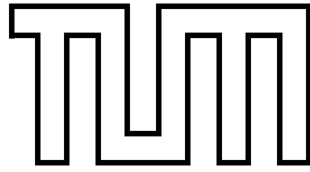


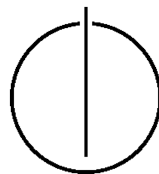

FAKULTÄT FÜR INFORMATIK

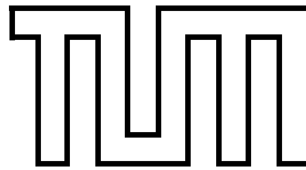
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN
Lehrstuhl IV
Software & Systems Engineering


Bachelor's Thesis in Informatik

# Index-based Model Clone Detection


Daniela Steidl




FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

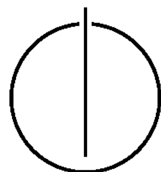
Lehrstuhl IV
Software & Systems Engineering


Bachelor's Thesis in Informatik

# Index-based Model Clone Detection

# Indexbasierte Erkennung von Modell-Klonen


| | |
|---|---|
| Author: | Daniela Steidl |
| Supervisor: | Prof. Dr. Dr. h.c. Manfred Broy |
| Advisor: | Benjamin Hummel |
| Date: | September $3^{th}$ 2010 |

I assure the single handed composition of this bachelor's thesis only supported by the declared resources.

München, September $3^{th}$ 2010 Daniela Steidl

# Abstract

In this work a novel index-based algorithm for clone detection in graph-based data-flow models is presented. Clone detection in graphs is the enumeration of all maximal, isomorphic, disjunctive, connected subgraphs. Our algorithm first enumerates all connected subgraphs of a given size, groups them according to isomorphism and then tries to find groups of maximal isomorphic subgraphs. As the problem of clone detection in graphs is known to be NP-complete, we cannot expect to find a polynomial solution. However, we present an algorithm with feasible runtime in practice, which is not complete, but produces satisfying results. By contrast to existing algorithms, our algorithm is both incremental and distributable, because it is index-based. We will show that the calculation of the index can be easily distributed over several machines and that the index can perform incremental updates efficiently. Large models in practice consist of several files. If only one of them changes, the index can be updated instead of running the entire clone detection algorithm again, which saves waiting time for software developers.

In this work, a detailed description of the algorithm can be found, as well as a brief runtime analysis and an evaluation on test models. We present the evaluation of our algorithm on graph-based models which were created with the common modeling tool MATLAB/-Simulink. Furthermore, we compare our algorithm with an existing algorithm for clone detection in graph-based models on several test models.

# Contents

**Abstract** iv

**1 Introduction** 1
1.1 Occurence of Clones . . . 1
1.2 Problems Caused by Clones . . . 1
1.3 Goal of Clone Detection in Graph-based Models . . . 2
1.4 Model-based Development Tools . . . 2
1.5 Contribution . . . 3
1.6 Outline . . . 5

**2 Basic Principles** 6
2.1 General Mathematical Definitions in Graphs . . . 6
2.2 Problem Definition of Clone Detection . . . 7
2.3 Preprocessing and Normalization . . . 7
2.4 Complexity of Clone Detection . . . 8

**3 Related Work** 9
3.1 Graph-based Clone Detection . . . 9
3.1.1 Heuristic Approach . . . 9
3.1.2 Exact and Approximate Clone Detection . . . 10
3.2 Index-based Code Clone Detection . . . 11

**4 Approach** 12
4.1 Architecture of the Algorithm . . . 12
4.2 Enumerating Subgraphs of Given Size . . . 13
4.3 Creating Clone Groups by Using Canonical Labeling . . . 15
4.3.1 Canonical Labeling . . . 15
4.3.2 Creating Clone Groups . . . 16
4.4 Merging Clone Groups . . . 16
4.4.1 Overview of Finding Maximal Clone Groups . . . 16
4.4.2 Key Generation for a Clone Group . . . 17
4.4.3 Merging Two Clone Groups with a Common Key and Equal Clone Pair Size . . . 21
4.4.4 Merging Two Clone Groups with a Common Key and Different Clone Pair Sizes . . . 22
4.4.5 Merging a Set of Clone Groups with a Common Key . . . 23
4.4.6 Finding Maximal Clone Groups . . . 23
4.4.7 Ordering of the Input . . . 26
4.5 Filtering Disjunctive Clone Groups . . . 27

4.6 Incremental Updates and Distributable Calculations . . . . . . . . . . . . . . 28

**5 Analysis of Complexity** **29**
5.1 Complexity of the Enumeration of Subgraphs of a Given Size . . . . . . . . 29
5.2 Complexity of Hashing with Canonical Labels . . . . . . . . . . . . . . . . . . 30
5.3 Complexity of Finding Maximal Clone Groups . . . . . . . . . . . . . . . . . 30

**6 Case Studies and Comparison to an Existing Algorithm** **33**
6.1 Test Models . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
6.2 Evaluation Criteria . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
6.3 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
6.4 Runtime . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
6.5 Memory Requirements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
6.6 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

**7 Conclusion and Future Work** **39**

**Bibliography** **40**

# 1 Introduction

In many situations during the development of a software system, software developers copy and paste parts of their code. In general the habit of copy-and-paste creates duplicated fragments in software systems, called *clones*. Clones are considered to be harmful by a large group among software engineers for several reasons. Therefore detecting clones in different kinds of software developing environments is an active current research topic. This chapter is an introduction to clone detection: It reveals the occurrence and the problems of clones, introduces to clone detection in graph-based models as well as to the main development tools and emphasizes the contribution of this work.

## 1.1 Occurence of Clones

Generally, fragments in software systems that are similar with respect to some definition of similarity [1] are called *clones*. Clones appear in several different environments according to current research, for example in classical programming languages or in model-based software development (MBD). In classical programming languages clones appear as duplicated code fragments, in MBDs as redundant model elements. In both environments, many programs contain a significant amount of duplicated code. This work focuses mainly on clone detection in graph-based data-flow models, which are created during a model-based software development process. In this area, only few clone detection algorithms exist. To explain the underlying idea of our algorithm, we will also study some existing algorithms to detect clones in code based environments.

## 1.2 Problems Caused by Clones

In the opinion of many software engineers, clones cause problems independent from the environment they appear in. They argue that cloning reduces productivity of software maintenance, because clones typically implement a common concept, which is used multiple times in the same code base. Hence a change to this concept needs to be carried out several times in all instances of the clone, which might be unknown. As the localization and consistent modification of all duplicates of a code fragment is very expensive, cloning potentially increases the maintenance effort. Additionally, redundant code increases the size of the program and thereby increases the maintenance effort which directly depends on the program volume in many cases. An empirical indication of the negative impact of cloning on maintainabilitiy can be found in [2]. Moden et al. analyze the modification history of a large COBOL legacy software system. They report that modules which contain clones have been significantly modified more often than modules without clones. Besides the higher expectation of maintenance costs, clones are a potential source for bugs, if not

all impacted clones are changed consistently. [3] discovers a large number of bugs uncovered by analyzing inconsistencies between code clones in open source projects.

Hence the field of clone detection has become an important area of research in the software engineering community. The general purpose of clone detection is to reveal the existence of duplicated code fragments. This helps software developers to either implement the common concept and remove clones or to manage existing clones and consistently carry out changes to all impacted clones.

## 1.3 Goal of Clone Detection in Graph-based Models

A clone consists of different redundant program elements, as mentioned above. In a graph model, a clone consists of subgraphs that are isomorphic. As software developers are interested in the biggest common concept of duplicated model elements rather than numerous small overlapping duplications, we aim to find clones of maximal size. To put it in a nutshell, clone detection in graphs is the enumeration of all maximal subgraphs, that are isomorphic. In Section 2, we will give a precise definition of the problem of clone detection in graphs and show that this problem is known to be NP-complete. Hence we can not expect to find an efficient (polynomial time) algorithm that enumerates all maximal clones.

## 1.4 Model-based Development Tools

To perform clone detection in model-based software development, the underlying development tool must be specified. For the purpose of this work, we assume to work only on graph-based data-flow models obtained from the model-based development tool MATLAB/-Simulink. MATLAB/Simulink is a commonly used tool, for example for models in the automotive sector of embedded systems. With MATLAB/Simulink you can model block diagrams consisting of blocks and lines to represent control engineering systems. Thus, blocks represent functions transforming input signals to output signals, lines represent signals that are exchanged between blocks. MATLAB/Simulink models can be interpreted as descriptions of time and value discrete control algorithms. Models in MATLAB/Simulink

(a) Entire model

(b) Subsystem on next lower level

Figure 1.1: Hierarchical modeling structure of MATLAB/Simulink

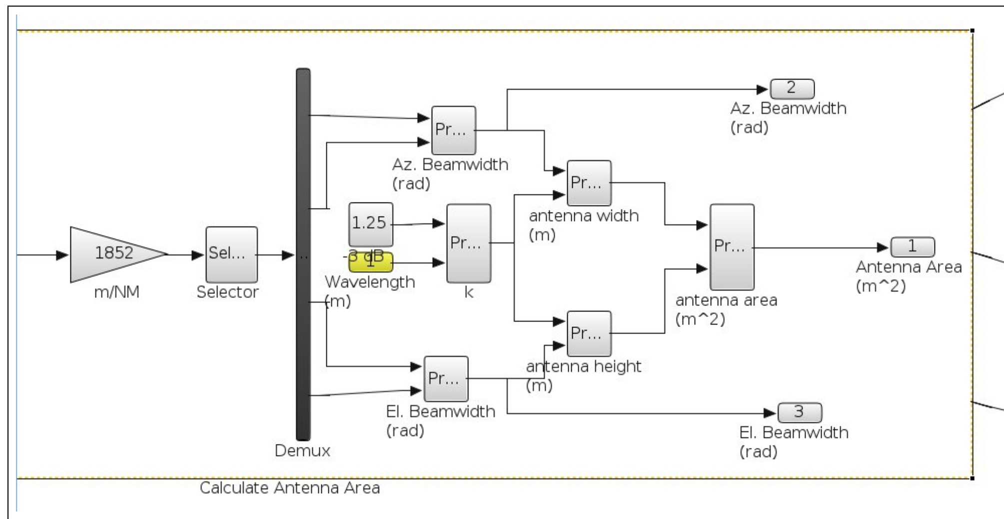



Figure 1.2: MATLAB/Simulink model on the lowest level without further subsystems

are organized in a recursive hierarchical structure with arbitrary many levels. A block in one level can either represent a simple function or a more complex subsystem on the next lower level. In both cases, a block represents a function with specified input and output ports. A specific hierarchical structure of a model in MATLAB/Simulink can be found in Figure 1.1 and 1.2. Figure 1.1 a) shows the entire model on the highest level of the model. Filled rectangles with a dotted boundary thereby represent a subsystem. One of the subsystems can be found in Figure 1.1 b). On the next lower level, this subsystem contains subsystems itself. A subsystem without further subsystems on the lowest level of the model can be found in Figure 1.2.

## 1.5 Contribution

A couple of algorithms have been developed for clone detection in code-based environments. However, clone detection algorithms in the context of model-based development are rare. In this section, we show the contribution of this work for detecting clones in model-based environments.

In the previous section, we introduced the underlying modeling tool MATLAB/Simulink. To extract a graph from a MATLAB/Simulink model, we use existing preprocessing and normalization methods from [4]. Preprocessing and normalization results in a directed labeled multi graph, where nodes correspond to blocks in a MATLAB/Simulink model and edges correspond to lines. The output graph only consists of information that is considered to be relevant for clone detection. More detailed information about preprocessing and normalization is presented in 2.3. Hence, this work does not improve any preprocessing or normalization methods.

The main contribution is a novel index-based algorithm for the problem of clone detection in graph-based data-flow models. The input of the algorithm is a directed labeled multi-graph, which is the result of preprocessing and normalization of MATLAB/Simulink models,

as described above. The output is a list of clones, e.g. groups of maximal subgraphs that are pairwise isomorphic. The algorithm consists of three steps. First, all subgraphs of a given size $k$ in the input graph are enumerated. Then subgraphs are inserted into a hash table with their canonical label as key. We use canonical labeling (described in detail in Section 4.3), because two subgraphs have the same canonical label if and only if they are isomorphic. Therefore the hash table consists of lists of isomorphic subgraphs, which form a group of clones. In the last step, clone groups are merged to find clones with maximal size. An abstraction of the three steps of the algorithm is shown in Figure 1.3. For reasons of simplicity, graphs are visualized as undirected, unlabeled graphs, although they are directed and labeled in reality.

The list of subgraphs with a given size $k$ is called the *index*. Existing algorithms read the entire model and detect all clones in a single step. Therefore, if the model changes, the entire detection has to be performed again. Our algorithm, by contrast, is able to perform incremental updates of the clone detection. If one file of the model changes, only the subgraphs of the model in this file need to be updated in the index. Subgraphs from other files remain unchanged. This makes our algorithm more efficient than existing algorithms, because incremental updates save waiting times. Furthermore, the calculation of the index can be distributed on several machines. All subgraphs of a model in a single file can be calculated on a separate machine. Afterward, all lists of subgraphs are simply merged to create the index.

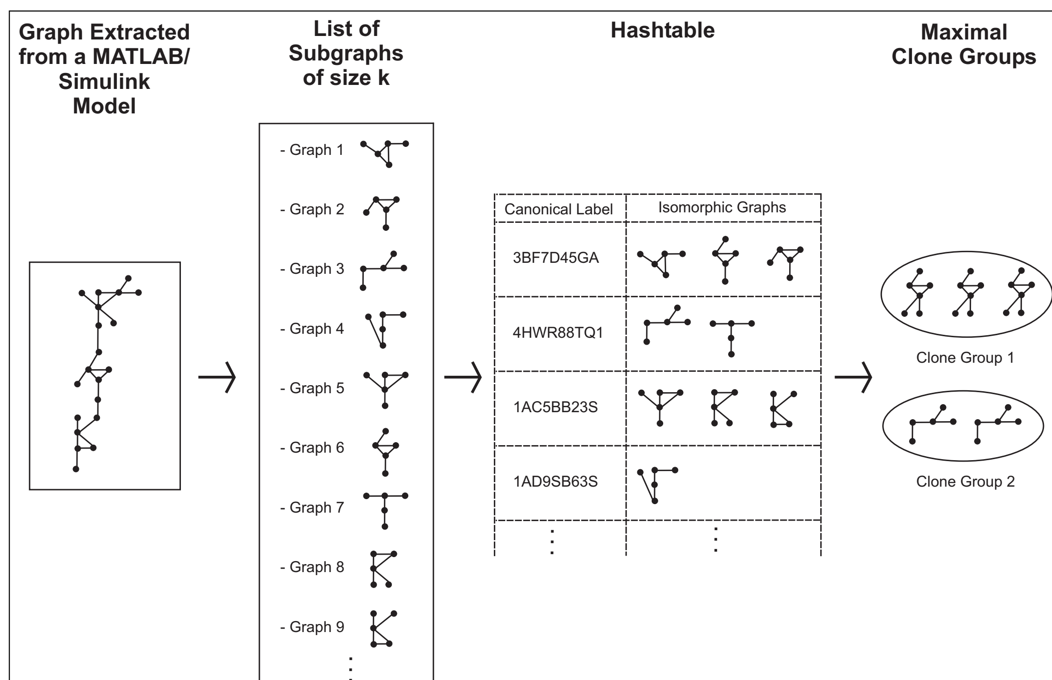



Figure 1.3: Overview of our new approach

## 1.6 Outline

This work is organized as follows: First, basic principles to understand clone detection are presented in Chapter 2, as well as our own specific problem definition. Then related research is presented in Chapter 3, both for code-based and model-based developments. In Chapter 4 our approach is explained in detail. Chapter 5 analyzes the complexity of the algorithm and Chapter 6 compares the algorithm with an existing algorithm on various test models. Chapter 7 shows our conclusion and future work.

# 2 Basic Principles

This chapter includes all definitions used for clone detection in this work. It first gives an overview over basic mathematical definitions in graph theory (Section 2.1) and then presents our specific definition for the problem of clone detection (Section 2.2). Section 2.3 and 2.4 show details of preprocessing and normalization methods and proof that clone detection is NP-complete.

## 2.1 General Mathematical Definitions in Graphs

In this section, we present the basic definitions in graph theory, that are used in this work. With the help of these definitions we can define the problem of clone detection in Section 2.2.

To define a clone we use the following fundamental definitions for graphs:

**Definition 2.1 (Undirected Graph)** *An* undirected graph *$G$ is a pair $G = (V, E)$ consisting of a finite set $V \neq \emptyset$ and a set $E$ of two-element subsets of $V$. The elements of $V$ are called* nodes. *An element $e = \{a, b\}$ is called an* edge *with* end nodes *$a$ and $b$. We say that $a$ and $b$ are* incident *with $e$ and that $a$ and $b$ are* adjacent *or neighbors of each other. The* size of a graph *denotes the number of nodes $|V|$.*

**Definition 2.2 (Directed Graph)** *A* directed graph *$G$ is a pair $G = (V, E)$ consisting of a finite set $V \neq \emptyset$ and a set $E \subseteq V \times V$ of directed edges.*

**Definition 2.3 (Labeled Graph)** *A directed (undirected)* labeled graph *$G = (V, E, L)$ is a directed (undirected) graph $G = (V, E)$ with a labeling function $L : V \cup E \to N$ for some set $N$ of labels.*

**Definition 2.4 (Multi Graph)** *A directed (undirected) labeled* multi graph *$G = (V, E, L)$ is a directed (undirected) graph $G = (V, E)$ where the set $E$ of edges is a multi set.*

**Definition 2.5 (Graph isomorphism)** *Two Graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are* isomorphic *iff there exists a bijection $f : V_1 \to V_2$ such that $\{x, y\} \in E_1 \iff \{f(x), f(y)\} \in E_2$.*

**Definition 2.6 (Graph isomorphism on labeled graphs)** *Two Graphs $G_1 = (V_1, E_1, L)$ and $G_2 = (V_2, E_2, L)$ are* isomorphic *iff there exists a bijection $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ such that for each $v \in V_1$ it holds $L(v) = L(f_V(v))$ and for each $e = (x, y) \in E_1$ it is both $L(e) = L(f_E(e))$ and $(f_V(x), f_V(y)) = f_E(e)$.*

**Definition 2.7 (Connectivity and $u-v-$path)** *An undirected Graph $G = (V, E)$ is called* connected *iff for each pair of nodes $u, v$ there exists a $u$-$v$-path in $G$. A $u$-$v$-*path *is a sequence of nodes $W = (u, w_1, ..., w_n, v)$ such that there are edges $e_1 = \{u, w_1\}, e_i = \{w_{i-1}, w_i\}\ \forall i = 2..n, e_{n+1} = \{w_n, v\}$ and all nodes are mutually distinct.*

**Definition 2.8 (Weak connectivity)** *A directed Graph $G = (V, E)$ is* weakly connected *if replacing all of its directed edges with undirected edges produces a connected (undirected) graph.*

## 2.2 Problem Definition of Clone Detection

With the definitions in Section 2.1, we can define a *clone* and the problem of clone detection in graph-based models:

**Definition 2.9 (Clone Pair)** *A* clone pair *is a pair of two weakly connected, directed, labeled multi graphs $G$ and $G'$ with $G \neq G'$, that are isomorphic. The* size *of a clone pair denotes the number of nodes of the graphs, which is the same for both graphs.*

**Definition 2.10 (Clone Group)** *A* clone group *is a set of graphs, in which any two graphs are a clone pair. The* size *of a clone group denotes the cardinality of the graph set.*

**Definition 2.11 (Disjunctive Clone Group)** *A* disjunctive clone group *is a clone group in which the node sets of any two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are disjunctive ($V_1 \cap V_2 = \emptyset$).*

**Definition 2.12 (Maximal Clone Group)** *A clone group $M$ is* maximal *if there is no other clone group $N$ such that each graph of $M$ is subgraph of at least one graph in $N$.*

**Definition 2.13 (Clone Detection in Graphs)** *Given a directed, labeled Graph $G = (V, E, L)$ and a constant $k \in \mathbb{N}$,* clone detection *is the enumeration of all maximal disjunctive clone groups where each clone pair has at least size $k$.*

For algorithms that potentially solve this problem, the size of $k$ is crucial. Usually $k$ is set to the minimal clone pair size, because clones with a size less than $k$ nodes can not be reported.

## 2.3 Preprocessing and Normalization

After giving a precise definition of the problem in the previous section, we now show details about preprocessing and normalization methods. In this work, we use the same methods as in [4]. Both preprocessing and normalization are used to extract a directed labeled multi graph from a MATLAB/Simulink model.

- In the preprocessing phase, the models are read. Thereby the hierarchical MATLAB/-Simulink structure is flattened by inlining all subsystems. Additionally, all unconnected lines are removed.
- The normalization assigns a label to each block and line consisting only of those attributes that are considered to be relevant for differentiating them. Normalization is used to remove semantically differences between blocks and lines that are not relevant for clone detection. The information included in the labels depends on the type of

clone that should be found. For blocks, at least the type of the block is included whereas semantically irrelevant details such as name or color are excluded. Furthermore, some block attributes are included, e.g. the value of the *Operator* attribute for the *RelationalOperator* block. For lines, generally the source and destination ports are stored with respect to some exceptions like the *Product* block. For this type of block the input ports do not have to be differentiated.

To summarize, the result of those two steps is a labeled multi-graph $G = (V, E, L)$ where nodes $V$ correspond to blocks and directed edges $E \subseteq V \times V$ correspond to lines. The labeling function $L$ maps nodes and edges to normalization labels from some set $N$.

## 2.4 Complexity of Clone Detection

At the end of this chapter we show that the problem of clone detection is NP-complete. Hence we cannot expect to find an efficient (polynomial) algorithm, which solves the problem and is correct and complete.

**Theorem 2.14 (NP-completeness of Clone Detection)** *The problem of Clone Detection is NP-complete.*

### Proof

As defined in the previous sections, clone detection in graph-based models is the enumeration of all maximal isomorphic subgraphs of a given graph $G = (V, E)$. To proof that clone detection is NP-complete, we show that another problem, which is known to be NP-complete, can be reduced to the problem of clone detection. The LCS problem (largest common subgraph-isomorphism decision problem) is known to be NP-complete ([5]), which is the problem of finding a common subgraph of two given graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with at least $k$ edges. The LCS problem can be reduced to the problem of clone detection as follows: You can decide the LCS problem by detecting all clones within a unified graph $G = (G_1 \cup G_2, E_1 \cup E_2)$ and verifying if the biggest clone pair has at least $k$ edges. $\square$

# 3 Related Work

This section gives a brief overview of the current state of the art of clone detection and describes some existing algorithms in detail that are relevant for this work. In Section 1.1 we already mentioned that clones appear in different environments. For the purpose of this work we focus on clone detection in graph-based data-flow models (3.1) and on clone detection in code-based environments (3.2).

## 3.1 Graph-based Clone Detection

In this section a couple of algorithms are presented, that perform clone detection on graph-based models.

### 3.1.1 Heuristic Approach

In [4], a heuristic approach is described that enumerates all maximal clones in a given graph extracted from MATLAB/Simulink models. To obtain such a graph, the authors use preprocessing and normalization methods as described in Section 2.3, which are also used in our approach. In a first step, the algorithm enumerates all pairs of clones, i.e. all pairs of subgraphs that are isomorphic, disjunctive and connected. To do that, the algorithm iterates over all possible pairings of nodes and proceeds in a breadth-first search manner from there. However, the authors do not use an exhaustive search. Instead, they use a heuristic to reduce the complexity. The heuristic provides an estimation of the similarity of a pair of nodes that includes not only the normalization labels but also the structure of the neighborhood of both nodes. While the algorithm iterates over all possible pairings of nodes, the heuristic is used to quickly find other pairs of nodes that can be combined with the current pair of nodes to a clone pair. In a second step the authors provide a method to combine clone pairs to a clone class. Therefore, they construct a new graph, in which vertices represent the node sets of a clone pair and edges are induced by the cloning relationship between them. Maximal clone classes are then the connected components of this graph.

By contrast to our algorithm, this algorithm does not work index-based. Therefore, it is not able to perform incremental updates. If parts of the model change, the entire clone detection needs to be performed again, starting with a new iteration over all possible pairings of nodes and a new search for each clone pair. Furthermore, calculations for the clone detection cannot be distributed over several machines, but must be performed on a single machine. The calculation of the index in our algorithm, by contrast, can be distributed over several machines. In Section 4.6, we will explain how incremental updates can be performed with our algorithm and how the calculation of the index can be distributed. These are the main advantages of our approach compared to the heuristic approach.

The heuristic algorithm in [4] is relevant for our work for several reasons: First, we use the preprocessing and normalization methods provided by the authors. Furthermore, we will use this algorithm for the evaluation of our algorithm in Chapter 6. There we compare both approaches on several test models.

### 3.1.2 Exact and Approximate Clone Detection

In [6], two different algorithms for clone detection in graphs are presented. The first algorithm, named *eScan*, enumerates exact clones with the same definition of a clone pair as defined in Chapter 2. The algorithm is based on the observation that each clone pair with $k$ edges can be derived from a clone pair with $k-1$ edges. So the algorithm starts with $L_1$, a set of all cloned subgraphs which consist of only one edge. These are all edges, that are repeated in the model. Out of all cloned subgraphs of size $k-1$, all cloned subgraphs of size $k$ are created by adding a single edge to the graphs of size $k-1$. If the result is still a cloned subgraph, then its clones and itself are added to $L_k$. This is done in a depth-first search manner. The authors provide a method of parent identification to ensure that no subgraph of size $k$ is created multiple times from different subgraphs of size $k-1$. After that all cloned subgraphs are contained in clone layers $L_1, L_2, ...L_{max}$. Then they are grouped layer by layer into clone groups. The resulting groups are filtered in order to obtain only maximal groups with respect to Definition 2.12.

The second algorithm of [6], named *aScan*, focuses on detecting approximate clones, i.e. non-overlapping weakly connected subgraphs that are similar in structure. To define a similarity measurement for those graphs, the authors use *Exas*, a vector-based representation and feature extraction method that can approximate the structure of a graph. A characteristic vector is assigned to each graph that represents the occurrence of certain structural features in a graph. Then, similarity of two graphs can be measured by an appropriate distance function of both characteristic vectors. If the distance is sufficiently small, then both graphs are considered to be similar in structure. The algorithm is based on the observation that two structural similar graphs must have an isomorphic core common part. Thereby, isomorphism is again defined by the authors' structural similarity function and minor conditions. Similar as in *eScan*, subgraphs of size $k$ are created based on subgraphs of size $k-1$ by adding one edge. By contrast, however, *aScan* performs a breadth-first search rather than a depth-first search. To reduce the complexity of the algorithm, pruning techniques are applied during the search. Then subgraphs are grouped to clone groups which are filtered in order to obtain only maximal groups with respect to Definition 2.12.

Both algorithm start with subgraphs of size 1 and continuously create subgraphs of size $k$ based on subgraphs of size $k-1$. If parts of the model change, then both algorithm need to be run again, because they are not able to be update their results incrementally. This is the main advantage of our algorithm compared to these algorithms since our approach is able to perform incremental updates. In Section 4.6 we will show how this can be done. As well as the algorithm presented in Section 3.1.1, the calculations of both algorithms can not be distributed over several machines neither. However, in Section 4.6, we will also show that part of the calculations in our approach can be distributed over several machines.

## 3.2 Index-based Code Clone Detection

After explaining current state of the art algorithms for clone detection in graph-based models in Section 3.1, this section presents former research for clone detection in code-based models. Thereby, we will primarily focus on one specific algorithm for index-based code clone detection ([7]). The architecture of this algorithm is similar to the one used in this work. We will first explain the existing algorithm in detail and in Chapter 4.1 we will show the parallels between the two approaches based on different underlying environments.

The contribution of the work presented in [7] is a novel algorithm for clone detection in code-based environments that is both incremental and scalable to very large code bases. The architecture as described in [7] exists of three components, namely preprocessing, detection and postprocessing. Both pre- and postprocessing algorithms are not novel, but used from the current state of the art clone detector *ConQAT* ([8]). Therefore, they are only briefly mentioned in this section. The bottleneck of the performance of the algorithm, especially scalability and the ability to perform incremental updates, is the detection phase. Hence this part of the algorithm will be described in detail. In particular, the parallels between our approach for graph-based models and the algorithm in [7] can be found in the clone detection phase.

First we give an overview over all three steps of the algorithm. Preprocessing reads code in each file from disk and performs a normalization on each statement of each file. Therefore, preprocessing results in a list of normalized statements. In the detection step equal substrings are found in the global list of normalized statement. So the detection phase results in cloning information on the level of statement sequences. Postprocessing creates cloning information on the level of code regions based on cloning information on the level of normalized statements.

Now, we describe the detection phase in detail. The central data structure used in this algorithm is the *clone index* which contains a mapping from sequences of normalized statements to their occurrences. The clone index is a list of tuples (file, statement index, sequence hash, info). *File* denotes the name of the file, *statement index* is the position in the list of normalized statements, represented as an integer, and *info* contains additional data. The most important field is the *sequence hash*, a hash code using MD5 hashing for the next $n$ normalized statements in the file starting from the statement index. The chunk length $n$ is usually set to the minimal clone length since no shorter clones can be detected. The core idea of the detection algorithm is the fact that if there are two entries in the index with the same hash sequence then a clone pair of size at least $n$ was found. After finding all statement sequences with the same hash value, the clone retrieval process tries to report only maximal clones, i.e. clone groups which are not completely contained in another clone group. The algorithm can be described in the following three steps: Splitting the input into sequences of length $n$, hashing those sequences and merging sequences to find maximal clone groups. Chapter 4.1 will show, how those three steps can be transfered to graph-based models.

We present this algorithm, because its architecture is similar to the architecture used in our approach. However, this algorithm is working on a different underlying environment, the code-based environment. Our algorithm, by contrast, will work on graph-based data-flow models.

# 4 Approach

In this chapter, details about our algorithm and its implementation can be found. In Section 4.1, we will give an overview of the algorithm and explain the three steps of its architecture. Each of the three steps of the algorithm is described in detail in a separate section. In Section 4.6 we explain why our algorithm is able to perform incremental updates and why parts of the calculations can be distributed over several machines.

## 4.1 Architecture of the Algorithm

From a high-level point of view, this section shows how the idea of index-based code clone detection from chapter 3.2 can be applied to model-based data-flow environments represented with graphs. Furthermore, this section gives an overview of the architecture of our algorithm.

As described in chapter 3.2 the core idea of index-based code clone detection can be summarized with three steps as follows: First, the algorithm creates a list of all continuous normalized statement sequences of length $n$ (chunk-length). Then those sequences are hashed in such a way that identical statement sequences (clone groups) can be easily found by comparing the hash value. If the hash value is identical, then the sequences were cloned. After that maximal clone groups are reported by merging clone groups of smaller size.

The basic principle of the three steps can be transferred to a graph-like representation of the model $G = (V, E)$. Instead of enumerating all continuous statement sequences of length $n$, we enumerate all weakly connected subgraphs $G' = (V', E')$ of $G$ with size $k$. Thereby $k$ represents the minimal clone size. We call this list of all weakly connected subgraphs *index*. According to the core idea of index-based code clone detection as described above, the next step is the hashing of potential clone candidates and getting a unique hash value. Whereas index-based clone detection uses MD5-hashing, we use canonical graph labeling (see Section 4.3) according to Definition 2.9: Two graphs should be reported as a clone pair if and only if they are isomorphic. Canonical graph labeling satisfies this condition, because two graphs have the same canonical label if and only if they are isomorphic. By using canonical labels as keys, we create a hash table where all graphs corresponding to the same key are pairwise isomorphic and therefore form a clone group according to Definition 2.10. The last step is the same as above. To report only maximal clones (see Definition 2.13) clone groups with common clone pairs must be merged to a bigger clone group.

Our algorithm can be quickly described with the following three steps of the clone detection phase: 1) Enumerate all weakly-connected subgraphs of a given size $k$. 2) Hash each graph based on its canonical label and find clone groups by comparing the labels. 3) Merge subgraphs to find maximal clone groups. To report only disjunctive clone groups as required by Definition 2.13, an existing post processing filter is applied as described in [6].

An overview of the algorithm in pseudo code can be found in Listing 4.1. As described in Chapter 2, we use existing preprocessing and normalization methods to extract a directed,

labeled graph from the model. This graph and the minimal clone size $k$ is the input of our algorithm. The output of the algorithm is a list of maximal, disjunctive clone groups according to Definition 2.13.

```
1  Algorithm: cloneDetection
2  Input: Directed Graph G=(V,E), int k
3  Output: List<CloneGroup> A list of maximal disjunctive clone groups
4
5  G = transformToUndirectedGraph(G);
6  List<Graph> subgraphs = enumerateSubgraphs(G, k);
7  List<CloneGroup> cloneGroups = createCloneGroups(subgraphs);
8  List<CloneGroup> maxCloneGroups = merge(cloneGroups);
9  List<CloneGroup> maxDisjunctiveCloneGroups = filter(maxCloneGroups);
10
11 return maxDisjunctiveCloneGroups;
```

Listing 4.1: Algorithm for Clone Detection

## 4.2 Enumerating Subgraphs of Given Size

This section describes the first step of the algorithm, the enumeration of all weakly-connected subgraphs of size $k$. To enumerate all those subgraphs, we use the algorithm shown in 4.2. The basic idea of the algorithm is to choose one node and enumerate all weakly-connected subgraphs of size $k$ if and only if they contain this node. Then all incident edges and the node itself are deleted and the next node can be chosen. This guarantees, that no subgraph is enumerated twice. As we are only interested in weakly-connected subgraphs, we can transform the directed input graph $G$ into an undirected graph $G'$ by replacing all of its directed edges with undirected edges and enumerate all connected subgraphs of $G'$.

```
1  Algorithm: enumerateSubgraphs
2  Input: Undirected Graph G=(V,E), int k
3  Output: List<Graph> A list of connected subgraphs of G of size k
4
5  List<Graph> connectedKSubgraphs;
6
7  for each Node v in V:
8    List<Node> visitedNodes = searchBFS(G, v, k-1);
9    List<List<Node>> kSubgraphs =
10     enumerateSubsets(visitedNodes, k-1);
11   for each List<Node> subgraphNodes in kSubgraphs:
12     subgraphNodes.add(v);
13     Graph subgraph = inducedSubgraph(subgraphNodes);
14     if (isConnected(subgraph))
15       connectedKSubgraphs.add(subgraph);
16   E = E \ {e in E: e is incident to v};
17   V = V \ {v};
18
19 return connectedKSubgraphs;
```

Listing 4.2: Enumeration of subgraphs of size $k$

Given one node $v \in V$, the algorithm first enumerates all (unconnected) subgraphs of size $k$ that contain this node. As the size of the subgraphs must be $k$, the subgraphs can only consist of nodes with a distance from $v$ of at most $k-1$. Thereby the distance between two nodes $u$ and $w$ denotes the length of the shortest path of all $u - v-$paths between $u$ and $w$. To find all nodes within a distance of at most $k-1$ from the selected node $v$ , the algorithm performs an undirected breadth first search. The start node of the search is $v$ and the search is limited to depth $k-1$. The breadth first search is implemented in such a way that it returns all visited nodes within the given depth except of the start node $v$. Then the algorithm enumerates all subsets of the visited nodes of size $k-1$ and adds the selected node $v$ to create subsets of size $k$. If the induced subgraph on one subset of visited nodes is connected, the induced subgraph is added to the output list.

**Theorem 4.1** *Given a directed graph $G$‘ transformed into an undirected graph $G$ and a minimal clone size $k$ as an input, then the algorithm* enumerateSubgraphs *as defined in Section 4.2 lists a subgraph of $G'$ if and only if the subgraph is weakly-connected and has exactly $k$ nodes.*

**Proof**

”$\Rightarrow$”
To show: If the algorithm lists a subgraph of $G$, then the subgraph is weakly-connected and has $k$ nodes.
In line 9 and 10, subsets of size $k-1$ are enumerated, which do not include the selected node $v$. After line 12, each subset includes the additional node $v$ and therefore is of size $k$. Then the induced subgraphs on each subset are tested on connectivity and are only added to the output, if they are connected. Hence the input graph $G$ is undirected, each undirected, connected subgraph of $G$ is a weakly-connected subgraph of the original directed graph $G'$ per definition.

”$\Leftarrow$”
To show: An arbitrary weakly-connected subgraph in $G$ with $k$ nodes is listed by the algorithm.
Let $o : V \rightarrow \mathbb{N}$ be the ordering of the nodes the algorithm uses in line 6 to select the next node. Without loss of generality, we assume the next node $v$ is selected according to $v = \min_{v \in V} o(v)$. Let $v_1, ...v_k$ be the nodes of the weakly-connected subgraph and $\bar{v} = min_{i=1..k} o(v_i)$. Then $\bar{v}$ is the first node of $v_1, ...v_k$ that is selected by the algorithm. As breadth first search with limited depth is complete, all nodes $v_1, ...v_k$ will be enumerated by the search with a depth limit of $k-1$, because all nodes have a distance from $\bar{v}$ of at most $k-1$. Then the algorithm lists all possible $k-1$-subsets of the visited nodes of the search, so the given subset is enumerated. Since the subgraph is weakly connect in $G'$, the induced subgraph in $G$ is connected and therefore added to the output of the algorithm.

□

To summarize, in this section, we have shown how all weakly connected subgraphs of a given size $k$ can be enumerated efficiently. In the following sections we will present methods how to create maximal clone groups based on the enumerated subgraphs.

## 4.3 Creating Clone Groups by Using Canonical Labeling

After all subgraphs of a given size $k$ have been enumerated, this section shows, how isomorphic subgraphs are found and how clone groups are generated.

### 4.3.1 Canonical Labeling

To find isomorphic subgraphs, we compare graphs based on their canonical labeling. The following definition can be found in [9].

**Definition 4.2 (Canonical Labeling)** *A canonical label of a Graph $G = (V, E)$ is a unique code that is invariant to the ordering of vertices $V$ and edges $E$. Thereby graphs that are not isomorphic have different canonical labels.*

The canonical label of a graph is based on a string representation of its adjacency matrix. To make the label invariant to the order of vertices and edges, a unique symmetric permutation of the matrix is used, for which the string is the lexicographically largest one. Section 4.3.1 will show in detail how this string is defined and calculated.

Canonical labels are used to generate clone groups because of the following property:

**Theorem 4.3** *Two Graphs $G_1 = (V_1, E_1, L)$ and $G_2 = (V_2, E_2, L)$ are isomorphic iff their canonical labeling is the same.*

**Proof**

The proof can be directly derived from Definition 4.2.

#### Computation of Canonical Labels

A simple way of calculating a canonical labeling is to create a string by concatenating the upper-triangle entries of a unique symmetric permutation of the graph's adjacency matrix. Thereby the unique permutation of the matrix is a permutation such that this string is the lexicographically largest one that can be obtained from all permutations. For a graph with $|V|$ vertices, the brute-force method of enumerating all possible strings has a complexity of $\mathcal{O}(|V|!)$ making it impractical even for graphs of moderate size.

As calculating the canonical labeling for a graph is equivalent to determining isomorphism between graphs, we can not expect to find an efficient (polynomial) solution. Determining graph isomorphism is a problem in computational complexity theory belonging to NP, but not known to belong to either of its (and, if P $\neq$ NP, disjoint) subsets: P and NP-complete.

However, the complexity of the algorithm can be reduced to $\mathcal{O}(\prod_i |V_i|!)$, if the vertices of the graph can be classified into some ordered subsets $V_1, V_2, ..., V_n$ ([10]). In our algorithm we use the degree of a vertex as a classification criterion so that each subset $V_i$ only contains vertices having the same degree. Then for each vertex $v$ a list $L_v = (a_1, a_2, \ldots a_n)$ is created where $a_i$ $(1 \leq i \leq n)$ represents the number of vertices in subset $V_i$ adjacent to $v$. Using this adjacency information each subset $V_i$ can be further partitioned into an ordered collection of subsets consisting of all vertices that have the same list. This process is repeated recursively until all subsets contain only nodes with identical lists or until

$\mathcal{O}(\prod_i |V_i|!)$ becomes sufficiently small to use the simple brute-force method as described above.

To summarize, a canonical label represents a unique code for a graph, invariant to the ordering of vertices and edges. Furthermore, a canonical label also defines a unique adjacency matrix, that specifies the resulting unique code of the graph. This matrix determines an ordering of the nodes and will be used later in Section 4.4.2.

### 4.3.2 Creating Clone Groups

To find isomorphic subgraphs, all enumerated subgraphs (Section 4.2), are inserted into a hash table with their canonical label as a key. With Theorem 4.3 it is easy to see that all subgraphs with identical keys are isomorphic. Therefore, the hash table consists of several lists, where each list contains subgraphs that are pairwise isomorphic and have size $k$. If a list contains only one subgraph, it is discarded and not further used. Hence, according to Definition 2.10, each list represents a clone group.

To summarize, step two of the algorithm takes a list of weakly-connected subgraphs of size $k$ as an input and returns a list of clone groups. However those clone groups may not be maximal according to Definition 2.12. Therefore, the last step of the algorithm is necessary, in which clone groups are merged together until they are maximal. This step is described in Section 4.4.

## 4.4 Merging Clone Groups

In the following section we show, how clone groups with a clone pair size of $k$ can be merged in order to obtain maximal clone groups as defined in Definition 2.12. In the following, we will first give a short overview of the algorithm in Section 4.4.1. Details of the algorithm are presented in Sections 4.4.2 until 4.4.7.

### 4.4.1 Overview of Finding Maximal Clone Groups

The input for the algorithm described in this section is a list of clone groups, the output is a list of maximal clone groups.

In a first step, each clone group is inserted into a hash table with several keys. Section 4.4.2 shows how keys are generated. The keys are designed in such a way that two clone groups with a clone pair size of $k$ and a common key can be merged to a clone group with a clone pair size of $k+1$. This merging process is described in Section 4.4.3. However, this merging process cannot be applied to two clone groups where one of them has a clone pair size greater than $k$. This drawback is explained in Section 4.4.4. As a next step, all clone groups that were inserted into the hash table with identical keys are merged together to one bigger clone group. Since each clone group $c$ was inserted several times into the hash table, $c$ is part of several bigger clone groups. Section 4.4.6 shows, how several bigger clone groups can be merged to find maximal clone groups and presents the complete algorithm. However, this algorithm is not invariant to the ordering of the input. The reasons for this drawback are explained in Section 4.4.7.

### 4.4.2 Key Generation for a Clone Group

In this section, we describe the generation of the keys for a given clone group. This algorithm is called "*createKeys*" with a clone group as an input, where each clone pair has size $k$. Let $l$ be the size of the clone group. Then the algorithm generates $k$ different keys for the given clone group. The keys are designed in such a way that two clone groups with a clone pair size $k$ and a common key can be merged to a clone group with a clone pair size of $k+1$.

#### Definition of a Key

To define a key for a clone group, we use the following definition:

**Definition 4.4 (Isomorphic Node List of a Clone Group)** *Let $c$ be a clone group of size $l$ and $G_1 = (V_1, E_1, L), G_2 = (V_2, E_2, L), \ldots, G_l = (V_l, E_l, L)$ isomorphic graphs with $k$ nodes each. Then a list of isomorphic nodes is a list with $l$ nodes $n_1, n_2, \ldots, n_l$ and for all $i = 1 \ldots l$ we have $n_i \in V_i$ and all nodes are pairwise isomorphic. Thereby two nodes $n_i$ and $n_j$ are isomorphic under the bijection $f_V$ (defining the isomorphism between the graphs), if $L(f_V(n_i)) = L(n_i)$ or $L(f_V(n_j)) = L(n_j)$.*

Hence clone group $c$ defines $k$ different isomorphic node lists.

Annotation: In some cases, the bijection $f_V$ defining the isomorphism between to graphs is not unique. For example, in graphs where all nodes have the same degree, several bijections exist. It follows that in those cases, the isomorphic node lists of a clone group are not unique neither.

With the help of this definition, we can define the structure of our key. An example to illustrate the structure can be found in Figure 4.1. Later we will explain, why we defined a key this way.

**Definition 4.5 (Key for a Clone Group)** *Let $c$ be a clone group of size $l$ and $G_1 = (V_1, E_1, L), G_2 = (V_2, E_2, L), \ldots, G_l = (V_l, E_l, L)$ isomorphic graphs with $k$ nodes each. $l_1 \ldots l_k$ are the isomorphic node lists defined by $c$. A key $K$ for a clone group $c$ consists of $l$ lists where each list contains $k-1$ nodes $n_1, \ldots, n_{k-1}$ and $n_i \in l_i$ $\forall i = 1 \ldots k-1$. The size of a key denotes the number of nodes in each list.*

With Definition 4.5 it follows that there exists $\binom{k}{k-1} = k$ distinct keys for each clone group $c$. Before a key $K$ is used to insert $c$ into the hash table, $K$ is sorted, so that neither the ordering of the lists in $K$ nor the ordering of the nodes in each list influences the equality of two keys.

To explain the reasons why we chose Definition 4.5 as our definition of a key, we first give a precise definition of the intuitive term "common key":

**Definition 4.6 (Common key)** *Let $K = \{l_1, \ldots, l_l\}$ be a key with size $n$ and $c_1 = \{g_1, \ldots, g_l\}$ and $c_2 = \{h_1, \ldots, h_l\}$ two clone groups. Clone group $c_1$ has a clone pair size $k_1 \geq n$, $c_2$ a clone pair size $k_2 \geq n$. Then $K$ is a common key of $c_1$ and $c_2$, if there exists a bijective function $m_1 : K \rightarrow c_1$ and a bijective function $m_2 : K \rightarrow c_2$ that fulfill the following property: $(m_1(l_i) = g_i) \Rightarrow l_i$ is subset of nodes of $g_i$ and $(m_2(l_i) = h_i) \Rightarrow l_i$ is subset of nodes of $h_i$ $\forall i = 1 \ldots l$.*

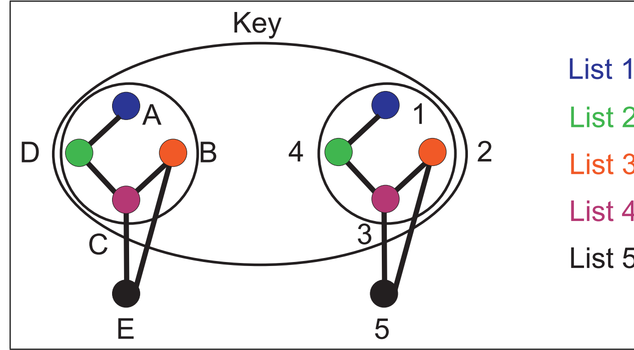



Figure 4.1: A key for a clone group of size 2 and a clone pair size of 5. The coloring of the nodes indicates the isomorphic node lists.

With the definition of the key of a clone group, we can show the following theorem:

**Theorem 4.7** *Let $c_1$ and $c_2$, $c_1 \neq c_2$, be clone groups with clone pair size $k$ and let $K$ be a common key of size $k-1$. Then $c_1$ and $c_2$ can be merged to a new clone group with a graph set $\mathcal{G}$ and a clone pair size $k+1$, where all graphs in $\mathcal{G}$ are pairwise isomorphic.*

**Proof**

If $c_1$ and $c_2$ have a common sorted key, then they have the same clone group size and the same clone pair size. Let $G_1 = (V_1, E_1, L) \in c_1$ and $G_2 = (V_2, E_2, L) \in c_2$ be two graphs. As $K$ is a common key, it follows that there exists a bijective mapping $m$ with $m(G_1) = G_2$ or $m(G_2) = G_1$ and $|V_1 \cap V_2| = k-1$. Hence $G_1$ and $G_2$ can be merged to a new graph $G_=(V, E, L)$ with $V = V_1 \cup V_2$ and $E = E_1 \cup E_2$. We show that the cardinality of $V$ is exactly $k+1$ for each new graph. It is trivial that $|V|$ cannot be greater than $k+1$ or less than $k$. We assume $|V| = k$ for one new Graph $G$ constructed from $G_1$ and $G_2$ as above. Then $G_1$ and $G_2$ had the same set of nodes and therefore the same set of edges, because each subgraph enumerated in Section 4.2 was a completely induced subgraph. Thus $G_1$ and $G_2$ are isomorphic and therefore were inserted into the same clone group according to Section 4.3. This is a contradiction to the assumption that $c_1$ and $c_2$ are not equal. As the graphs of each clone group $c_1$ and $c_2$ are pairwise isomorphic all merged graphs with a common size of $k+1$ will be isomorphic as well and therefore form a new clone group. $\square$

With the help of Theorem 4.7 it is obvious that a key fulfills the desired property in 4.4.1: Keys are designed in such a way that two clone groups with a clone pair size $k$ and a common key of size $k-1$ can always be merged to a clone group with a clone pair size of $k+1$. Note, that defining a key of a clone group with any size smaller than $k-1$ does not fulfill the desired property, because the graphs of the new clone group do not necessarily have the same size and therefore are not isomorphic. A specific example can be found in Figure 4.3, where the size of the subgraphs is $k = 4$ and the key has size 2, which is smaller than $k-1 = 3$. The figure shows, that if the crucial property is not fulfilled, the clone groups can not be merged, although they have a common key.

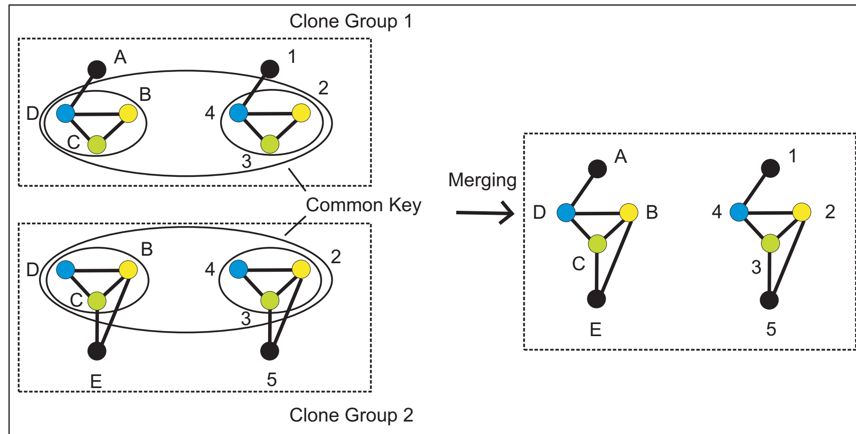



Figure 4.2: Merging Two clone groups with a common key $K$.

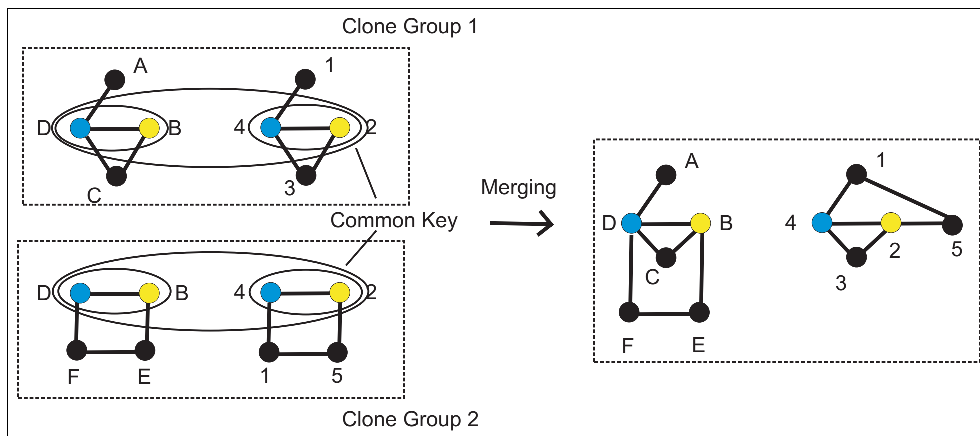



Figure 4.3: Merging Two clone groups with a common key $K$ and a key size smaller than $k-1$ leads to a failure in the merging process.

### Details of the Implementation of a Key

In this paragraph, we show, how isomorphic node lists according to Definition 4.4 can be implemented efficiently. Each list of isomorphic nodes for graphs $G_1 = (V_1, E_1, L), G_2 = (V_2, E_2, L), \ldots, G_l = (V_l, E_l, L)$ with a clone pair size of $k$ can be implemented efficiently in the following way: Let $a_i$ be the unique adjacency matrix of graph $G_i$, specified by the canonical labels as defined in Section 4.3. Since $G_1, \ldots, G_l$ all belong to the same clone group, they are isomorphic and hence their canonical label is the same. It follows that all adjacency matrices $a_1, \ldots, a_l$ specify the same ordering for each node set $V_i$: $o_i : [k] \to V_i$ such that all nodes $o_1(j), o_2(j), \ldots o_l(j)$ are isomorphic $\forall j \in \{1 \ldots k\}$. In our implementation the orderings are stored together with the canonical label and therefore the generation of $k$ isomorphic node lists can be done by enumerating all nodes $o_1(j), o_2(j), \ldots o_l(j),\ j = 1 \ldots k$.

### Drawback

Unfortunately, there exists a drawback of the method as explained above: In the following we will show that the reverse of Theorem 4.7 does not hold true in general: If two different clone groups $c_1$ and $c_2$ with the same size and the same clone pair size $k$ can be merged to a new clone group with a graph set $G$ and a clone pair size $k+1$, where all graphs in $G$ are pairwise isomorphic, then our algorithm does not necessarily compute a common key $K$. Let us give an example, which will show this.

Figure 4.4 shows two clone groups of size 2 and a clone pair size of 4. They could obviously be merged with a common key $K = \{\{B, C, D\}, \{2, 3, 4\}\}$. However, this key might not be computed by our algorithm. As both graphs in clone group 1 are complete (each node is adjacent to all other nodes), there are multiple valid orderings (as defined in the previous section), such that two isomorphic nodes are enumerated at the same position. Both $(A, B, C, D), (1, 2, 3, 4)$ and $(A, B, C, D), (2, 3, 4, 1)$ are valid orderings that could be the results of the canonical label computation. Let us consider the second ordering $(A, B, C, D), (2, 3, 4, 1)$. According to this ordering, the isomorphic nodes

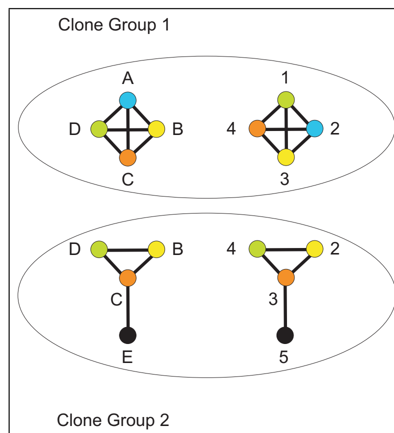



Figure 4.4: Two clone groups that can be merged, but no common key is found. The coloring of the nodes indicates the isomorphic node lists.

lists are $\{A,2\},\{B,3\},\{C,4\},\{D,1\}$. Based on those lists, all possible keys are: $K_1 = \{\{A,B,C\},\{2,3,4\}\}$, $K_2 = \{\{A,B,D\},\{2,3,1\}\}$, $K_3 = \{\{A,C,D\},\{2,4,1\}\}$ and $K_4 = \{\{B,C,D\},\{3,4,1\}\}$. This shows that the required key $K$ to merge both clone groups is not computed and therefore the reverse of Theorem 4.7 does not hold true.

However, we expect this problem to occur only in few cases in practical applications. The problem mainly occurs in regular graphs. (A graph is $k$-regular, if all nodes have degree $k$) But we expect graphs in real-world models to be sparse graphs and not to be regular due to the structure of MATLAB/Simulink models.

### 4.4.3 Merging Two Clone Groups with a Common Key and Equal Clone Pair Size

In this section, it is shown, how two clone groups with a common key are merged to a new clone group. This algorithm is called *mergeCloneGroups*. The input of the algorithm are two distinctive clone groups $c_1$ and $c_2$ and a common key $K$. Both $c_1$ and $c_2$ have a clone group size $l$ and a clone pair size $k$. Let $G = \{g_1 \dots g_l\}$ be the graphs of $c_1$ and $H = \{h_1 \dots h_l\}$ the graphs of $c_2$. The key contains $l$ lists $K = \{l_1 \dots l_l\}$ with $k-1$ nodes in each list.

To merge $c_1$ and $c_2$ to a new clone group $c$, two bijective mappings $m_1$ from $K$ to $G$ and $m_2$ from $K$ to $H$ are required. Thereby a list $l_i \in K$ can be mapped onto a graph $G_j = (V_j, E_j, L)$, if $l_i \subset V_j$. With the help of those two mappings the new clone group $c$ with a graph set $P$ can be constructed as $P = \{p : p = g \cup h, g \in G, h \in H, \exists i : m_1(l_i) = g \wedge m_2(l_i) = h\}$. However, those mappings $m_1$ and $m_2$ do not have to be unique. A list $l_i \in K$ could be a subset of more than one graph as the set of nodes of the graphs are not required to be disjunctive at this point of time in the algorithm. To find one possible bijective mapping from $K$ to $G$ (and from $K$ to $H$ respectively), we use the following definition:

**Definition 4.8 (Matching number)** *Let $c$ be a clone group of size $l$ with isomorphic graphs $G = \{G_1 = (V_1, E_1, L), G_2 = (V_2, E_2, L), \dots G_l = (V_l, E_l, L)\}$ and let $K = \{l_1 \dots l_l\}$ be a key for $c$. Then the matching number $\lambda$ of a Graph $G_i = (V_i, E_i, L) \in G$ is defined as $\lambda(G_i) = |\{j \in \{1, 2 \dots l\} : list_j \subset V_i\}|$.*

The matching number of a graph denotes the number of lists in $K$ that can be mapped onto this graph. With the help of the matching number of a graph, our algorithm finds a bijective matching as shown in Listing 4.3. The algorithm loops through all lists $l_i$ of the key. For each list $l_i$, it creates a set of graphs, which are possible matching partners. Among them, the algorithm chooses the graph with the minimal matching number. In other words: The algorithm chooses the graph that has the least possibilities to be matched with a different list.

Since $K$ was a key of $c$, such a bijective mapping $m : K \to G$ exists. However, as our algorithm proceeds in a greedy-manner, it might fail to find a mapping between all elements in $K$ and $G$ so that some elements remain unmatched. In most cases, we expect that the mapping $m$ is unique, so the set of possible matching partners has a cardinality of 1 in each step of the iteration. Then this mapping is found by the greedy algorithm. If the mapping is not unique, the greedy algorithm could fail. But then, the clone groups are not

```
1  Algorithm: createMapping
2  Input: Clone group c with isomorphic graphs
3         G = {G1=(V1, E1, L), G2=(V2,E2, L), ... ,Gl=(Vl,El, L)} and
4         Key K = {list1 ,... , listl}
5  Output: A bijective matching m from K to G
6
7  for each list list_i in K do
8    Set<Graph> S = {Graph G' in G: list_i is subset of V'};
9    Graph H = argmin{lambda(G): G in S};
10   K = K \ {list_i};
11   G = G \ {H}
12   m(list_i) = H;
13 return m;
```

Listing 4.3: Construction of bijective mapping from a key to a clone group

disjunctive and will be discarded in Section 4.5, as we only aim to find disjunctive clone groups according to Definition 2.13. If the greedy algorithm fails, it does not affect the output of our algorithm.

Once, such a mapping $m$ is found, no other possible mappings are constructed. This can lead to incompleteness of the algorithm. But as the problem of clone detection is NP-complete (see Section 2.4) we do not aim to find a complete solution, but a fast and sufficient approximation.

### 4.4.4 Merging Two Clone Groups with a Common Key and Different Clone Pair Sizes

In this section, we examine, if two clone groups which have the same clone group size and a common key, but a different clone pair size, can be merged.

The algorithm *mergeCloneGroups* tries to merge $c_1$ and $c_2$ as shown in Section 4.4.3. We consider a merging process of two clones to fail, if the graphs of the resulting clone group are not pairwise isomorphic.

**Theorem 4.9** *Let $c_1 = \{G_1 \dots G_l\}$ and $c_2 = \{H_1 \dots H_l\}$ be two different clone groups, where $c_1$ has a clone pair size $k_1 \geq n+2$ , $c_2$ a clone pair size $k_2 \geq n+1$. Let $K$ be a common key $K = \{list_1 \dots list_l\}$ with size $n$. Although $K$ is a common key, merging $c_1$ and $c_2$ does not necessarily create a new clone group where all graphs are pairwise isomorphic.*

#### Proof

Due to the size of the key $n$, not all graphs in the result of the merging process must have the same size. In the proof of Theorem 4.7 $|V_g \cap V_h| = n$ holds for each two graphs $G = (V_g, E_g)$ and $H = (V_h, E_h)$, that should be merged. However, in this case $|V_g \cap V_h|$ can be greater than $n$ for one pair of graphs $\{g, h\}$ and equal to $n$ for a different pair of graph. So the resulting graphs do not have the same size and therefore can not be isomorphic. A specific example can be found in Figure 4.5.

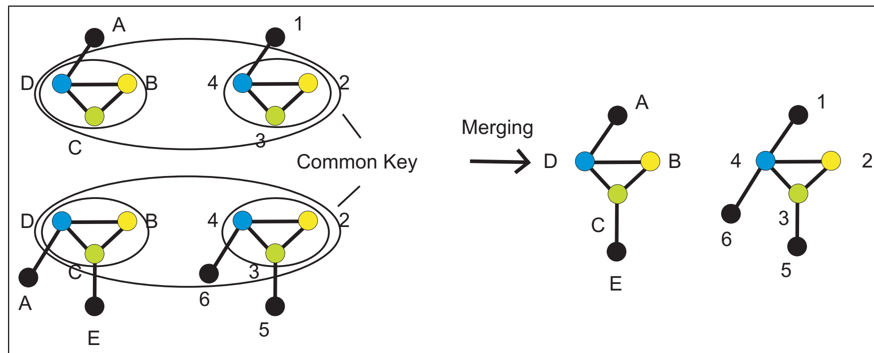



Figure 4.5: Example of a failure in the merging process. The coloring of the nodes indicates the isomorphic node lists.

### 4.4.5 Merging a Set of Clone Groups with a Common Key

In this section we explain how several clone groups, that have a common key, can be merged. This algorithm is called *mergeCloneGroupSet*.

Let $C$ be a list of $n$ clone groups $C = \{c_1 \dots c_n\}$ where all clone groups have a clone pair size of $k$ at the beginning and let $K$ be the common key of all clone groups with size $k-1$. The basic idea is to recursively merge the first two clone groups in $C$ according to Section 4.4.3, remove both of them from $C$ and add the newly created clone to the head of the list. In each step $i$ of the recursion ($i = 1 \dots n-1$) the newly created clone should have a clone pair size of $k+i$. As empirical studies show, this works in most cases. However, the merging process of the first two clones of $C$ could fail in a recursion depth $i$ for $i \geq 2$. We consider a merging process of two clones to fail if the graphs of the resulting clone group are not pairwise isomorphic. For recursion depth $i = 1$ both clone groups have a clone pair size of $k$. For a key with size $k-1$ we have shown in Theorem 4.7 that all graphs in the result of the merging process are pairwise isomorphic and therefore build a valid clone group. For a recursion depth $i \geq 2$ we are trying to merge a clone group of clone pair size $k+i-1$ with a clone group with clone pair size $k$ and a key with size $k-1$. For this case we have shown in Theorem 4.9 that the merging process could possibly fail. However this is rare in practice. At recursion depth $i$, let $c$ be the second clone group in $C$ where the algorithm fails to merge for the first time. Then $c$ is removed from $C$, no merging process takes place, and the algorithm continues.

### 4.4.6 Finding Maximal Clone Groups

With the help of Sections 4.4.2, 4.4.3 and 4.4.5 we can now present the entire algorithm to find maximal clone groups.

#### Creating the Hash Table

As described above, all clone groups, created like in Section 4.3, are inserted into a new hash table with $k$ different keys. This is shown in pseudo code in Listing 4.4. The basic idea

```
Algorithm: createHashTable
Input:  List<Clone group> cloneGroups
Output: A Hashtable<Key, Clone group>

Hashtable<Key, Clone group> hashTable;
for each cloneGroup in cloneGroups do
  Set<Key> keys = createKeys(cloneGroup)
  for (int i = 0; i < k; i++){
    hashTable.insert(keys.get(i), cloneGroup);
  }
return hashTable;
```

Listing 4.4: Generation of a hash table for clone groups

behind this process is the fact that clone groups that have a common key can be merged to a bigger clone group as described in Section 4.4.5.

### Entire Algorithm

Before we present the complete algorithm for finding maximal clones, we use the following definition to shorten further notations:

**Definition 4.10 (Mergeable Clone Groups)** *Two clone groups are called mergeable iff they have a common key.*

The algorithm to find all maximal clone groups (see Listing 4.5) proceeds as follows: It takes a hash table, generated as described in Listing 4.4, as an input. The output is a list of maximal clones. The algorithm starts with an empty list of maximal clones, $list_{max}$. Then the algorithm loops through all clone groups, created like in Section 4.3. Each clone group and all its mergeable clone groups are merged and a new clone group $c$ is created as shown in Listing 4.6. If $c$ is a subset of any existing clone group in $list_{max}$, then the algorithm continues in the loop. If $c$ can be merged with any existing clone group $c'$ in such a way that all resulting graphs are pairwise isomorphic, then this will be done and the new clone group replaces $c'$ in $list_{max}$. Later we will give more details of the internal representation which helps to perform the test, if $c$ can be merged with any clone group. If $c$ can not be merged with any existing clone group, $c$ is inserted as a new element at the end of $list_{max}$. The entire algorithm can be found in Listing 4.5, where *mergeAll* represents the algorithm in Listing 4.6.

### Internal Representation

At the end of this section, we give some details about the implementation of the algorithm listed in Listing 4.5. The main purpose of the internal representation is to perform an efficient test whether the current clone group $c$ can be merged with an existing clone group in $list_{max}$. Therefore, the algorithm keeps an additional list in memory besides $list_{max}$. We call this list $list_{unmerged}$. During the algorithm, both lists are kept in sync in such a way that for each maximal clone group $\bar{c}$ at position $i$ in $list_{max}$, $list_{unmerged}$ contains at position $i$ a list of all clone groups that were merged to create $\bar{c}$.

```
1  Algorithm: findMaximalClones
2  Input: A Hashtable<Key, Clone group> hashtable
3  Output: List<CloneGroup> with maximal clone groups
4
5  List<CloneGroup> cloneGroups = hashtable.getValues();
6  List<CloneGroup> list_max;
7  for each cloneGroup in cloneGroups do:
8    CloneGroup c = mergeAll(hashtable, cloneGroup);
9    if (c is contained in a clone group in list_max)
10     continue in loop;
11   else if (c is mergeable with a clone group c' in list_max){
12       list_max.remove(c');
13       list_max.add(mergeCloneGroups(c,c');
14   } else
15       list_max.add(c);
16 return list_max;
```

Listing 4.5: Algorithm to find maximal clones

```
1  Algorithm: mergeAll
2  Input: A Hashtable<Key, Clone group> hashtable and clone group c
3  Output: CloneGroup
4
5  List<Keys> keys = createKeys(cloneGroup);
6  CloneGroup merged
7
8  for each key: keys do
9      List<CloneGroup> list = hashtable.getList(key);
10     CloneGroup mergedList = (mergeCloneGroupSet(list, key))
11     merged = mergeCloneGroups(merged, mergedList);
12
13 hastable.removeKeys(keys);
14 return merged;
```

Listing 4.6: Method mergeAll: Creation of a clone group based on one clone group and all its mergeable clone groups

We now describe how both lists are kept in sync. In line 8, a new clone group $c$ is created based on the current clone group and all its mergeable clones with respect to Definition 4.10. In this step, before the merging process to create $c$ is executed, the current clone group and all its mergeable clone groups are stored in a separate list, $list_c$. Now, we look at each case of the if-else-structure in Listing 4.5.

- To test, if $c$ is already contained in $list_{max}$, the algorithm iterates over $list_{unmerged}$ and tests, if all clone groups in $list_c$ are contained in $list_{unmerged}$ at a certain position $i$. If this is the case, nothing needs to be done and the algorithm continues.
- To test, if $c$ can be merged with an existing clone group, the algorithm iterates over $list_{unmerged}$ again. This time, it checks only, if one of the clone groups in $list_c$ is contained in $list_{unmerged}$ at a certain position $i$. We denote this clone group as $c_{common}$. If $c_{common}$ exists, then the current clone group $c$ can be merged with the existing clone

group at position $i$ in $list_{max}$. Therefore, we choose one key of $c_{common}$ as the common key and merge both clone groups. If the merge was valid, the newly created clone group replaces the old one at position $i$ in $list_{max}$. To keep both lists in sync, all clone groups of $list_c$ are added to $list_{unmerged}$ at position $i$. If the merge was not valid, we continue iterating until we find another common clone group.

- If $c$ is neither contained in another clone group nor can be merged with an existing clone group, it is added at the end of $list_{max}$. Additionally, $list_c$ is added at the end of $list_{unmerged}$.

Now, let us assume we would not use the internal representation as described above. To test, if a clone group $c_1 = \{G_1 = (V_1, E_1), \dots G_l = (V_l, E_l)\}$ is contained in a clone group $c_2 = \{H_1 = (W_1, F_1), \dots H_l = (W_l, F_l)\}$, it must be tested for each combination of $c_1 \times c_2$, if $V_i$ is subset of $H_j$ for all $i, j = 1 \dots l$. Even if that could be done in feasible runtime, if $l$ is small, our internal representation has another advantage. Finding the common clone group $c_{common}$ in the merging process automatically determines one possible key for the merging process. Otherwise, the calculation of the common key would be more complex.

### 4.4.7 Ordering of the Input

In this section we show that the algorithm in Listing 4.5 is not invariant with respect to the ordering of the input.

The output depends on the ordering of the clone groups that are returned from the hash table (Line 5). We observe that both in line 8 and in line 13 merging clone group sets and two clone groups, respectively, can lead to new clone groups with a clone pair size greater than k, which are inserted into the list $list_{max}$. In Section 4.4.4 we showed that merging two clone groups with both clone pair sizes greater than k may lead to a failure in the merging process and so a valid new clone group might not be created. Depending on the time when this failure occurs, the output of the algorithm varies.

Let us give an example, visualized in Figure 4.6: Let $A, B, C$ be the first three elements in a possible ordering of the clone groups. Then let $\{D, E\}$ be all mergeable clone groups of $A$, $\{D\}$ the mergeable clone group of $B$ and $\{E\}$ of $c$. In the first iteration, the algorithm merges $A, D, E$ to a new clone group $A^*$, which is inserted to $list_{max}$. In the second iteration the new clone group $B^*$ is created by merging $B$ and $D$. Then $B^*$ is merged with $A^*$ to $A^*B^*$. Then, in the third iteration a failure occurs: Let $C^*$ be the result of the merging process of $C$ and $E$. Then $C^*$ is not merged with $A^*B^*$, because both clone groups have a clone pair size greater than $k$. So the result will be $A^*B^*$ and $C^*$. We now assume a different ordering of the input such that $A^*$ is first merged with $C^*$ to $A^*C^*$. Then $B^*$ can not be merged with $A^*C^*$ and the result is different from the one before, namely $A^*C^*$ and $B^*$. Figure 4.6 visualizes the problem. In a first step, clone groups $A^*$, $B^*$ and $C^*$ are created. Then it depends on the ordering of further merging processes. If first $A^*$ and $B^*$ are merged, then the result is $A^*B^*$ and $C^*$, because the merging process of $A^*B^*$ and $C^*$ causes a failure. In clone group $C^*$ nodes $H$ and 8 are isomorphic. However, in clone group $A^*B^*$, node $H$ is isomorphic to node 7 and node 8 does not exist in neither one of the graphs of $A^*B^*$. So the resulting graphs of the merging process differ in the number of nodes and therefore can not be isomorphic.

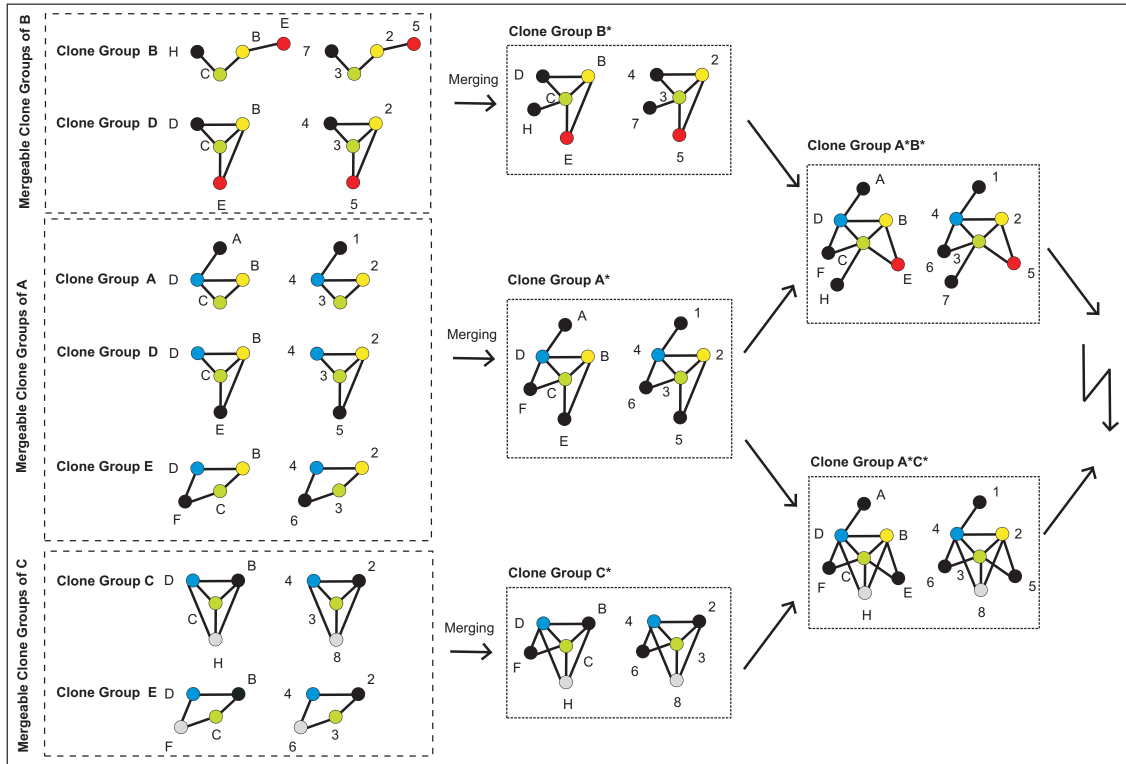



Figure 4.6: Iterations during the merging process. Depending on whether $A*$ and $B*$ or $A*$ and $C*$ are merged first, the output varies.

Because an indeterministic algorithm is not desirable, we return a sorted list of clone groups in line 5. Hence the output of our algorithm is deterministic. However, it might not find all clone groups with maximal clone pair size. As the problem of clone detection is NP-complete (see Section 2.4), we can not expect to find a correct and complete solution in polynomial time. But our evaluation on models in Chapter 6 showed that our algorithm still detects more clones than previous existing solutions.

## 4.5 Filtering Disjunctive Clone Groups

This section describes the filtering process to report only disjunctive clone groups. Therefore, a new graph is constructed for each clone group reported in Section 4.4. Nodes in the new graph correspond to the graphs of the clone group. Edges connect two nodes if and only if the corresponding graphs have disjunctive vertices sets. Then the Bron-Kerbosch algorithm, as presented in [11], is applied to enumerate all maximal cliques of the graph. All cliques that contain at least two nodes are then reported as a clone group. After this filtering process, all clone groups are disjunctive.

## 4.6 Incremental Updates and Distributable Calculations

In Section 1 and 3.1 we already mentioned the two main advantages of our algorithm over existing algorithms for clone detection: Our algorithm is able to perform incremental updates and can be distributed. In this section, we will give details about both features.

**Distributable Calculations**

As explained in Section 4.1 the *index* of our algorithm is a list of all weakly-connected subgraphs of size $k$ of the input graph. The calculation of this list can be distributed over several machines as follows: If the model consists of several files, then preprocessing and normalization of each file are independent from each other and can be distributed over several machines. Assuming the input graph exists of several connected components, the enumeration of subgraphs in each component is independent from other components as well. Therefore, the calculation of the index can be distributed over several machines. After all calculations are finished, the collection of lists of subgraphs only needs to be merged to one list, the *index*.

Furthermore, the merging process can also be distributed over several machines. However, distributing the calculations as described in Section 4.4 is not trivial and prior knowledge is needed. In the following we describe one possibility how the merging process can be distributed. First of all, a new graph is generated, where nodes represent clone groups created like in Section 4.3 and two nodes are adjacent if and only if there exists a common key. In this graph each connected component represents a set of clone groups that can be merged. However, with the help of Section 4.4, it is clear that clone groups in one connected component can not necessarily be merged to one single clone group. This would involve merging clone groups with a common key, but a different clone pair size, and therefore can possibly fail. So the merging process as described in Section 4.4 is still necessary for each connected component in the new graph. Each of the merging processes in a connected component of the new graph can then be distributed over several machines, because clone groups in different connected components do not have a common key and therefore can not be merged. After the distributed merging processes are finished, the concatenation of each list of maximal clone groups on a single machine is the output of the algorithm.

**Incremental Updates**

The index structure of our algorithm can be updated incrementally. Assume the model exists of several files and a change by the software developer is made to only one file. In addition, we assume that each subgraph in the *index* has a reference to the file, which it was extracted from. This is currently not implemented, but can be added quickly. Then only those subgraphs, defined in the file that changed, need to be removed from the index. The new subgraphs of the changed file can be calculated and added to the index. Thereby all other subgraphs remain unchanged. This incremental update saves waiting time compared to a scenario where the entire clone detection needs to be performed again.

# 5 Analysis of Complexity

This chapter will discuss the runtime complexity of the algorithm. Thereby we analyze the complexity of each of the three steps of the algorithm (see Chapter 4.1) separately. However, we do not aim to give a detailed and complete study of the complexity but only a brief description to show that the runtime is not exponential in the size of the input. The complexity will be very high in worst case scenarios like complete or almost complete graphs. But such graphs are rare in practice. By contrast we expect the models to consist of sparse graphs $G = (V, E)$ where $|E| = \mathcal{O}(|V|)$. Therefore the evaluation of the runtime on realistic models is more important than a detailed study of the complexity to show that the algorithm can be run feasible fast. Chapter 6 shows our evaluation on several test models.

## 5.1 Complexity of the Enumeration of Subgraphs of a Given Size

In this section, we present an approximation of the complexity of the first step of the algorithm. In the first step, all weakly-connected subgraphs of a given size are enumerated.

**Theorem 5.1** *Let $b$ be the maximal degree of a node in a given undirected graph $G = (V, E)$, $k$ the minimal clone size and $n$ the size of $V$. Then the algorithm* enumerateSubgraphs *as defined in 4.2 has a runtime complexity of $\mathcal{O}(n \cdot b^k \cdot \binom{b^{k-1}}{k-1} \cdot (k^2 + k \cdot b))$*

**Proof**

We first calculate the complexity of the algorithm components and then of the algorithm itself. In line 7 the algorithm uses breadth first search. Breadth first search has a complexity of $\mathcal{O}(|V| + |E|)$. As the breadth first search is limited to depth $k-1$, it can visit at most $b^{k-1}$ nodes and $\frac{b \cdot b^{k-1}}{2}$ edges. Therefore the complexity of breadth first search is

$$\mathcal{O}(b^{k-1} + \frac{b \cdot b^{k-1}}{2}) = \mathcal{O}(b \cdot b^{k-1}) = \mathcal{O}(b^k) \tag{5.1}$$

Let $n'$ be the number of visited nodes: $n' = \mathcal{O}(b^{k-1})$. Then the algorithm enumerates all $k-1$-subsets of $n'$ with a complexity of

$$\mathcal{O}(\binom{n'}{k-1}) = \mathcal{O}(\binom{b^{k-1}}{k-1}) \tag{5.2}$$

For each subset, the induced subgraph is created and tested on connectivity. As each subset has $k$ nodes, the induced subgraph can be calculated in

$$\mathcal{O}(\binom{k}{2}) = \mathcal{O}(k^2) \tag{5.3}$$

The connectivity test on an undirected graph with $k$ nodes can be implemented with depth first search which has a complexity of

$$\mathcal{O}(k + |E|) = \mathcal{O}(k + \frac{k \cdot b}{2}) = \mathcal{O}(k \cdot b) \tag{5.4}$$

Because the algorithm performs a breadth first search and a subset enumeration for each node, creates the induced subgraph for each subset and tests it on connectivity, the overall complexity of the algorithm is

$$\mathcal{O}(n \cdot b^k \cdot \binom{b^{k-1}}{k-1} \cdot (k^2 + k \cdot b)) \tag{5.5}$$

□

## 5.2 Complexity of Hashing with Canonical Labels

This section examines the complexity of creating a hash table for all enumerated subgraphs from step 1 with their canonical labels as a key.

In Section 4.3, we described an efficient way to calculate canonical labels. We classified the vertices of the graph into ordered subsets and started recursively to further partition each subset. [10] showed that in this case, the complexity of the algorithm is $\mathcal{O}(\prod_i |V_i|!)$ for $V_1, V_2, ..., V_n$ subsets of the vertices set. This process continues recursively until $\mathcal{O}(\prod_i |V_i|!)$ becomes sufficiently small to use simple brute-force methods as described in Section 4.3.

In general, the runtime of computing canonical labeling is exponential in $|V|$. However, we calculate the canonical labels only once in the algorithm, after the enumeration of all subgraphs of size $k$. Therefore canonical labels are only calculated for graphs where $|V|$ is bounded by $k$, which is constant during the entire algorithm. So the algorithm is not exponential in the size of the input.

We do not further examine the runtime of creating canonical labels, because we only aim to show that the overall complexity of our algorithm is not exponential.

## 5.3 Complexity of Finding Maximal Clone Groups

In this section, we give a brief overview of the complexity of finding maximal clone groups. However, we do not examine the complexity of each single step of the algorithm, since we only want to show that our algorithm is not exponential in the size of the input.

### Complexity of Key Generation

In the following we examine the complexity of creating all keys for a given clone group.

**Theorem 5.2** *All keys for a clone group with clone group size $l$ and clone pair size $k$ can be generated in $\mathcal{O}(k^2 \cdot l)$.*

**Proof**

We first determine the memory requirements for all keys. According to Definition 4.5 a key for a clone group consists of $l$ lists where each list contains $k-1$ nodes. As there exist $k$ different keys for each clone group, the memory requirements are in $\mathcal{O}(k \cdot (k-1) \cdot l) = \mathcal{O}(k^2 \cdot l)$. To select the nodes for each list, the isomorphic node lists of the clone group are required. In Section 4.4.2 we explained already that they can be easily calculated based on the adjacency matrix of the canonical labels. Accessing one single node in the matrix is done in $\mathcal{O}(1)$. Since the canonical labels were already calculated, there are no additional calculation costs. Therefore the complexity of generating all keys is bounded by the memory requirement of $\mathcal{O}(k^2 \cdot l)$. □

### Complexity of Merging Clone Groups

We first examine the merging process of two clone groups with clone group size $l$ and clone pair size $k$.

**Theorem 5.3** *Two clone groups with clone group size $l$ and clone pair size $k$ and a common key can be merged in $\mathcal{O}(l^2 \cdot k^2)$.*

**Proof**

Under the assumption that we know that a common key exists, the bijective mapping from the key to both clone groups needs to be calculated. Therefore we calculate the matching number of each graph $G = (V, E)$ with $|V| = k$ (see Definition 4.8). The verification, if $list_j$ is subset of $V$, can be done in $\mathcal{O}(k)$ for each of the $l$ lists. This leads to a complexity of $\mathcal{O}(l \cdot k)$ to calculate the matching number for each graph. The matching numbers of all $l$ graphs are then calculated in $\mathcal{O}(l^2 \cdot k)$. Then the entire complexity of the algorithm in Listing 4.3 is bounded by $\mathcal{O}(l^2 \cdot k)$ hence there are no additional more expensive costs in the main loop. Once the matching from key to both clone group is generated, the clone groups can be merged in $\mathcal{O}(l \cdot (|V|^2 + |E|^2))$ by unifying both node and edge sets for each graph. Under the assumption that all graphs are sparse graphs, this leads to a total complexity of $\mathcal{O}(l^2 \cdot k + l \cdot (|V|^2 + |E|^2) = \mathcal{O}(l^2 \cdot k + \cdot l \cdot k^2) = \mathcal{O}(l^2 \cdot k^2)$. □

From this observation, the following theorem can be derived:

**Theorem 5.4** *A set of $s$ clone groups can be merged in $\mathcal{O}(s \cdot l^2 \cdot \bar{k}^2)$.*

**Proof**

In a set, represented as a list, of $s$ clone groups, the first two clone groups can be merged consecutively, are then removed from the list and the newly created clone group is added at the head of the list. This leads to $s-1$ merging processes with costs of $\mathcal{O}(l^2 \cdot \bar{k}^2)$, if $\bar{k}$ is the maximal clone pair size created during merging processes. Therefore the total costs are bounded by $\mathcal{O}((s-1) \cdot l^2 \cdot \bar{k}^2) = \mathcal{O}(s \cdot l^2 \cdot \bar{k}^2)$. □

We observe that the costs for the key generation can be neglected.

### Complexity of Finding Maximal Clone Groups

To give a meaningful complexity of the entire algorithm that finds all maximal clone groups is difficult since it depends on several quantities that can not be estimated in advance. The number $m$ of clone groups found in Section 4.3, for example, varies depending on the underlying model as well as the number of mergeable clone groups $n$ found for each clone group. Therefore we only give a short approximation and use the same notation as in Section 4.4.6: $list_{max}$ is the list of maximal clone groups, $list_{unmerged}$ is the list of lists of unmerged clone groups, $c$ is the newly created clone group in each iteration and $list_c$ denotes a list of all mergeable clone groups of $c$.

**Theorem 5.5** *Let c be the number of clone groups in the input. Then algorithm Listing 4.5 can be run in* $\mathcal{O}(m^3 \cdot \bar{l}^2 \cdot \bar{k}^2)$ *where* $\bar{l}$ *is the maximal clone group size and* $\bar{k}$ *the maximal clone pair size.*

**Proof**

The number of iterations in the loop of the algorithm in Listing 4.5 is bounded by $m$. In each iteration, the current clone group is merged with all its mergeable clone groups, which are in $\mathcal{O}(m)$. Merging those clone groups is therefore in $\mathcal{O}(m \cdot l^2 \cdot \bar{k}^2)$. After this merge, the algorithm tests if $c$ is already contained in $list_{max}$. To perform this test, the algorithm iterates over $list_{unmerged}$. The length of the list is bounded by $\mathcal{O}(m)$. For each entry of the list, the algorithm tests, if the current clone group is contained. The length of each entry of the list is also bounded by $\mathcal{O}(m)$. To perform the test, the current clone group is compared with each clone group in each entry of the list. This makes a total of $\mathcal{O}(m^2)$ comparisons. Each comparison can be implemented in $\mathcal{O}(l \cdot \bar{k})$, if $\bar{k}$ is the maximal clone pair size and $l$ the size of both clone groups, if both clone groups are sorted. So the entire test hast a complexity of $\mathcal{O}(m^2 \cdot l \cdot \bar{k})$. The second test, whether $c$ can be merged with an existing clone group, is also bounded by $\mathcal{O}(m^2 \cdot l \cdot \bar{k})$. All other costs in Listing 4.5 can be neglected. Therefore we approximate the complexity of finding maximal clone groups with $\mathcal{O}(m^3 \cdot \bar{l}^2 \cdot \bar{k}^2)$, where $\bar{l}$ is the maximal clone group size and $\bar{k}$ the maximal clone pair size.

□

# 6 Case Studies and Comparison to an Existing Algorithm

In this chapter we compare our algorithm to an existing algorithm for graph-based clone detection as presented in [4]. Therefore, we present our test models (Section 6.1), define evaluation criteria (Section 6.2), evaluate both algorithms (Section 6.3) and compare run-time and memory requirements of both algorithms (Section 6.4 and 6.5).

## 6.1 Test Models

This section presents MATLAB/Simulink models used for evaluation of our algorithm and for comparison to existing methods. In Table 6.1, the sizes of different models can be found.

| Name of model | Number of nodes | Number of edges |
|---|---|---|
| FUELSYS | 106 | 141 |
| AERO | 78 | 92 |
| CLUTCH | 61 | 72 |
| TICTACFLOW | 42 | 47 |
| IEEE | 282 | 325 |

Table 6.1: Size (number of nodes and edges) of different models, used for evaluation.

## 6.2 Evaluation Criteria

In this section, we present several evaluation criteria, which we will use in Section 6.3 to compare our approach with the existing algorithm as presented in [4].

To evaluate our algorithm, we created a statistic for each model over the following four values:

- Number of clone groups: Total number of maximal, disjunctive clone groups, found by the algorithm
- Clone group size (see Definition 2.10)
- Clone pair size (see Definition 2.9)
- Clone size: The clone size denotes the product of clone group size times clone pair size

For the values of clone group size, clone pair size and clone size, we calculated the average, the averaged deviation, the minimum and the maximum. Per definition, the minimal clone group size is always 2, the minimal clone pair size is $k$.

## 6.3 Evaluation

For the comparison of our approach with the algorithm given in [4], we set the minimal clone size $k$ in the input of our algorithm to $k = 3$. Manual inspection of the results of [4] on the models as given in 6.1 revealed that in some results the maximal clone size is bounded by 3. Since $k$ denotes the minimal clone size that can be found by our algorithm, we chose $k = 3$. We consider values for k less than 3 to be inappropriate. Clones with a size of 2 or 1 are very unlikely to implement a common concept, which is worth being extracted. To draw a meaningful comparison, we filtered the output of algorithm [4] such that it contains only clone groups with a minimal size of 3 as well.

To compare both algorithms, we created the same statistic on their outputs. Those statistics can be found in the following paragraphs, where $A_N$ denotes our approach and $A_O$ denotes the existing algorithm in [4].

**Comparison on model FUELSYS**

| | **Clones** | **Clone Group Size** | | | | **Clone Pair Size** | | | | **Clone Size** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | av | dev | min | max | av | dev | min | max | av | dev | min | max |
| $\mathbf{A_N}$ | 1 | 3 | 0 | 3 | 3 | 4 | 0 | 4 | 4 | 12 | 0 | 12 | 12 |
| $\mathbf{A_O}$ | 1 | 3 | 0 | 3 | 3 | 4 | 0 | 4 | 4 | 12 | 0 | 12 | 12 |

Table 6.2: Comparison of both algorithms $A_N$ and $A_O$ on model FUELSYS.

In this model, both algorithms enumerate only one clone group with a clone group size of 3 and a clone pair size of 4. Manual inspection of both outputs reveals that both algorithms find the same clone group.

**Comparison on model AERO**

| | **Clones** | **Clone Group Size** | | | | **Clone Pair Size** | | | | **Clone Size** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | av | dev | min | max | av | dev | min | max | av | dev | min | max |
| $\mathbf{A_N}$ | 4 | 2 | 0 | 2 | 2 | 3 | 0 | 3 | 3 | 6 | 0 | 6 | 6 |
| $\mathbf{A_O}$ | 2 | 2 | 0 | 2 | 2 | 3 | 0 | 3 | 3 | 6 | 0 | 6 | 6 |

Table 6.3: Comparison of both algorithms $A_N$ and $A_O$ on model AERO.

The comparison and manual inspection reveal that algorithm $A_O$ enumerates 2 clone groups, whereas our algorithm finds two additional clone groups, which are not enumerated by $A_O$. Figures 6.1 and 6.2 show one clone group, that is found by both algorithms,

and an additional clone group, which is only enumerated by our algorithm. This example demonstrates, why our algorithm is very likely to find more clone groups than algorithm $A_O$ does. As explained in Section 3.1, algorithm $A_O$ uses a search to detect clones. However, this search is not complete, but guided by a heuristic, which prunes the search tree. Therefore, we expect that the clone group in Figure 6.2 can not be enumerated by $A_O$, because parts of it are already contained in clone group 6.1 and therefore not considered for any other clone group.

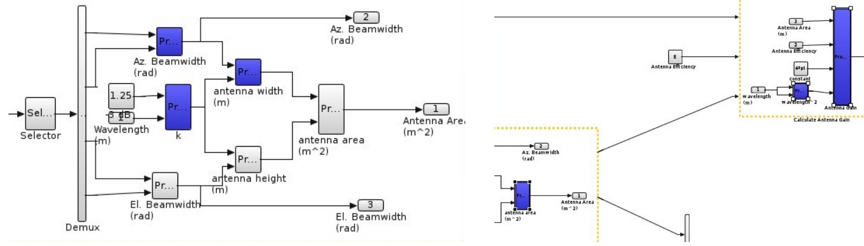



Figure 6.1: Clone group found by both algorithms, visualized with ConQAT, [8].

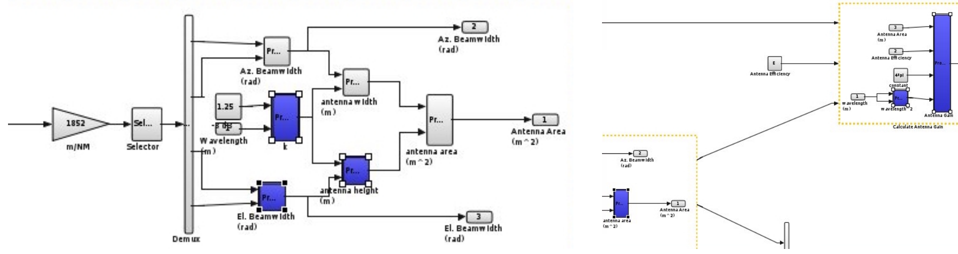



Figure 6.2: Additional clone group found by our algorithm, visualized with ConQAT, [8].

### Comparison on model CLUTCH

| | Clones | Clone Group Size | | | | Clone Pair Size | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | av | dev | min | max | av | dev | min | max |
| $\mathbf{A_N}$ | 25 | 3,24 | 1,26 | 2 | 5 | 3,56 | 0,90 | 3 | 7 |
| $\mathbf{A_O}$ | 4 | 3 | 0,5 | 2 | 4 | 5,25 | 1,75 | 3 | 7 |

Table 6.4: Comparison of both algorithms $A_N$ and $A_O$ on model CLUTCH.

| | Clone Size | | | |
|---|---|---|---|---|
| | av | dev | min | max |
| $\mathbf{A_N}$ | 11,28 | 4,47 | 6 | 21 |
| $\mathbf{A_O}$ | 14,75 | 3,125 | 12 | 21 |

Table 6.5: Comparison of both algorithms $A_N$ and $A_O$ on model CLUTCH.

With the model CLUTCH as an input, our algorithm enumerates 25 clones opposed to 4 enumerated by algorithm $A_O$. In addition to that, our algorithm finds one clone group with a clone group size of 5, whereas the maximal clone group size of $A_O$ is 4. Although the average clone group size, the average clone pair size and the average clone size of $A_O$ is greater than of $A_N$, $A_O$ does not find bigger clones than $A_N$ does. Instead, the maximums of those three criteria of $A_N$ are greater or equal to the maximums of $A_O$. However $A_N$ finds a lot more number of clone groups than $A_O$, including more smaller clone groups, which reduce the average of those three criteria.

**Comparison on model IEEE**

| | **Clones** | **Clone Group Size** | | | | **Clone Pair Size** | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | av | dev | min | max | av | dev | min | max |
| $\mathbf{A_N}$ | 34 | 2,35 | 0,66 | 2 | 8 | 3,68 | 0,99 | 3 | 9 |
| $\mathbf{A_O}$ | 11 | 2,55 | 0,99 | 2 | 8 | 5,55 | 3,07 | 3 | 19 |

Table 6.6: Comparison of both algorithms $A_N$ and $A_O$ on model IEEE.

| | **Clone Size** | | | |
|---|---|---|---|---|
| | av | dev | min | max |
| $\mathbf{A_N}$ | 8,94 | 4,44 | 6 | 40 |
| $\mathbf{A_O}$ | 13,82 | 9,92 | 6 | 40 |

Table 6.7: Comparison of both algorithms $A_N$ and $A_O$ on model IEEE.

The comparison of both algorithms on model IEEE reveals two significant differences in the output. First, our algorithm enumerates 34 clones, whereas the other algorithm enumerates 11. However, algorithm $A_O$ finds one clone group with a clone pair size of 19, whereas our algorithm has a maximal clone pair size of 9. To our best knowledge, we assume, that in this case, the heuristic used in [4], was very accurate and detected a clone group with a large clone pair size of 19 and a clone group size of 2. Our algorithm, by contrast, could not detect a clone group with a similar clone pair size. However, it found several clone groups with clone group size 2 and a smaller clone pair size. In our current implementation, these could not be merged to one bigger clone group. This is due to the incompleteness of our merging process as shown in Section 4.4.

Furthermore, in this model we make the same observation as in the previous model. The average of clone group, clone pair and clone size are smaller for our approach $A_N$, but the maximums of the clone group size and the clone size are the same. The maximum of the clone pair size for $A_O$ is greater as described above. We explain the smaller average of clone group size and clone size with $A_N$ finding more clones, among them a couple of smaller one reducing those averages.

**Comparison on model TICTACFLOW**

| | Clones | Clone Group Size | | | | Clone Pair Size | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | av | dev | min | max | av | dev | min | max |
| $\mathbf{A_N}$ | 4 | 2,25 | 0,375 | 2 | 3 | 3,25 | 0,375 | 3 | 4 |
| $\mathbf{A_O}$ | 3 | 2 | 0 | 2 | 2 | 3,67 | 0,44 | 3 | 4 |

Table 6.8: Comparison of both algorithms $A_N$ and $A_O$ on model TICTACFLOW.

| | Clone Size | | | |
|---|---|---|---|---|
| | av | dev | min | max |
| $\mathbf{A_N}$ | 7,5 | 2,25 | 6 | 12 |
| $\mathbf{A_O}$ | 7,33 | 0,88 | 6 | 8 |

Table 6.9: Comparison of both algorithms $A_N$ and $A_O$ on model TICTACFLOW.

On the model TICTACFLOW, our algorithm finds one clone group with clone group size 3, whereas algorithm $A_O$ finds only clone group with size 2. This leads to a higher average in both the clone group size and the clone size.

## 6.4 Runtime

In this section, we present the runtime of our algorithm compared to the existing algorithm as presented in [4]. The following times were measured on an Intel Centrino CPU with 1.836 GHz, 3 GB RAM and 2 GB Cache. The code is written in Java 1.6 within the Eclipse platform 3.5.2 under the operating system Ubuntu-Linux. The runtimes represent the current state of the art of our approach. The code has not been optimized yet and we expect to be able to significantly reduce both the runtime and the memory requirements of the algorithm. Table 6.10 shows the result of the comparison.

| | TICTACFLOW | CLUTCH | AERO | FUELSYS | IEEE |
|---|---|---|---|---|---|
| $\mathbf{A_N}$ | 9s | 19s | 11s | 57s | 357s |
| $\mathbf{A_O}$ | 6s | 6s | 6s | 6s | 7s |

Table 6.10: Comparison of the runtime of both algorithms $A_N$ and $A_O$

The comparison of the runtime of both algorithms shows that $A_O$ can be run significantly faster than $A_N$ at the moment. Indepent from the size of the models, $A_O$ needs 6 or 7 seconds to be executed. Our algorithm needs 12 seconds for the smallest models TICTACFLOW, twice as long as $A_O$. For the largest models IEEE it requires 319 seconds, almost 46 times longer than $A_O$. This shows that analyzing and optimizing the runtime will be a major aspect of future work.

## 6.5 Memory Requirements

After comparing the runtime, we compare the memory requirements of both algorithms on the same machine as in Section 6.4. Also in terms of memory requirements the code has not been optimized yet and we expect to be able to reduce the memory requirements significantly. The results of the comparison can be seen in Table 6.11.

| | TICTACFLOW | CLUTCH | AERO | FUELSYS | IEEE |
|---|---|---|---|---|---|
| $\mathbf{A_N}$ | 90.112kB | 135.936kB | 97.408kB | 154.304kB | 154.944kB |
| $\mathbf{A_O}$ | 74.688kB | 78.976kB | 78.208kB | 75.520kB | 101.952kB |

Table 6.11: Comparison of the memory requirements of both algorithms $A_N$ and $A_O$

The comparison reveals that algorithm $A_O$ is constantly able to perform the clone detection with a memory requirement between 74.000 kB and 79.000 kB for the four smallest models. Although the number of nodes and edges in model IEEE are approximately twice the number of nodes and edges in model FUELSYS, the memory requirements only increased to 102.000 kB (IEEE) compared to 75.520 kB (FUELSYS). Our approach requires more memory, starting with 90.000 kB for the smallest model TICTACFLOW up to 155.000 kB for the largest model IEEE.

## 6.6 Summary

To draw a conclusion, our algorithm enumerates generally speaking more clones than algorithm $A_O$ and sometimes finds clones with a bigger clone group size. However, due to the incompleteness in our merging process, algorithm $A_O$ can possibly find clone groups with larger clone pair sizes. Both in terms of runtime and memory requirements the existing algorithm $A_O$ performs better. However our algorithm is able to perform incremental updates and calculations can be distributed, because it is index-based. This is the main advantages of $A_N$ over $A_O$, because it can reduce the runtime on large real-world models significantly.

# 7 Conclusion and Future Work

In this work, we presented a novel index-based algorithm for clone detection in graph-based data-flow models. By contrast to all existing algorithms, our approach is both incremental and distributable. We presented details about its implementation and gave an analysis of the complexity, which showed that the runtime is not exponential in the size of the input. Furthermore, we evaluated our algorithm on test-models and compared it with an existing algorithm for clone detection in graph-based models extracted from Matlab/Simulink. We showed that our algorithm is able to compete with the existing algorithm in most cases. In a couple of test models, our algorithm found clones with a bigger clone group size.

However, our algorithm may possibly detect clone groups with a smaller clone pair size than the existing algorithm used for comparison. Therefore, one main aspect of future work will be the optimization of the merging process. We want to improve the merging process in such a way, that our algorithm is also able to detect clone groups with large clone pair sizes. In addition to that, we will optimize both memory requirements and the runtime of our algorithm as the comparison to an existing algorithm revealed that our new approach requires more memory and takes longer to be executed. We aim to reduce the runtime of our algorithm so that we can run it on large real-world models with thousands of nodes.

# Bibliography


[1] R. Koschke. Survey of research on software clones. In *Dagstuhl Seminar Proceedings: Duplication, Redundancy, and Similarity in Software*, Schloss Dagstuhl, 2007.

[2] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato, and Ken-ichi Matsumoto. Software quality analysis by code clones in industrial legacy software. In *METRICS '02: Proceedings of the 8th International Symposium on Software Metrics*, page 87, Washington, DC, USA, 2002. IEEE Computer Society.

[3] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-based detection of clone-related bugs. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 55–64, New York, NY, USA, 2007. ACM.

[4] Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Bernhard Schätz, Stefan Wagner, Jean-François Girard, and Stefan Teuchert. Clone detection in automotive model-based development. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 603–612, New York, NY, USA, 2008. ACM.

[5] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[6] Nam H. Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. Complete and accurate clone detection in graph-based models. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 276–286, Washington, DC, USA, 2009. IEEE Computer Society.

[7] Benjamin Hummel, Elmar Jürgens, Lars Heinemann, and Michael Conradt. Index-based code clone detection: Incremental, distributed, scalable. In *Accepted for puplication at ICSM '10*, Timisoara, Romania, 2010. IEEE Computer Society.

[8] ConQAT (Continuous Quality Assessment Toolkit) web home. http://www.conqat.org.

[9] Michihiro Kuramochi and George Karypis. Finding frequent patterns in a large sparse graph. *Data Min. Knowl. Discov.*, 11(3):243–271, 2005.

[10] Ronald C. Read and Derek G. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1(4):339–363, 1977.

[11] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, 1973.