, B)$ is the normalized version of the kernel. A further normalization step using the weights of each string can be applied:

$$\bar{k2}_{w\geq 4}(A, B) = \frac{k2_{w\geq 4}(A, B)}{\sqrt{k2_{w\geq 4}(A, A) \times k2_{w\geq 4}(B, B)}} = \frac{k2_{w\geq 4}(A, B)}{weight\_k2_{w\geq 4}(A) \times weight\_k2_{w\geq 4}(B)}. \tag{3.39}$$

$$\bar{k2}_{w\geq 4}(A, B) = \frac{1248}{64 \times 52} = \frac{1248}{3328} \approx 0.375. \tag{3.40}$$

In this case, the kernel emits a similarity score of 37.5% between A and B.

We have seen how each of the proposed kernels is an evolution of the previous one. In theory, more *kastx* kernels could be designed by changing the way the weights are summarized, as long as the four rules are respected.

Summary

In this chapter, we introduced to the reader a string representation and three novel string kernel functions for the comparison of those strings. We also showed how generic trees, Clang Abstract Syntax Trees, LLVM Intermediate Representations and I/O traces can be converted to the mentioned string form. A proof of the concepts here presented is developed in the next chapter in the form of a more detailed example.

4 Proof of Concept

In the previous chapter we introduced to the reader a string representation based on weights, as well as three novel string kernel functions to compare them. In this chapter we present a proof of the concept by means of an example with synthetic strings using symbols of three different alphabets. The idea behind is to show how the kernels are able to separate the strings into the expected classes. The first section (4.1) explains the reasons why the ***blended spectrum kernel*** *was selected as the baseline kernel. Section 4.2 presents the structure of the alphabets per class, while Section 4.3 details the strings that compound each class. Section 4.4 explains how the experiment is configured. Section 4.5 shows the results of running the three kernels and presents the comparison with the results of the baseline algorithm. Finally, Section 4.6 discusses the complexity order of the developed algorithms.*

4.1 Blended Spectrum Kernel as Baseline Kernel

The intention of this thesis is to explore the application of string kernels in the problem of detection of patterns in I/O traces and Intermediate Representations. The *blended spectrum kernel* has been successfully used for the classification of proteins, but not in the mentioned problem of this research.

Given the particular form of the string representation proposed here, where a group of subsequent tokens can encode more meaningful information than a single one, we discarded the *bag-of-characters* and the *bag-of-words* kernels. Experimental evaluation done during this thesis (see Chapters 5 and 6) also showed that the *k-spectrum kernel* was not successful at finding an acceptable clustering, a task where the *blended spectrum kernel* had a better performance.

A characteristic of the *blended spectrum kernel* is that it takes into account all the *k-spectrum* kernels below a certain given k (the name *blended* reflects the fact that is a mix of other kernels). Hence, the smaller the value, the cheaper the computation gets.

On the contrary, all the proposed kernels in this thesis take into account the values above the cut weight. For them, the smaller the cut weight value, the more expensive the computation gets.

4.2 Classes and Alphabets

In order to test the viability of our approach, we have created a synthetic set of strings, which can be divided in two broad categories:

- *Primary Classes (A,B and C):* Primary classes are characterized by having their own alphabet. None of the elements of an alphabet are present in the others. For simplification purposes, the alphabets of the primary classes do not have more than seven elements.

- *Mixed Class (X):* The alphabet for class X is given by the union of all alphabets. The idea behind the use of a mixed class is to see how this class is placed in a data visualization analysis with respect to the primary ones.

The expectation is to see the four classes clearly separated, with the mixed one in the middle. All the alphabets are listed in Table 4.1.

4.3 Classes and Strings

Each primary class contains four strings (see Tables 4.2, 4.3 and 4.4). Two of the strings are pure, which means they are compound only by symbols of the class alphabet. The other two are impure, which means they present elements from other alphabets; in any case, the external part represents no more than a quarter of the string. The idea behind the usage of impure strings is to see how they are located with respect to the pure ones.

The mixed class contains only two examples (see Table 4.5). The notion of pure and impure strings does not apply for this class.

4.4 Experiment Configuration

For the sake of simplicity, the following rules are applied:

- All strings have the same size (27 tokens).
- Each token (symbol) has an implicit weight of 1, so the weight of each string is always 27.
- As the tokens are single characters, the enclosing symbols `[` and `]` are omitted.

The three proposed kernels are applied over the 14 strings using a **cut weight** from 1 to 27; the idea is to see how the selection of the cut weight affects the formation of the clusters. The obtained kernel matrices are analyzed using Kernel PCA [SSM97]; the dimensionality reduction of this technique allows the detection of linearly separable clusters. For the sake of visualization, the first two principal components are used to plot the data items position, with the first component as the horizontal axis and the second component as the vertical axis.

4.5 Results

In this section we discuss the observed patterns and their relationship with the cut weight selection. We show that, although all kernels present two patterns where the separation of classes is clear, the proposed kernels are more sensible to the selection of the cut weight than the baseline kernel.

4.5.1 Observed Patterns

The three proposed kernels show a similar behavior, totally different to the baseline algorithm. For this reason, we have only included the figures of one kernel, the *kast spectrum kernel.* A different behavior from the baseline kernel is expected, as the matching strategy used by both classes of algorithms is different.

A alphabet	`a,b,c,d,e,f,g`
B alphabet	`1,2,3,4,5`
C alphabet	`s,t,u,v,x,y,z`
X alphabet	`a,b,c,d,e,f,g,1,2,3,4,5,s,t,u,v,x,y,z`

Table 4.1: Alphabets for Primary and Mixed Classes

Pure Strings	
A0	`abcdefgacdefgabcdefgabcdefg`
A1	`cccdefgacdefgabcdefgabcdecc`
Impure Strings	
A2 (contains elements of B)	`cccdefgacd12345cdefgabcdecc`
A3 (contains elements of C)	`cccdefgacdefgstuvxzgabcdecc`

Table 4.2: Strings for Primary Class A

Pure Strings	
B0	`123451234512345123451234512`
B1	`223311123451234512345145231`
Impure Strings	
B2 (contains elements of A)	`223abcde3451234512345145231`
B3 (contains elements of C)	`2233111234512345l234stuvx31`

Table 4.3: Strings for Primary Class B

Pure Strings	
C0	`stuvxzystuvxzystuvxzystuvxz`
C1	`stuvxzystuvxzystuvxzytuvzxy`
Impure Strings	
C2 (contains elements of B)	`stuvxzystuvxzystuvxzyt12345`
C3 (contains elements of A)	`abcdezystuvxzystuvxzytuvzxy`

Table 4.4: Strings for Primary Class C

X0	`1234stuvxabcdstuvx123abcdxy`
X1	`1as34stuvxabcdstuvx122xev4f`

Table 4.5: Strings for Mixed Class X

Pattern 1: Clear inter-cluster separation and meaningful intra-cluster distances

Seen on the following kernels: All (*kastx* kernel family and baseline kernel).

Details: For small values of the cut weight, all kernels (1 to 3 for the *kast spectrum kernel*, and 1 to 5 for the rest, including the baseline kernel), pure strings tend to be closer to each other than to the impure strings (see Figures 4.1 and 4.2). Recall that the proposed kernels have a similar clustering scheme, hence, we have included the figures of one kernel only. Notice that all classes are clearly separated and the mixed class (class X) is always in the middle. The impure strings tend to be located to the side of the class where the external symbols belong.

The same pattern is seen again with the *blended spectrum kernel* at higher cut weight levels (15 to 27).

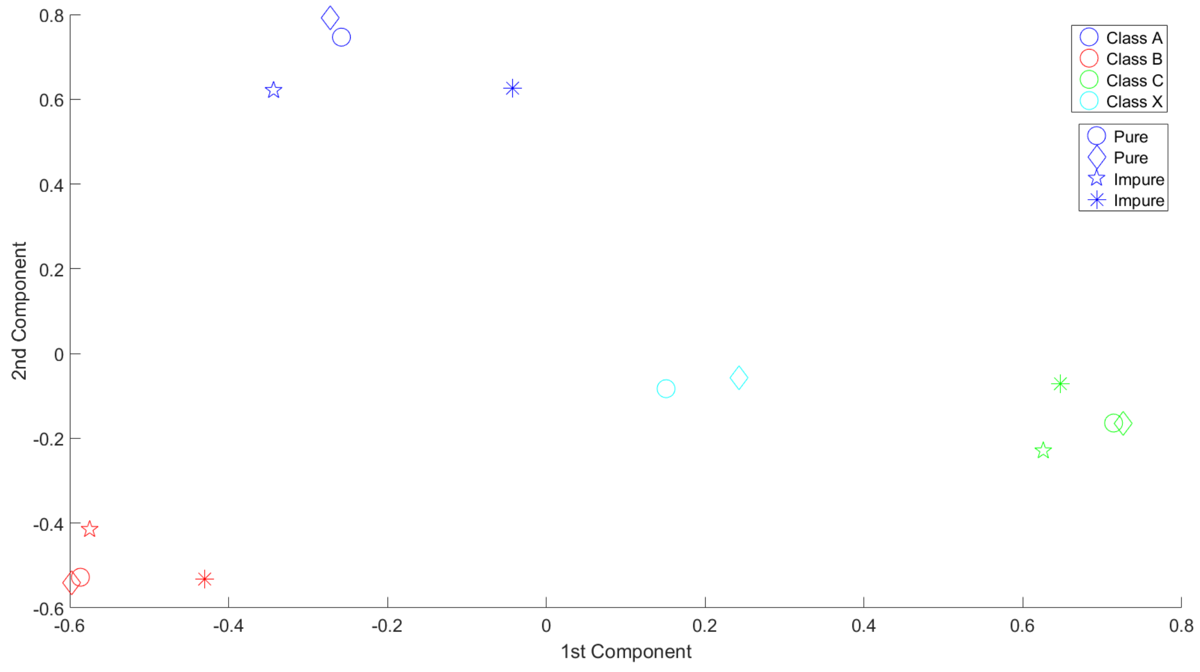


Figure 4.1: Pattern 1 Observed in Kernel PCA for Blended Spectrum Kernel in Synthetic Strings (cut weight = 5).

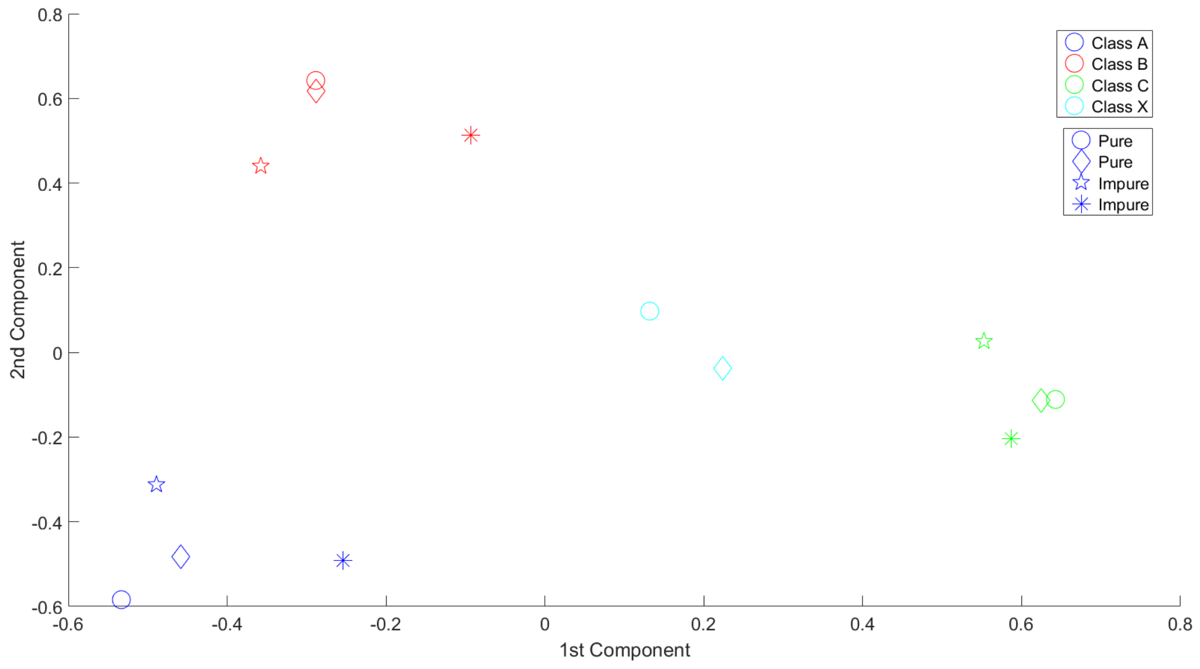


Figure 4.2: Pattern 1 Observed in Kernel PCA for Kast Spectrum Kernel in Synthetic Strings (cut weight = 3).

Pattern 2: Clear inter-cluster separation

Seen on the following kernels: All (*kastx* kernel family and baseline kernel).

Details: Pure strings are preserved on their respective groups for cut weights from 1 till 14 (see Figures 4.3 and 4.4), while some impure strings tend to join the mixed class. The intra cluster separation between pure and impure strings no longer holds for all classes. However, the mixed class is still located in the middle.

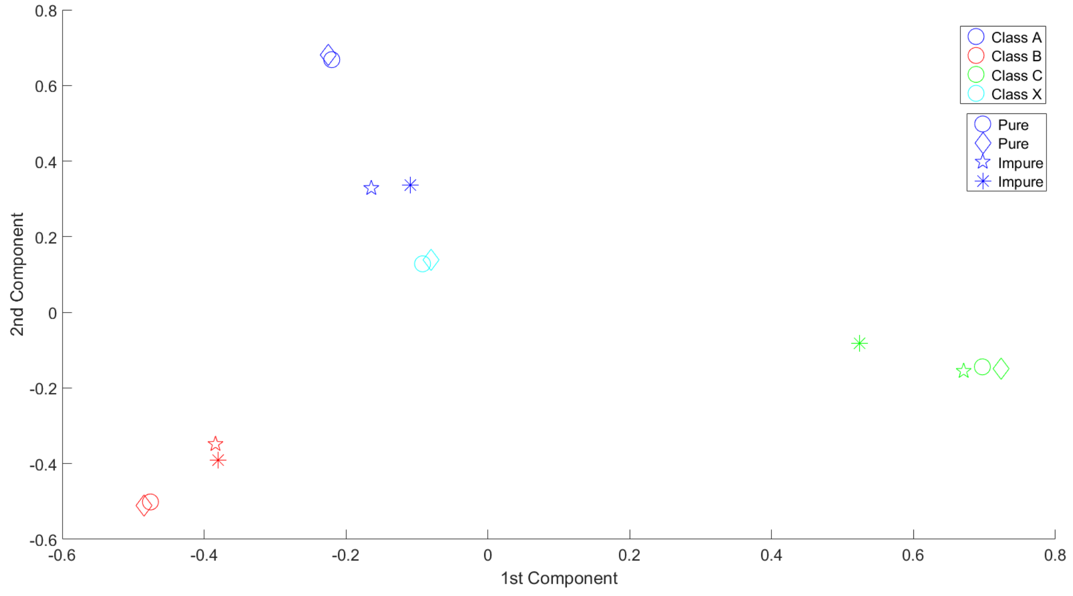


Figure 4.3: Pattern 2 Observed in Kernel PCA for Blended Spectrum Kernel in Synthetic Strings (cut weight = 14).

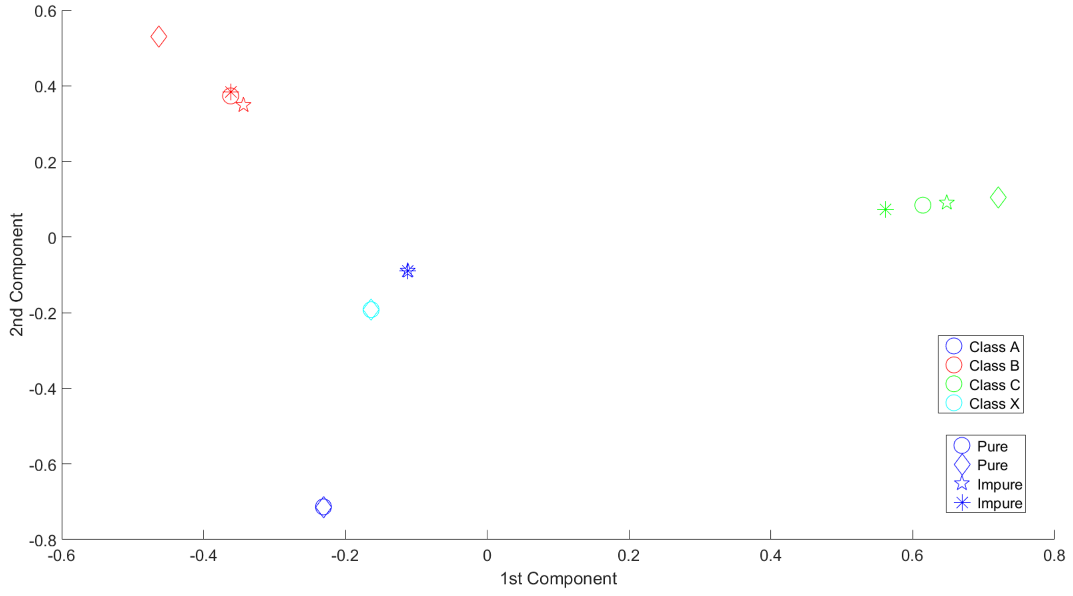


Figure 4.4: Pattern 2 Observed in Kernel PCA for Kast Spectrum Kernel in Synthetic Strings (cut weight = 14).

Pattern 3: One class tended to merge the mixed class

Seen on the following kernels: *kastx* kernel family only.

Details: For cut weights from 15 till 21, the pure strings of one of the primary classes are no longer in a separate cluster, and tend to join the mixed class (see Figure 4.5).

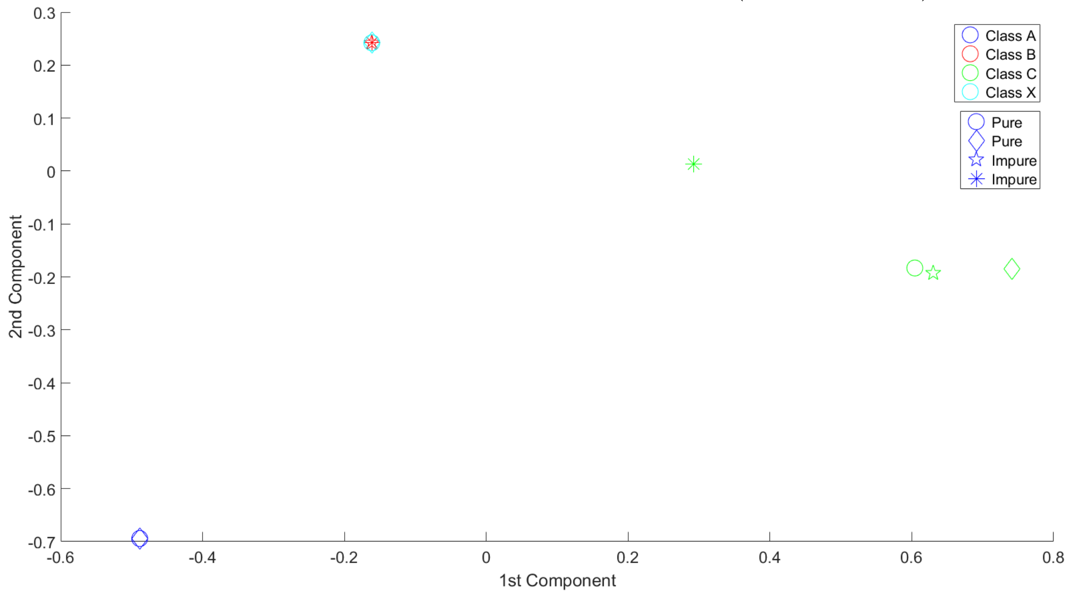


Figure 4.5: Pattern 3 in KPCA for Kast Spectrum Kernel (cut weight = 21).

Pattern 4: Two classes tended to merge the mixed class

Seen on the following kernels: *kastx* kernel family only.

Details: With cut weights of 22 and 23, only the pure strings of one class are separated from the mixed cluster (see Figure 4.6).

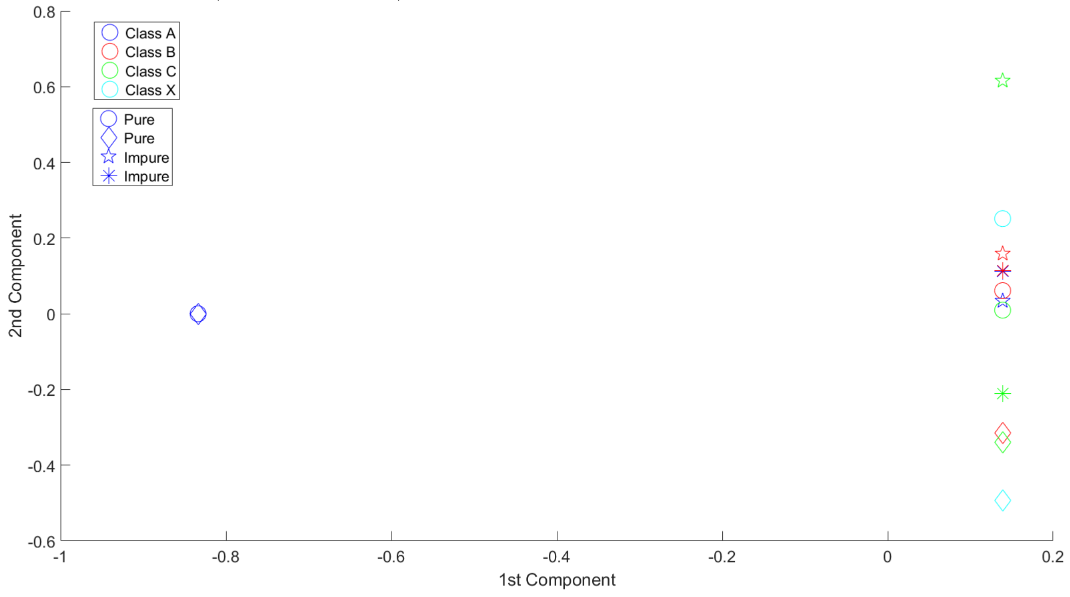


Figure 4.6: Pattern 4 in KPCA for Kast Spectrum Kernel (cut weight = 23).

Pattern 5: No pattern

Seen on the following kernels: *kastx* kernel family only.

Details: For cut weights of 24 till the size of the strings (27), no class patterns are detected.

4.5.2 Remarks

From a certain cut weight close to half the size of the strings, the proposed kernels show a different behavior to the baseline kernel. The baseline kernel presents only the Pattern 1 and 2, which are the ones showing the clearest cluster separation. Because this kernel adds up the spectrums below the cut weight, it stabilizes once the maximum substring size is reached. On the contrary, the proposed kernels take into account substrings whose weight is above the cut weight; hence, if the cut weight is greater than the largest matching substring size, no patterns will be detected.

The last observation clearly gives a hint: in order to detect meaningful patterns, the cut weight should be small enough to cover at least the size of all the meaningful largest matching substrings among all observations. In the given example, half the size of the strings is the decision boundary.

The size of the strings in this example is too small to show differences between the proposed kernels but have helped to prove the stability of the approach. To see major differences among the kernels, the strings have to be longer, as it will be seen in the following chapters.

4.6 Complexity Considerations

All developed kernels present the same general code structure (see Listing 4.1), varying only in the most nested loops, where the weights are summarized. In order to simplify this analysis, let us assume the following:

- The strings have the same size.
- All tokens can be represented by a single character, which makes unnecessary the consideration of a token separator.
- The weight of each token is 1, so the number of tokens corresponds to both the weight and the size of the string, and its is represented by n.

4.6.1 Worse Case Complexity

In order to consider the worse case scenario, the cut weight is set to 1 and the strings to be compared, namely, A and B, have no substrings in common. Recall that in our kernels, the cut weight acts as a lower limit, and all the weights above it must be considered. The algorithm is analyzed from the inner loops to the outer loops.

Inner Loops

Loop 3 starts at line 20 and loop 4 at line 29. A reference string is searched in B using the `find` algorithm of C++, which has, in this case a worse complexity order of $O(n^2)$. If a match is found, the corresponded match is marked down in a separate copy, which is important to keep track of the matching progress. Same is done in string A. This step is necessary to ensure that all copies of the reference string in A are also marked down. Due to the fact that the two inner loop are independent from each other, the combined complexity of them corresponds to the summation of their complexities:$O(n^2) + O(n^2) = O(n^2)$.

Outer Loops

Loop 1 starts at line 6 and loop 2 at line 16. Starting from the largest substring of A, each substring of A is extracted as a reference substring. The search starts always from left to right. This operation is performed by two external loops which have a computational cost of $n(n-1)/2$, as it represents a classical combinatorial loop. The complexity order until this two loops is: $O(n^2)$.

Total Complexity

Due to the fact that the inner loops are nested inside the outer loops, the overall complexity is given by the multiplication of their values: $O(n^2) \times O(n^2) = O(n^4)$. In this case, the weight of the substring corresponds to its length. However, when the weights can not be guessed, the algorithm precises a third loop nested inside the inner loops that collects the weights of each token, with linear complexity. This would increase the complexity of the inner loops to $O(n^3)$, hence increasing the complexity of the whole algorithm to $O(n^6)$.

4.6.2 Best Case Complexity

The two conditions after the first loop permit to avoid further execution by checking in the marked down strings how many tokens are still available for comparison. If the number of available tokens is not sufficient, the loop skips to a lower number of tokens. The best case scenario occurs when both strings are equal. In this case, there are no more tokens left, which is control by the second condition (line 12), which interrupts the execution of the loop. This means that the overall complexity of the algorithm is given only by the `find` function of C++, which has, in the best case, the complexity order of: $\Omega(n)$, when both strings are equal.

4.6.3 Baseline Algorithm Complexity

In order to perform the experimentation, we developed our own implementation of the baseline algorithm, using a similar code structure for the *kastx* kernel family (see Listing 4.1). The only difference is that, for the baseline algorithm, the worse case scenario occurs when the cut weight is set to the maximum weight, in this case, the length of the string, opposite to our kernels. Similarly, the best case happens when the cut weight is set to 1. This happens because, in the baseline algorithm, the cut weight acts as the upper limit, and all the weights below must be taken into consideration.

```
float* kernelCalculation(string StringA, string StringB, string MarkedA, string MarkedB, int n)
{
    float *featureVector;
    int remainingTokensA = n;
    int remainingTokensB = n;
    for(int i = n; i > 0 ; i--) // LOOP 1
    {
        if(remainingTokensA < i || remainingTokensB < i )
        {
            continue;
        }
        if(remainingTokensA == 0 || remainingTokensB == 0 )
        {
            break;
        }
        for(int j = 0; j <= n-i ; j++) // LOOP 2
        {
            string subString = StringA.substr(j,i);
            int p = 0;
            while(( p = StringB.find(subString, p)) != string::npos) // LOOP 3
            {
                if(MarkedStringB.substr(p,i) == subString)
                {
                    MarkedStringB.replace(p,i,i,'_');
                    remainingTokensB = remainingTokensB - i;
                }
            }
            int p = j + i;
            while(( p = StringA.find(subString, p)) != string::npos) // LOOP 4
            {
                if(MarkedStringA.substr(p,i) == subString)
                {
                    MarkedStringA.replace(p,i,i,'_');
                    remainingTokensA = remainingTokensA - i;
                }
            }
            // Feature vector calculation
            ...
        }
    }
    return featureVector;
}
```

Listing 4.1: C++ Code Segment Scheme of the Kernel Calculation.

Summary

We have shown how the proposed kernels correctly separate a group of synthetic strings into the expected classes they belong, and how they perform with respect to a baseline algorithm. Additionally, we have discussed their complexity. Next chapter is dedicated to present the results of the application of these kernels in a problem of the real world, namely, the detection of patterns in I/O traces. There, the kernels show a different behavior among each other.

5 Comparison of I/O Traces

A preliminary experimentation stage with I/O traces was the first application of the kastx kernel family in a real domain. This phase helped to refine the kernel design and paved the way for the main highlight of this work, which is the comparison of intermediate representations (see Section 6). In that sense, in this chapter, we present those initial findings. Understanding I/O for data-intense applications is the foundation for the optimization of these applications. The classification of the applications according to the expressed I/O access pattern eases the analysis. The three kernels have been compared against the ***blended spectrum kernel****, a kernel found in the literature that is noted to be good to find similarities among strings. The similarity matrices have been analyzed using two well-known clustering algorithms: Hierarchical clustering and kernel principal component analysis. Section 5.1 is dedicated to detail the experiment configuration. Section 5.2 shows the results obtained with the baseline kernel. Sections 5.3, 5.4 and 5.5 present the results for the kernels proposed here.*

5.1 Experiment Configuration

It is on the interest of this thesis to study the suitability of the proposed strategy to find similarities among four distinct classes of I/O access patterns, which have been obtained from two different parallel I/O benchmarks:

- *The IOR HPC Benchmark:* It is used for benchmarking parallel file systems that use POSIX, MPIIO, or HDF5 interfaces [LMM12].
- *The FLASH I/O Benchmark:* It measures the performance of the FLASH parallel HDF 5 output. FLASH is scientific tool for modeling astrophysical thermonuclear flashes [Fry+00].

5.1.1 Preprocessing

The I/O traces are organized in the following classes of storage access:

- *Class A (Flash I/O):* 10 traces. Characterized for containing contiguous `write` operations with diverse byte values that are not present in the other classes.
- *Class B (Random POSIX I/O):* 4 traces. These ones present `lseek` operations not seen elsewhere.
- *Classes C and D (Sequential I/O):* 4 traces each. 8 in total. Classes C and D do not have any remarkable difference among them, a fact that has been confirmed by experimentation. However, they come from different runs, that is why we present them here separately.

Strings Representation	Using Byte Information	Ignoring Byte Information
Number of tokens	32 - 111	32 - 45
Weight range	1473 - 8386	1460 - 8387

Table 5.1: Ranges for both String Representations for I/O traces (the byte information corresponds to the number of bytes involved in the I/O operations).

The traces have a size that ranges between 1504 and 16587 lines. For each trace, 4 additional synthetic copies have been created. Such copies introduce small mutations on the pattern; the idea behind these mutations is the need to create access patterns that are, in theory, closer to a determined example than to the rest of the category members. The creation of synthetic I/O traces has been used before to facilitate the analysis of the behavior of programs whose source code cannot be accessed, mainly due to copyright restrictions [Beh+14]. This results in a total of 110 examples for our similarity study.

5.1.2 String Conversion and Comparison

As seen previously (see Chapter 3), it is necessary to filter the operations that are not important for the formation of patterns. Deleting the unnecessary information reduces the noise and facilitates the detection of similarities. After this filtering step, each trace is converted into the two proposed string representations; Table 5.1 details the respective ranges in number of tokens and weights for each of them. The three kernels here proposed have been applied to the set of strings, as well as the *blended spectrum kernel*. The selected **cut weight** values are the following: $\{2^1, 2^2, ..., 2^n\} : n = 10$. If the resulting matrices present negative eigenvalues, those values are replaced by zeros and the matrices are rebuilt. After that, all the similarity matrices are analyzed with both Kernel Principal Component Analysis (Kernel PCA) and Hierarchical Clustering (HC), the latest using the simple linkage method.

5.2 Baseline Kernel: Blended Spectrum Kernel

Hereby we present first the results of the baseline kernel, as this facilitates the comparison with the proposed kernels. The results of the experiment have shown that the baseline kernel presents a bad performance at separating classes, as it has been able to identify only two groups, one conformed by all the examples in class A, and another one conformed by all the examples of the other three classes. In other words, this kernel has detected only those patterns related to the Flash I/O benchmark, regardless of the string representation involved. However, as it is explained next, those strings that keep the byte information from the I/O traces are less sensitive to the changes of the cut weight than the ones that ignore it. Let us review each case in detail.

5.2.1 Clustering with Strings Using Byte Information

When the experimentation involves strings that keep the byte information from the I/O traces, the variation in the cut weight value does not affect significantly the clustering scheme. Unfortunately, the obtained clustering is not good enough to separate all the classes. For example, with Hierarchical Clustering, two clusters are conformed, one corresponding to class A (Flash I/O), while the other one gathers all the remaining 3 classes (B-C-D) (see Figure 5.1). The results are even worse with Kernel PCA: although class A is still separated from the rest, it is shown divided into two clusters (see Figure 5.2). This shows that, for the baseline kernel, the internal differences of the members of this class have the same strength as the differences among classes, which does not correspond with the expected behavior.

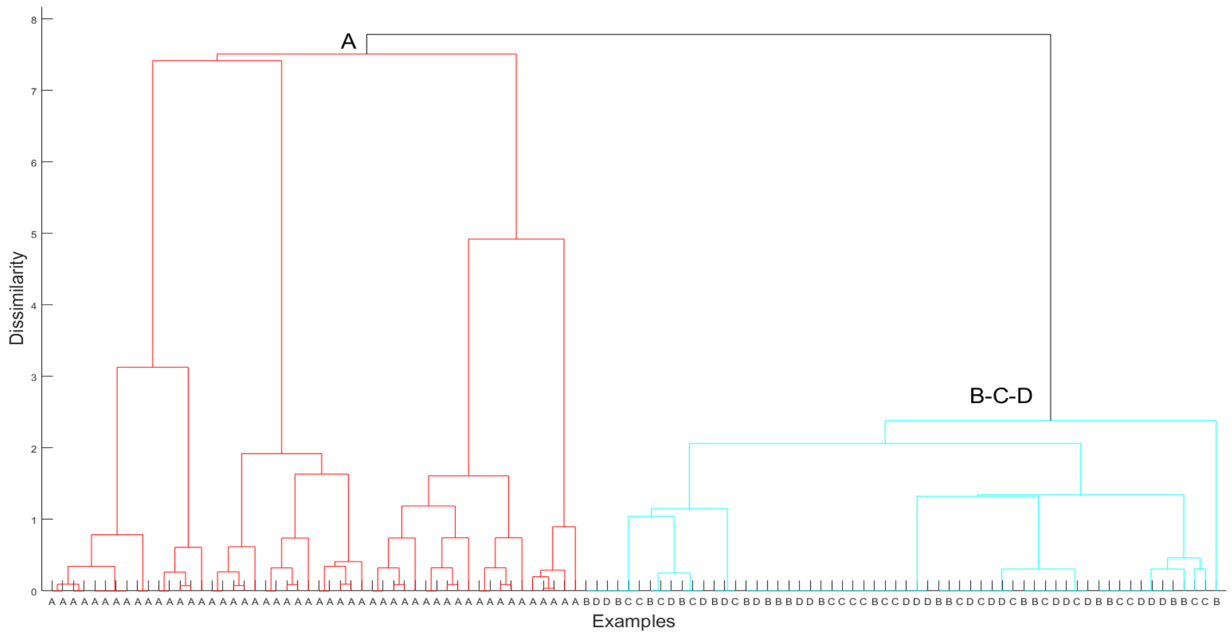


Figure 5.1: Hierarchical Clustering for Blended Spectrum Kernel in I/O Traces using Byte Information (Cut Weight = 2).

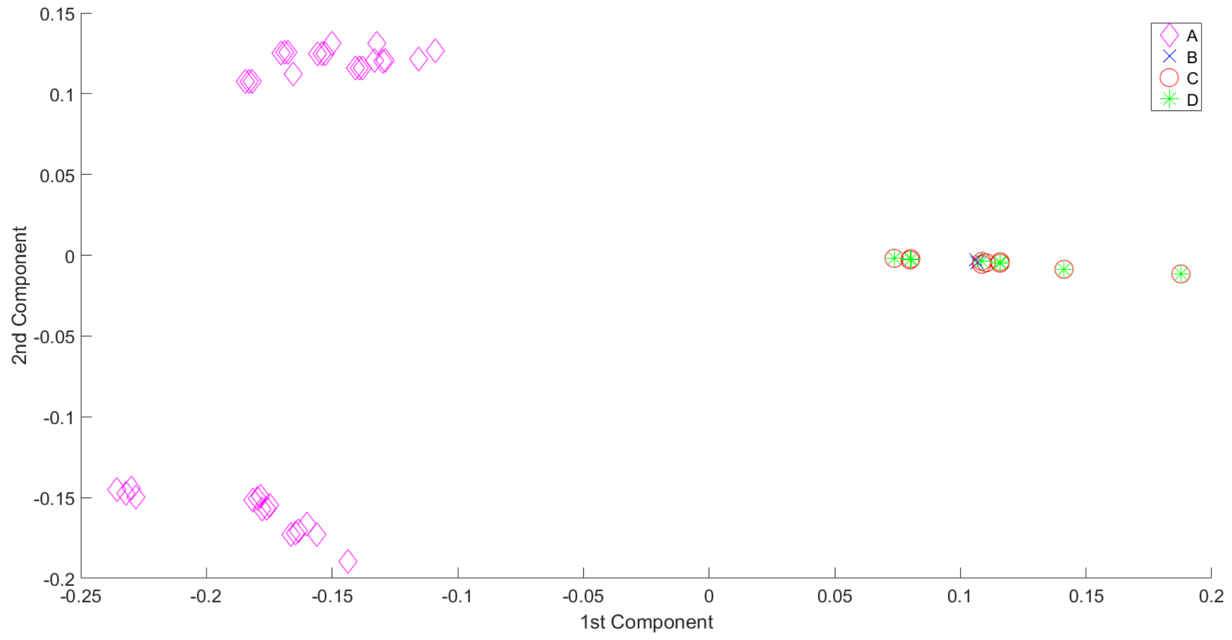


Figure 5.2: Kernel PCA for Blended Spectrum Kernel in I/O Traces using Byte Information (Cut Weight = 2).

5.2.2 Clustering With Strings Ignoring Byte Information

When the experimentation involves strings that ignore the byte information from the I/O traces, and additionally, small cut weight values are used, the results of both clustering techniques are not satisfactory, because the members of all classes appear mixed together, and no clear cluster formation is seen. In order to achieve at least the separation of one class, higher cut weight values have to be used, e.g. 512 (see Figure 5.3). Only then the results are similar to those obtained by the other category of strings. Still, class A (Flash I/O) keeps being divided into two clusters with Kernel PCA (see Figure 5.4).

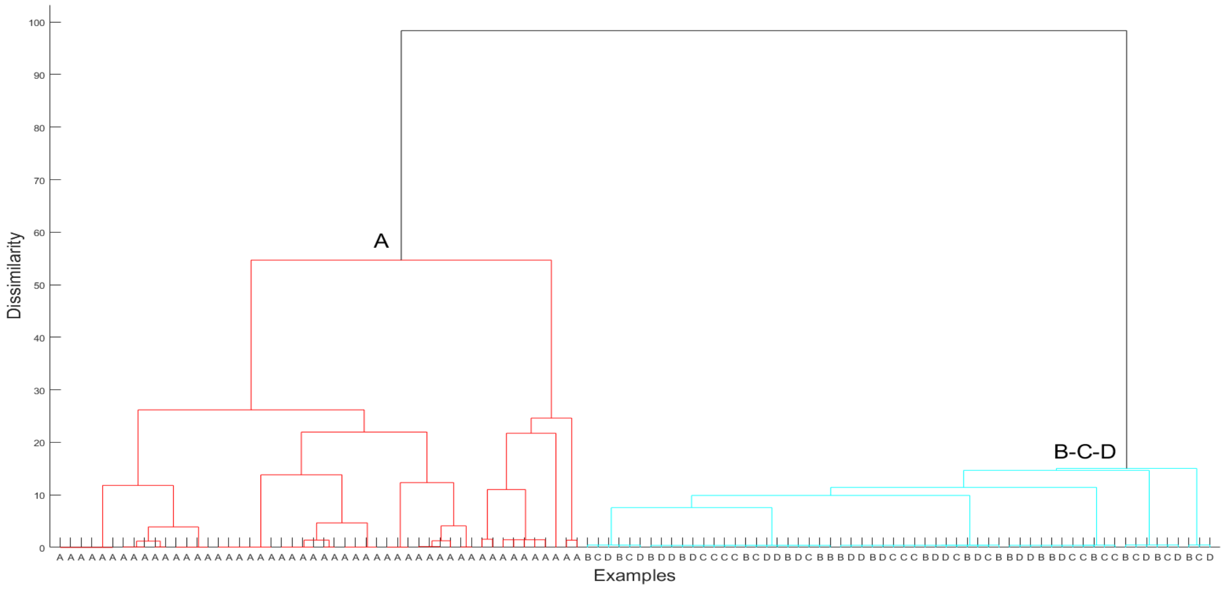


Figure 5.3: Hierarchical Clustering for Blended Spectrum Kernel in I/O Traces ignoring Byte Information (Cut Weight = 512).

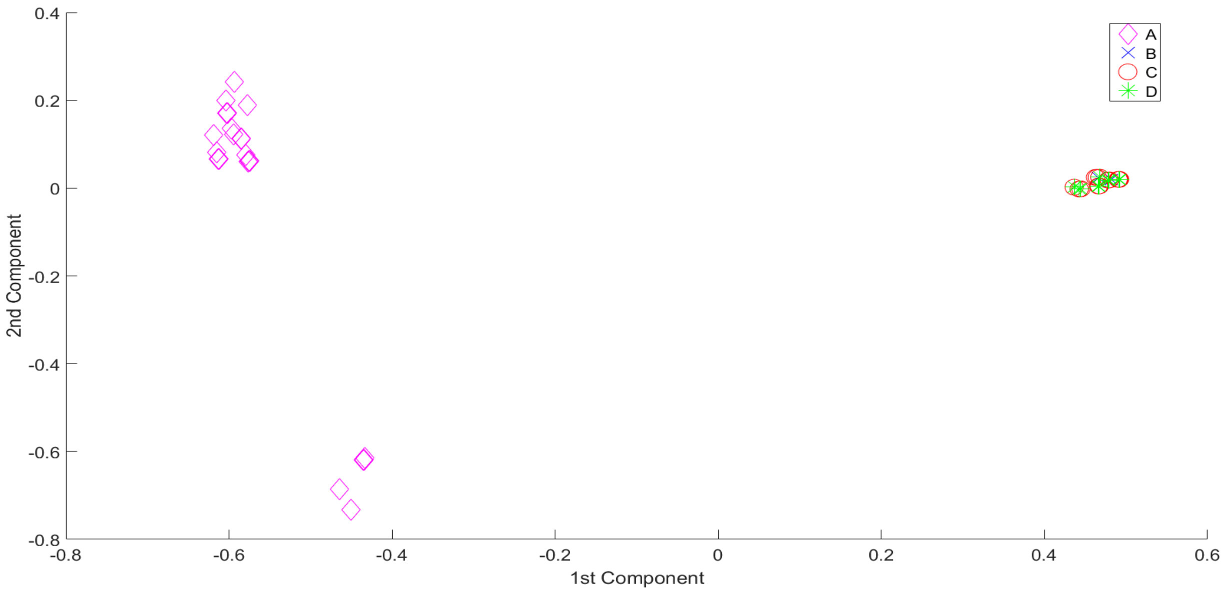


Figure 5.4: Kernel PCA for Blended Spectrum Kernel in I/O Traces ignoring Byte Information (Cut Weight = 512).

5.2.3 Remarks

The baseline kernel presents the following particularities: On the one hand, small cut weight values suffice for comparing strings using the byte information, which maintains the comparison cheap and the parametrization efforts low. On the other hand, strings lacking the byte information are shorter, hence, their comparison is cheaper. But they require higher cut weight values to work properly, which incurs in higher computational costs and additional parametrization efforts. In any case, only the patterns coming from the Flash I/O benchmark have been detected by this kernel.

5.3 Kast Spectrum Kernel

The first kernel to be tested is the *kast spectrum kernel*. This kernel has shown a better performance than the baseline kernel, as it is able to separate 3 clusters, while the baseline kernel have identified only 2. Two of the clusters correspond to two classes, namely, the Flash I/O (class A) and the POSIX I/O (class B). The third cluster gathers together Sequential Access I/O (classes C and D), which corresponds to the expectation, as examples of these classes are very similar among each other. Similar as with the baseline kernel, the use of strings that keep the byte information are not affected significantly by the selection of the cut weight, while the other category of strings work only if the cut weight is high.

5.3.1 Clustering with Strings Using Byte Information

The application of the first of the proposed kernel functions (*kast spectrum kernel*) over strings that preserve the byte information from the I/O operations, achieves the best results when a small cut weight is used, e.g. 2. The fact that small cut weights are sufficient to achieve a meaningful clustering, eases the parametrization of the comparison process. It is remarkable that both learning algorithms clearly separate the same 3 clusters (see Figures 5.5 and 5.6):

- Class A (Flash I/O).
- Class B (POSIX I/O).
- Classes C and D (Sequential I/O).

The kernel captures the similarity between classes C and D, as they share the same pattern. It is important to notice that there are no misplaced examples on any of the groups in the HC analysis. For higher cut weight values, e.g. 512, the same clusters are identified, with the difference that the intra cluster distances in the HC are smaller. Notice also that with Kernel PCA, Flash I/O samples are divided into two parts; however, the distance between these two parts is negligible in comparison with the inter-cluster distances. This shows that the new kernel is more capable to generalize the I/O patterns than the baseline kernel.

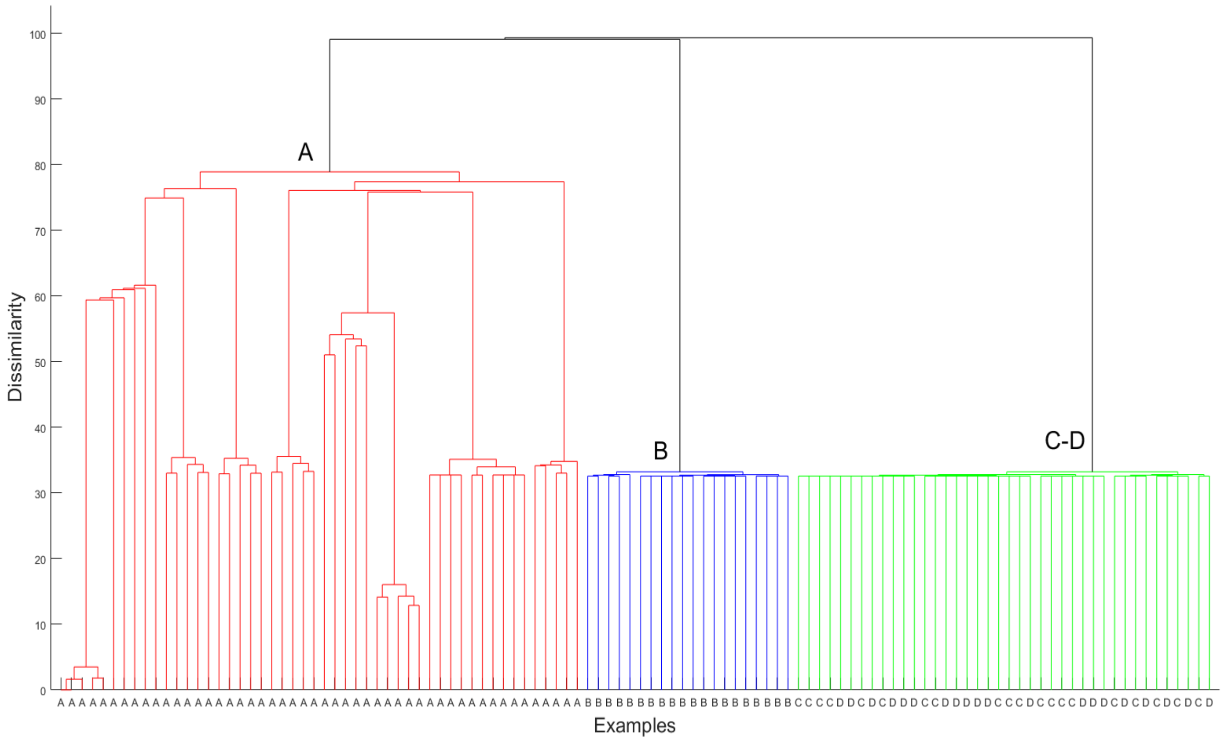


Figure 5.5: Hierarchical Clustering for Kast Spectrum Kernel in I/O Traces using Byte Information (Cut Weight = 2).

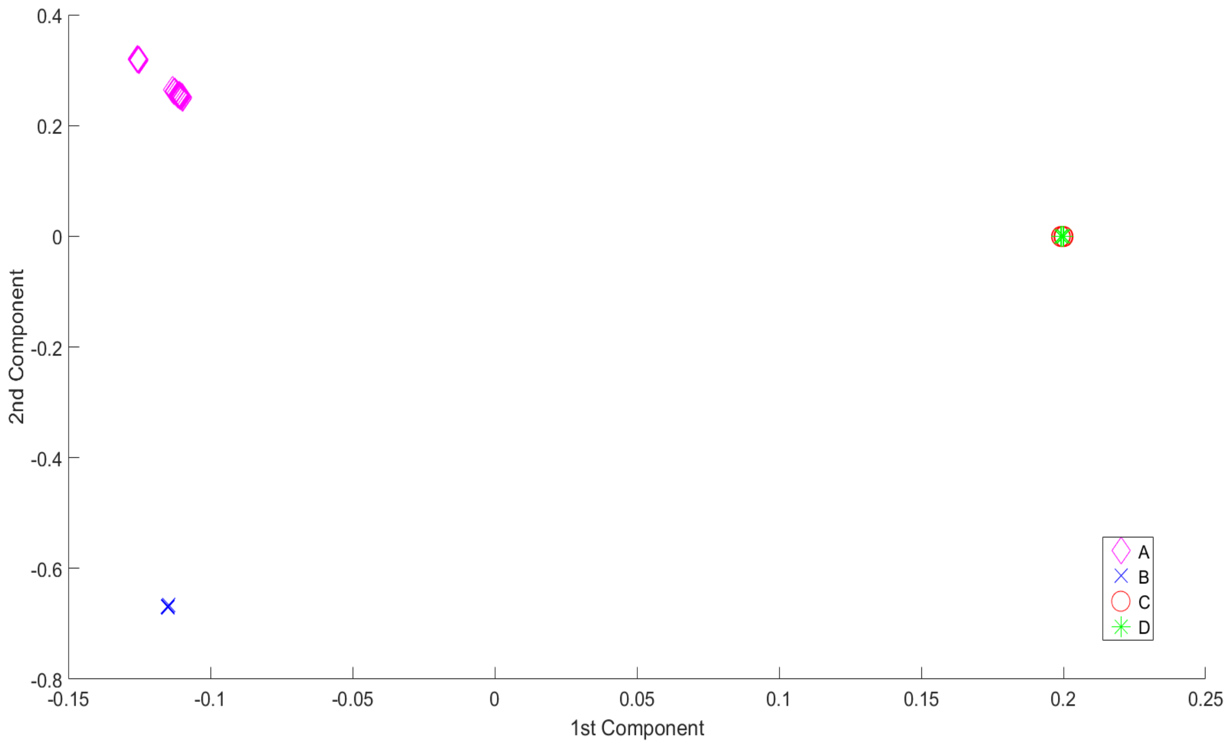


Figure 5.6: Kernel PCA for Kast Spectrum Kernel in I/O Traces using Byte Information (Cut Weight = 2).

5.3.2 Clustering with Strings Ignoring Byte Information

In the case of the strings that ignore the byte information, such clear separation of clusters is not achieved when using small cut weights. In order to obtain the same three clustering groups identified using the other string category, the weight value has to be increased considerably, e.g. 512 (see Figures 5.7 and 5.8).

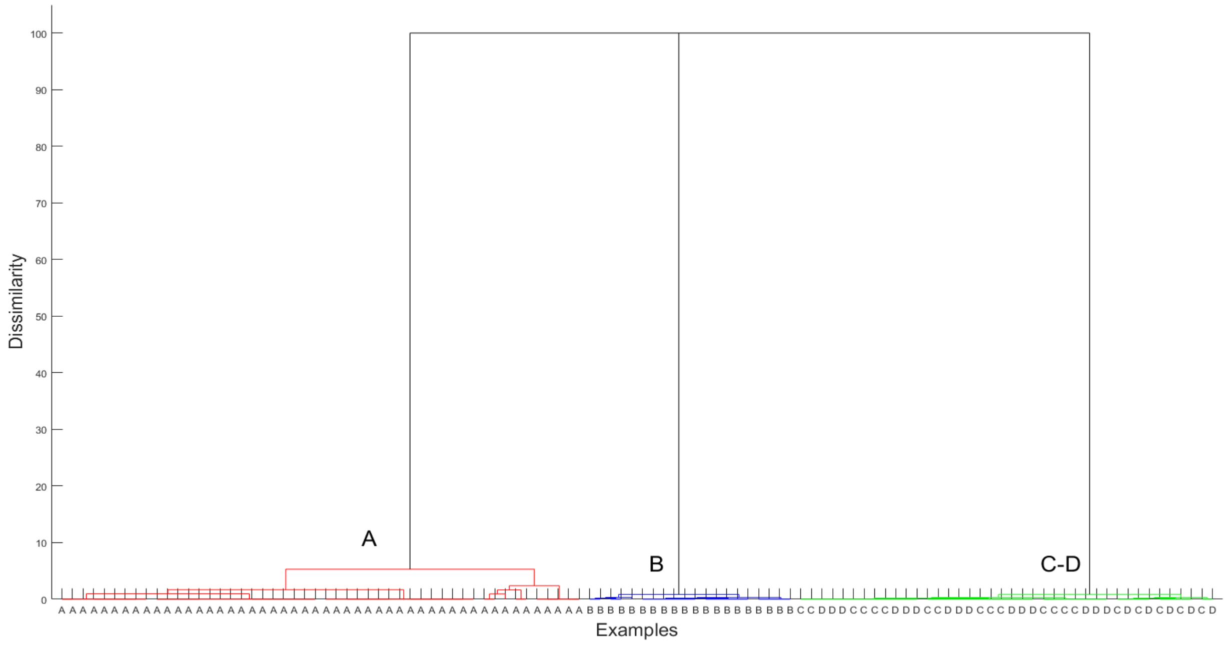


Figure 5.7: Hierarchical Clustering for Kast Spectrum Kernel in I/O Traces ignoring Byte Information (Cut Weight = 512).

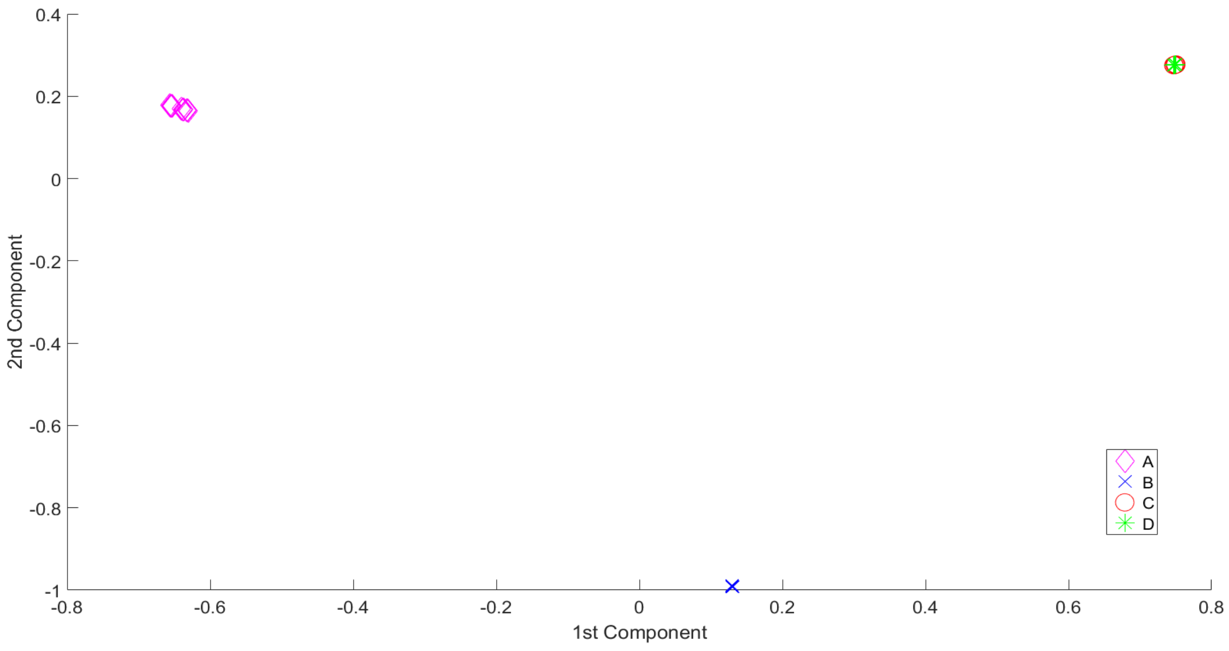


Figure 5.8: Kernel PCA for Kast Spectrum Kernel in I/O Traces ignoring Byte Information (Cut Weight = 512).

5.3.3 Remarks

On the one hand, the usage of higher cut weights is appropriate for finding general categories, with lower computational costs. On the other hand, the usage of lower cut weights is appropriate for discriminating better among examples, with higher computational costs. However, strings lacking the byte information yield good results with higher cut weight values only.

5.4 Kast1 Spectrum Kernel

The second of the developed kernels is the *kast1 spectrum kernel*. Experimentation shows that this kernel also performs better than the baseline kernel, due to the fact that it detects more patterns: *i)* Flash I/O, *ii)* POSIX I/O, and *iii)* Sequential I/O. In fact, it also outperforms the *kast spectrum kernel*, as with both categories of strings, the obtained clustering does not change significantly when the cut weight is modified. Recall that this kernel was designed as an improved version of the *kast spectrum kernel*, that reduces the noise introduced by small matches.

5.4.1 Clustering with Strings Using Byte Information

When this kernel is applied over strings that keep the byte information, the clustering results are similar to those ones from the previous kernel for small cut weight values, e.g. 2 (see Figures 5.9 and 5.10), as well as for higher values. This means that class A and class B are presented separated on their respective clusters, while classes C and D are mixed together in another cluster.

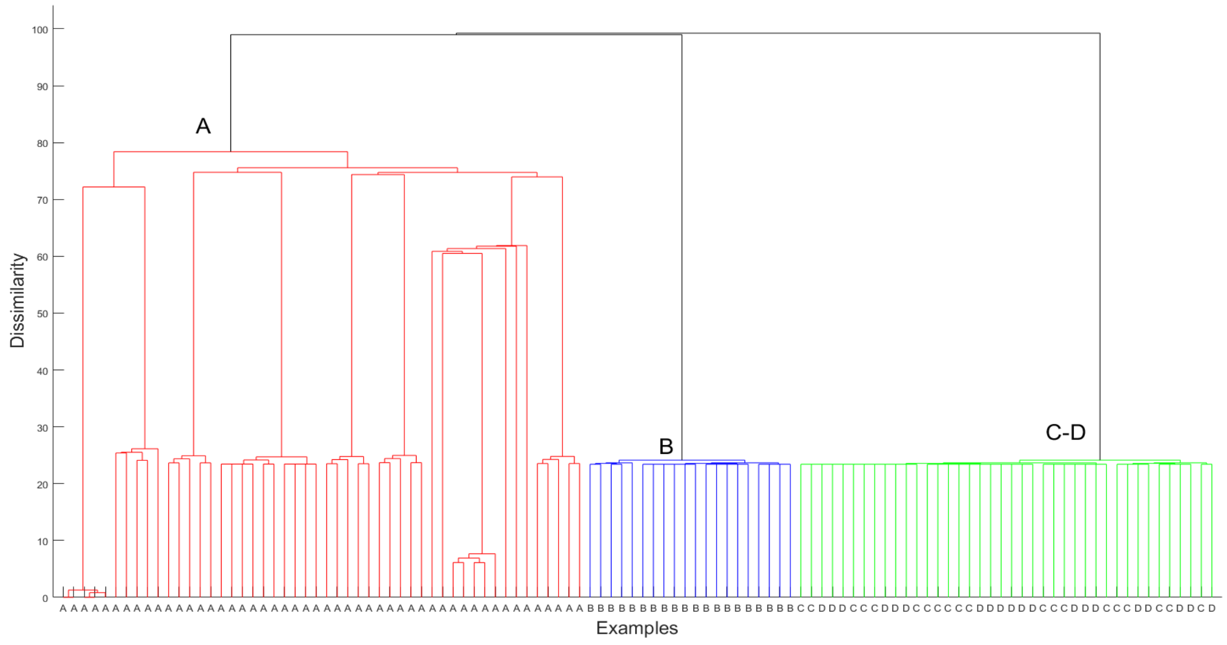


Figure 5.9: Hierarchical Clustering for Kast1 Spectrum Kernel in I/O Traces using Byte Information (Cut Weight = 2).

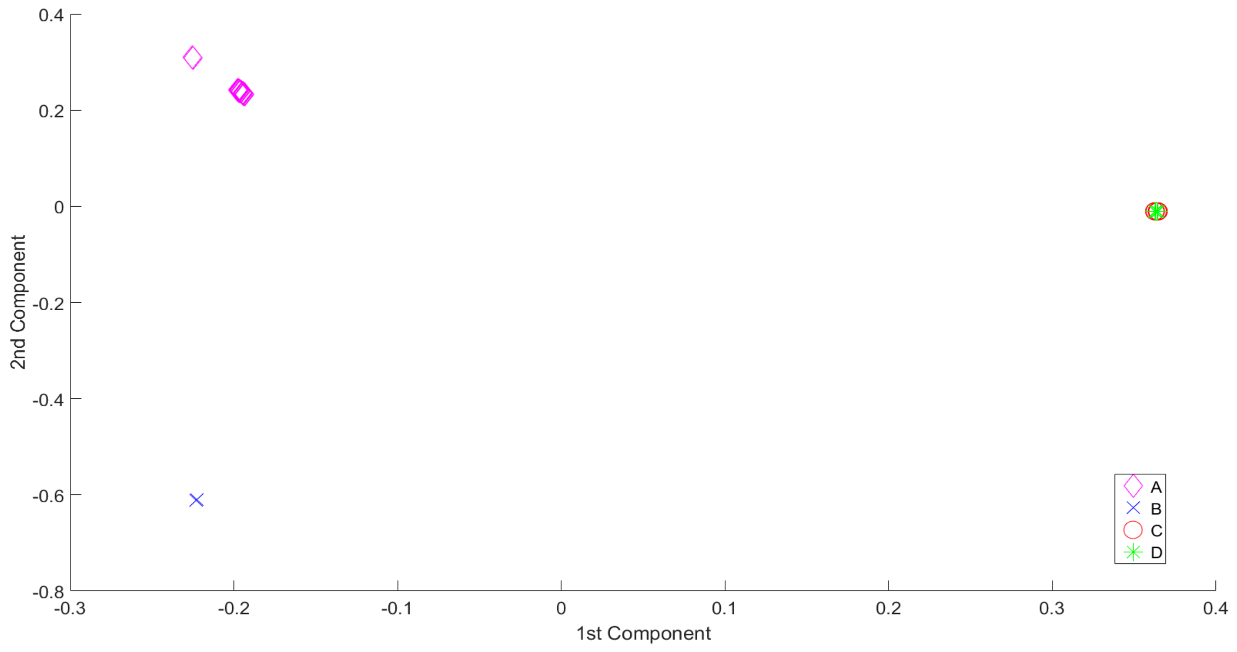


Figure 5.10: Kernel PCA for Kast1 Spectrum Kernel in I/O Traces using Byte Information (Cut Weight = 2).

5.4.2 Clustering with Strings Ignoring Byte Information

An advantage of this kernel, in contrast with the previous kernel, is that, when it is applied over strings that ignore the byte information, it performs also well with small cut weight values, e.g. 2 (see Figures 5.11 and 5.12). In other words, this kernel is less sensitive to the selection of the cut weight when this category of strings is used.

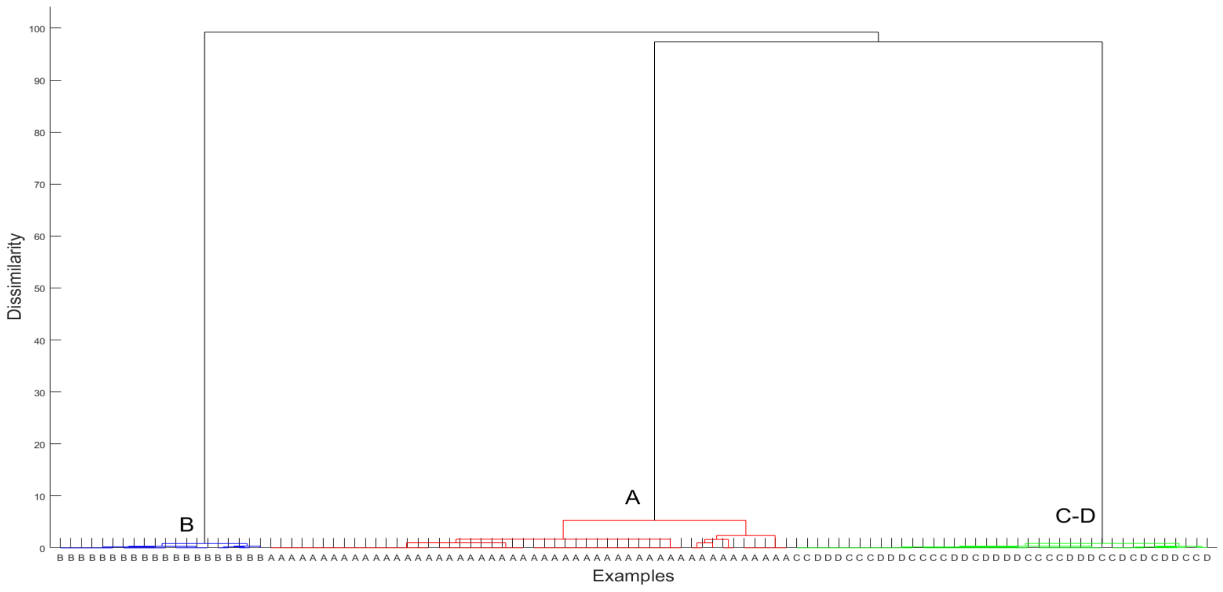


Figure 5.11: Hierarchical Clustering for Kast1 Spectrum Kernel in I/O Traces ignoring Byte Information (Cut Weight = 2).

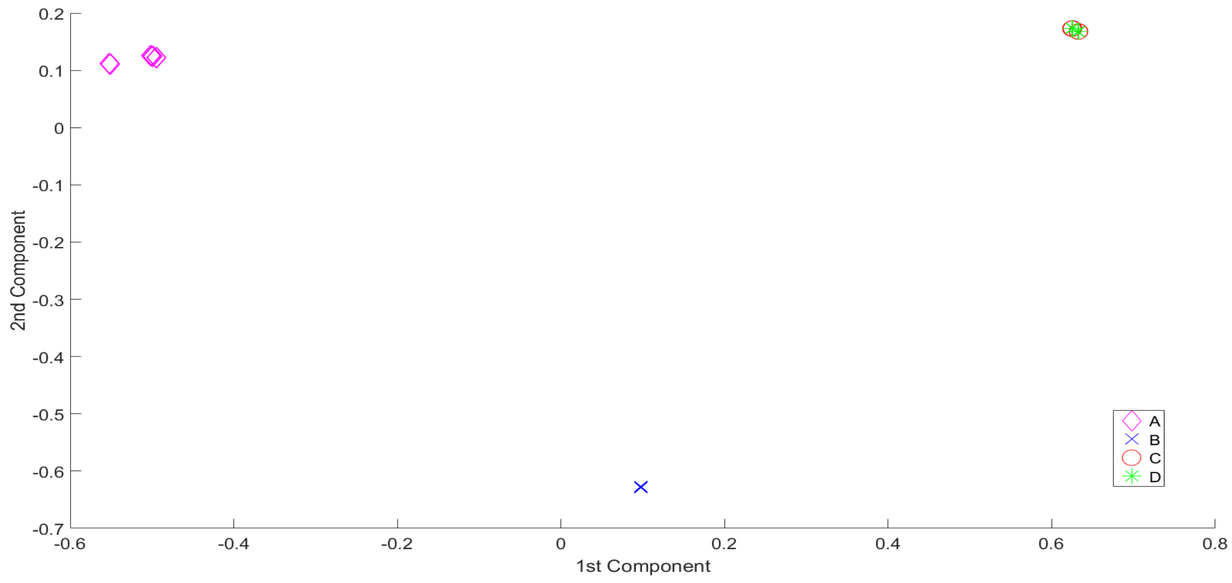


Figure 5.12: Kernel PCA for Kast1 Spectrum Kernel in I/O Traces ignoring Byte Information (Cut Weight = 2).

5.4.3 Remarks

Regardless of the cut weight or the string category used, Kernel PCA divides class A into two parts, but the distance among those parts is not comparable to distance among clusters. Recall that, with the baseline kernel, this does not hold. The clustering obtained with the *kast1 spectrum kernel* reflects more the nature of the I/O traces than the clustering thrown by the baseline kernel. Moreover, both string categories are less sensitive to the selection of the cut weight, which supposes also an improvement with respect to the *kast spectrum kernel*.

5.5 Kast2 Spectrum Kernel

The last kernel that has been tested is the *kast2 spectrum kernel*. The results are similar to the *kast1 spectrum kernel*, as it outperforms both the baseline kernel and the *kast spectrum kernel*. Here too, the selection of the cut weight does not have a significant effect on the clustering. However, the intra-cluster distances do not show remarkable changes with both small and high cut weight values. This stability is an advantage with respect to all other kernels analyzed. Recall that this kernel changes the way that weights are summarized and introduces a penalization value. This way it is possible to reflect more naturally the matching between two strings.

5.5.1 Clustering with Strings Using Byte Information

In contrast with the previous kernels, when strings that keep the byte information are utilized, the usage of small or high cut weight values does not show a significant difference in the inter-cluster and intra cluster distances. The usual clusters are found: *i)* Flash I/O, *ii)* POSIX I/O, and *iii)* Sequential I/O. Small cut weight values present a compact cluster formation (see Figures 5.13 and 5.14) High cut weight values with HC (see Figure 5.15) yield a similar scheme, while with Kernel PCA (see Figure 5.16), Class A is divided into two parts, but the distance among those two parts is negligible compared to the inter-cluster distances. This is a big advantage in comparison to the baseline kernel.

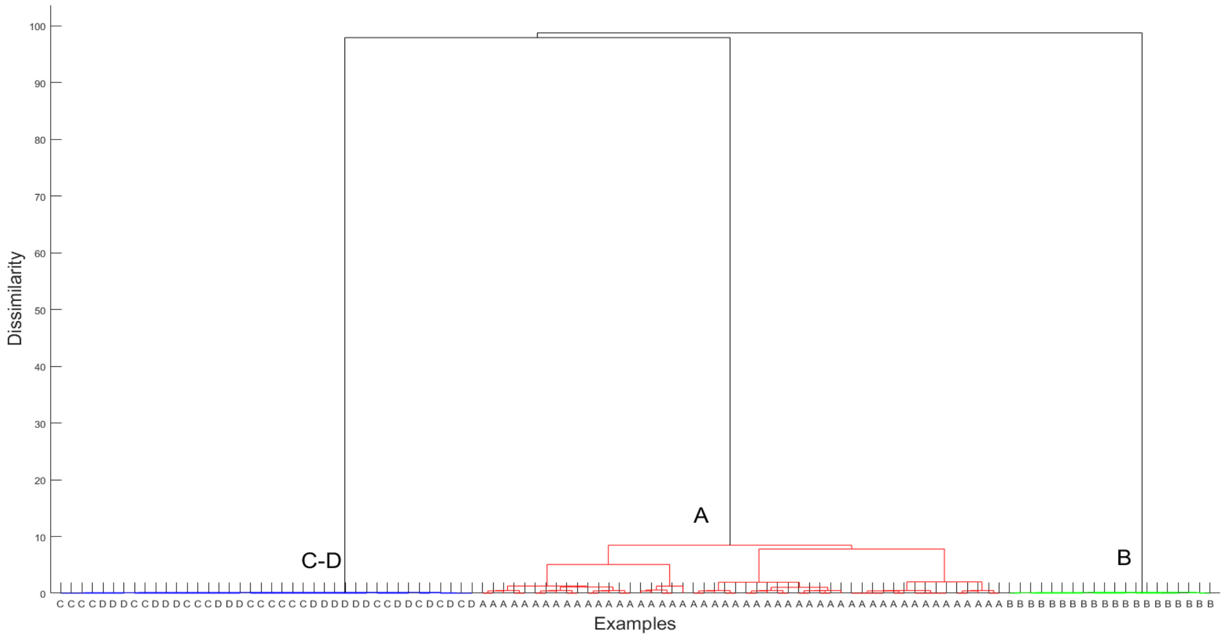


Figure 5.13: Hierarchical Clustering for Kast2 Spectrum Kernel in I/O Traces using Byte Information (Cut Weight = 2).

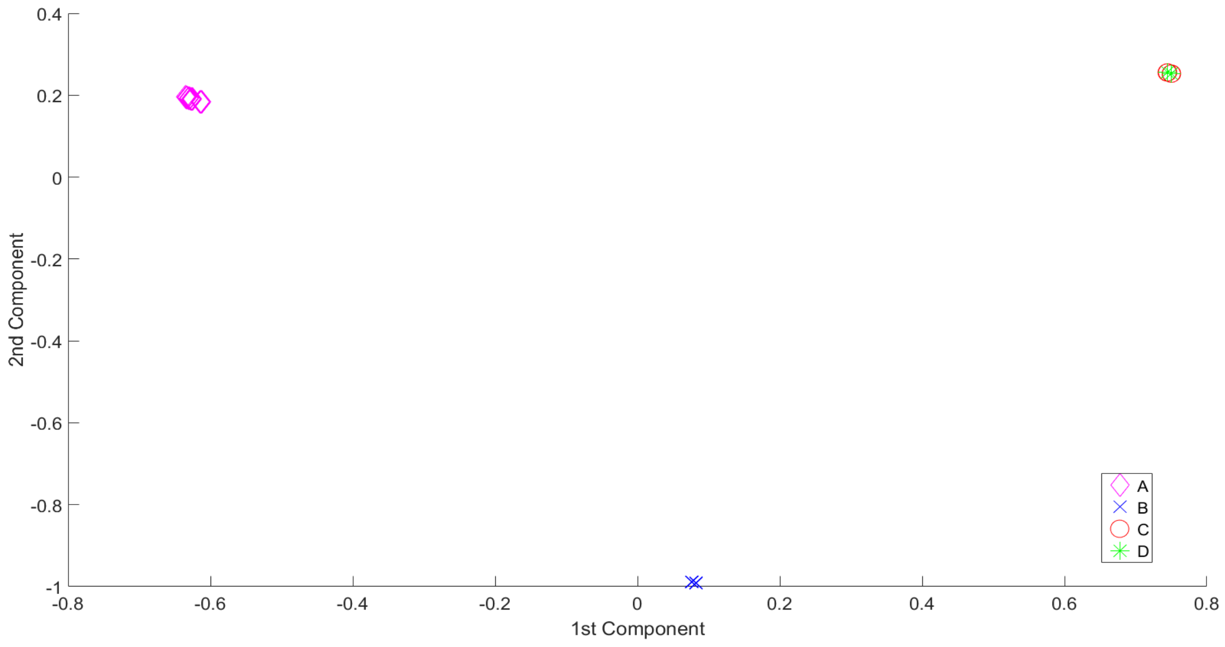


Figure 5.14: Kernel PCA for Kast2 Spectrum Kernel using byte information (Cut Weight = 2).

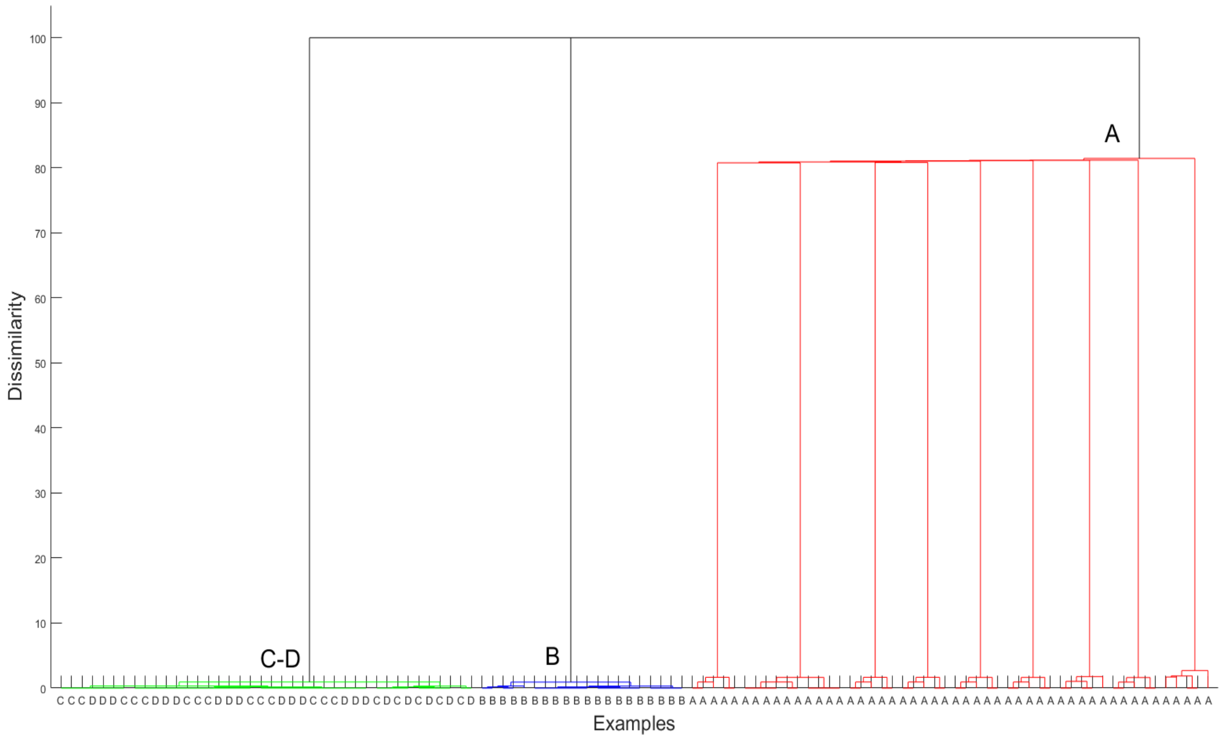


Figure 5.15: Hierarchical Clustering for Kast2 Spectrum Kernel in I/O Traces using Byte Information (Cut Weight = 512).

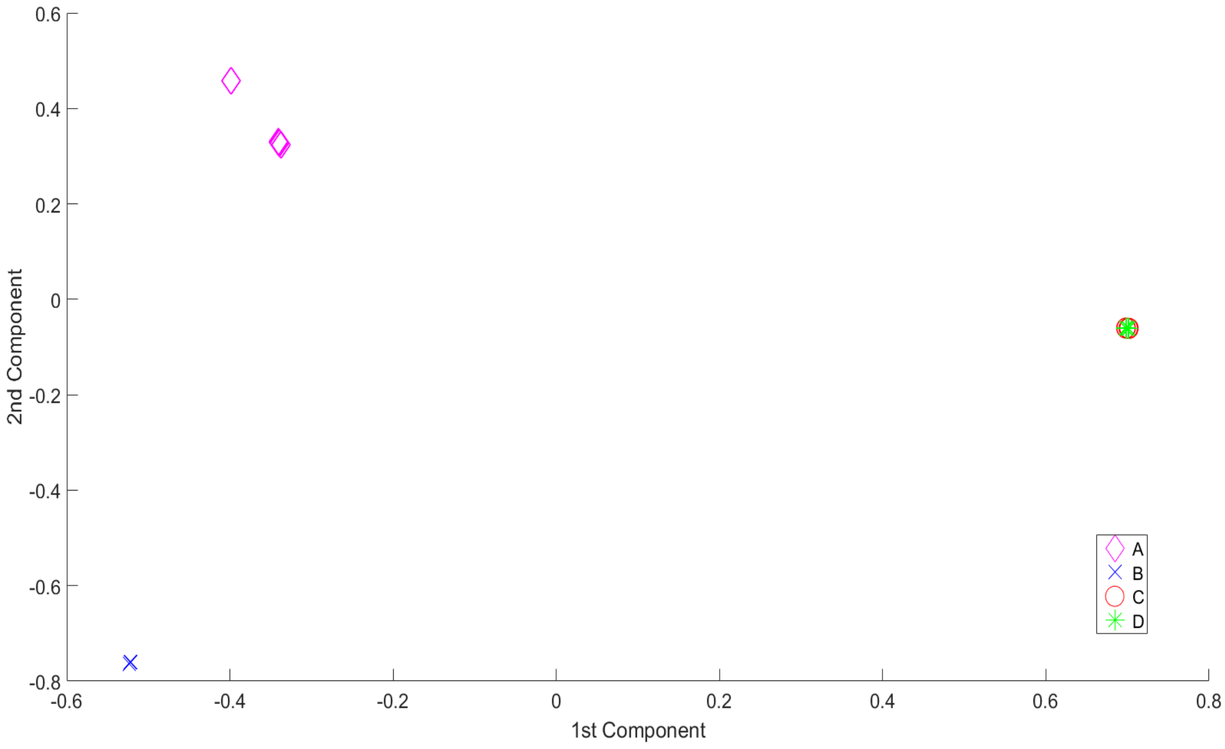


Figure 5.16: Kernel PCA for Kast2 Spectrum Kernel in I/O Traces using Byte Information (Cut Weight = 512).

5.5.2 Clustering with Strings Ignoring Byte Information

In the case of the strings that ignore the byte information, such clear separation of clusters is not so easily achieved when using small cut weights. However, if the cut weight value is increased, e.g. 512, the usual 3 groups are found (see Figures 5.17 and 5.18). The fact that the cut weight value infers on the clustering makes the parametrization a key point when using this string category.

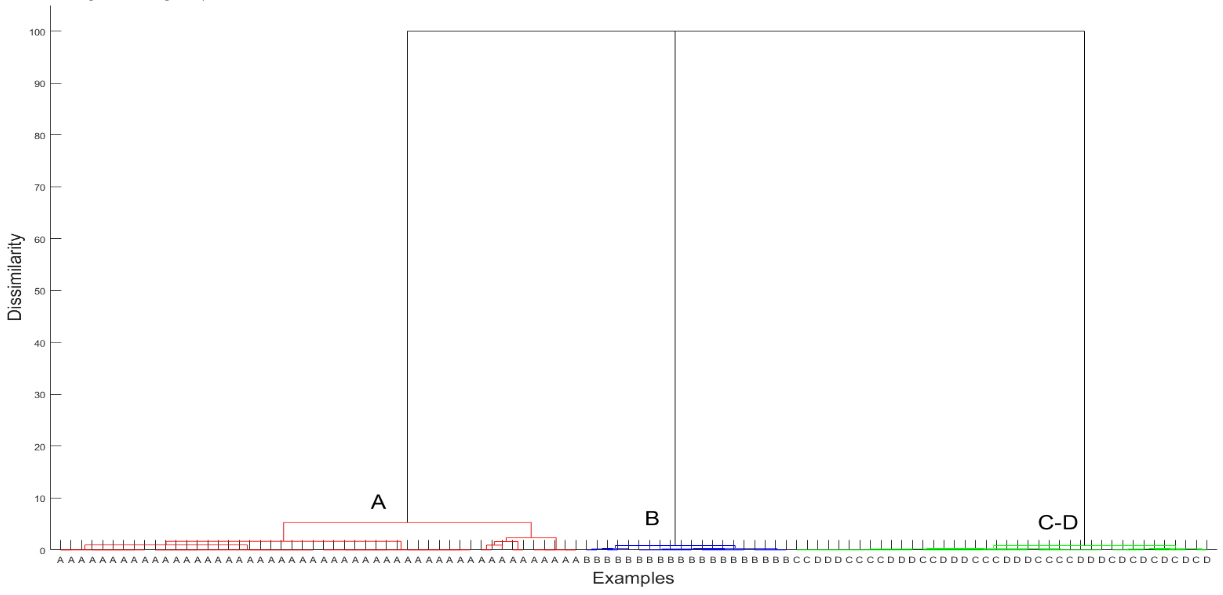


Figure 5.17: Hierarchical Clustering for Kast2 Spectrum Kernel in I/O Traces ignoring Byte Information (Cut Weight = 512).

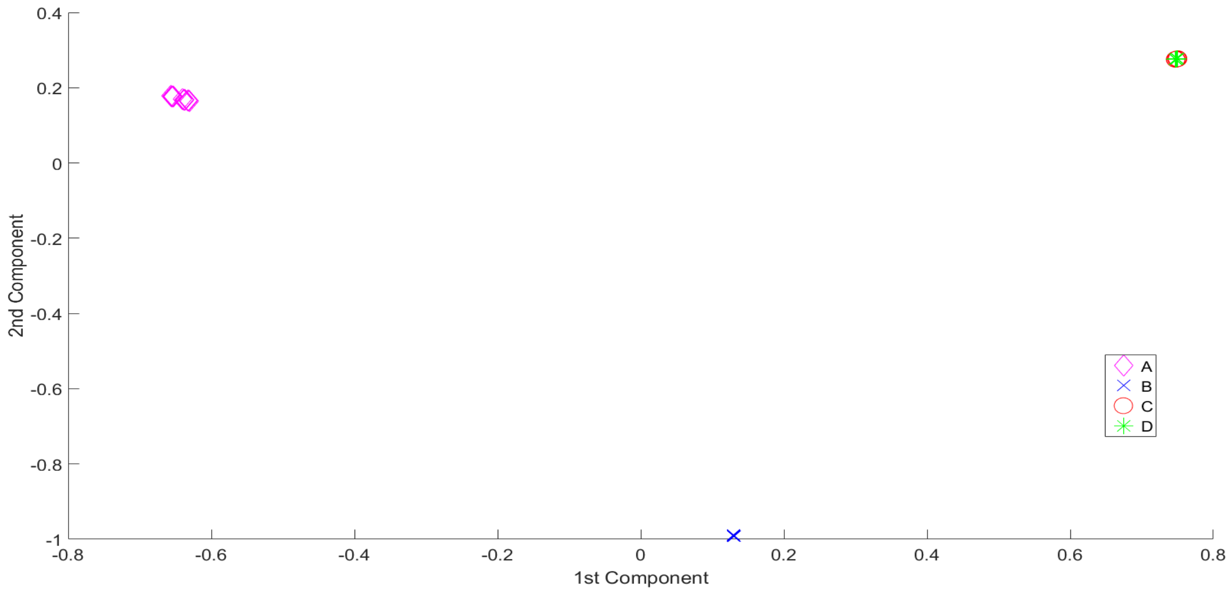


Figure 5.18: Kernel PCA for Kast2 Spectrum Kernel in I/O Traces ignoring Byte Information (Cut Weight = 512).

5.5.3 Remarks

It has been concluded that the usage of the *kast2 spectrum kernel* with strings that preserve the byte information of the I/O traces, is the best option to detect the patterns in the given examples, due to the fact that both inter-cluster and intra-cluster are not significantly affected by the selection of the cut weight value. This reduces the parametrization efforts from the user.

Summary

This chapter was dedicated to show the experimental results of the comparison of strings coming from I/O traces. The family of ***kastx*** *kernels performed better than the baseline kernel for the clustering of a given set of traces, as it was able to identified three different patterns, one more than the baseline kernel. From this family of kernels, the* ***kast2 spectrum kernel*** *was the one showing the best stability, as the selection of the cut weight value did not affect either the cluster formation nor the intra-cluster distances. The three kernels presented here were able to find clusters that reflected the similarity on the domain of I/O patterns. However, I/O pattern recognition is not the major domain where these kernels have been tested. Next chapter shows the results of a similar experiment with the Intermediate Representations of the LLVM Compiler.*

6 Comparison of Intermediate Representations

*In this chapter, we extend the application of the family of **kastx kernels** to the analysis of the Intermediate Representations of a popular compiler. The experiment has been designed to see how well these kernels perform at separating four different classes of functions written in C language, using separately their Clang ASTs and LLVM IRs as the source of information. In Section 6.1, we explain the configuration of the experiment. Section 6.2 is dedicated to describe the findings with the baseline kernel, namely the **blended spectrum kernel**. Sections 6.3, 6.4 and 6.5 are dedicated to each one of the string kernels which are the main contribution of this thesis.*

6.1 Experiment Configuration

In the previous chapter, we showed how the family of **kastx kernels** have outperformed the baseline kernel on the problem of detecting patterns on a set of I/O traces. The major domain where this new family of kernels has been tested is the comparison of the intermediate representations of a compiler. For doing that, we have created a collection of code pieces that manifest certain relations among each other. From a broad perspective, the collection has been organized in four classes of functions. From a closer perspective, each code piece has five different implementations (clones). The idea behind these two approaches is to identify how well the analyzed kernels perform at separating general classes and what is the role of the clone types in the conformation of them. For this experiment, 20 functions have been written manually. They are divided in four broad classes:

- **Class A: Matching Functions.** They are string kernels of the literature, but the name was changed to avoid confusion when referring to the kernels used in this study. These functions can be divided in two groups:
  - *Group I: Size-based Matching.* These functions search for matching substrings of a given size, regardless of the presence of separators and words.
    - K-spectrum: it tries to find matching substrings of size k.
    - Blended spectrum: it searches for matching substrings of size k or less.
    - Bag-of-characters: it aims to find matching substrings of size 1.
  - *Group II: Delimiter-based Matching.* The matching is restricted to substrings enclosed between delimiters, regardless of the size.
    - bag-of-words functions: the delimiter can be a blank space.
    - bag-of-sentences functions: it is similar to the latter but with two delimiters, one for the opening, one for the closing.

- **Class B: Sort Functions.** As the name suggests it, they are a popular set of functions used for sorting numbers:
  - Bubble sort.
  - Insert sort.
  - Selection sort.
  - Heap sort.
  - Merge sort.
- **Class C: 3D Stencils.** Stencils are operations performed on a structured grid, where the value of a cell is calculated using the values of the surrounding cells. In the stencils here used, the initial value of the cell itself is always taken into account and the selected operation is simply the summation of all values.
  - Compact stencil: Compact stencils take into account the values all the neighboring cells.
  - Side stencil: this one only takes into account neighboring cells sharing the same position in two axes.
  - Edge stencil: this stencil takes into account neighboring cells sharing the same position in only one axis.
  - Vertex stencil: it only takes into account neighboring cells on a diagonal position.
  - Non-compact stencil 1 additional layer: Non-compact stencils go a few layers further the neighboring cells.
- **Class D: 2D Stencils.** Similar to the previous class, with the number of dimension reduced by one. Obviously, there is no room for a Side stencil.
  - Compact stencil.
  - Edge stencil.
  - Vertex stencil.
  - Non-compact stencil 1 additional layer.
  - Non-compact stencil 2 additional layers.

It is also the interest of this thesis to study the structure of clone types inside each function class; for that reason, each of them has been implemented in five different variants with the same functionality:

- Original version: written from scratch.
- *Type-1* Clone: It is created automatically from the original version. It presents changes in the number of spaces, break lines and comments.
- *Type-2* Clone: It is also generated automatically from the original version. It shows variable renaming and changes in data types.
- *Type-3* Clone: In order to create it, the *Type-2* Clone is manually restructured in some lines of code (insertions, deletions and modifications) without affecting the result.

- *Type-4* Clone: It is manually created and is a different implementation that delivers the same result.

This results then in a set of 100 examples for the study, whose size ranges from 32 to 124 lines of code[1]. We have leveraged the Clang AST of each code sample and converted it into string. Same has been done with the LLVM IR. Several similarity matrices have been obtained by running the baseline kernel function over those strings. The same has been done with the kernel functions proposed on this work. We have tested the following cut weight range: $\{2^0, 2^2, ..., 2^n\}$ for $n = 9$. Note that when the computed matrices present negative eigenvalues, these eigenvalues have been replaced by zeros and the matrices have been recalculated using the new eigenvalues. All the similarity matrices have been analyzed with both Kernel Principal Component Analysis (Kernel PCA) and Hierarchical Clustering (HC), the latter using the simple linkage method.

6.2 Baseline Kernel: Blended Spectrum Kernel

We start with the analysis of the *blended spectrum kernel*, whose results are the basis for the comparison with the family of **kastx** kernels. For the baseline kernel, a basic scheme of clusters has been achieved: *i)* matching functions, *ii)* sort functions and *iii)* stencils. The experiment also shows clearly the following ranking in distances for clone types with respect to the original version: *i) Type-1*, *ii) Type-2*, *iii) Type-3* and *iv) Type-4* clones. *Type-1* clones are almost overlapped with the original versions. This ranking reflects the theoretical definition of clone types. However, *Type-4* clones have the tendency to break the class clustering. Let us see how is the clustering for each intermediate representation.

6.2.1 Clustering with Clang Abstract Syntax Trees

When the baseline kernel is used in conjunction with the Clang ASTs, this kernel shows only a good performance with the hierarchical clustering (HC) technique, as Kernel PCA fails to create meaningful clusters. This behavior is seen not only in the broad class separation but also in the clone type separation.

Clustering by Function Class

The analysis of the complete collection of examples shows that the best results with this kernel are obtained when using a cut weight of 16. HC separates the examples in 3 clusters (see Figure 6.1): *i)* matching functions, *ii)* sort functions and *iii)* stencils. However, three code pieces (*Type-4* clones of sort functions a.k.a class B), are misplaced in the cluster of matching functions (class A). Unfortunately, Kernel PCA presents an unclear clustering, with class A being the only one separated from the rest.

[1]Available under https://git.wr.informatik.uni-hamburg.de/raul.torres/kast_test_functions.

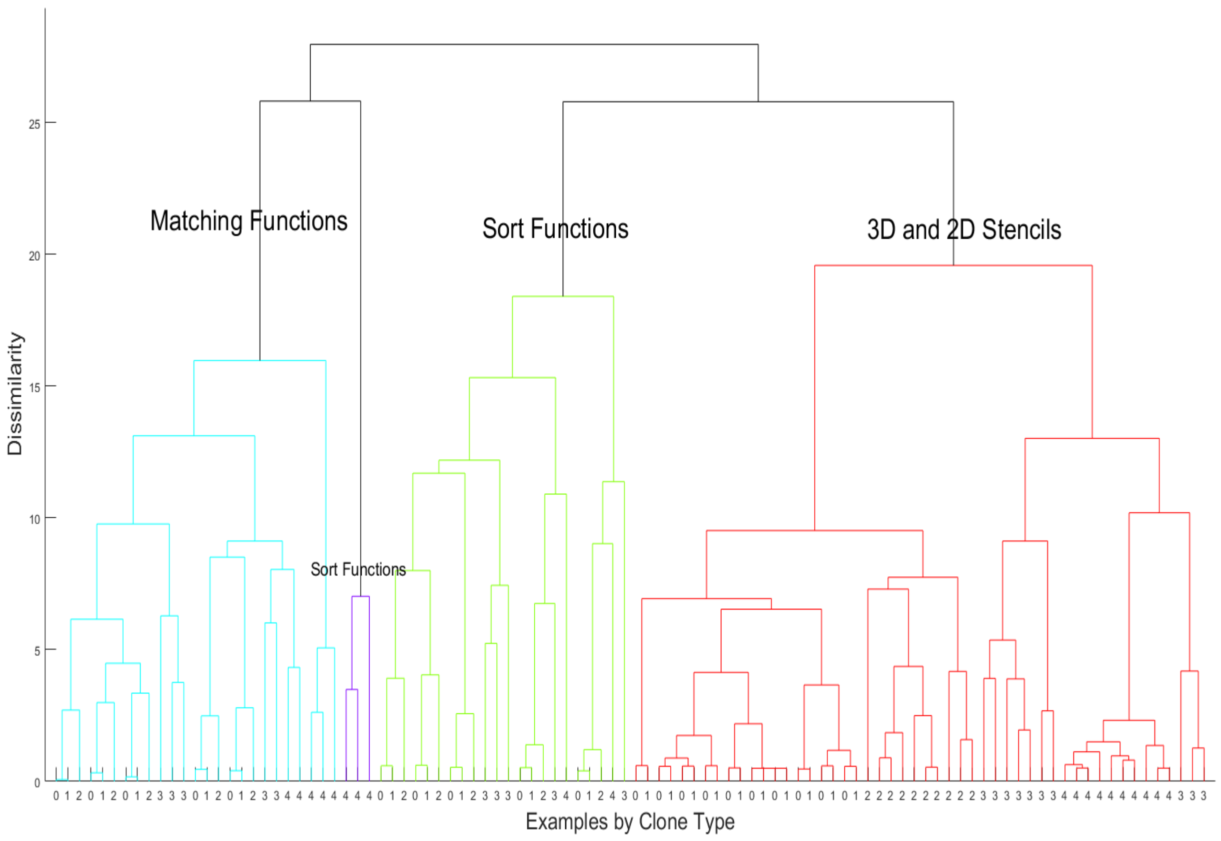


Figure 6.1: Hierarchical Clustering for Blended Spectrum Kernel using ASTs (Cut Weight = 16).

Clustering by Clone Type

Let us see now the clustering inside each function class and how clone types are organized around the original version. With Clang ASTs, it is possible to detect patterns inside all function classes.

Class A (Matching Functions): HC and Kernel PCA show a similar cluster formation inside class A (see Figures 6.2 and 6.3). Notice that group I (size-based matching functions) and Group II (delimiter-based matching functions) are clearly separated. For all functions, *Type-1* and *Type-2* clones are found very close to its corresponding original version, conforming a small cluster, while *Type-3* and *Type-4* clones are located at a larger distance from it. It is noticeable that there is not a clear tendency of clones of the same type to conform clusters.

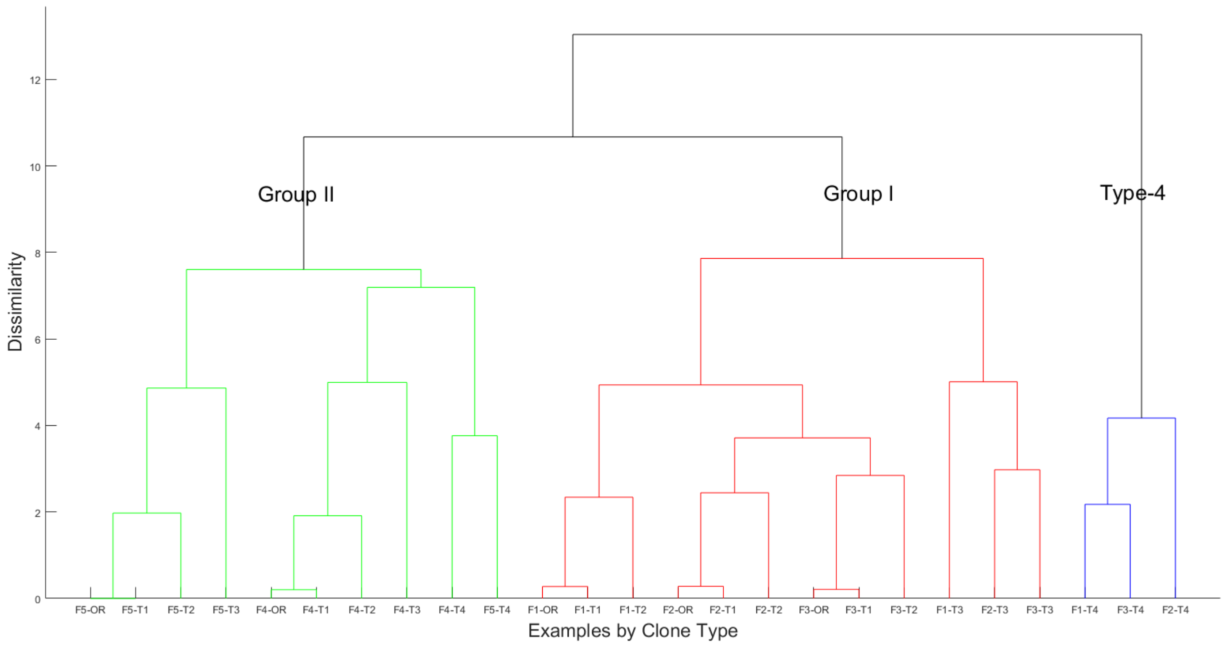


Figure 6.2: Hierarchical Clustering for Blended Spectrum Kernel using ASTs of Matching Functions (Cut Weight = 16).

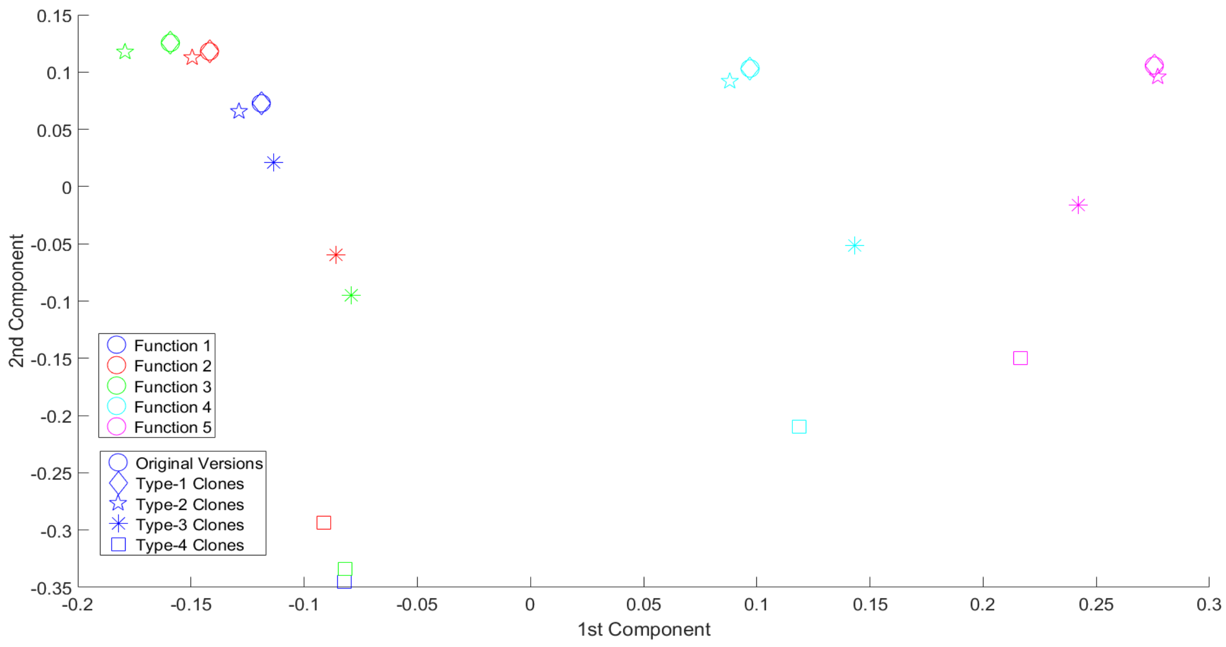


Figure 6.3: KPCA for Blended Spectrum Kernel using ASTs of Matching Functions (Cut Weight = 16).

Class B (Sort Functions): With this class, only HC shows conclusive results (see Figure 6.4). Merge sort and heap sort are separated completely in single branches. Additionally, the distance ranking to the original version is the same as in the previous class. In this class also, clones do not tend to conform clusters.

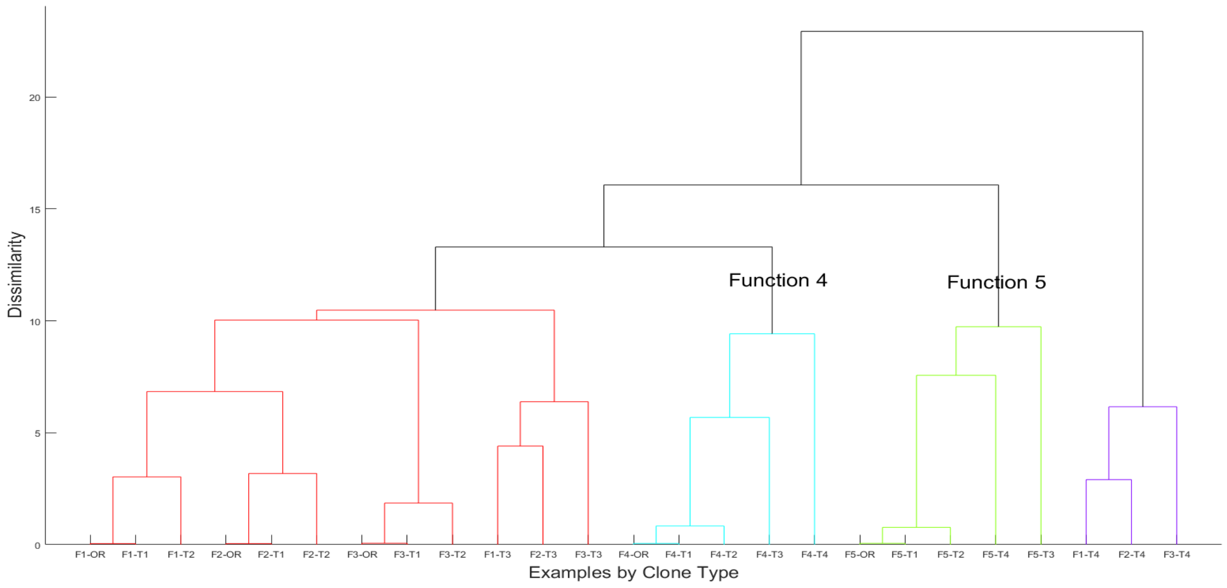


Figure 6.4: Hierarchical Clustering for Blended Spectrum Kernel using ASTs of Sort Functions (Cut Weight = 16).

Class C (3D Stencils): HC and Kernel PCA show a similar behavior at clustering class C (see Figures 6.5 and 6.6). The clustering, however, shows a different behavior to the previous classes, as clones of the same type tend to be grouped together. In comparison with the previous classes, the original functions of this class are more similar among each other; due to this fact, they tend to conform a cluster. Both clustering algorithms conglomerate all the original functions and *Type-1* clones in a single cluster. Close to that first cluster, another one is conformed by all *Type-2* clones. The most distanced cluster is conformed by all *Type-4* clones, while *Type-3* clones fail to be clustered altogether.

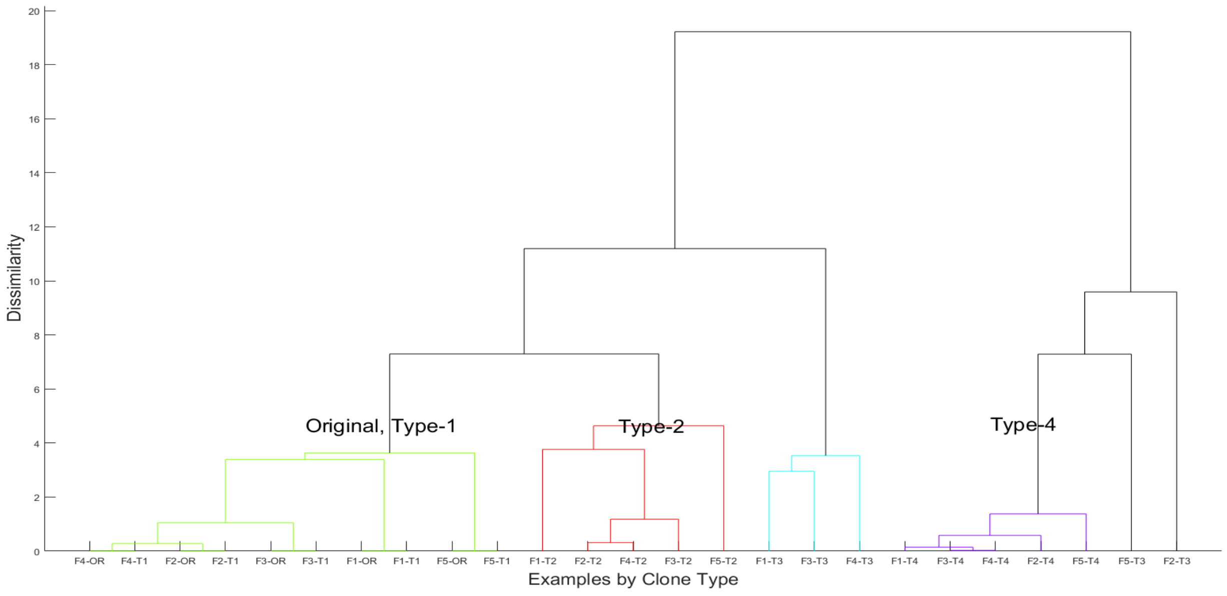


Figure 6.5: Hierarchical Clustering for Blended Spectrum Kernel using ASTs of 3D Stencils (Cut Weight = 16).

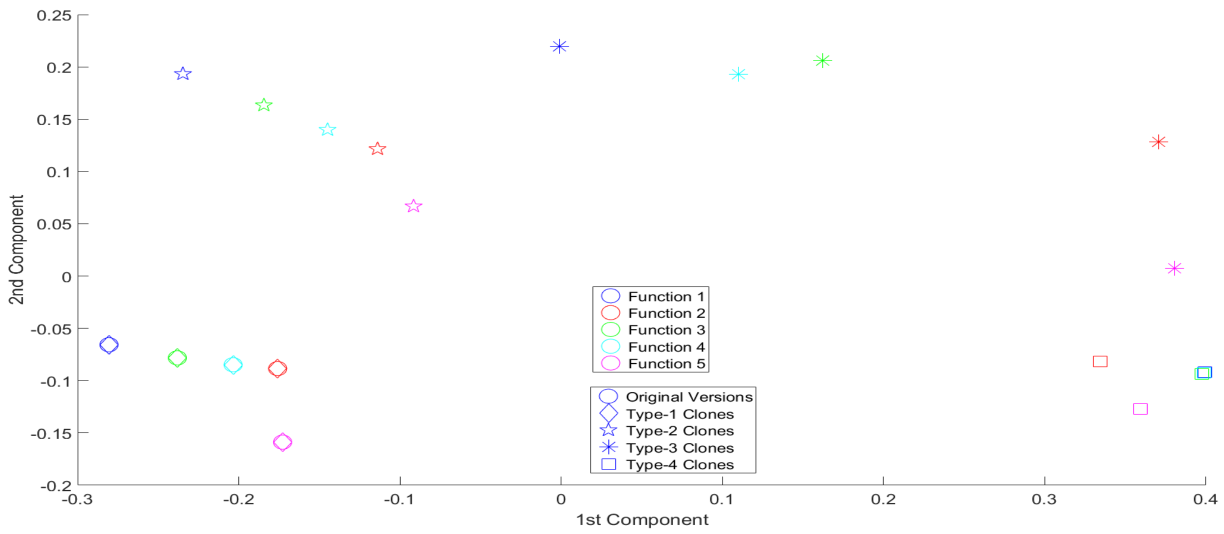


Figure 6.6: KPCA for Blended Spectrum Kernel using ASTs of 3D Stencils (Cut Weight = 16).

Class D (2D Stencils): With this class, only HC shows conclusive results (see Figure 6.7), with a similar behavior presented by class C (3D stencils).

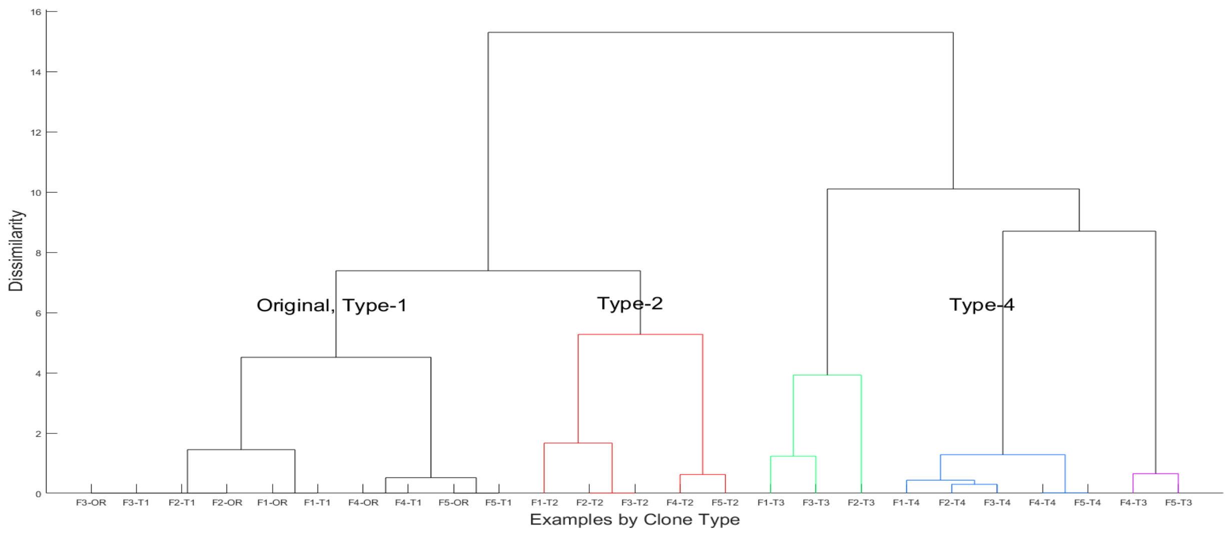


Figure 6.7: Hierarchical Clustering for Blended Spectrum Kernel using ASTs of 2D Stencils (Cut Weight = 16).

The results of applying the baseline kernel to Clang ASTs have been presented. The following section details the results obtained with the LLVM intermediate representations.

6.2.2 Clustering with LLVM Intermediate Representations

Let us change now the intermediate representation. When the baseline kernel is used in conjunctions with LLVM IRs instead, the results of hierarchical clustering and Kernel PCA are very similar. Let us see that in detail.

Clustering by Function Class

In the broad test, the best clustering is achieved when using a cut weight of 64 (see Figures 6.8 and 6.9). In comparison with the Clang ASTs, LLVM IRs allow a partial distinction between 3D and 2D stencils, hence improving the clustering results. However, *Type-4* clones keep being the most differentiated structures, showing a tendency to create clusters that break the class separation.

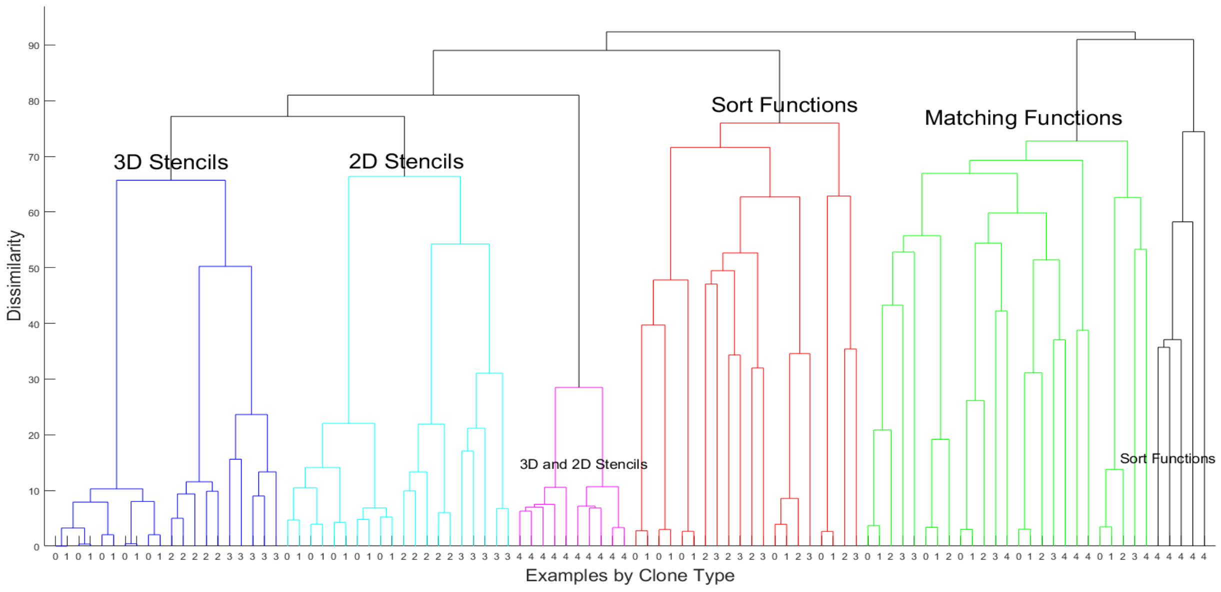


Figure 6.8: Hierarchical Clustering for Blended Spectrum Kernel using IRs (Cut Weight = 64).

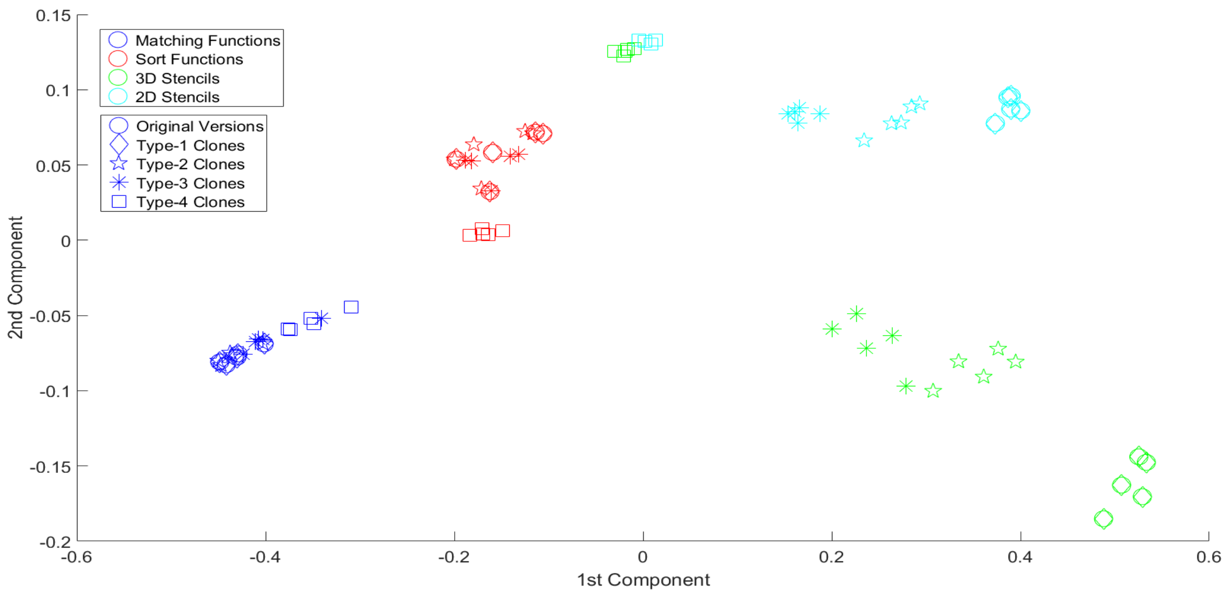


Figure 6.9: KPCA for Blended Spectrum Kernel using IRs (Cut Weight = 64).

Clustering by Clone Type

Let us look inside the separation inside each function class. With LLVM IRs, the detection of meaningful patterns inside classes is not always possible.

Class A (Matching Functions): The clustering results are not conclusive. Both Kernel PCA and HC show a separation between group I and group II. However, group I does not conform a compact cluster and all its members seem to be sparse.

Class B (Sort Functions): In this class, the clustering results are not conclusive either. The only clear pattern inside this group is the formation of a cluster with all *Type-4* clones.

Class C (3D Stencils): Both clustering algorithms yield a similar pattern detection performance in class C. The patterns favor the formation of clusters of clone types rather than clusters of functions, same like when using Clang ASTs. However, with Clang ASTs, *Type-3* clones do not present a compact formation. This changed with LLVM IRs, as with HC they do conform a single branch (see Figure 6.10), and with Kernel PCA they present a solid cluster formation together with *Type-2* clones (see Figure 6.11).

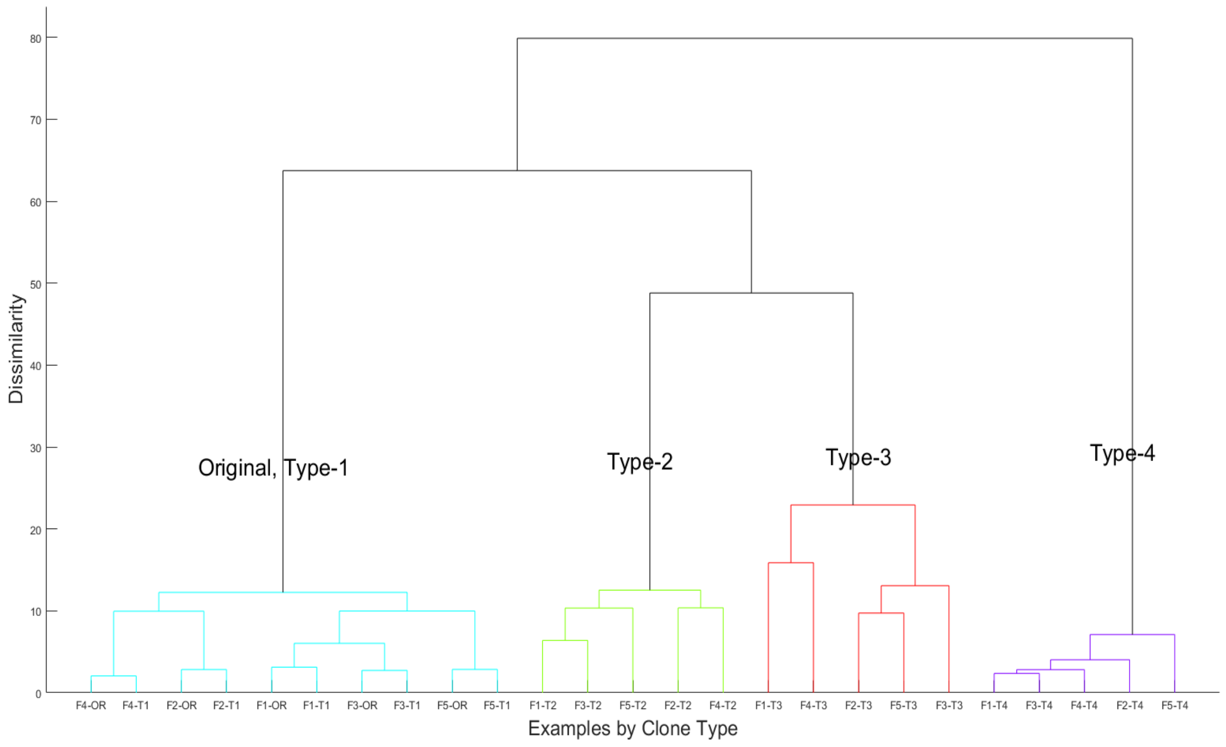


Figure 6.10: Hierarchical Clustering for Blended Spectrum Kernel using IRs of 3D Stencils (Cut Weight = 64).

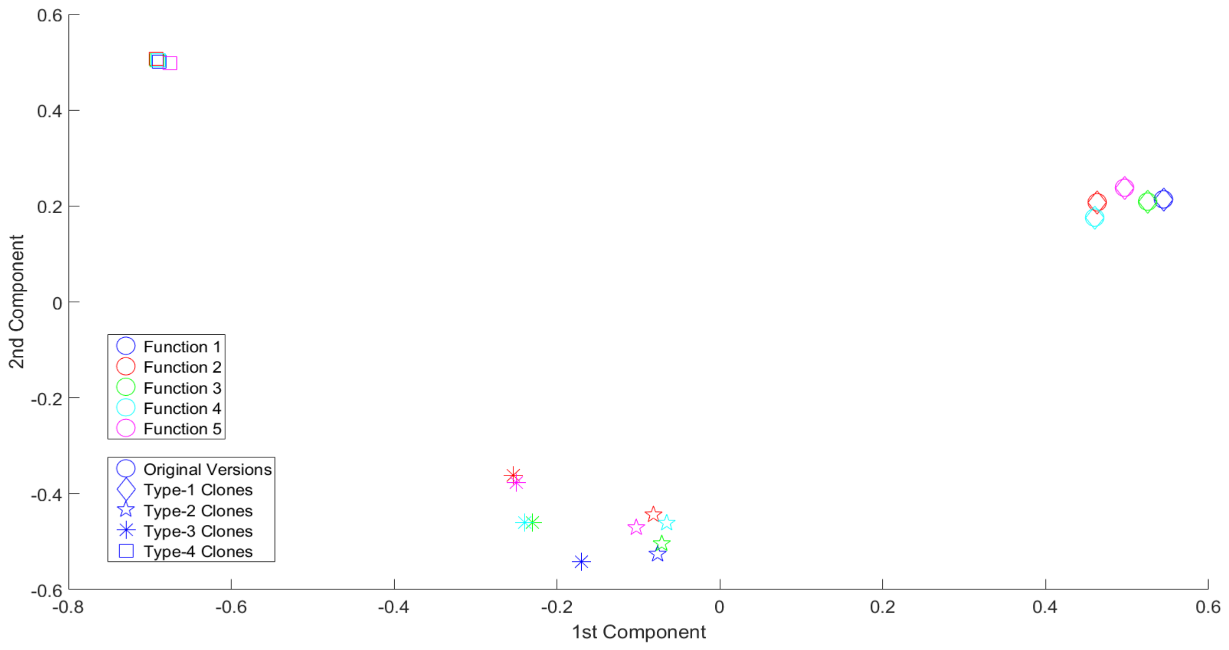


Figure 6.11: KPCA for Blended Spectrum Kernel using IRs of 3D Stencils (Cut Weight = 64).

Class D (2D Stencils): With this class, the results are similar to those obtained with 3D stencils (see Figures 6.12 and 6.13). The major difference is that *Type-3* clones are clustered all together and separated from *Type-2* clones.

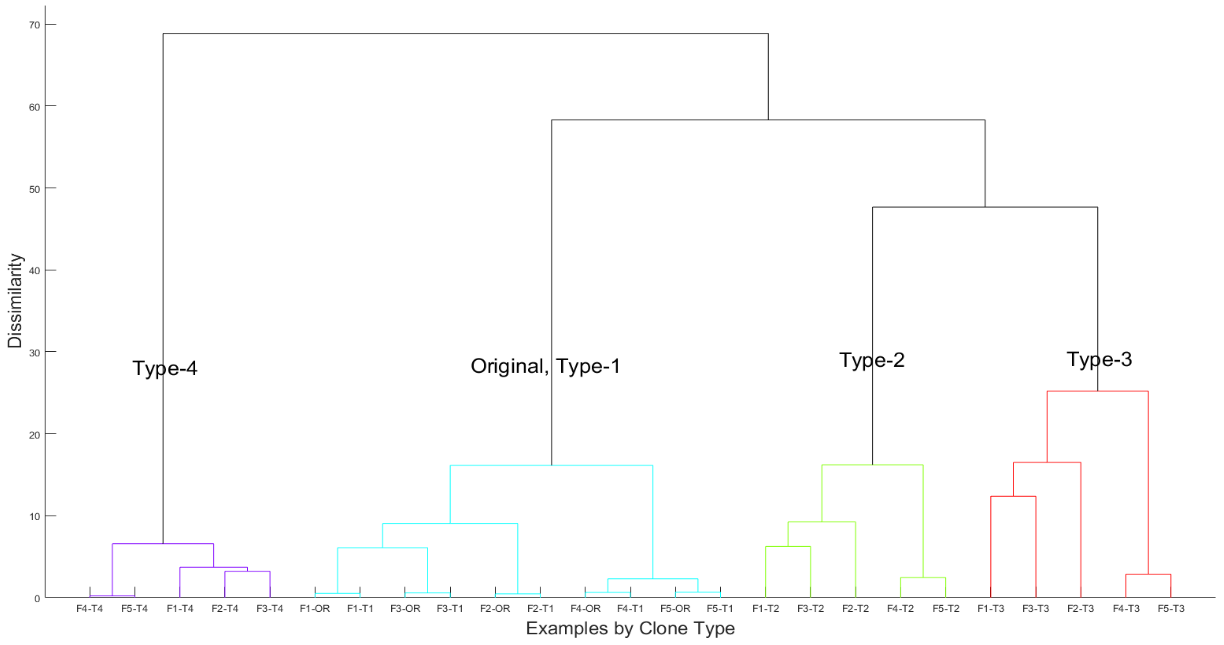


Figure 6.12: Hierarchical Clustering for Blended Spectrum Kernel using IRs of 2D Stencils (Cut Weight = 64).

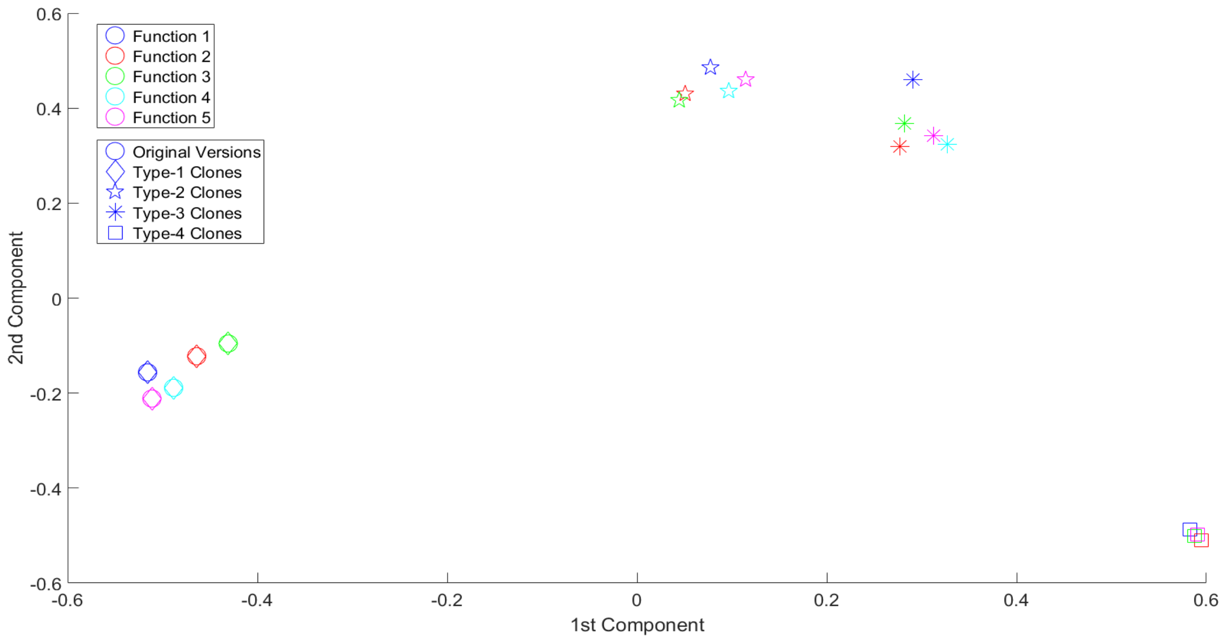


Figure 6.13: KPCA for Blended Spectrum Kernel using IRs of 2D Stencils (Cut Weight = 64).

6.2.3 Remarks

For the baseline kernel, the use of LLVM IRs yields a clearer class separation than the use of Clang ASTs. However, for the detection of patterns inside classes, Clang ASTs are more effective. Experimentation also shows that Hierarchical Clustering is more effective on detecting patterns than Kernel PCA. This kernel is sensitive to the strong differences in the implementation of *Type-4* clones; this is reflected on the fact that several of these clones are found in an incorrect class or conforming a separate cluster.

6.3 Kast Spectrum Kernel

Let us review the results with the first kernel of the **kastx** family: the *kast spectrum kernel*. The following conclusions learned from experimentation show that there is no significant advantage on the usage of this kernel over the baseline kernel:

- *Similarities with the baseline kernel:* For the *kast spectrum kernel*, the same basic scheme of clusters is achieved, though hierarchical clustering is more effective on the task than Kernel PCA.

- *Differences with the baseline kernel:* Contrary to the baseline kernel, the use of Clang ASTs yields a clearer class separation than the use of LLVM IRs. However, the intra-cluster distances are larger than those of the baseline kernel.

- *Disadvantages with respect to the baseline kernel:* While the *kast spectrum kernel* only shows the expected ranking in distances for clone types when using Clang ASTs, the baseline kernel shows this behavior with both representations.

- *Advantages with respect to the baseline kernel:* If the *kast spectrum kernel* is applied over Clang ASTs, no misplaced examples are found. In this case, the strong semantical differences of *Type-4* clones do not have a significant effect on the general clustering scheme.

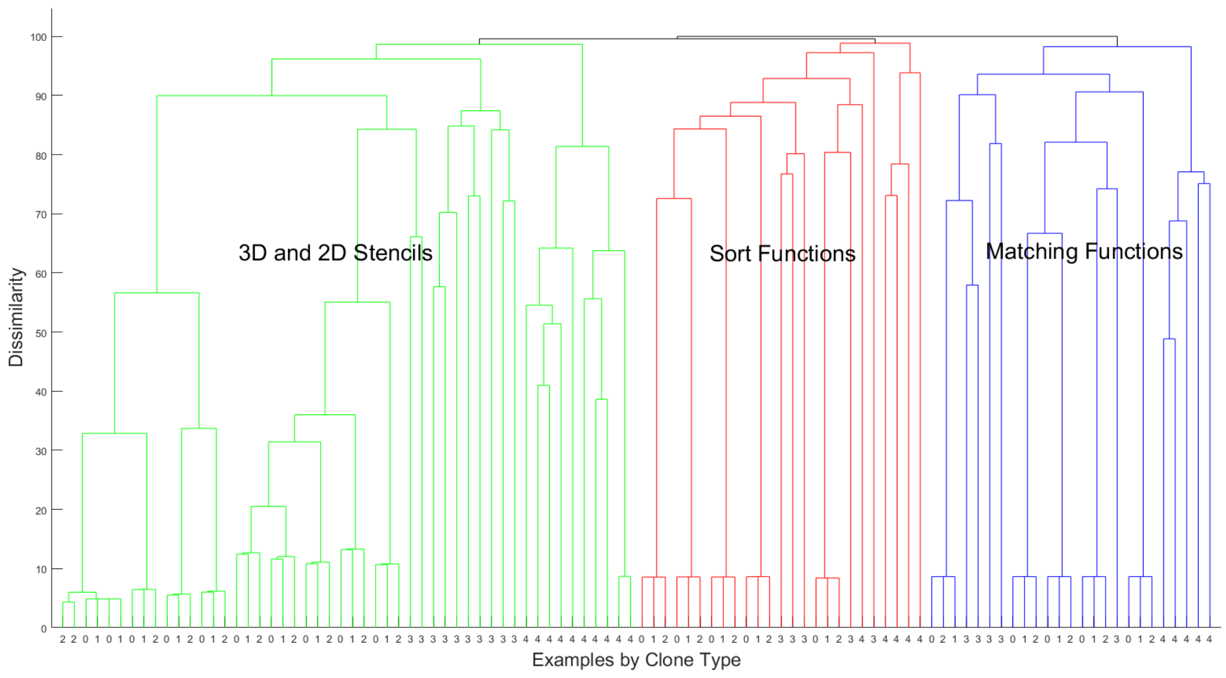
6.3.1 Clustering with Clang Abstract Syntax Trees

If the *kast spectrum kernel* is applied over Clang ASTs, the ranking in distances to the original version correspond to the expectation according to the definition of clone types: *Type-1* and *Type-2* clones are found almost overlapped to the original version, while *Type-3* and *Type-4* clones are situated at further distances.

Clustering by Function Class

In the problem of detecting general classes, only the use of Hierarchical Clustering shows that this kernel is able to separate the usual 3 clusters without misplaced examples (see Figure 6.14). One difference with the baseline kernel is that the intra-cluster distances are larger. Unfortunately, the results of Kernel PCA are not conclusive, as there is no clear formation of clusters.

Figure 6.14: Hierarchical Clustering for Kast Spectrum Kernel using ASTs (Cut Weight = 64).



Clustering by Clone Type

The analysis of each function class does not show a remarkable improvement with respect to the baseline kernel. Class A and class B, for example, do not show additional patterns.

Class C (3D Stencils): The clustering achieved with Kernel PCA is clearer than the one with HC. On the one hand, HC is able to separate together only *Type-4* clones (see Figure 6.15). On the other hand, Kernel PCA shows how *Type-3* and *Type-4* clones tend to be gathered together with their own type (see Figure 6.16). Moreover, an additional pattern is detected: the non-compact stencil (Function 5) is located at a larger distance from the compact stencils.

Figure 6.15: Hierarchical Clustering for Kast Spectrum Kernel using ASTs of 3D Stencils (Cut Weight = 64).

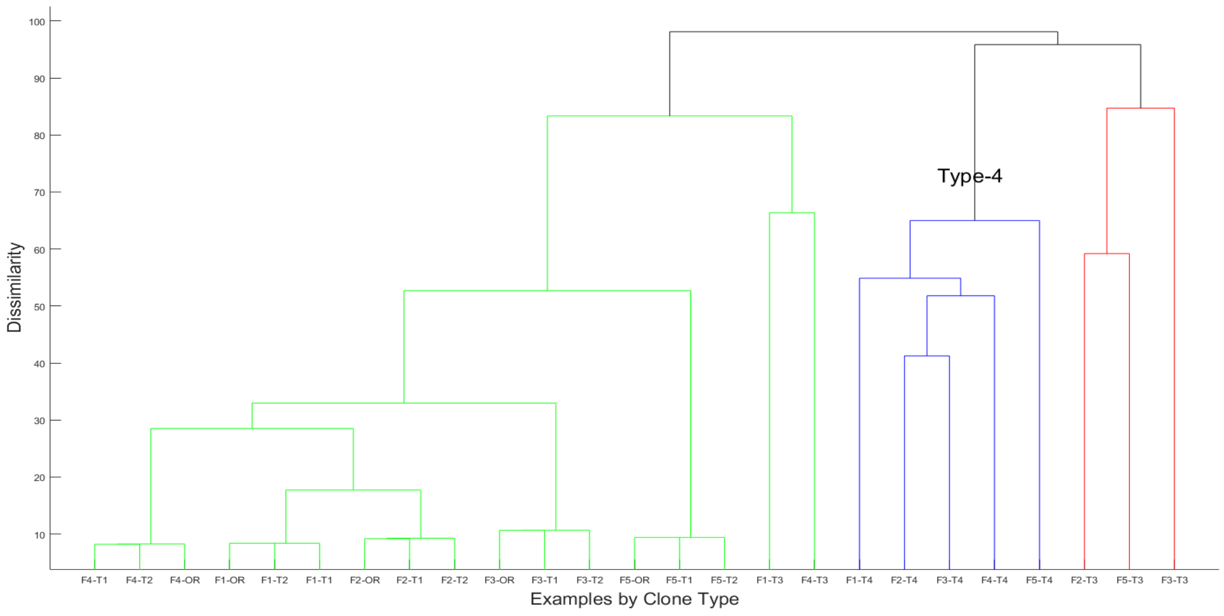
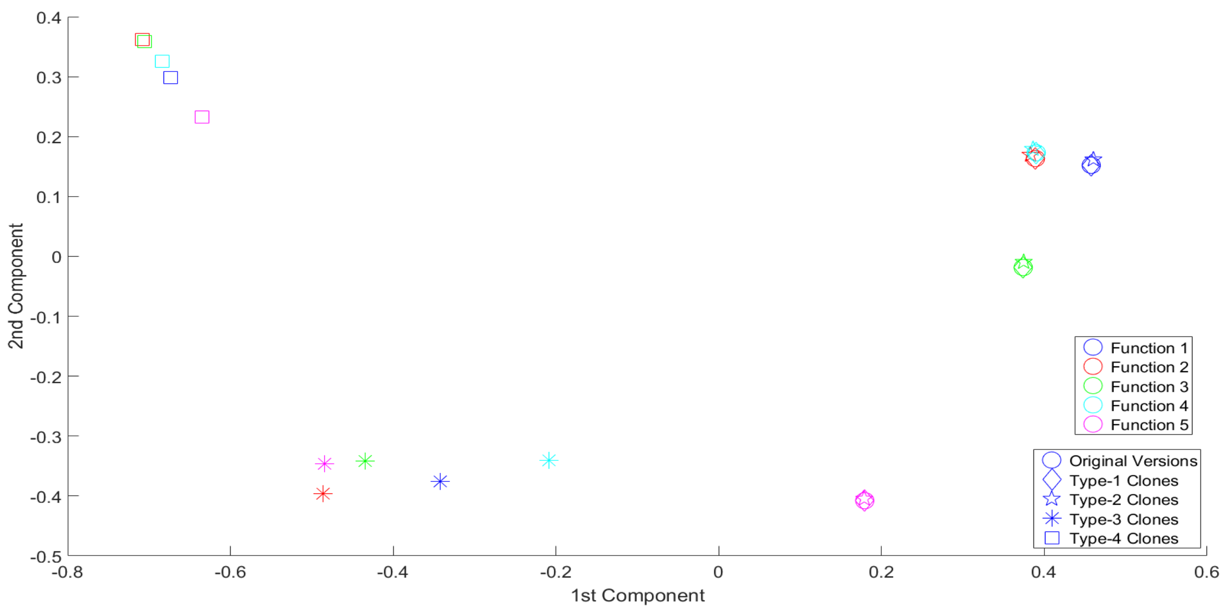


Figure 6.16: KPCA for Kast Spectrum Kernel using ASTs of 3D Stencils (Cut Weight = 64).



Class D (2D Stencils): Here, *Type-3* and *Type-4* clones create clear compact formations with both clustering algorithms (see Figures 6.17 and 6.18). The new pattern is also detected: the non-compact stencils (Functions 4 and 5) conform a small cluster far from the rest.

Figure 6.17: Hierarchical Clustering for Kast Spectrum Kernel using ASTs of 2D Stencils (Cut Weight = 64).

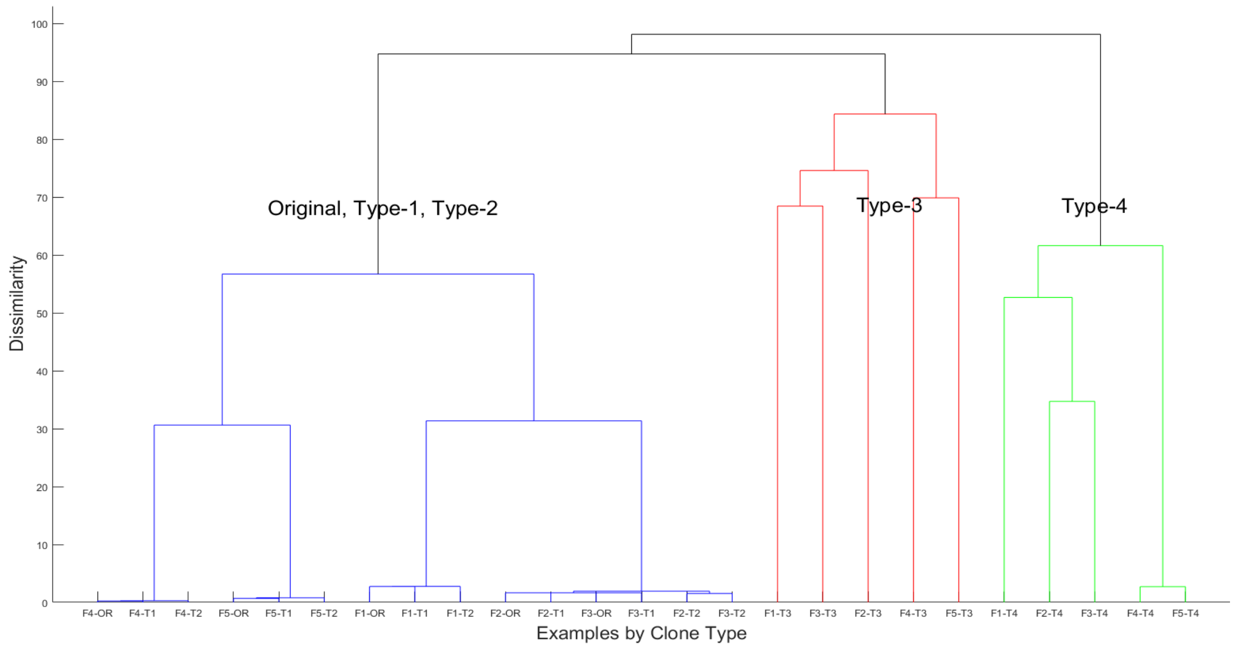
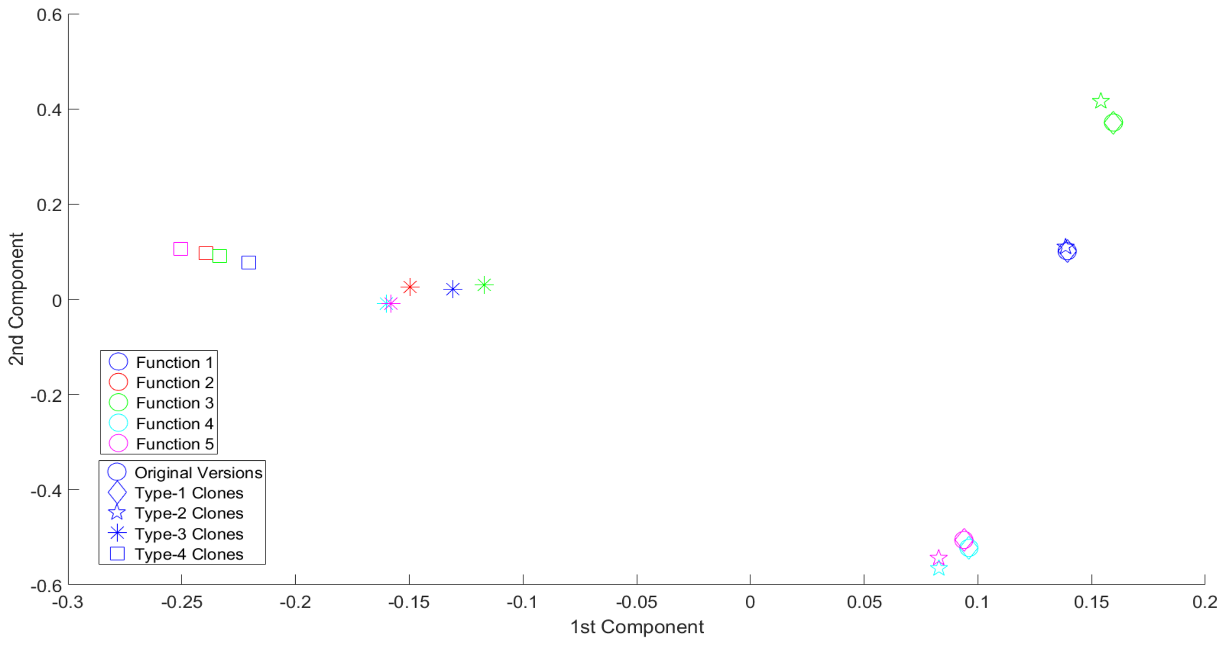


Figure 6.18: KPCA for Kast Spectrum Kernel using ASTs of 2D Stencils (Cut Weight = 64).



Those are the results corresponding to the analysis of Clang ASTS. The following section presents the findings obtained with LLVM IRs.

6.3.2 Clustering with LLVM Intermediate Representations

The usage of LLVM IRs with the *kast spectrum kernel* presents some differences. When using LLVM IRs, the ranking in distances to the original version does not correspond completely to the expectation according to the definition of clone types: as expected, *Type-1* clones are found almost overlapped to the original version, while *Type-4* clones are situated at further distance. However, some *Type-3* clones are located closer to their corresponding original version than the *Type-2* clones. Let us see that in detail.

Clustering by Function Class

In the problem of separating broad classes of functions, Hierarchical Clustering shows that this kernel is able to separate the usual 3 clusters, but with misplaced examples (see Figure 6.19). Same like with the baseline kernel, the misplaced examples are all *Type-4* clones of the class B (Sort functions). It still holds that the intra-cluster distances are larger than with the baseline kernel. Unfortunately, Kernel PCA keeps failing at showing a good clustering in this problem.

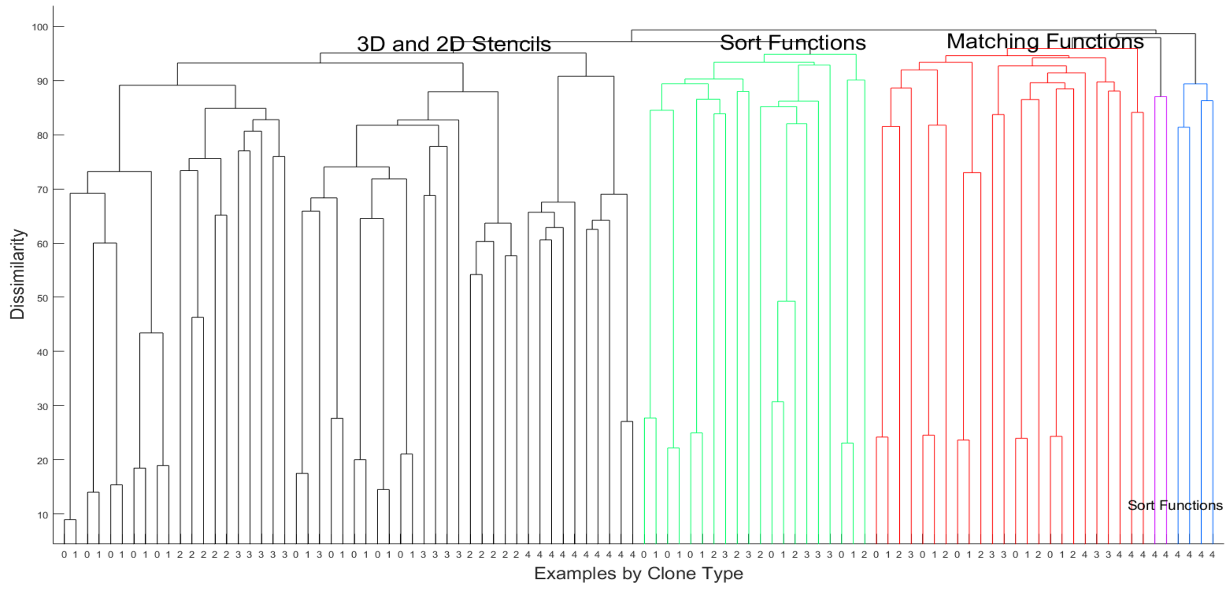


Figure 6.19: Hierarchical Clustering for Kast Spectrum Kernel using IRs (Cut Weight = 32).

Clustering by Clone Type

In the problem of organizing functions inside a determinate class according to the clone type they belong, there is no remarkable improvement with respect to the baseline kernel. For matching functions (class A) and sorting functions (class B), there are no additional detected patterns. The separation between group I and group II inside matching functions is not detected. Only stencil functions show a meaningful clustering inside each class, but without distinction between compact and non-compact stencils.

Class C (3D Stencils): The clustering achieved by Kernel PCA is clearer than HC. On the one hand, HC manages to collect all *Type-2* and *Type-4* clones in their respective clusters (see Figure 6.20). On the other hand, Kernel PCA achieves the same plus the separation of *Type-3* clones (see Figure 6.21). However, these *Type-3* clones are localized closer to the original versions than *Type-2* clones. Unfortunately, there are no clear signs of separation of the non-compact stencil from the rest.

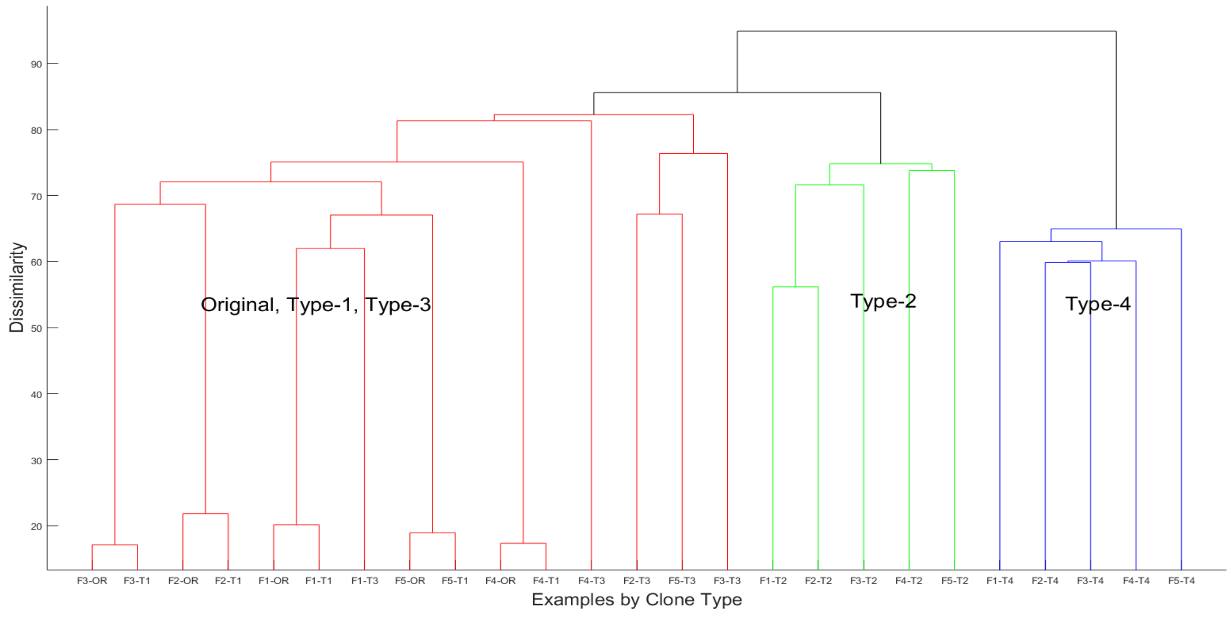


Figure 6.20: Hierarchical Clustering for Kast Spectrum Kernel using IRs of 3D Stencils (Cut Weight = 32).

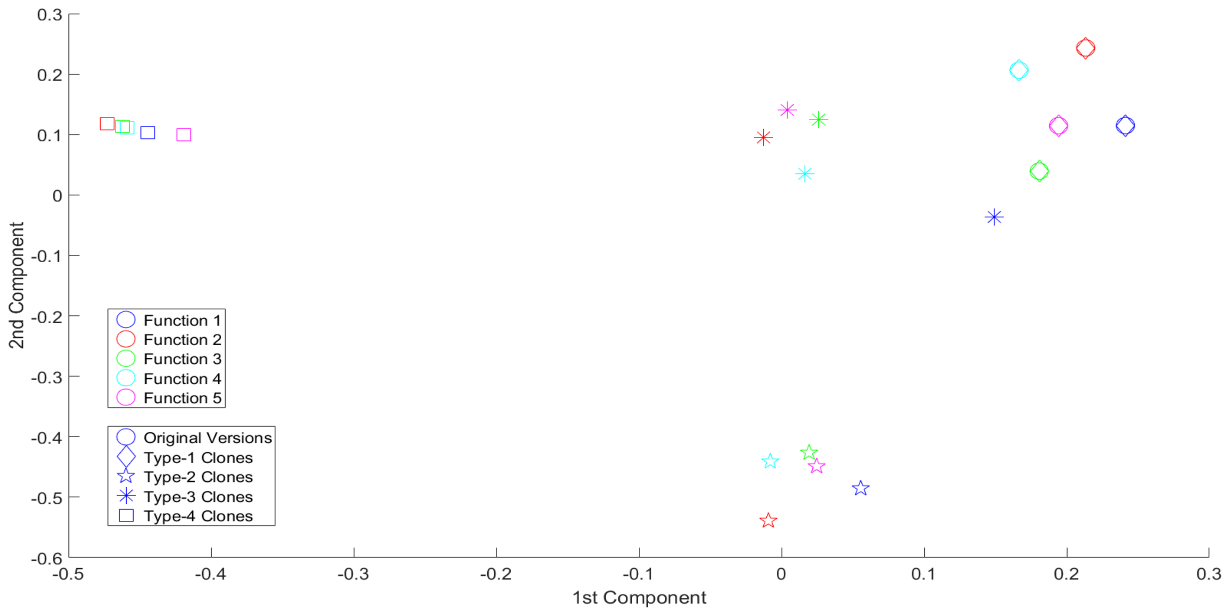


Figure 6.21: KPCA for Kast Spectrum Kernel using IRs of 3D Stencils (Cut Weight = 32).

Class D (2D Stencils): Inside 2D stencils, clones of the same type conform clear compact formations with Kernel PCA only. HC can only separate completely *Type-4* clones (see Figure 6.22), while Kernel PCA achieves the separation for *Type-2*, *Type-3* and *Type-4* clones (see Figure 6.23). In any case, there are no signs of separation between compact and non-compact stencils.

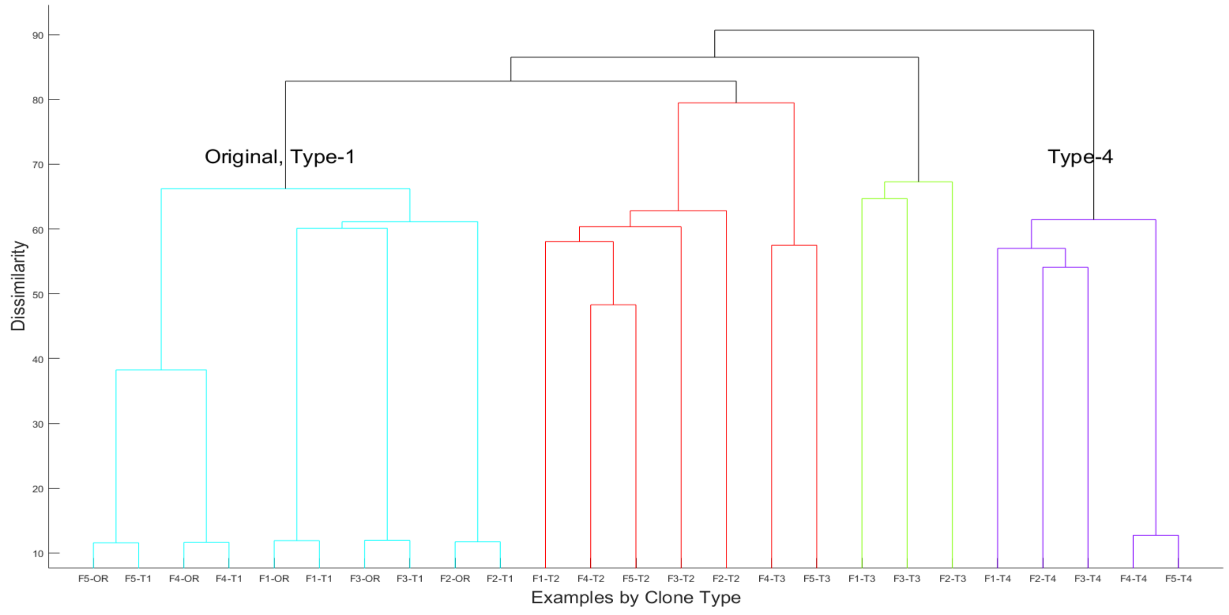


Figure 6.22: Hierarchical Clustering for Kast Spectrum Kernel using IRs of 2D Stencils (Cut Weight = 32).

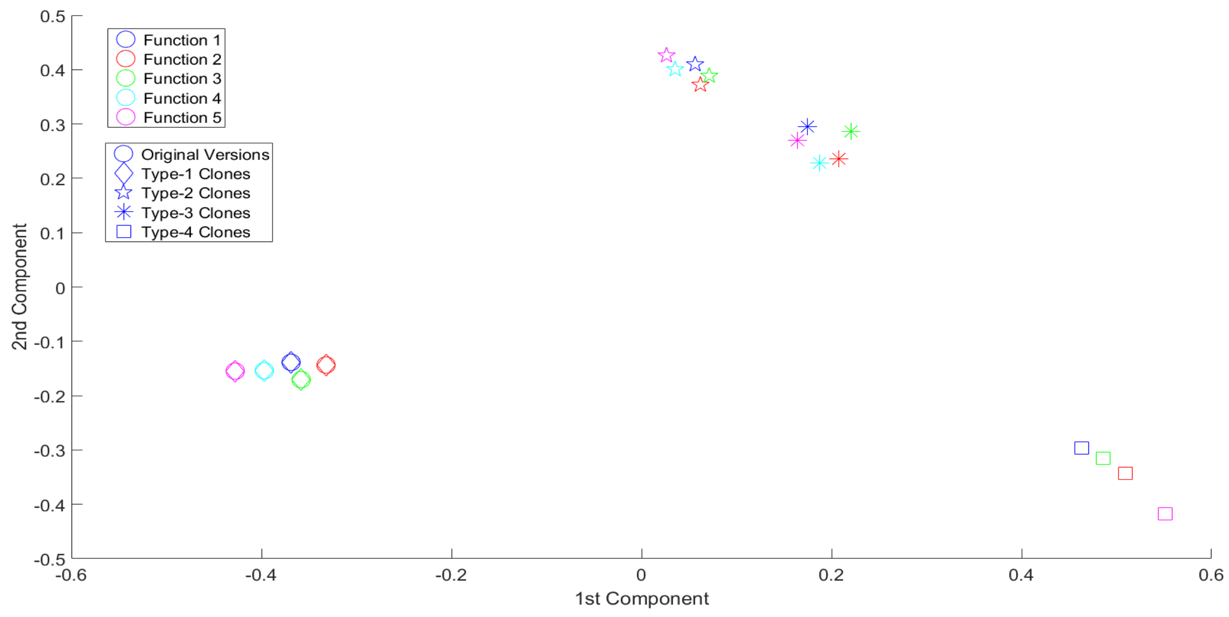


Figure 6.23: KPCA for Kast Spectrum Kernel using IRs of 2D Stencils (Cut Weight = 32).

6.3.3 Remarks

The application of the *kast spectrum kernel* over Clang ASTs shows a better separation of classes of functions in the problem of general clustering, than its application over LLVM IRs. In the particular case of the stencil classes, the separation between compact and non-compact stencils is a new pattern not seen with the baseline kernel. However, the separation between group I and group II inside class A (matching functions) is not achieved.

6.4 Kast1 Spectrum Kernel

In this section, the results of the analysis of the second kernel from the **kastx** family are covered, namely, the *kast1 spectrum kernel.* For this kernel, the experiment shows that it yields a better cluster separation than the baseline kernel:

- *Similarities with the baseline kernel:* For the *kast1 spectrum kernel*, the usual basic scheme of clusters is achieved, with hierarchical clustering being more effective on the task than Kernel PCA. Another common point is, that the ranking in distances for clone types with respect to the original version correspond to the expectation according to the theory.
- *Differences with the baseline kernel:* The major difference with respect to the baseline kernel is that the intra-cluster distances are larger.
- *Advantages with respect to the baseline kernel:* In the fist place, with this kernel, there are no misplaced examples. Secondly, *Type-4* clones do not create clusters outside their respective classes. Thirdly, it is important to notice that both LLVM IRs and Clang ASTs yield similar clustering results. Fourthly, when using Clang ASTs, the original version of a function and its corresponding *Type-1* and *Type-2* clones are found almost overlapped.

Recall that this kernel was designed as an improved version of the *kast spectrum kernel*, that reduces the noise introduced by small matches. The following sections detail the behavior of this kernel with Clang ASTs and LLVM IRs respectively.

6.4.1 Clustering with Clang Abstract Syntax Trees

If the *kast1 spectrum kernel* is applied over Clang ASTs, original versions with their corresponding *Type-1* and *Type-2* clones are gathered together very close to each other. This is new with respect to all the previous kernels.

Clustering by Function Class

In the problem of detecting classes as general patterns, the best results for Hierarchical Clustering are obtained using a cut weight of 16, showing no misplaced examples on any of the clusters (see Figure 6.24). Unfortunately, Kernel PCA results are not conclusive, as no clear cluster formation can be seen.

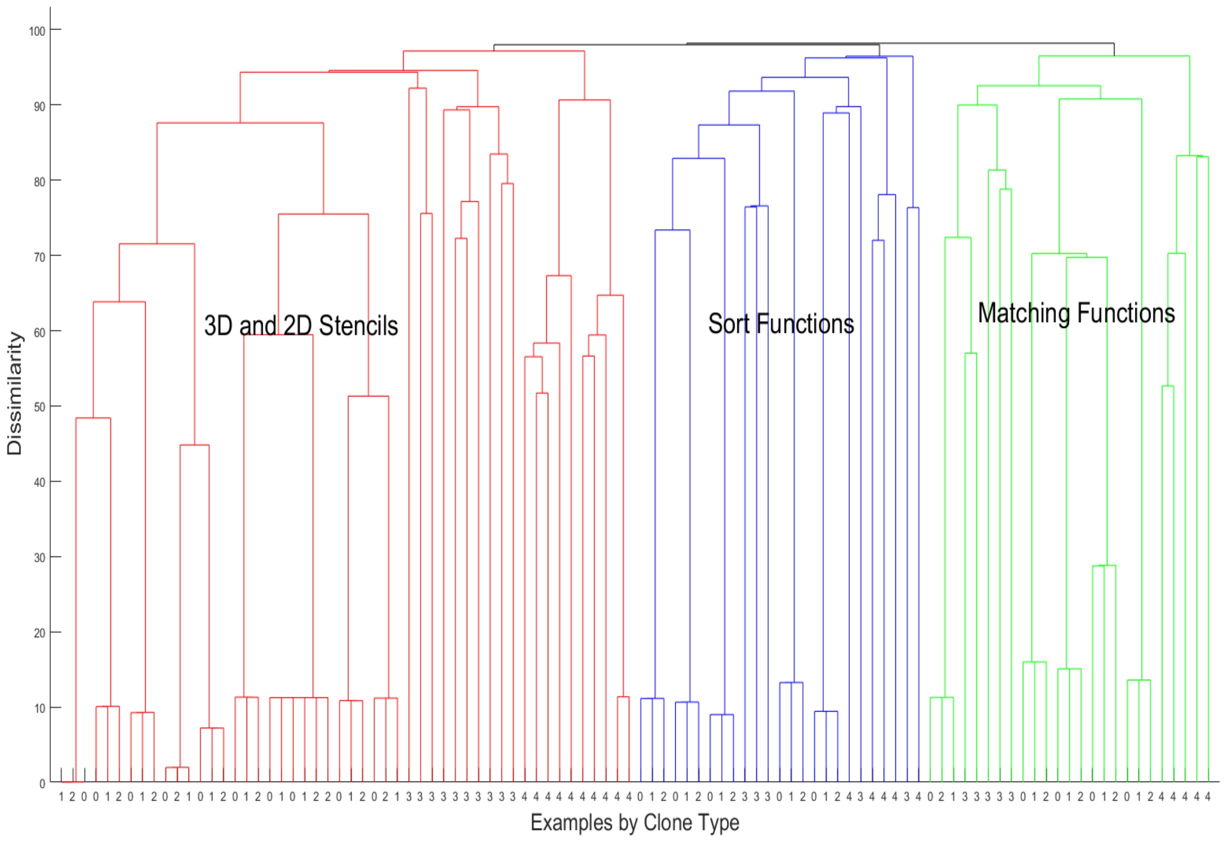


Figure 6.24: Hierarchical Clustering for Kast1 Spectrum Kernel using ASTs (Cut Weight = 16).

Clustering by Clone Type

When it comes to see the internal organization of each class of functions, both Kernel PCA and HC show that the original versions and their corresponding *Type-1* and *Type-2* clones conform single clusters. It is important to notice *Type-1* and *Type-2* clones seem to be at a similar distance from the original version in both HC and Kernel PCA. For class A and class B, no other significant patterns are found.

Class C (3D Stencils): HC shows three branches, one for *Type-3* clones, one for *Type-4* clones and one for the rest (see Figure 6.25) This findings are confirmed by the results of Kernel PCA; with it, five different small clusters are formed that comprise each original function an its corresponding *Type-1* and *Type-2* clones (see Figure 6.26). This supposes an improvement in comparison with the baseline kernel, which could only gather the original versions with its *Type-1* clone. Additionally, *Type-3* clones of all examples conform a compact cluster away from the original versions. The same behavior is observed with *Type-4* clones, which were located further away. Unfortunately, there is not clear separation between non-compact and compact stencils.

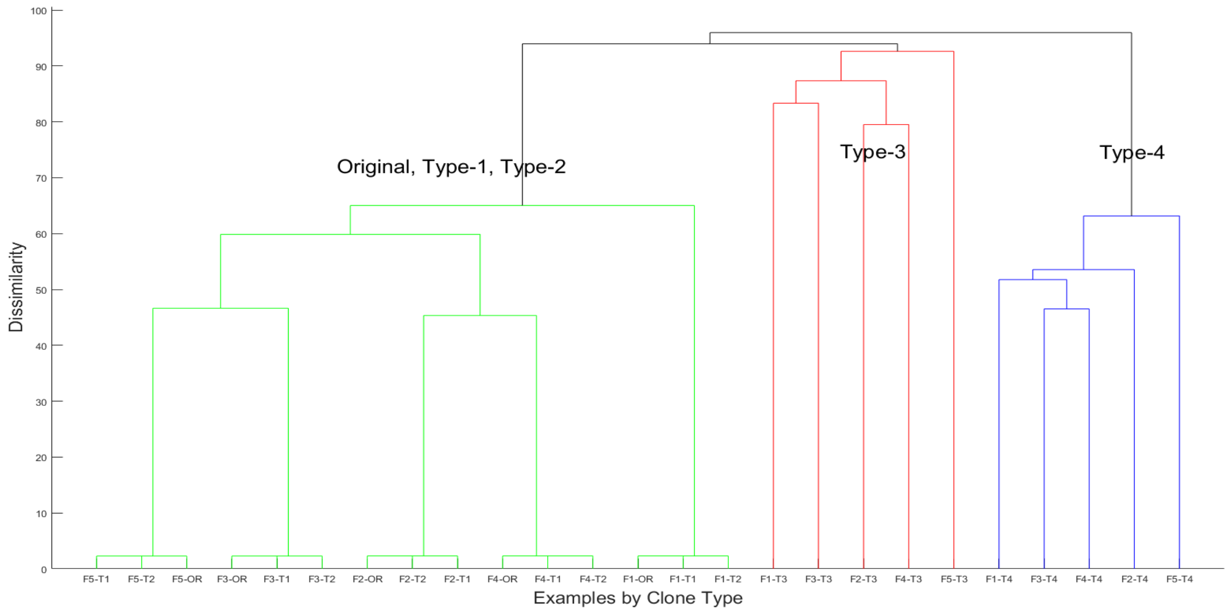


Figure 6.25: Hierarchical Clustering for Kast1 Spectrum Kernel using ASTs of 3D Stencils (Cut Weight = 16).

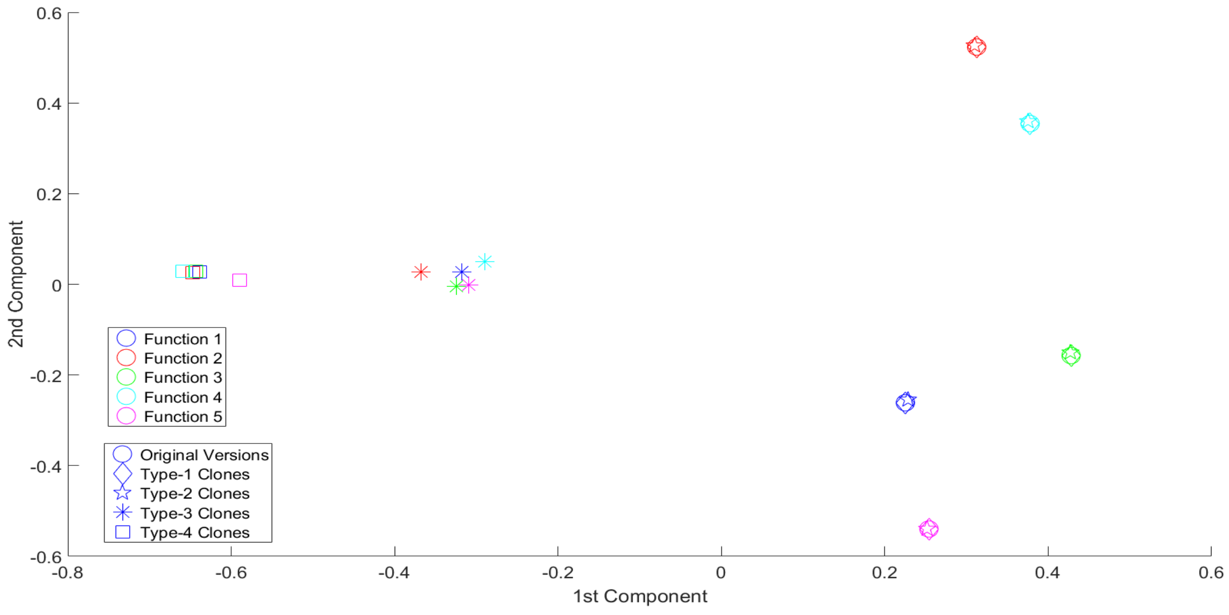


Figure 6.26: KPCA for Kast1 Spectrum Kernel using ASTs of 3D Stencils (Cut Weight = 16).

Class D (2D Stencils): HC shows a similar behavior presented by 3D stencils (see Figure 6.27). However, in Kernel PCA, the edge and vertex stencils are not separated but mixed in the same small cluster (see Figure 6.28). In this case, the non-compact stencils appear separated from the compact stencils.

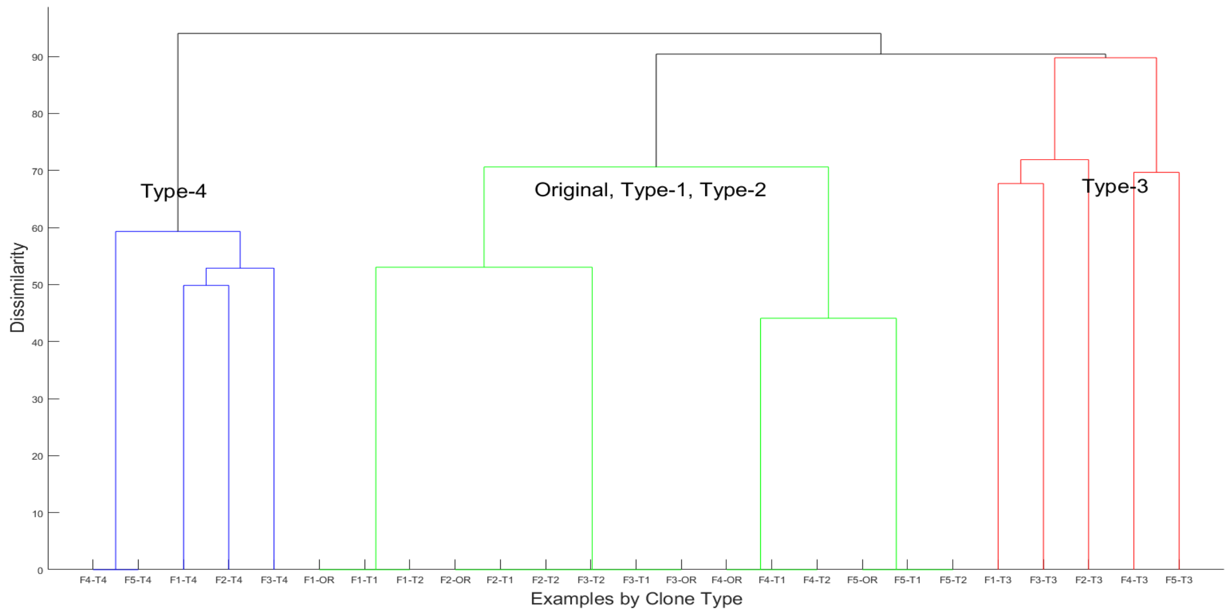


Figure 6.27: Hierarchical Clustering for Kast1 Spectrum Kernel using ASTs of 2D Stencils (Cut Weight = 16).

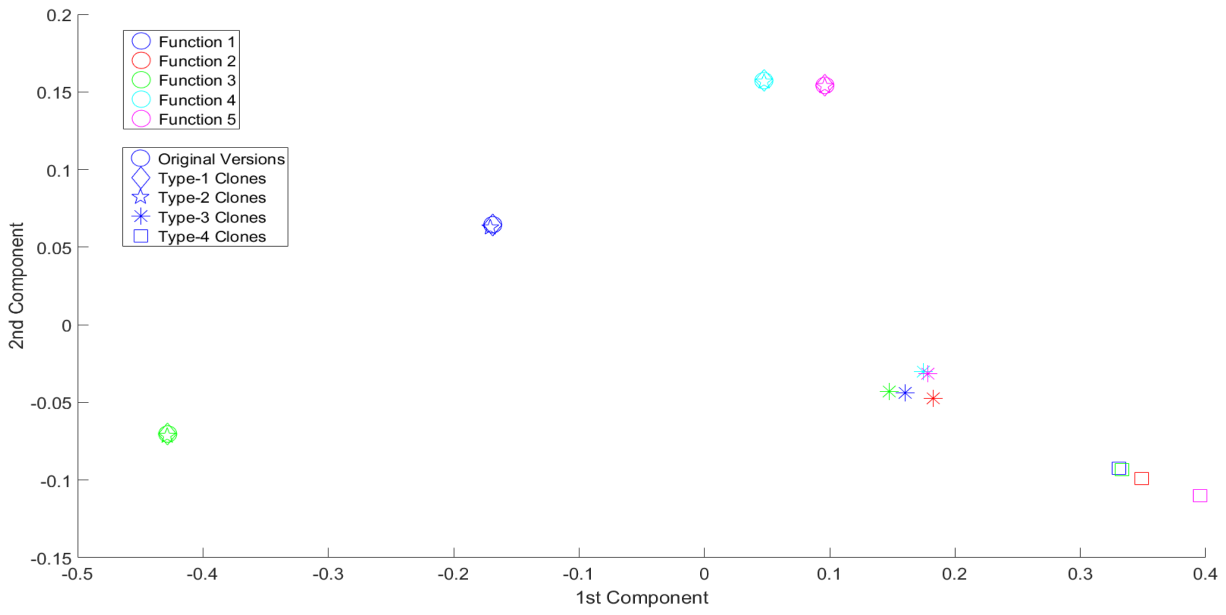


Figure 6.28: KPCA for Kast1 Spectrum Kernel using ASTs of 2D Stencils (Cut Weight = 16).

Next section shows the results of using LLVM IRs instead of Clang ASTs.

6.4.2 Clustering with LLVM Intermediate Representations

The usage of this kernel with LLVM IRs achieves the usual clustering scheme of three groups with no misplaced examples. In the internal analysis of each class, the expected organization of clone types is detected.

Clustering by Function Class

In the general problem of separating functions according to the class they belong, the best results for hierarchical clustering are obtained when using a cut weight value of 1. It is important to remark that there are no misplaced examples. However, some intra-cluster distances appear to be too large. HC clearly detects the same 3 usual clusters (see Figure 6.29). However, Kernel PCA failed at that task.

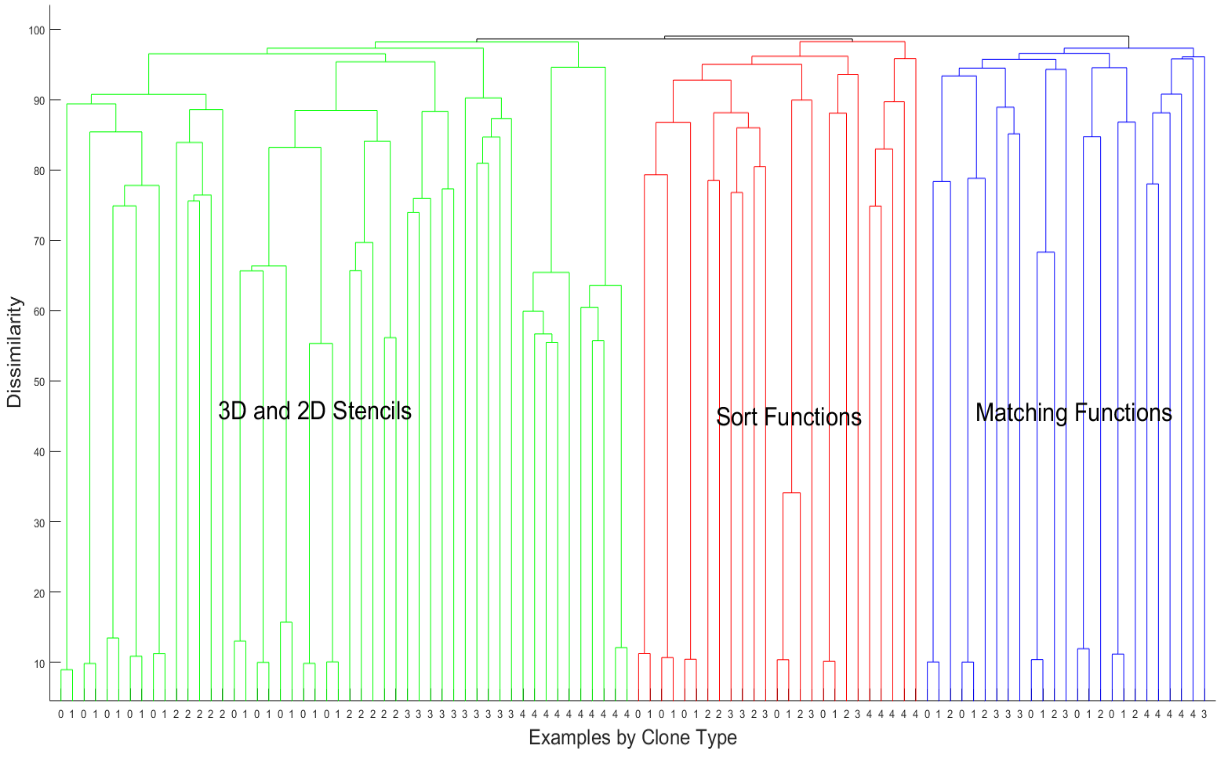


Figure 6.29: Hierarchical Clustering for Kast1 Spectrum Kernel using IRs (Cut Weight = 1).

Clustering by Clone Type

A deeper look into each class shows that, with both Kernel PCA and HC, the original versions and their corresponding *Type-1* clones tend to conform single clusters, while *Type-2* and *Type-3* have the tendency to agglomerate on the same cluster at considerable distance from their corresponding original versions. *Type-4* clones have the tendency to conform a separate cluster with the longest distance to the original versions. As expected, *Type-2* clones are located closer to the original version, followed by *Type-3* and *Type-4* clones. This kernel is sensible to the changes in the LLVM IRs of *Type-2* clones, which put them far from the original version. However, due to the fact that code modifications follow similar rules, they are placed closer to clones of the same type that might not have the same semantics but contain the similar structure. class A and class B did not show any other interesting pattern.

Class C (3D Stencils): HC shows 4 defined branches, one for the original versions and *Type-1* clones, and one for each remaining type (see Figure 6.30). All this is confirmed by Kernel PCA, where the original functions and *Type-1* clones conglomerate in a sparse cluster, except from the non-compact stencil. The non-compact stencil differs from the other stencils because it has a few additional instructions that correspond to the extra layer calculation. Type-2, *Type-3* and *Type-4* clones conform three different clusters, each one with a clear compact formation (see Figure 6.31).

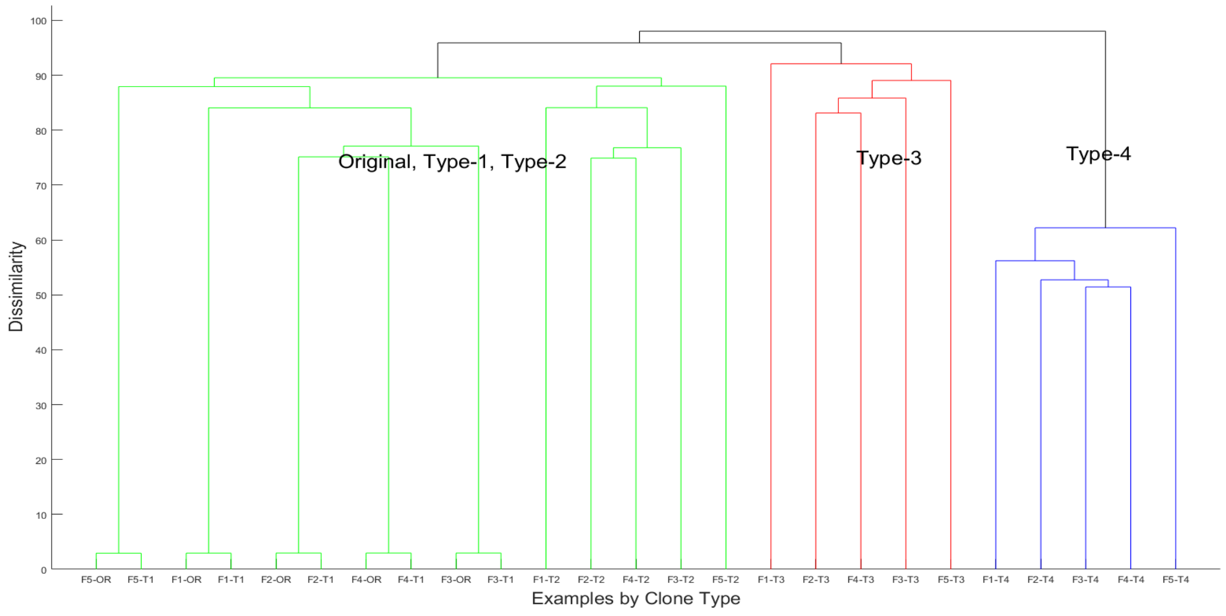


Figure 6.30: Hierarchical Clustering for Kast1 Spectrum Kernel using IRs of 3D Stencils (Cut Weight = 1).

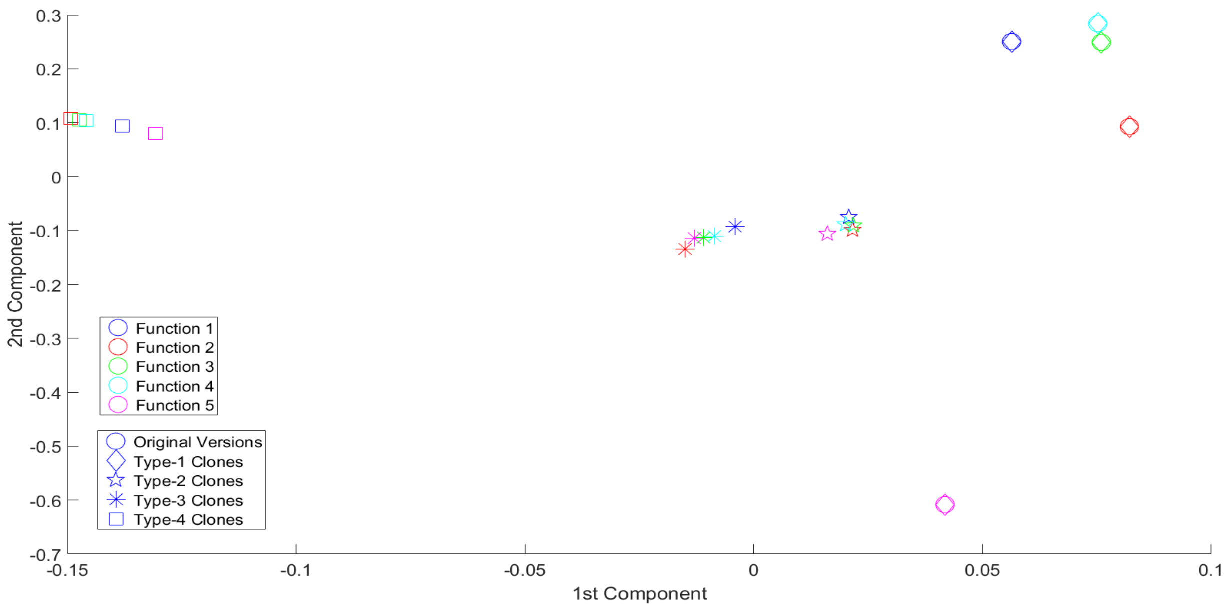


Figure 6.31: KPCA for Kast1 Spectrum Kernel using IRs of 3D Stencils (Cut Weight = 1).

Class D (2D Stencils): This class presents a similar behavior like 3D stencils, with the non-compact stencils closer to each other than to other stencils (see Figures 6.32 and 6.33).

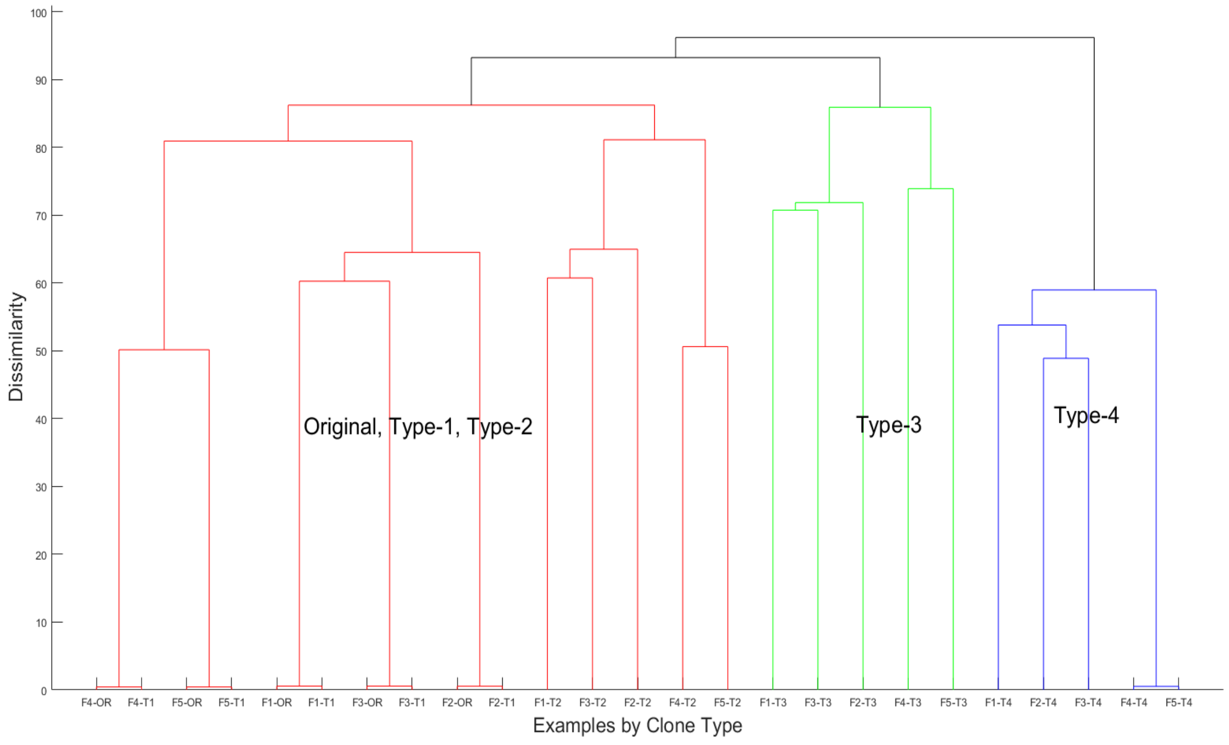


Figure 6.32: Hierarchical Clustering for Kast1 Spectrum Kernel using IRs of 2D Stencils (Cut Weight = 1).

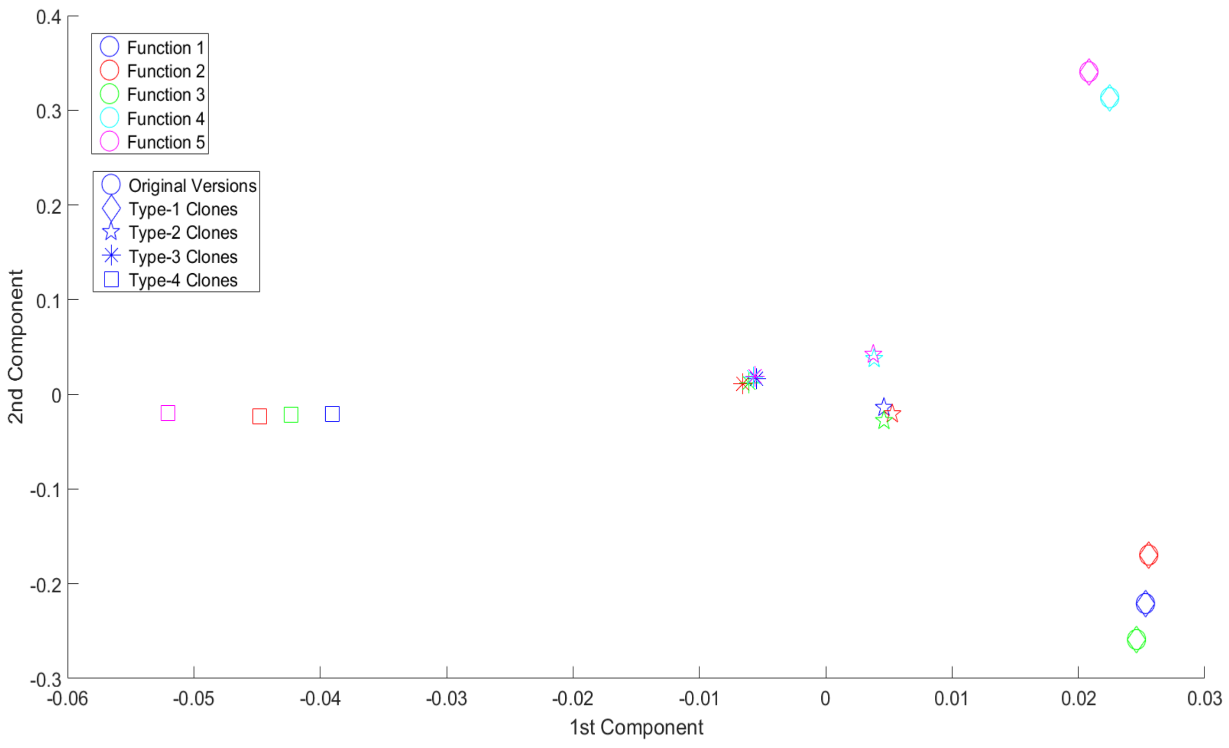


Figure 6.33: KPCA for Kast1 Spectrum Kernel using IRs of 2D Stencils (Cut Weight = 1).

6.4.3 Remarks

The *kast1 spectrum kernel* has a good performance with the two intermediate representation from LLVM. This is an improvement with respect to the *kast spectrum kernel.*

6.5 Kast2 Spectrum Kernel

The last kernel of the family of **kastx** kernels is the *kast2 spectrum kernel.* This kernel presents a better performance in comparison with the baseline kernel:

- *Similarities with the baseline kernel:* For the *kast2 spectrum kernel*, the same basic scheme of clusters is achieved. The ranking in distances for clone types with respect to the original version corresponds to the expectation according to the theory. Additionally, the intra-cluster distances are small too.
- *Differences with the baseline kernel:* The major difference with respect to the baseline kernel is that *Type-4* clones tend to create clusters outside their respective classes only when using Clang ASTs. The baseline kernel shows this behavior when using both representations.
- *Advantages with respect to the baseline kernel:* On the one hand, when using Clang ASTs, the original version of a function and its corresponding *Type-1* and *Type-2* clones are found almost overlapped. On the other hand, using LLVM IRs, no misplaced examples are found. This kernel is the only one where Hierarchical Clustering and Kernel PCA manifest the same clustering patterns.

Recall that this kernel changes the way that weights are summarized and introduces a penalization value. This way it is possible to reflect more naturally the matching between two strings.

6.5.1 Clustering with Clang Abstract Syntax Trees

The application of the *kast2 spectrum kernel* over Clang ASTs shows the usual groups of clusters; it also reflects congruent clone relationships. This kernel is also capable to separate some classes into the expected groups.

Clustering by Function Class

The first problem is the separation of functions into classes. With Clang ASTs, the best results are obtained when using a cut weight value of 32. This kernel clearly detects the same 3 clusters with both algorithms (see Figures 6.34 and 6.35).

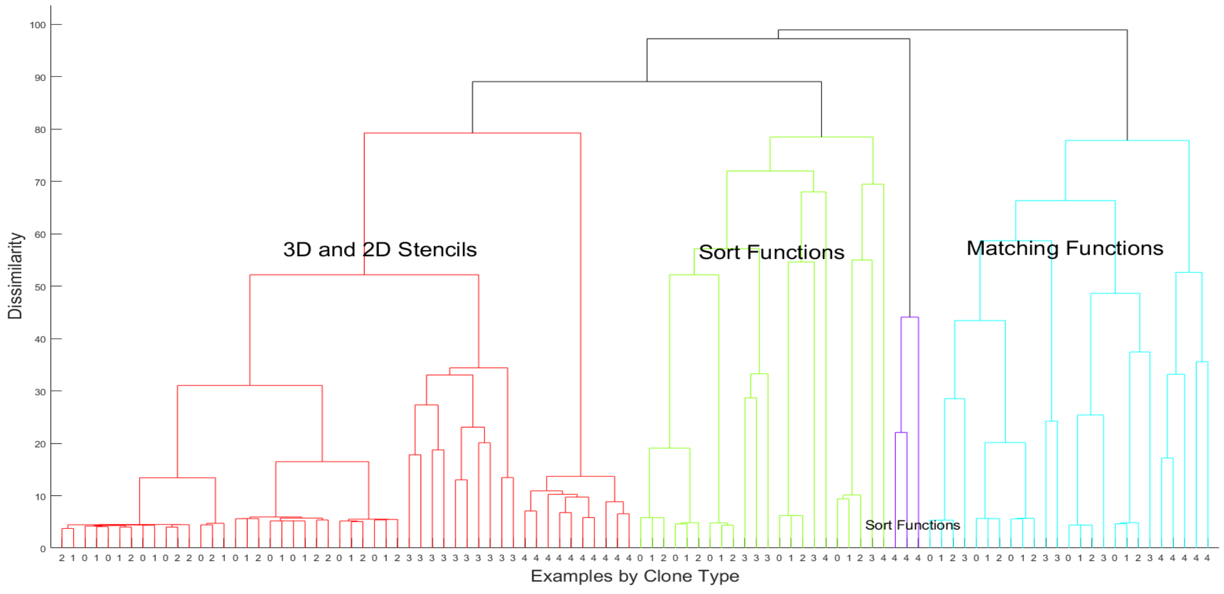


Figure 6.34: Hierarchical Clustering for Kast2 Spectrum Kernel using ASTs (Cut Weight = 32).

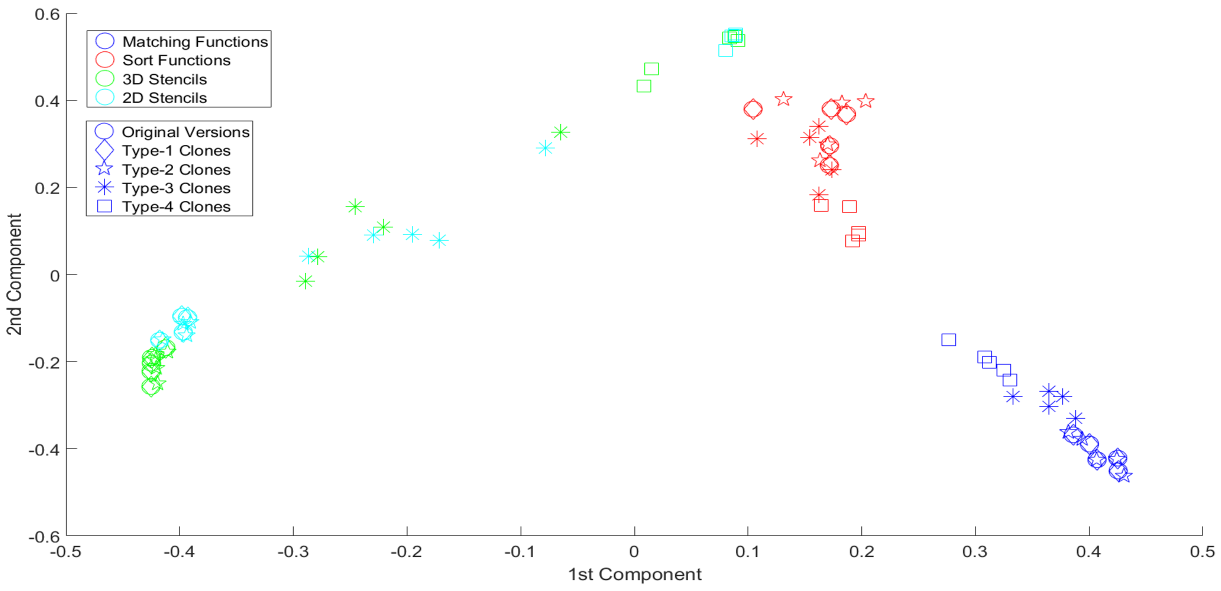


Figure 6.35: KPCA for Kast2 Spectrum Kernel using ASTs (Cut Weight = 32).

Clustering by Clone Type

The second problem is the organization of code clones inside each class. Both Kernel PCA and HC, show that the original versions and their corresponding *Type-1* and *Type-2* clones conform single clusters when compared only with examples of the same general group. It is important to notice *Type-1* and *Type-2* clones seem to be at a similar distance from the original version in both HC and Kernel PCA. *Type-3* and *Type-4* clones are found outside those clusters. As expected, *Type-3* clones are located closer to their respective original versions than *Type-4* clones.

Class A (Matching Functions): With HC, Group I and II seem to be separated. However, *Type-4* clones are grouped on a separated branch (see Figure 6.36). The clustering is better with Kernel PCA, where it can be seen that *Type-4* clones respect the discrimination between Group I and Group II (see Figure 6.37).

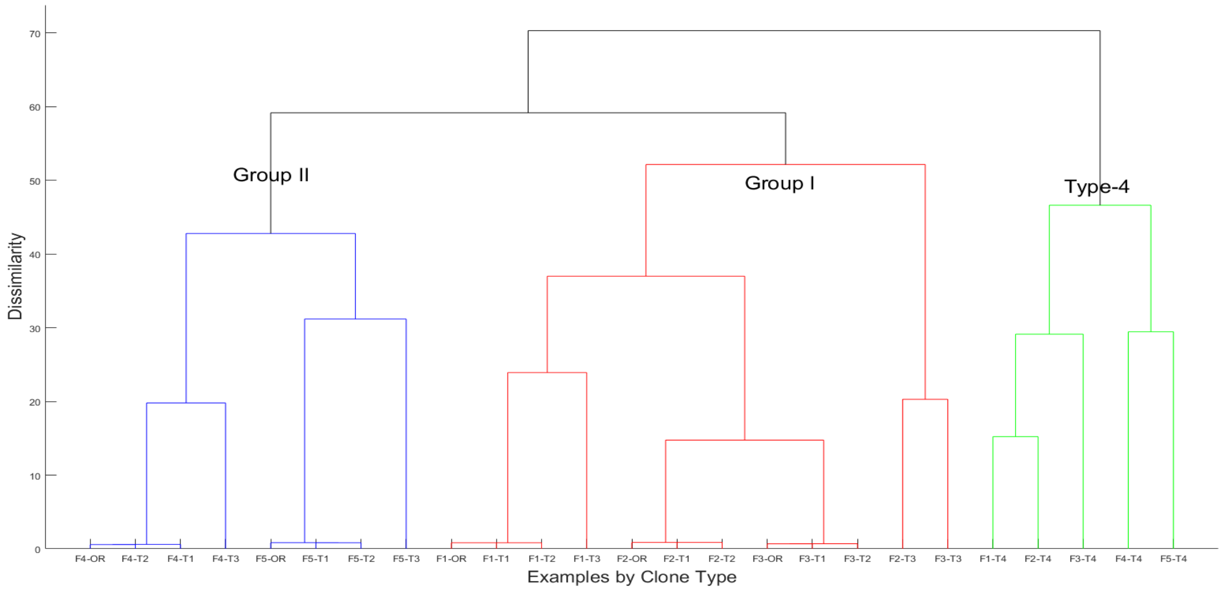


Figure 6.36: Hierarchical Clustering for Kast2 Spectrum Kernel using ASTs of Matching Functions (Cut Weight = 32).

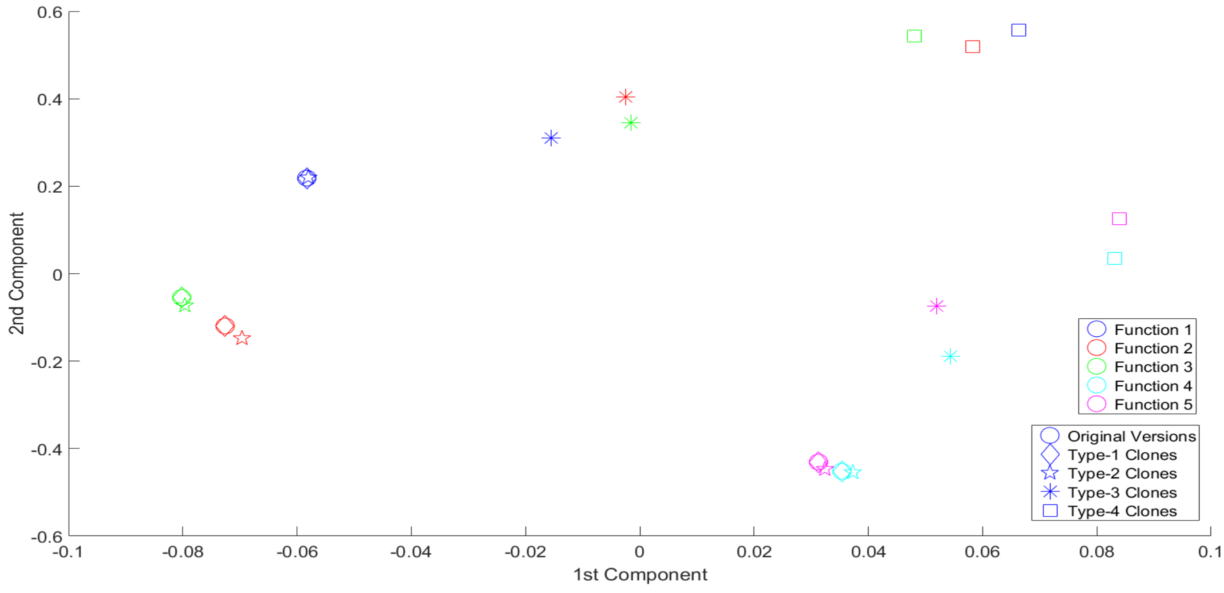


Figure 6.37: KPCA for Kast2 Spectrum Kernel using ASTs of Matching Functions (Cut Weight = 32).

Class B (Sort Functions): No additional patterns were observed.

Class C (3D Stencils): HC manifests three branches, one for the *Type-3* clones, one for *Type-4* clones and one for the rest (see Figure 6.38). See how in Kernel PCA all the original functions and *Type-1* and *Type-2* clones conglomerate in a single cluster (see Figure 6.39). The *Type-3* clones of all programs conform a sparse cluster away from the original versions. The same behavior is observed with *Type-4* clones. A distinction between compact and non-compact stencils is evident in Kernel PCA, with the non-compact stencil separating from the main conglomeration, and in HC, with a branch collecting compact stencils, while another one does the same with the non-compact one. This is not seen with the *blended spectrum kernel*.

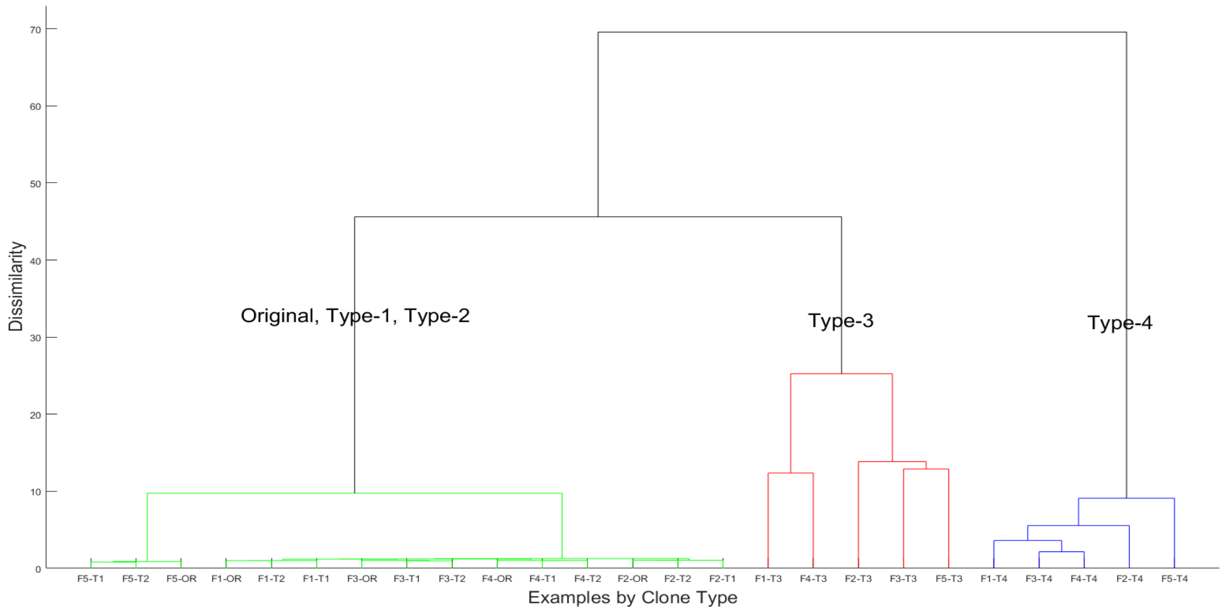


Figure 6.38: Hierarchical Clustering for Kast2 Spectrum Kernel using ASTs of 3D Stencils (Cut Weight = 32).

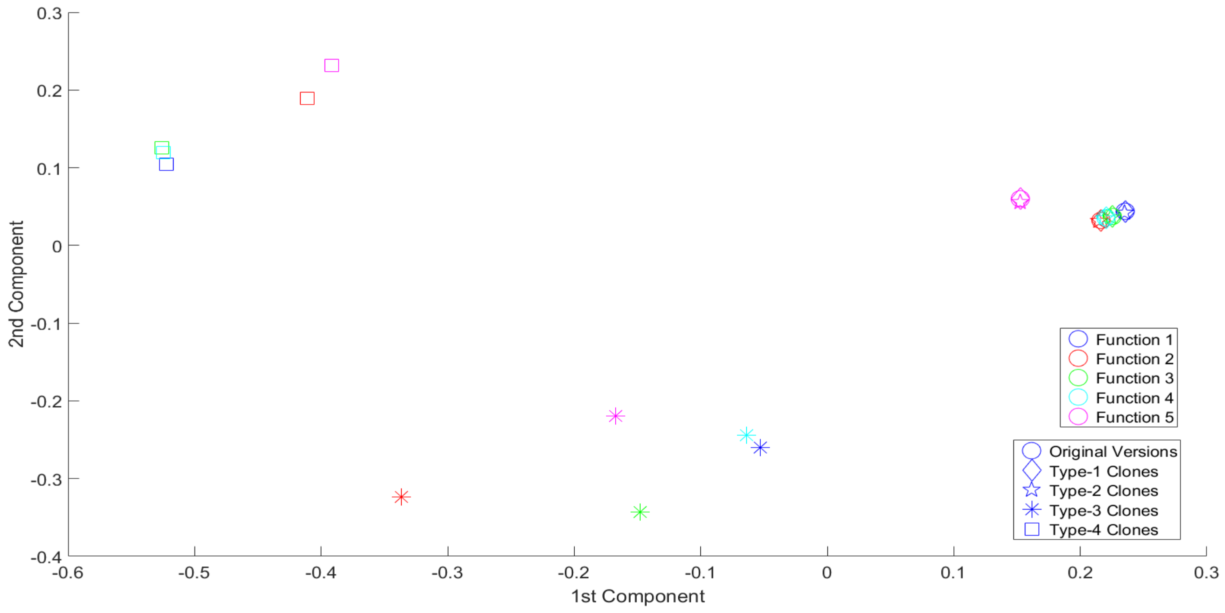


Figure 6.39: KPCA for Kast2 Spectrum Kernel using ASTs of 3D Stencils (Cut Weight = 32).

Class D (2D Stencils): 2D stencils show a similar behavior presented by 3D stencils (see Figures 6.40 and 6.41). Here also both algorithms are able to distinguish between compact and non-compact stencils.

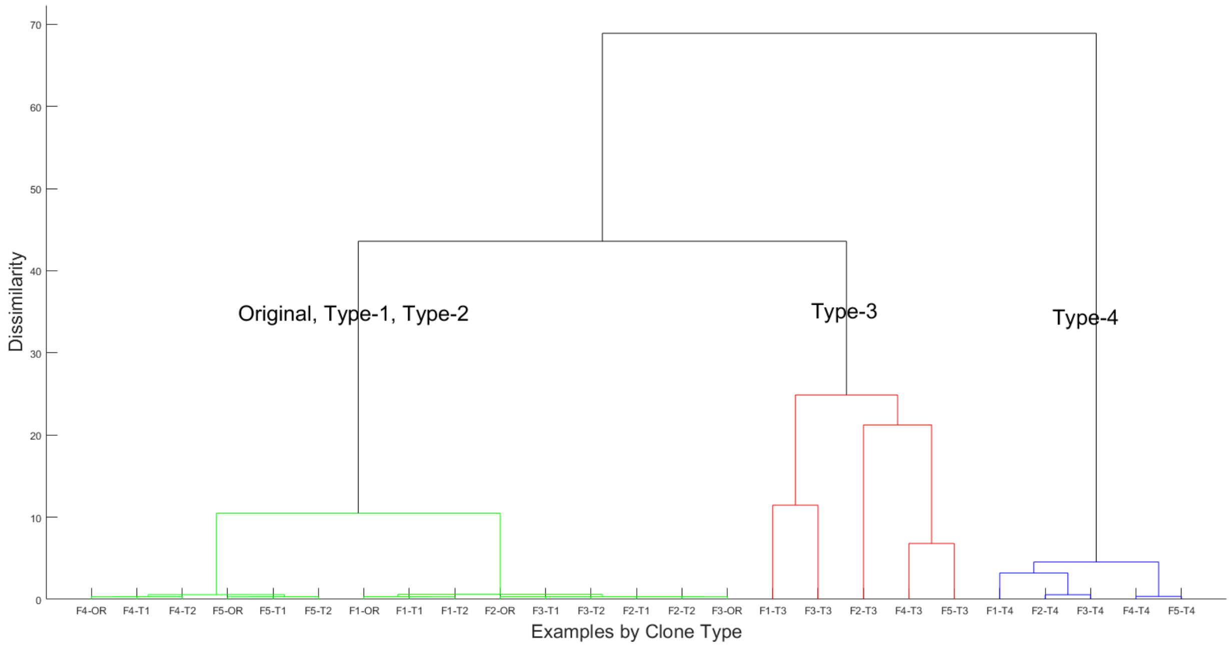


Figure 6.40: Hierarchical Clustering for Kast2 Spectrum Kernel using ASTs of 2D Stencils (Cut Weight = 32).

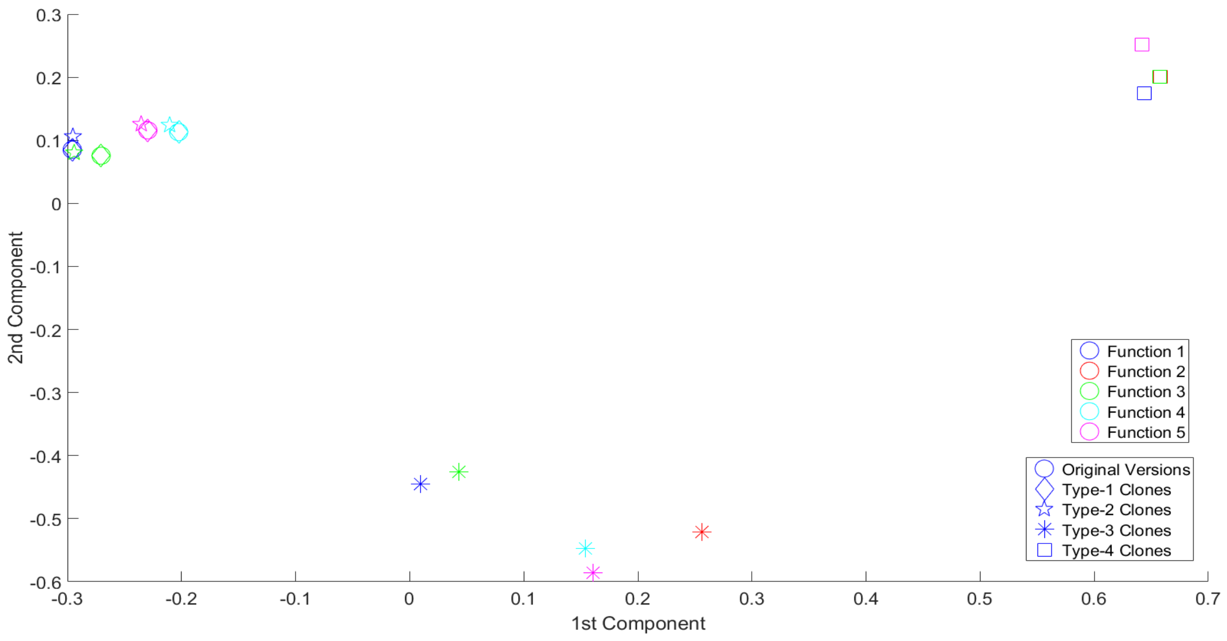


Figure 6.41: KPCA for Kast2 Spectrum Kernel using ASTs of 2D Stencils (Cut Weight = 32).

Next, we detail the results with the other intermediate representation.

6.5.2 Clustering with LLVM Intermediate Representations

With LLVM IRs, the *kast2 spectrum kernel* presents a trade off between separating clone types (intra-cluster distances) while separating classes (inter-cluster distances).

Clustering by Function Class

In the problem of finding classes as patterns, the best results are obtained when using a cut weight value of 32. It is important to remark that there are no misplaced examples when using HC (see Figure 6.42). This kernel clearly detects the usual 3 clusters with both algorithms. With both clustering techniques, the intra-cluster distances are smaller than the ones obtained by the previous kernel. Small intra-cluster distances make a cluster more compact and distinguishable from others. This easies the detection of function classes. A closer inspection of Kernel PCA (see Figure 6.43) depicts that some examples of all classes are drastically separated from the rest, sometimes conforming totally separated clusters. As expected, these exceptional clusters are mostly conformed by all the *Type-4* clones of their class, which correspond to the the most different implementations to the original versions. Such finding is confirmed by HC, where *Type-4* clones tend to be organized under a separate branch inside their respective class.

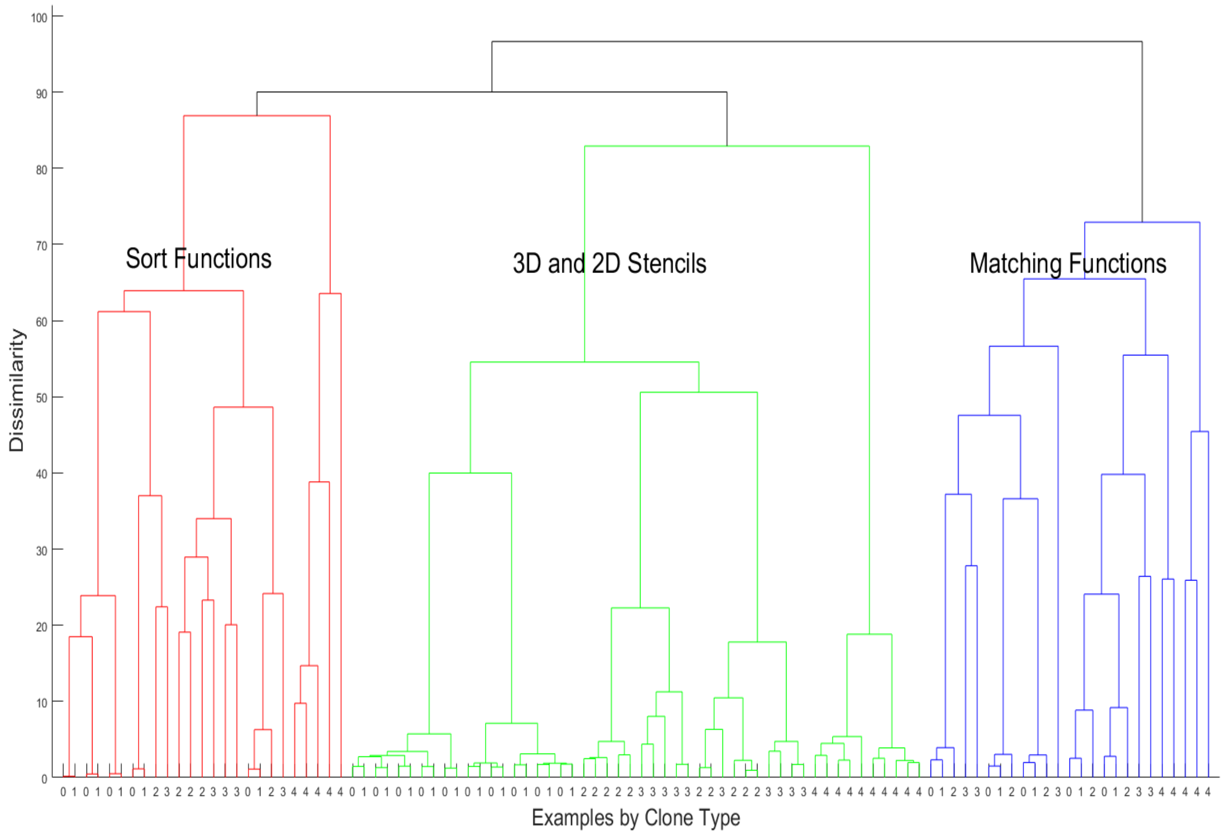


Figure 6.42: Hierarchical Clustering for Kast2 Spectrum Kernel using IRs (Cut Weight = 32).

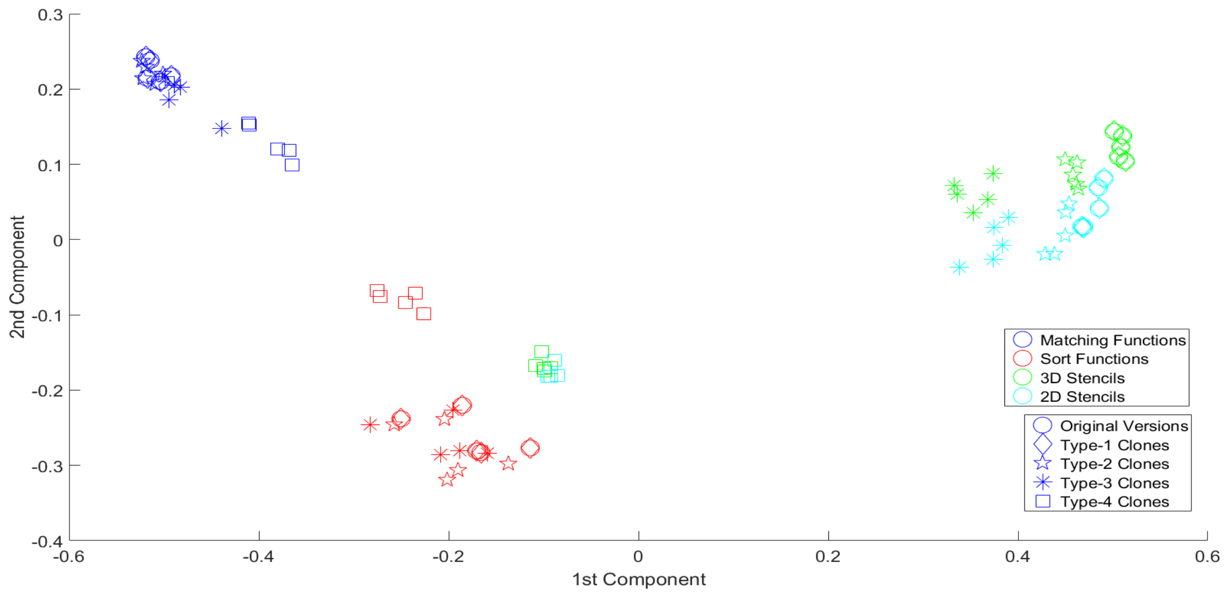


Figure 6.43: KPCA for Kast2 Spectrum Kernel using IRs (Cut Weight = 32)

Clustering by Clone Type

Here both clustering techniques grouped original versions and their corresponding *Type-1*. However, the distance between the original versions and their corresponding *Type-2* clones has been significantly reduced. This represents an enhancement on the clone detection capabilities. *Type-3* and *Type-4* clones are the most distanced examples.

Class A (Matching Functions): Both clustering algorithms were able to clearly separate Group I and II. (see Figures 6.44 and 6.45).

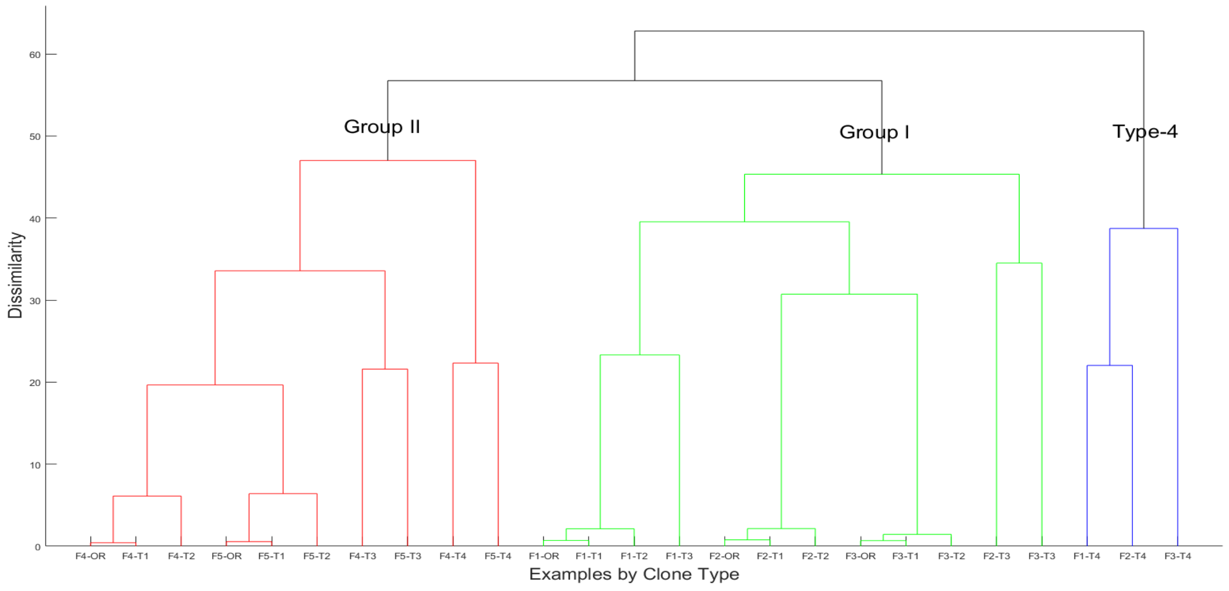


Figure 6.44: Hierarchical Clustering for Kast2 Spectrum Kernel using IRs of Matching Functions (Cut Weight = 32).

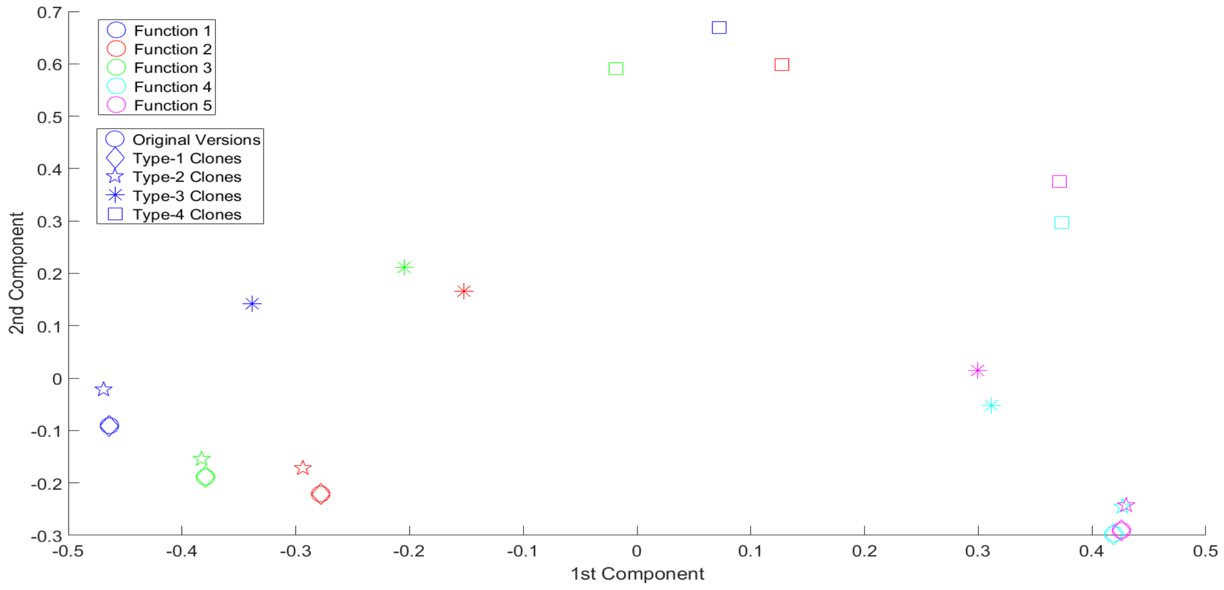


Figure 6.45: KPCA for Kast2 Spectrum Kernel using IRs of Matching Functions (Cut Weight = 32).

Class C (3D Stencils): HC creates four branches, one for the *Type-2* clones, one for the *Type-3* clones, one for *Type-4* clones and one for the rest (see Figure 6.46). With Kernel PCA, all the original functions and *Type-1* clones conglomerate in a compact cluster (see Figure 6.47). *Type-2* clones and *Type-3* clones locate themselves on the neighboring clusters with a less compact formation than *Type-4* clones, which clearly show a solid agglomeration. The drastic changes in code seem to put away these clones from the original version and puts them closer to other clones that might not have the same semantics but contain the similar structure. With LLVM IRs, a distinction between compact and non-compact stencils is not possible.

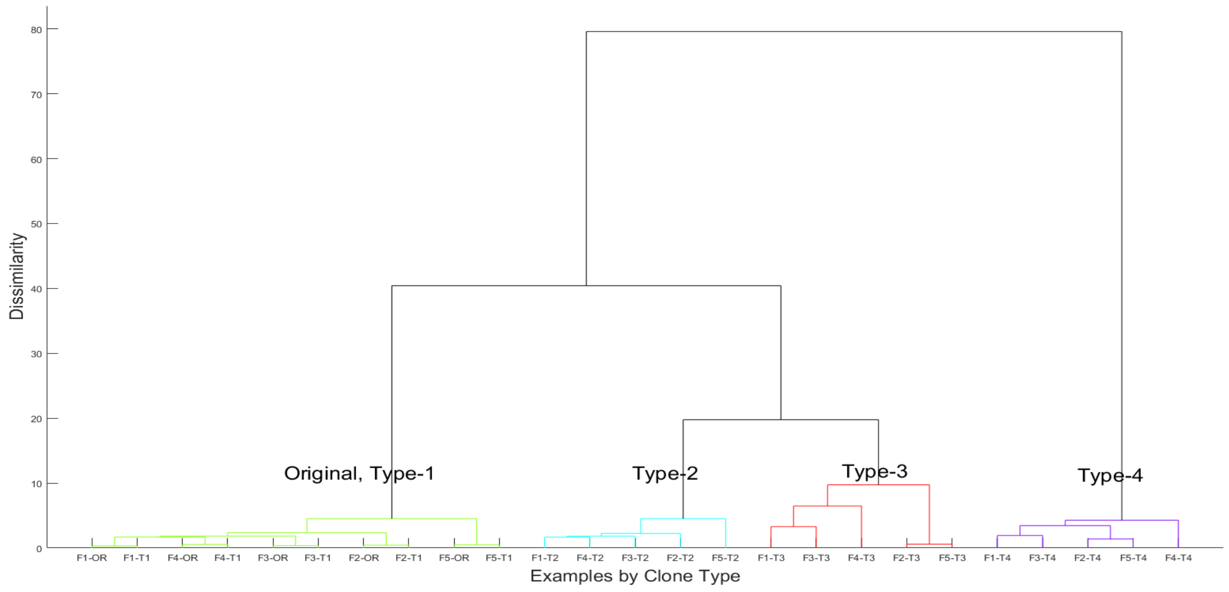


Figure 6.46: Hierarchical Clustering for Kast2 Spectrum Kernel using IRs of 3D Stencils (Cut Weight = 32).

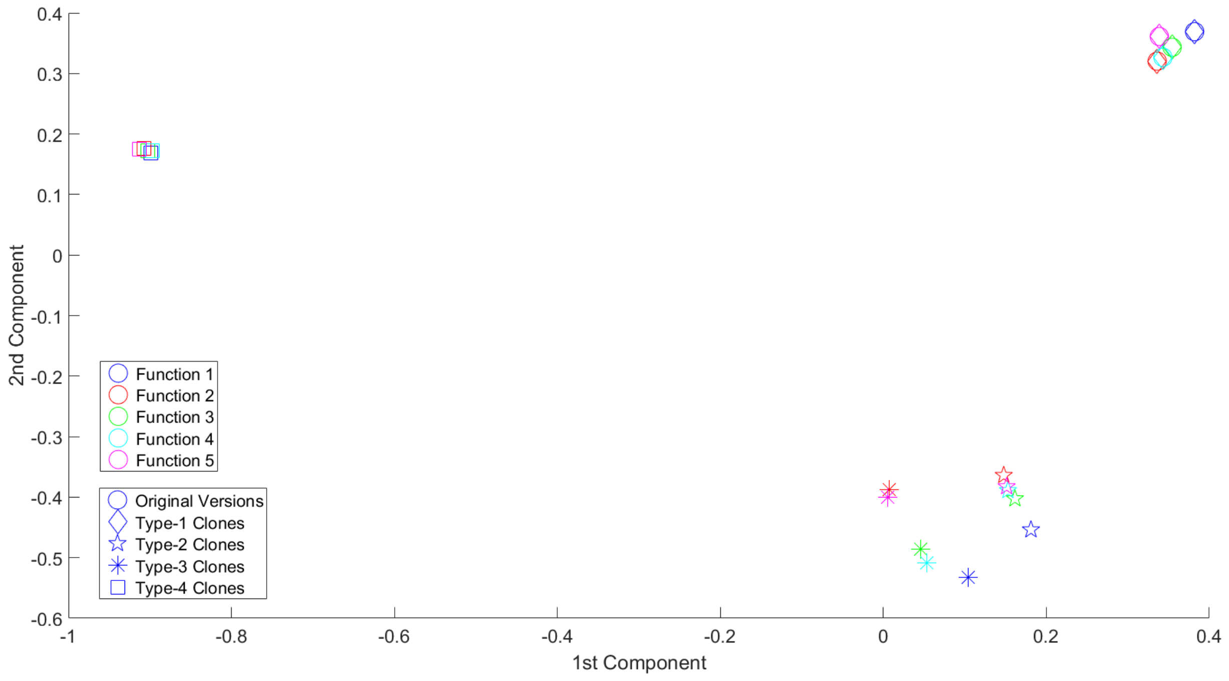


Figure 6.47: KPCA for Kast2 Spectrum Kernel using IRs of 3D Stencils (Cut Weight = 32).

Class D (2D Stencils): It shows a similar behavior presented by 3D stencils, but with more compact patterns (see Figures 6.48 and 6.49).

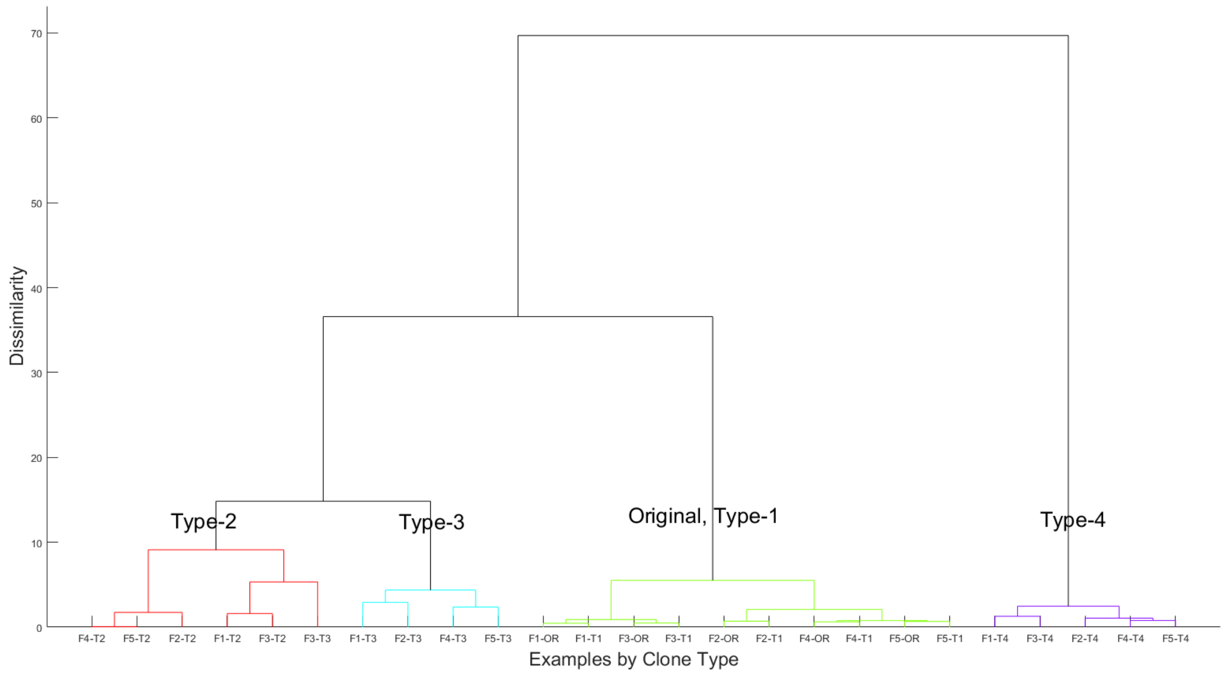


Figure 6.48: Hierarchical Clustering for Kast2 Spectrum Kernel using IRs of 2D Stencils (Cut Weight = 32).

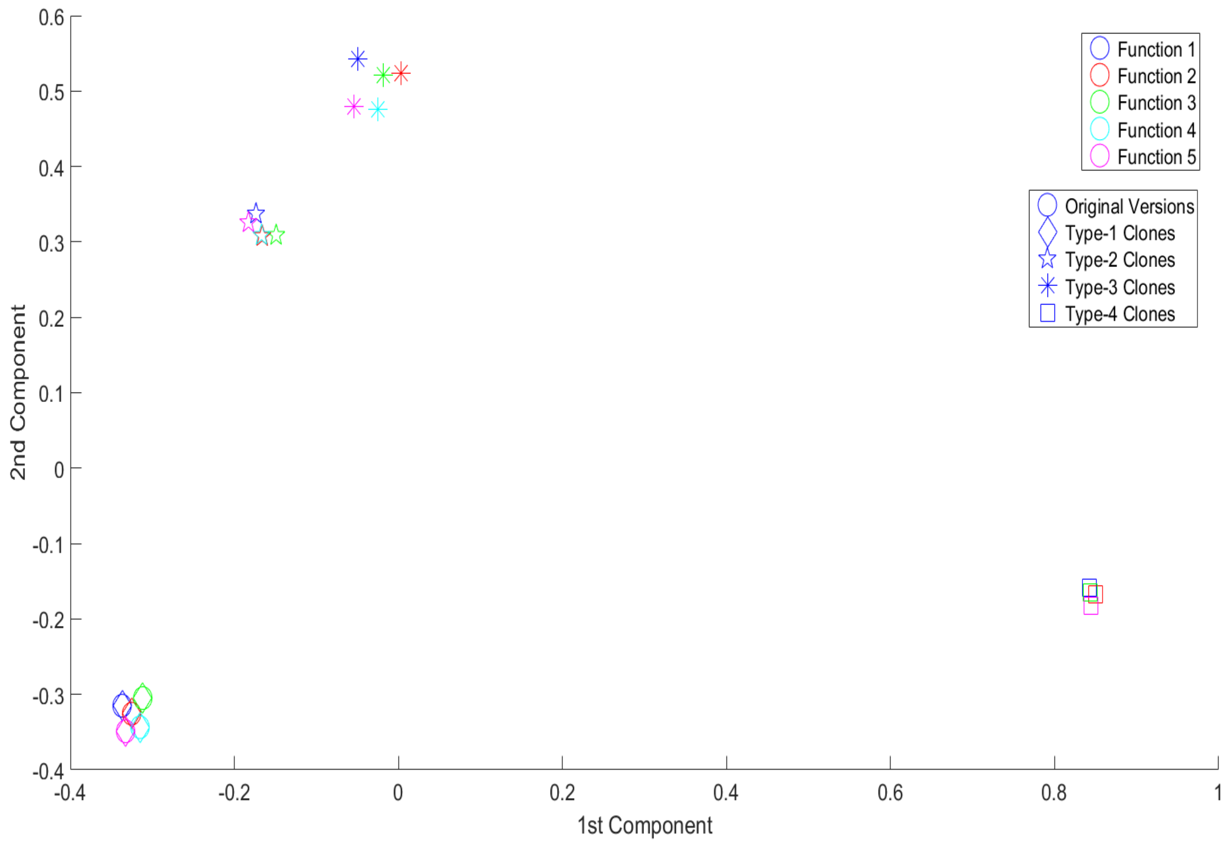


Figure 6.49: KPCA for Kast2 Spectrum Kernel using IRs of 2D Stencils (Cut Weight = 32).

6.5.3 Remarks

On the one hand, the application of the *kast2 spectrum kernel* applied over Clang ASTs allows the detection of function classes that reflect the nature of the code pieces; this combination also permits the detection of subclasses inside each class, as well as the depiction of a well structured organization of clones around the original versions. On the other hand, the usage of LLVM IRs supposes a downgrade in the intra-cluster detection capabilities, but instead achieves a general clustering with no misplaced examples.

Summary

*This chapter showed to the reader the application of the three proposed string kernel functions with strings coming from the intermediate representations of a popular compiler (LLVM), and their performance in comparison with the baseline kernel (**blended spectrum kernel**). Two of the proposed kernels presented a better performance than the baseline kernel, namely the **kast1** and **kast2 spectrum kernel**. This demonstrates that our contribution is suited for different domains, as in the previous chapter similar results were presented in the area of I/O pattern recognition. In the following chapter we have collected the conclusions of this work and possible ways of extending it in the future.*

7 Conclusions and Future Work

In this thesis we have shown how the I/O traces of a program, as well as its intermediate representations, can be represented as a string of weighted tokens, which subsequently are used to extract patterns. The resulting strings have been compared using three novel kernel functions proposed by the author. The kast, kast1 and kast2 spectrum kernels emit similarity matrices between examples that can be later analyzed by a clustering algorithm.

7.1 General Remarks

The use of strings with associated weights is one of the pillars of the new method that we have proposed, since these weights allow the compression of the information encoded in the strings, as well as the modulation of the feature value that characterizes and distinguishes each of the new kernels that we have developed.

One of the parameters of greater relevance for our comparison strategies is the cut weight. In the first instance, the higher is the cut value, the smaller is the number of comparisons made by the algorithm and therefore the speed of it increases; however, a smaller cut weight allows to consider short sequences in the contribution to the similarity of the strings, hence improving the pattern discovery performance. We have identified in our test cases that a cut weigh below 64 was a value that allowed to separate clusters in an adequate way.

In the specific case of synthetic strings, it has been found that our kernels do not perform better than baseline kernel; since the baseline kernel is oriented to take values lower than the cut weight, it stabilizes the clustering results once the longest matching substring is reached. Despite this, our kernels achieve the same clustering by decreasing the value of the cut weight, although this increases the computational cost.

Beyond this, in the two case studies of the real world presented in this research, the developed kernels show a better clustering performance than the baseline kernel.

7.2 Comparison of I/O traces

The method for converting the trees into strings has proved to be effective, thanks to: i) the hierarchical nesting via the definition of nodes like `[ROOT]`, `[HANDLE]` and `[BLOCK]`, ii) the reduction of the information via the simplification of adjacent nodes that shared names and number of bytes.

It has been on the interest of this thesis to study the suitability of the proposed strategy to find similarities among four different classes of I/O access patterns. For all proposed kernels, both hierarchical clustering and Kernel PCA have yielded similar results. The best results have been obtained when the string representation takes into account the byte information of the operations and the cut weight is small. It has been observed that the cut weight determines the granularity of the search, while the usage of the byte information permits the separation between examples of the same cluster. These findings clearly show that both the proposed string representation and the comparison method are suitable to compare I/O access patterns of a parallel application.

The baseline kernel (*blended spectrum kernel*) presents the worse performance, as it is able to identify only 2 patterns. On the contrary, the kernels proposed here have been able to identify 3 patterns:

- Class A (Flash I/O).
- Class B (POSIX I/O).
- Classes C and D (Sequential I/O).

The intrinsic similarity between classes C and D is captured by our kernels. The *kast spectrum kernel* works well only with strings using byte information and small weight values. The *kast1 spectrum kernel* works well with small weights, regardless of the byte information. However, the *kast2 spectrum kernel* performs better than all the other kernels, as the selection of the cut weight does not have a significant effect on the clustering. This means that the speed of the comparison could be increased with this kernel by selecting higher values of the cut weight.

7.3 Comparison of Intermediate Representations

It has been shown that the conversion of Clang Abstract Syntax Trees into strings is done in an intuitive way following a pre-order traversal. In the case of LLVM Intermediate Representations it was necessary to apply a preliminary step to convert the representation into tree to capture hierarchical relationships among the instructions.

With this experiment, we aimed to depict the efficacy of the string kernels at separating 4 different classes of functions written in C language. We have used separately their Clang ASTs and LLVM IRs as the source of information. Each function has been implemented in five different variants with the same functionality, to reflect the notion of clone types. All kernels, including the baseline kernel, have been able to detect the same scheme of clusters:

- Class A: String kernels.
- Class B: Sort functions.
- Classes C and D: 3D and 2D stencils.

The intuitive similarity of 3D and 2D stencil calculations has been clearly captured by all kernels. The expected ranking in distances for clone types with respect to the original version has been observed by all the kernels, excepting by the *kast spectrum kernel*. Two types of clones have exhibit the same patterns with all the kernels: *Type-1* clones are found almost overlapped with the original versions., while *Type-4* clones have the tendency to break the class clustering by conforming a cluster by themselves. In general, both representations (Clang ASTs and LLVM IRs) yield similar results. Regarding the clustering algorithm, Hierarchical Clustering has been more effective on the task than Kernel PCA.

The *kast spectrum kernel* does not offer a substantial advantage over the baseline kernel. While the *kast spectrum kernel* only shows the theoretical ranking in distances for clone types when using Clang ASTs, the baseline kernel shows this behavior with both representations.

On the contrary, the *kast1* and *kast2 spectrum kernels* exhibit a better cluster separation, evidencing that, when considering only the weights of the independent *valid matching substrings*, it is possible to obtain a better differentiation of the four types of clones and while maintaining a meaningful separation of the three major function classes: matching functions, sort functions and stencils. With the *kast1 spectrum kernel* there are no misplaced examples. Additionally, *Type-4* clones do not tend to create clusters outside their respective classes. With the *kast2 spectrum kernel*, both representations yield a similar cluster scheme. This kernel is the only one where Hierarchical Clustering and Kernel PCA are effective for separating classes.

These results indicate that these novel comparison methods can be promisingly utilized to find similarities in source code snippets.

7.4 Applications and Future Work

The proposed solution is susceptible to improvements that will pave the way for a broader application in other domains.

7.4.1 Improvements

We saw how the cut weight is a parameter that has to be defined by the user. Testing all the possible cut weights will only increase the complexity order of the algorithm. A future improvement for the comparison method is the automatic selection of this value. This could be done by some heuristics that could profile the strings and determine an acceptable cut weight according to the size of the data or its nature. Another way of automatically establishing the cut weight is to relate it to the minimum size in numbers of lines that the code or the I/O traces must have. For example, the user might decide that only code snippets with more than 20 lines are of his interest; the key here would be to determine the average weight that code snippets of that size might have.

From all the conversion strategies presented in this work, only the one related to LLVM IRs does not introduce a compression step. A further step would consist in the study of the effect of uncompressed strings for Clang ASTs and I/O traces in the final similarity score, as well as in the performance of the algorithm. This can be used as a guideline for the selection of the most adequate string representation form.

Another possibility of exploration is the comparison of the string kernel proposed here against other types of comparison methods that are not string kernels, e.g. tree kernels applied directly over the ASTs, or source metrics applied directly over the code. Further studies in these direction would assess the capabilities of the string kernels and would serve as guide for researchers that are constantly looking for new and better ways of comparing code.

In this thesis, the focus has been placed into the intermediate representations of C programs. However, given that the comparison objects are weighted strings, the ASTs or IRs generated by other languages could also be used. A possible study would try to determine how much does the similarity of the same program change when it is implemented in different languages, e.g. C vs Python, or Fortran vs Java.

Finally, one of the most promising extensions of this work could be related to the creation of new *kastx* kernels, and its corresponding experimentation.

7.4.2 Applications in Computer Science

One of the possible applications of these kernels is the implementation of a code smell detection plug-in inside a integrated development environment (IDE). The plug-in could verify if a particular piece of code indicated by the user, contains bad code that might affect maintainability or performance. The verification can be done against a well-studied collection of code pieces that exhibit good and bad programming code practices.

On the same direction, they may be utilized in the prediction of the presence of bad patterns inside server side I/O log files of an specific application in a distributed system. For example, on a first phase, a significant amount of I/O traces containing bad patterns could be automatically labeled as problematic by a benchmark. On a second stage, those patterns are converted into weighted strings and compared with any of the string kernels proposed in this work. As a result, a kernel matrix among the bad examples is obtained, with each row of it associated to a determined bad pattern. For new logs coming from the application, the respective kernel values will be calculated, and the respective label will be inferred using only the information from the kernel matrix.

7.4.3 Applications in Other Fields

Following the same idea, the string representation and the proposed kernels can be used for performing comparison of DNA sequences or to find similarity among data structures taken from chemical properties. For example, the molecular electrostatic potential (MEP) of a molecule can be represented as a tree [DMT13]. The trees of a collection of molecules with drug affinity can be flattened to the string representation that we proposed in this work, and posteriorly they can be compared and analyzed with our kernels, in order to determine which compound is a candidate as replacement of another compound. This could be of great interest to find alternative compounds for designing less expensive medicaments.

Publications

In order to share the findings of this work with the scientific community, we participated in several conferences and journals. The following is a list of the publications that resulted from that exercise:

- “A Novel String Representation and Kernel Function for the Comparison of I/O Access Patterns”, in Parallel Computing Technologies. Springer International Publishing, 2017, pp. 500–512.
- “Comparison of Clang Abstract Syntax Trees using String Kernels”, in International Conference on High Performance Computing and Simulation (HPCS 2018). IEEE, 2018, pp. 106-113.
- “A Similarity Study of I/O Traces via String Kernels”, in the Journal of Supercomputing, Special Issue: Parallel Computing Technologies 2018. Springer US, 2018, pp. 1-13.

Bibliography

[Agr+98] Rakesh Agrawal et al. "Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications". In: *SIGMOD Rec.* 27.2 (June 1998), pp. 94–105. ISSN: 0163-5808. DOI: `10.1145/276305.276314`.

[AlE+05] Raihan Al-Ekram et al. "Cloning by Accident: An Empirical Study of Source Code Cloning across Software Systems". In: *Across Software Systems.International Symposium on Empirical Software Engineering (ISESE'05.* 2005, pp. 376–385.

[Beh+14] Babak Behzad et al. "Automatic Generation of I/O Kernels for HPC Applications". In: *Proceedings of the 9th Parallel Data Storage Workshop.* PDSW '14. New Orleans, Louisiana: IEEE Press, 2014, pp. 31–36. ISBN: 978-1-4799-7025-4. DOI: `10.1109/PDSW.2014.6`. URL: `http://dx.doi.org/10.1109/PDSW.2014.6`.

[Beh+15] Babak Behzad et al. "Pattern-driven Parallel I/O Tuning". In: *Proceedings of the 10th Parallel Data Storage Workshop.* PDSW '15. Austin, Texas: ACM, 2015, pp. 43–48. ISBN: 978-1-4503-4008-3. DOI: `10.1145/2834976.2834977`.

[Bel+07] Stefan Bellon et al. "Comparison and Evaluation of Clone Detection Tools". In: *IEEE Trans. Softw. Eng.* 33.9 (Sept. 2007), pp. 577–591. ISSN: 0098-5589. DOI: `10.1109/TSE.2007.70725`. URL: `http://dx.doi.org/10.1109/TSE.2007.70725`.

[Bet14] Bradley Beth. *A Comparison of Similarity Techniques for Detecting Source Code Plagiarism.* 2014.

[BHS07] Gokhan BakIr, Thomas Hofmann, and Bernhard Scholkopf. *Predicting Structured Data.* Ed. by The MIT Press. The MIT Press, 2007.

[Bil05] Philip Bille. "A survey on tree edit distance and related problems". In: *Theoretical computer science* 337.1 (2005), pp. 217–239.

[Bro+00] S. Browne et al. "A Portable Programming Interface for Performance Evaluation on Modern Processors". In: *Int. J. High Perform. Comput. Appl.* 14.3 (Aug. 2000), pp. 189–204. ISSN: 1094-3420. DOI: `10.1177/109434200001400303`. URL: `http://dx.doi.org/10.1177/109434200001400303`.

[BS05] Arini Balakrishnan and Chloe Schulze. "Code obfuscation literature survey". In: *CS701 Construction of compilers* 19 (2005).

[BW13] Upul Bandara and Gamini Wijayarathna. “Source code author identification with unsupervised feature learning”. In: *Pattern Recognition Letters* 34.3 (2013), pp. 330–334. ISSN: 0167-8655. DOI: `http://dx.doi.org/10.1016/j.patrec.2012.10.027`.

[Byn+08] Surendra Byna et al. “Parallel I/O prefetching using MPI file caching and I/O signatures”. In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press. 2008, p. 44.

[Cha02] Moses S Charikar. “Similarity estimation techniques from rounding algorithms”. In: *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. ACM. 2002, pp. 380–388.

[Cor+10] A. Corazza et al. “A Tree Kernel based approach for clone detection”. In: *Software Maintenance (ICSM), 2010 IEEE International Conference on*. Sept. 2010, pp. 1–5. DOI: `10.1109/ICSM.2010.5609715`.

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. *A taxonomy of obfuscating transformations*. Tech. rep. Department of Computer Science, The University of Auckland, New Zealand, 1997.

[CV05] R. Cilibrasi and P. M.B. Vitanyi. “Clustering by Compression”. In: *IEEE Trans. Inf. Theor.* 51.4 (Apr. 2005), pp. 1523–1545. ISSN: 0018-9448. DOI: `10.1109/TIT.2005.844059`. URL: `http://dx.doi.org/10.1109/TIT.2005.844059`.

[CX12] Silvio Cesare and Yang Xiang. *Software Similarity and Classification*. Springer-Verlag London, 2012.

[Dea+03] Thomas R. Dean et al. “Agile Parsing in TXL”. In: *Automated Software Engineering* 10.4 (Oct. 2003), pp. 311–336. ISSN: 1573-7535. DOI: `10.1023/A:1025801405075`. URL: `https://doi.org/10.1023/A:1025801405075`.

[Dev01] Open64 Developers. *Open64 compiler and tools*. 2001.

[DMT13] Edgar E Daza, Julio Maza, and Raul Torres. “Molecular electrostatic potential as a graph”. In: *Current computer-aided drug design* 9.2 (2013), pp. 272–280.

[DS12] P. Danphitsanuphan and T. Suwantada. “Code Smell Detecting Tool and Code Smell-Structure Bug Relationship”. In: *Engineering and Technology (S-CET), 2012 Spring Congress on*. May 2012, pp. 1–5. DOI: `10.1109/SCET.2012.6342082`.

[DW15] Shilpa Dang and Shahid Ahmad Wani. “Performance evaluation of clone detection tools”. In: *International Journal of Science and Research (IJSR)* (2015), pp. 1903–1906.

[Fen+15] Bo Feng et al. “IOSIG+: on the Role of I/O Tracing and Analysis for Hadoop Systems”. In: *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*. IEEE. 2015, pp. 62–65.

[Fry+00] B Fryxell et al. “FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes”. In: *The Astrophysical Journal Supplement Series* 131.1 (2000), p. 273.

[Fu+16] Deqiang Fu et al. “WASTK: An Weighted Abstract Syntax Tree Kernel Method for Source Code Plagiarism Detection”. In: *Scientific Programming Volume 2017 (2017), Article ID 7809047, 8 pages* (2016).

[Gei+08] Markus Geimer et al. “The SCALASCA Performance Toolset Architecture”. In: *Proc. of the International Workshop on Scalable Tools for High-End Computing (STHEC), Kos, Greece.* June 2008, pp. 51–65.

[Gen01] Marc G Genton. “Classes of kernels for machine learning: a statistics perspective”. In: *Journal of machine learning research* 2.Dec (2001), pp. 299–312.

[GLF04] Thomas Gartner, John W. Lloyd, and Peter A. Flach. “Kernels and Distances for Structured Data”. In: *Machine Learning* 57 (3 2004), pp. 205–232. ISSN: 0885-6125. URL: `http://dx.doi.org/10.1023/B:MACH.0000039777.23772.30`.

[God09] Nils Gode. “Evolution of Type-1 Clones”. In: *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation.* SCAM ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 77–86. ISBN: 978-0-7695-3793-1. DOI: `10.1109/SCAM.2009.17`. URL: `http://dx.doi.org/10.1109/SCAM.2009.17`.

[Gop+11] Siddharth Gopal et al. “Statistical Learning for File-Type Identification”. In: *Proceedings of the 2011 10th International Conference on Machine Learning and Applications and Workshops - Volume 01.* ICMLA ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 68–73. ISBN: 978-0-7695-4607-0. DOI: `10.1109/ICMLA.2011.135`.

[Gri02] Arthur Griffith. *GCC: The Complete Reference.* 1st ed. New York, NY, USA: McGraw-Hill, Inc., 2002. ISBN: 0072224053.

[GT99] David Gitchell and Nicholas Tran. “Sim: A Utility for Detecting Similarity in Computer Programs”. In: *SIGCSE Bull.* 31.1 (Mar. 1999), pp. 266–270. ISSN: 0097-8418. DOI: `10.1145/384266.299783`.

[Gun+98] Steve R Gunn et al. “Support vector machines for classification and regression”. In: *ISIS technical report* 14.1 (1998), pp. 5–16.

[Hau99] David Haussler. *Convolution Kernels on Discrete Structures.* Tech. rep. University of California at Santa cruz, 1999.

[HP14] Seong-Bae Park Hyun-Je Song and Se Young Park. “Computation of Program Source Code Similarity by Composition of Parse Tree and Call Graph”. In: *Mathematical Problems in Engineering* (2014).

[HRV10] Jurriaan Hage, Peter Rademaker, and Nike van Vugt. “A comparison of plagiarism detection tools”. In: *Utrecht University. Utrecht, The Netherlands* 28 (2010).

[HTF03] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning*. Corrected. Springer, July 2003. ISBN: 0387952845. URL: `http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20%5C&path=ASIN/0387952845`.

[Inf17] The LLVM Compiler Infrastructure. *Clang: A C language family frontend for LLVM*. May 2017. URL: `https://clang.llvm.org/index.html`.

[J Q00] Daniel J. Quinlan. “ROSE: Compiler Support for Object-Oriented Frameworks.” In: 10 (June 2000), pp. 215–226.

[Jia+07] Lingxiao Jiang et al. “DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones”. In: *Proceedings of the 29th International Conference on Software Engineering*. ICSE ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 96–105. ISBN: 0-7695-2828-7. DOI: `10.1109/ICSE.2007.30`. URL: `http://dx.doi.org/10.1109/ICSE.2007.30`.

[JMF99] A. K. Jain, M. N. Murty, and P. J. Flynn. “Data clustering: a review”. In: *ACM Comput. Surv.* 31.3 (Sept. 1999), pp. 264–323. ISSN: 0360-0300. DOI: `10.1145/331499.331504`.

[JPl17] JPlag. *JPlag: Detecting software plagiarism*. May 2017. URL: `https://jplag.ipd.kit.edu/`.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. “CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code”. In: *IEEE Trans. Softw. Eng.* 28.7 (July 2002), pp. 654–670. ISSN: 0098-5589. DOI: `10.1109/TSE.2002.1019480`. URL: `http://dx.doi.org/10.1109/TSE.2002.1019480`.

[Klu11] Michael Kluge. “Comparison and End-to-End Performance Analysis of Parallel Filesystems”. PhD thesis. Technische Universität Dresden, 2011.

[KR10] Ricardo Koller and Raju Rangaswami. “I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance”. In: *Trans. Storage* 6.3 (Sept. 2010), 13:1–13:26. ISSN: 1553-3077. DOI: `10.1145/1837915.1837921`.

[KR87] Richard M Karp and Michael O Rabin. “Efficient randomized pattern-matching algorithms”. In: *IBM Journal of Research and Development* 31.2 (1987), pp. 249–260.

[Kri01] Jens Krinke. “Identifying similar code with program dependence graphs”. In: *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on.* IEEE. 2001, pp. 301–309.

[Kun+14] Julian M. Kunkel et al. "The SIOX Architecture – Coupling Automatic Monitoring and Optimization of Parallel I/O". In: *Supercomputing: 29th International Conference, ISC 2014, Leipzig, Germany, June 22-26, 2014. Proceedings.* Ed. by Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer. Cham: Springer International Publishing, 2014, pp. 245–260. ISBN: 978-3-319-07518-1. DOI: `10.1007/978-3-319-07518-1_16`. URL: `https://doi.org/10.1007/978-3-319-07518-1_16`.

[Kun13] Julian Martin Kunkel. "Simulation of Parallel Programs on Application and System Level". eng. PhD thesis. Von-Melle-Park 3, 20146 Hamburg: Universität Hamburg, 2013. URL: `http://ediss.sub.uni-hamburg.de/volltexte/2013/6264`.

[Kun14] S.Y. Kung. *Kernel Methods and Machine Learning.* Cambridge University Press, 2014. ISBN: 9781139867634. URL: `https://books.google.de/books?id=HXQ9AwAAQBAJ`.

[LA04] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04).* Palo Alto, California, Mar. 2004.

[LA14] Bruno Cardoso Lopes and Rafael Auler. *Getting Started with LLVM Core Libraries.* Packt Publishing, 2014. ISBN: 1782166920, 9781782166924.

[LEN02] C. Leslie, E. Eskin, and W. S. Noble. "The spectrum kernel: A string kernel for SVM protein classification". In: *Proceedings of the Pacific Symposium on Biocomputing.* Vol. 7. 2002, pp. 566–575.

[Lev66] Vladimir I Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals". In: *Soviet physics doklady.* Vol. 10. 8. 1966, pp. 707–710.

[Liu+14] Yang Liu et al. "Automatic Identification of Application I/O Signatures from Noisy Server-Side Traces". In: *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14).* Santa Clara, CA: USENIX, 2014, pp. 213–228. ISBN: ISBN 978-1-931971-08-9. URL: `https://www.usenix.org/conference/fast14/technical-sessions/presentation/liu`.

[LMM12] William Loewe, T McLarty, and C Morrone. *IOR benchmark.* 2012.

[Luo+15] Xiaoqing Luo et al. "Hpc i/o trace extrapolation". In: *Proceedings of the 4th Workshop on Extreme Scale Programming Tools.* ACM. 2015, p. 2.

[LV02] Yihua Liao and V.Rao Vemuri. "Use of K-Nearest Neighbor classifier for intrusion detection". In: *Computers & Security* 21.5 (2002), pp. 439–448. ISSN: 0167-4048. DOI: `http://dx.doi.org/10.1016/S0167-4048(02)00514-X`.

[MKZ06] Hermann A Maurer, Frank Kappe, and Bilal Zaka. "Plagiarism-a survey." In: *J. UCS* 12.8 (2006), pp. 1050–1084.

[MR02] Tara M. Madhyastha and Daniel A. Reed. “Learning to classify parallel input/output access patterns”. In: *IEEE Transactions on Parallel and Distributed Systems* 13.8 (2002), pp. 802–813.

[MVL03] Mika Mäntylä, Jari Vanhanen, and Casper Lassenius. “A Taxonomy and an Initial Empirical Study of Bad Smells in Code”. In: *Proceedings of the International Conference on Software Maintenance*. ICSM ’03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 381–. ISBN: 0-7695-1905-9. URL: `http://dl.acm.org/citation.cfm?id=942800.943571`.

[NFH05] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. “Understanding Source Code Evolution Using Abstract Syntax Tree Matching”. In: *SIGSOFT Softw. Eng. Notes* 30.4 (May 2005), pp. 1–5. ISSN: 0163-5948. DOI: `10.1145/1082983.1083143`. URL: `http://doi.acm.org/10.1145/1082983.1083143`.

[NMC16] Ricardo Nobre, Luiz G. A. Martins, and João M. P. Cardoso. “A Graph-based Iterative Compiler Pass Selection and Phase Ordering Approach”. In: *SIGPLAN Not.* 51.5 (June 2016), pp. 21–30. ISSN: 0362-1340. DOI: `10.1145/2980930.2907959`. URL: `http://doi.acm.org/10.1145/2980930.2907959`.

[Par15] Terence Parr. *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf, Dec. 11, 2015. 328 pp. ISBN: 1934356999. URL: `http://www.ebook.de/de/product/19253876/terence_parr_the_definitive_antlr_4_reference.html`.

[PCA12] Eunjung Park, John Cavazos, and Marco A. Alvarez. “Using Graph-based Program Characterization for Predictive Modeling”. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. CGO ’12. San Jose, California: ACM, 2012, pp. 196–206. ISBN: 978-1-4503-1206-6. DOI: `10.1145/2259016.2259042`. URL: `http://doi.acm.org/10.1145/2259016.2259042`.

[PMP02] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. “Finding plagiarisms among a set of programs with JPlag”. In: *J. UCS* 8.11 (2002), p. 1016.

[PS15] Mayur Pandey and Suyog Sarda. *LLVM Cookbook*. Packt Publishing, 2015.

[Rat88] John W. Ratclif. *Pattern Matching: the Gestalt Approach*. July 1988. URL: `http://www.drdobbs.com/database/pattern-matching-the-gestalt-approach/184407970`.

[RC08] Chanchal K Roy and James R Cordy. “NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization”. In: *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. IEEE. 2008, pp. 172–181.

[RCK09] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach”. In: *Science of Computer Programming* 74.7 (2009), pp. 470–495. ISSN: 0167-6423. DOI: `https://doi.org/10.1016/j.scico.2009.02.007`. URL: `http://www.sciencedirect.com/science/article/pii/S0167642309000367`.

[RKC17] Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. “A comparison of code similarity analysers”. In: *Empirical Software Engineering* (Oct. 2017). ISSN: 1573-7616. DOI: `10.1007/s10664-017-9564-7`. URL: `https://doi.org/10.1007/s10664-017-9564-7`.

[SC04] John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis*. New York, NY, USA: Cambridge University Press, 2004. ISBN: 0521813972.

[SF83] Alberto Sanfeliu and King-Sun Fu. “A distance measure between attributed relational graphs for pattern recognition”. In: *IEEE transactions on systems, man, and cybernetics* 3 (1983), pp. 353–362.

[SPP07] Alok Sharma, Arun K. Pujari, and Kuldip K. Paliwal. “Intrusion detection using text processing techniques with a kernel based similarity measure”. In: *Computers & Security* 26.7–8 (2007), pp. 488–495. ISSN: 0167-4048. DOI: `http://dx.doi.org/10.1016/j.cose.2007.10.003`.

[SR14] Jeffrey Svajlenko and Chanchal K. Roy. “Evaluating Modern Clone Detection Tools”. In: *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*. ICSME ’14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 321–330. ISBN: 978-1-4799-6146-7. DOI: `10.1109/ICSME.2014.54`. URL: `http://dx.doi.org/10.1109/ICSME.2014.54`.

[SSM97] Bernhard Scholkopf, Alexander Smola, and Klaus-Robert Muller. “Kernel principal component analysis”. In: *Artificial Neural Networks ICANN97*. Vol. 1327. Lecture Notes in Computer Science. 10.1007/BFb0020217. Springer Berlin Heidelberg, 1997, pp. 583–588. URL: `http://dx.doi.org/10.1007/BFb0020217`.

[SWA03] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. “Winnowing: Local Algorithms for Document Fingerprinting”. In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’03. New York, NY, USA: ACM, 2003, pp. 76–85. ISBN: 1-58113-634-X. DOI: `10.1145/872757.872770`.

[TC11] Linda Torczon and Keith Cooper. *Engineering A Compiler*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012088478X.

[THW10] Jan Treibig, Georg Hager, and Gerhard Wellein. “LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments”. In: *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*. ICPPW ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 207–216. ISBN: 978-0-7695-4157-0. DOI: `10.1109/ICPPW.2010.38`. URL: `http://dx.doi.org/10.1109/ICPPW.2010.38`.

[Tik+10] Domonkos Tikk et al. “A Comprehensive Benchmark of Kernel Methods to Extract Protein–Protein Interactions from Literature”. In: *PLoS Computational Biology* (2010).

[Tor11] R. Torres. “Parallel Computing System for the efficient calculation of molecular similarity based on negative electrostatic potential”. MA thesis. Universidad Nacional de Colombia, 2011.

[Udd+11] Md Sharif Uddin et al. “On the effectiveness of simhash for detecting near-miss clones in large scale software systems”. In: *Reverse Engineering (WCRE), 2011 18th Working Conference on*. IEEE. 2011, pp. 13–22.

[VBR16] Tijana Vislavski, Zoran Budimac, and Gordana Rakic. “Towards the Code Clone Analysis in Heterogeneous Software Products”. In: *Proceedings of the Fifth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, Budapest, Hungary, August 29-31, 2016.* 2016, pp. 89–96. URL: `http://ceur-ws.org/Vol-1677/paper11.pdf`.

[VS03] S. V. N. Vishwanathan and Alexander J. Smola. “Fast Kernels for String and Tree Matching”. In: *Advances in Neural Information Processing Systems 15*. MIT Press, 2003, pp. 569–576. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.4.9887`.

[Wan+15] Chunyan Wang et al. “Identification and elimination of platform-specific code smells in high performance computing applications”. In: *International Journal of Networking and Computing* 5.1 (2015), pp. 180–199.

[Wei73] Peter Weiner. “Linear Pattern Matching Algorithms”. In: *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973)*. SWAT ’73. Washington, DC, USA: IEEE Computer Society, 1973, pp. 1–11. DOI: `10.1109/SWAT.1973.13`. URL: `https://doi.org/10.1109/SWAT.1973.13`.

[Wis93] Michael J Wise. “String similarity via greedy string tiling and running Karp-Rabin matching”. In: *Online Preprint, Dec* 119 (1993).

[YB07] Li Yujian and Liu Bo. “A normalized Levenshtein distance metric”. In: *IEEE transactions on pattern analysis and machine intelligence* 29.6 (2007), pp. 1091–1095.

[Zha+08] Min Zhang et al. “Semantic Role Labeling Using a Grammar-Driven Convolution Tree Kernel”. In: *Audio, Speech, and Language Processing, IEEE Transactions on* 16.7 (Sept. 2008), pp. 1315–1329. ISSN: 1558-7916. DOI: `10.1109/TASL.2008.2001104`.

List of Figures

List of Listings

List of Tables

Eidesstattliche Versicherung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Dissertationsschrift selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ort, Datum	Unterschrift