Codeklonerkennung mit Dominatorinformationen

Dissertation

zur Erlangung des akademischen Grades

doctor rerum naturalium (Dr. rer. nat.)

vorgelegt dem Rat der Fakultät für Mathematik und Informatik der Friedrich-Schiller-Universität Jena

von

M.Sc. André Schäfer
geboren am 10.01.1994 in Eisenach

Gutachter
1. Prof. Dr. habil. Wolfram Amme
2. Prof. Dr. Thomas S. Heinze, Gera
3. Prof. Dr. Rainer Koschke, Bremen

Tag der öffentlichen Verteidigung:
07. Juli 2023

Abstract

If an existing function in a software project is copied and reused (in a slightly modified version), the result is a code clone. If there was an error or vulnerability in the original function, this error or vulnerability is now contained in several places in the software project. This is one of the reasons why research is being done to develop powerful and scalable clone detection techniques. In this thesis, a new clone detection method is presented that uses paths and path sets derived from the dominator trees of the functions to detect the code clones. A dominator tree is a special form of the control flow graph, which does not contain cycles. The dominator tree based method has been implemented in the StoneDetector tool and can detect code clones in Java source code as well as in Java bytecode. It has equally good or better recall and precision results than previously published code clone detection methods. The evaluation was performed using the BigCloneBench. Scalability measurements showed that even source code with several 100 million lines of code can be searched in a reasonable time.

In order to evaluate the bytecode based StoneDetector variant, the BigCloneBench files had to be compiled. For this purpose, the Stubber tool was developed, which can compile Java source code files without the required libraries. Finally, it could be shown that using the register code generated from the Java bytecode, similar recall and precision values could be achieved compared to the source code based variant.

Since some machine learning studies specify that very good recall and precision values can be achieved for all clone types, a machine learning method was trained with dominator trees. It could be shown that the results published by the studies are not reproducible on unseen data.

Zusammenfassung

Wird eine bestehende Funktion in einem Softwareprojekt kopiert und (in leicht angepasster Form) erneut genutzt, entsteht ein Codeklon. War in der ursprünglichen Funktion jedoch ein Fehler oder eine Schwachstelle, so ist dieser Fehler beziehungsweise diese Schwachstelle jetzt an mehreren Stellen im Softwareprojekt enthalten. Dies ist einer der Gründe, weshalb an der Entwicklung von leistungsstarken und skalierbaren Klonerkennungsverfahren geforscht wird. In der hier vorliegenden Arbeit wird ein neues Klonerkennungsverfahren vorgestellt, das zum Detektieren der Codeklone Pfade und Pfadmengen nutzt, die aus den Dominatorbäumen der Funktionen abgeleitet werden. Ein Dominatorbaum wird aus dem Kontrollflussgraphen abgeleitet und enthält keine Zyklen. Das Dominatorbaum-basierte Verfahren wurde in dem Werkzeug StoneDetector umgesetzt und kann Codeklone sowohl im Java-Quelltext als auch im Java-Bytecode detektieren. Dabei hat es gleich gute oder bessere Recall- und Precision-Werte als bisher veröffentlichte Codeklonerkennungsverfahren. Die Wert-Evaluierungen wurden dabei unter Verwendung des BigCloneBenchs durchgeführt. Skalierbarkeitsmessungen zeigten, dass sogar Quellcodedateien mit mehreren 100-Millionen Codezeilen in angemessener Zeit durchsucht werden können.

Damit die Bytecode-basierte StoneDetector-Variante auch evaluiert werden konnte, mussten die Dateien des BigCloneBench kompiliert werden. Dazu wurde das Stubber-Tool entwickelt, welches Java-Quelltextdateien ohne die benötigten Abhängigkeiten kompilieren kann. Schlussendlich konnte somit gezeigt werden, dass mithilfe des aus dem Java-Bytecode generierten Registercodes ähnliche Recall- und Precision-Werte im Vergleich zu der Quelltext-basierten Variante erreicht werden können.

Da einige Arbeiten mit maschinellen Lernverfahren angeben, bei allen Klontypen sehr gute Recall- und Precision-Werte zu erreichen, wurde ein maschinelles Lernverfahren mit Dominatorbäumen trainiert. Es konnte gezeigt werden, dass die von den Arbeiten veröffentlichten Ergebnisse nicht auf ungesehenen Daten reproduzierbar sind.

Inhaltsverzeichnis

1 Einleitung und Motivation **1**

2 Begriffsbildung der Codeklonerkennung **7**

2.1 Codeklon 7

2.2 Wie entstehen Klone? 9

2.3 Klontypen 12

2.4 Strukturen bei der Codeklonerkennung 15

2.5 Evaluation von Klonerkennungsverfahren 20

3 Stand der Forschung **25**

3.1 Textuelle Verfahren 26

3.2 Lexikalische Verfahren 27

3.3 Syntaktische Verfahren 28

3.4 Semantische Verfahren 30

3.5 Hybride Verfahren 31

3.6 Maschinelle Lernverfahren 32

4 Dominatorbasierte Klonerkennung **37**

4.1 Der grundlegende Ansatz zur Codeklonerkennung 37

4.2 Vorgehen beim Klonerkennungsansatz 41

4.2.1 Finden ähnlicher Funktionen und Unterfunktionen 41

4.2.2 Entdeckung ähnlicher Unterblöcke 42

4.2.3 Vergleich von Pfaden 43

4.2.4 Vergleich von Pfadmengen 49

5 Dominatorbasierte Klonerkennung für Sourcecode 51

5.1 Recall-Wert 56

5.2 Precision-Wert 58

5.3 Detaillierter Vergleich mit Klonerkennungssystemen 60

5.4 Laufzeitverhalten 65

6 Dominatorbasierte Klonerkennung für nicht Sourcecode-basierte Darstellungen 69

6.1 Java-Bytecode-basierter Klonerkennungsbenchmark 69

6.2 Stubber 71

6.2.1 Funktionsweise 73

6.2.1.1 Generelle Anpassungen 73

6.2.1.2 BigCloneBench/BigCloneEval Integration 77

6.2.2 Evaluation 78

6.3 Java-Bytecode-basierte Klonerkennungsansätze 80

6.3.1 Stackcode-basierte Klonerkennung 84

6.3.2 Registercode-basierte Klonerkennung 87

6.3.3 Vergleich StoneDetector mit iClones 90

7 Maschinelles Lernen zur Klonerkennung 95

7.1 CDLH-Teildatensatz 96

7.2 Graph-Matching-Network 96

7.3 Experimente zu GMNs und Dominatorbäumen 98

7.3.1 Experimente zum Vergleich von DT+GMN mit anderen Tools 99

7.3.2 Experimente zur Separation von Trainings- und Evaluationsdatensatz 101

7.3.3 Experimente mit genaueren Kloninformationen 103

7.4 Diskussion 105

8 Zusammenfassung **107**

Literatur **113**

Abbildungsverzeichnis **135**

Tabellenverzeichnis **137**

Kapitel 1

Einleitung und Motivation

Das Kopieren von bestehendem Code und dessen Wiederverwendung mit kleineren oder größeren Änderungen ist in der Softwareentwicklung eine häufige Praxis, die zu duplizierten oder sehr ähnlichen Codefragmenten, sogenannte *Codeklonen* [127], führt. Diese Praxis ist sehr weit verbreitet und so gibt es auch Codeklone in großen quelloffenen Softwareprojekten, in denen empirische Studien erhebliche Anteile an dupliziertem Code gefunden haben [103, 125]. Es gibt zwar verschiedene Gründe, duplizierten Code zu nutzen, jedoch sind die Auswirkungen umstritten und teilweise haben sich erhebliche Nachteile bei der Nutzung von dupliziertem Code gezeigt. Die Identifizierung von Codeklonen ist daher eine wichtige Aufgabe im Softwareentwicklungsprozess und dient zudem dem Programmverständnis von großen Anwendungen. Es gibt keinen einheitlichen Typ eines Codeklones. Codeklone können von zwei identischen Codeteilen bis hin zu zwei Codefragmenten reichen, welche dieselbe Funktionalität auf völlig unterschiedliche Weise erfüllen. Gerade diese Codeklone mit einer geringen syntaktischen Übereinstimmung sind schwierig zu identifizieren.

In dieser Arbeit wird das entwickelte StoneDetector-System vorgestellt. Der Name StoneDetector steht dabei für ***S**ub- and **St**ructural Cl**one Detector***, da mithilfe von StoneDetector insbesondere Subklone und strukturelle Klone aufgespürt werden können. Ein Beispiel für ein strukturelles Klonpaar ist dabei in der Abbildung 1.1 zu sehen. Die beiden Programmfragmente führen die exakt gleiche Funktionalität aus, benutzen dafür jedoch verschiedene strukturelle Programmierkomponenten. Das StoneDetector-System implementiert eine textuelle und pfadorientierte Technik zur Erkennung von Codeklonen in Java-Quellcode und auch im Java-Bytecode. Zu diesem Zweck werden ähnliche Codefragmente, d. h. Java-Methoden, mit einer

```
1  int[][] fillMatrixRandom1(int matrix[][]) {
2      Random random = new Random(1);
3      for (int i=0; i< matrix.length; i++) {
4          for (int j=0; j<matrix[i].length; j++) {
5              matrix[i][j] = random.nextInt();
6          }
7      }
8      return matrix;
9  }
```

```
1  int[][] fillMatrixRandom2(int matrix[][]) {
2      Random random = new Random(1);
3      int i = 0;
4      while (i<matrix.length) {
5          int j=0;
6          while(j<matrix[i].length) {
7              matrix[i][j] = random.nextInt();
8              j++;
9          }
10         i++;
11     }
12     return matrix;
13 }
```

Abbildung 1.1: Schwer zu identifizierender struktureller Klon.

Kombination aus Java-Quellcode beziehungsweise Java-Bytecode und Kontrollflussinformationen abgeglichen. Die Kontrollflussinformationen werden in der Form von Dominatorbäumen, einer spezialisierten Kontrollflussdarstellung, bei der Erkennung von Codeklonen einbezogen. Dominatorbäume unterstützen eine Überprüfung auf Übereinstimmung von Pfaden im Vergleich zu anderen bekannten Codedarstellungen wie Kontrollflussgraphen besser, da eine Pfadableitung aus Dominatorbäumen aufgrund der fehlenden Schleifen im Vergleich zu Kontrollflussgraphen vereinfacht umgesetzt werden kann. Ein Vergleich mit den weit verbreiteten Klondetektoren NiCad [126], SourcererCC [131], CCAligner [166], Deckard [63], iClones [54] und Oreo [130] wird zeigen, dass der in StoneDetector implementierte Ansatz oft mehr Codeklone mit vergleichbarer Präzision und in angemessener Zeit findet.
StoneDetector kann verwendet werden, um exakte und nahezu identische Codeklone sowie Codeklone mit größerer syntaktischer Varianz (vgl. Abbildung 1.1) zu erkennen. Die große Mehrheit der aktuellen Klonerkennungswerkzeuge ist nicht in der Lage, ein solches Klonpaar wie in Abbildung 1.1 zu erkennen. Das in dieser Arbeit

vorgeschlagene Verfahren erkennt dieses Klonpaar sogar mit einem Distanz-Schwellenwert von 0. Im Gegensatz dazu ist NiCad erst in der Lage, den Codeklon bei einem Schwellenwert von mindestens 60% zu erkennen, d. h. es muss bis zu 60% syntaktische Unterschiede bei der Erkennung von Klonen akzeptieren. CCAligner erkennt das Klonpaar erst mit einem Schwellenwert von 75%.

Die Arbeit ist wie folgt aufgebaut. In den weiteren Abschnitten dieses Einleitungskapitels werden zunächst die zu evaluierenden Thesen dieser Arbeit vorgestellt und anschließend Gründe genannt, weshalb das Detektieren von Codeklonen eine wichtige (Forschungs-)Aufgabe ist. Im Kapitel 2 werden grundlegende Begrifflichkeiten, wichtige Strukturelemente und Evaluationsmöglichkeiten in der Codeklonerkennung aufgeführt. Das Kapitel 3 beinhaltet eine detaillierte Aufarbeitung ähnlicher und verwandter Arbeiten im Bereich der Klonerkennung. Der allgemeine dominatorbasierte Klonerkennungsansatz wird anschließend im Kapitel 4 vorgestellt. Eine erste Instanziierung des allgemeinen Ansatzes wird im Kapitel 5 vorgenommen, wobei Codeklone auf Basis des Java-Quellcodes identifiziert werden. Zwei weitere Instanziierungen des allgemeinen Dominatorbaum-basierten Klonerkennungsansatzes werden in Kapitel 6 vorgestellt. In diesem Kapitel werden Codeklone sowohl auf Stackcode-Basis als auch auf Registercode-Basis identifiziert. Neuere maschinelle Lernwerkzeuge geben an, dass sie fast alle Klone unabhängig von den syntaktischen Unterschieden mit einer enorm hohen Genauigkeit detektieren können. Um den Aussagen dieser Veröffentlichungen auf den Grund zu gehen, werden in Kapitel 7 die Ergebnisse verschiedener Experimente vorgestellt. Bei den Experimenten wurde ein maschinelles Lernverfahren mithilfe von Dominatorbäumen trainiert. Abschließend folgt eine Zusammenfassung in Kapitel 8.

Thesen

Die vorliegende Arbeit versucht in ihren Kapiteln an der Beantwortung der Frage zu arbeiten, ob die folgenden Thesen belegbar sind:

1. Dominatorinformationen können zur Quelltext-Klonerkennung verwendet werden und erzielen bei hohen Precision-Werten vergleichbare oder bessere Recall-Ergebnisse als bisher bekannte Klonerkennungsverfahren.

2. Klonerkennungsverfahren unter Verwendung von Dominatorbäumen entdecken im Gegensatz zu bisherigen Verfahren vermehrt auch Klone vom Typ-3 und Typ-4, insbesondere Subklone und strukturelle Klone.

3. Die Verwendung von Dominatorbäumen bei der Quelltext-Klonerkennung ist effizient durchführbar und zudem auf große Datenmengen anwendbar.

4. Für bereits kompilierte Programmbeschreibungen, die in Stackcode oder Registercode vorliegen, können effizient die Quelltext-Klone abgeleitet werden.

5. Dominatorbäume können zur Klonerkennung mit maschinellen Lernverfahren verwendet werden.

6. Bisher beim maschinellen Lernen verwendete Benchmarkprogramme, wie beispielsweise der CDLH-Datensatz, sind nicht geeignet, um maschinelle Lernverfahren zur Klonerkennung gleichzeitig zu trainieren und zu evaluieren.

Veröffentlichungen

Wesentliche Teile der vorliegenden Arbeit wurden in der einen oder anderen Form bereits veröffentlicht. Der allgemeine dominatorbasierte Klonerkennungansatz aus Kapitel 4 sowie die Quelltext-basierte Klonerkennung aus Kapitel 5 wurden in diesen Arbeiten [134, 137] veröffentlicht. Details zur Erzeugung des Klonerkennungsbenchmarks für Java-Bytecode-basierte Klondetektoren aus dem Abschnitt 6.2 können in [135] nachgelesen werden. Die Experimente zu maschinellen Lernverfahren in der Codeklonerkennung aus Kapitel 7 wurden in [136] diskutiert.

Nachteile von Klonen und warum man sie aufspüren möchte

Codeklone können negative Auswirkungen auf die Qualitäten, die Wartbarkeit und die Wiederverwendbarkeit haben. Aus diesem Grund ist es wichtig, Klone zu detektieren und auch an der Entwicklung von effizienterer und genauerer Klonerkennungssoftware zu forschen. Roy und Cordy [128] haben zu diesem Thema in ihrer Arbeit viele Gründe recherchiert, die hier zusammengefasst wiedergegeben werden. Wenn ein Codefragment eine Schwachstelle (Bug) enthält und dieses Programmfragment mit keiner oder nur wenigen Änderungen kopiert wird, dann bleibt die

Schwachstelle eventuell bestehen. Das nennt man dann ***Bug Propagation*** [66, 99] und das führt dazu, dass die Software insgesamt fehleranfälliger wird. Auch beim Kopieren eines korrekten Programmfragments kann es zu Problemen kommen, denn beim Kopieren und Wiederverwenden muss die Entwicklerin beziehungsweise der Entwickler das Fragment an die Gegebenheiten anpassen. Bei diesem Prozess kann es zu **Neuen Fehlern** kommen und Schwachstellen entstehen im System [19, 68]. Das gilt es zu vermeiden und aus diesem Grund sollten erst keine Kopien von Programmfragmenten und somit Klone angelegt werden. Falls allerdings bereits Codeklone im System sind, so sollten diese aus der Software entfernt werden. Auch das Warten der Software kann erschwert werden, wenn ein möglicher Fehler nicht nur an einer Stelle aus dem System entfernt werden muss, sondern auch noch aus den entsprechenden Kopien [105]. Möchte man nur einen Teil der Software wiederverwenden, dann können Codeklone diesen Vorgang erschweren. Also sollte ein **Schlechtes Design** [110] vermieden werden. Auch das **Erweitern oder Verbessern eines Systems wird erschwert** [65, 105], wenn Codeklone im System vorhanden sind. Durch den duplizierten Code wird mehr Aufmerksamkeit und Zeit benötigt, um das System zu verstehen und Änderungen oder Neuimplementierungen vorzunehmen. Durch Codeklone können auch die **benötigten Ressourcen steigen** [65]. Wenn es aufgrund von Softwareklonen mehr Quelltext gibt, so dauert der Kompilierungsprozess entsprechend länger, wodurch eventuell mehr Hardware- und Stromkosten entstehen. Bei *emdedded Systemen*, wie beispielsweise kleinen Mikrochips oder in Mikrocontrollern, gibt es nur eine sehr begrenzte Menge an Speicher. Codeklone können dazu führen, dass die entsprechenden *embedded Systeme* durch neuere ausgetauscht werden müssen, weil der Speicherplatz nicht mehr ausreichend ist. Werden Klone aufgespürt, dann können die Funktionen zusammengeführt werden [123], um Ressourcen einzusparen. Durch das Aufspüren von ähnlichen Programmfragmenten kann auch ***Malware*** **entdeckt** werden [30, 32, 164]. Wenn wichtige Programmfragmente einer Malware identifiziert wurden, so kann nach diesen Fragmenten in anderen Softwareprojekten gesucht werden, um so ähnliche Malwaresegmente aufzuspüren. Ein gleiches Vorgehen kann auch bei **Plagiats- oder Copyrightsverletzungen** angewendet werden. Auch bei der **Softwareevolutionsforschung** werden Codeklone für die Analyse genutzt [5, 6, 42, 107, 114].

Codeklone können aber unter gewissen Bedingungen auch positive Auswirkungen haben. Aber in genau diesen Szenarien ist es auch hilfreich, diese Codeklone zu identifizieren. Ein Codeklon könnte beispielsweise entstehen, wenn ein Codefragment besonders elegant, effizient oder gut verständlich implementiert ist und aus

diesem Grund oft kopiert wurde. Wenn das der Fall ist, würde sich dieses Programmfragment besonders gut für einen Kandidaten eignen, der in einer Programmerweiterungsbibliothek aufgenommen werden könnte [28, 41]. Das Detektieren eines Codeklons kann auch zum **Verbessern des Programmverständnisses** [122] beitragen. Wenn zum Beispiel eine Funktion gefunden wird, die komprimierte Daten entpackt, dann lässt sich aus Kopien dieser Funktion in anderen Softwareteilen schließen, dass in diesen Softwareteilen auch mit komprimierten Daten gearbeitet wird. Wenn eine Funktion oder Methode in der Lage ist, mehrere Aufgaben zu erfüllen, so kann aus den verschiedenen Einsatzgebieten der Methodenklone auf die Anwendungsmöglichkeiten für die Methode geschlossen werden. Das erleichtert das Verstehen darüber, wie die Methode funktioniert.

Kapitel 2

Begriffsbildung der Codeklonerkennung

In diesem Kapitel werden grundlegende Themen zur Codeklondetektion behandelt. Es wird versucht, in der Literatur eine Definition für einen Klon zu finden. Weiterhin wird geklärt, wie Klone entstehen, welche Strukturen bei der Codeklonerkennung zum Einsatz kommen und welche Möglichkeiten es gibt, ein Klonerkennungsverfahren zu evaluieren.

2.1 Codeklon

Bevor eine Begriffserklärung für einen *Klon* gesucht werden kann, muss zuerst eine Beschreibung für ein *Programmfragment* gefunden werden.

> Ein **Programmfragment** ist ein Teilprogramm, welches entweder aus einer kompletten Quellcodedatei oder nur aus einigen nicht-leeren Zeilen des Quelltextes besteht. Dabei kann es sich aber auch um kompilierte Versionen der Quellcodedateien, wie beispielsweise Bytecode- oder Binärcodedateien, handeln.

In dieser Arbeit werden Programmfragment, Codefragment und Programm(code)-segment als Synonyme genutzt. Zwei Codefragmente, die einen Codeklon zueinander bilden, nennt man auch *Klonpaar*. Wenn mehrere Codefragmente Klone zueinander sind, so bilden die Codefragmente eine *Klonklasse*.

> Nach Roy und Cordy [128] ist ein Codefragment, welches identische oder ähnliche Codefragmente besitzt, im Allgemeinen ein **Codeklon** zu eben diesen Codefragmenten.

Leider ist in der Arbeit von Roy und Cordy [128] nicht klar definiert, was *Ähnlichkeit* zwischen Codefragmenten bedeutet und es kann auch keine generische Definition von einem Codeklon abgeleitet werden. Aus diesem Grund werden hier die Klondefinitionen aus anderen Veröffentlichungen wiedergegeben. Baxter et al. [19] definieren Klone als ähnliche Codefragmente. Dabei geben die Autoren eine Schwellenwert-basierte Definition für ähnliche *Abstrakte Syntaxbäume* an. In verschiedenen weiteren Arbeiten [71,100] wurden Schwellenwert-basierte Definitionen für die *Ähnlichkeit* zwischen Programmfragmenten gesucht. Kamiya et al. [69] geben eine eher unpräzise formulierte Definition für Codeklone an. Sie definieren Klone als Teile einer oder mehrerer Sourcecodedateien, die *identisch* oder *ähnlich* zueinander sind. Mit *identisch* ist hier gemeint, dass eine exakte Kopie angelegt wurde. Jedoch wird auch in dieser Arbeit keine Definition für die *Ähnlichkeit* gegeben.

Um das Problem mit der Definition der *Ähnlichkeit* zu umgehen, kombinierten einige Veröffentlichungen [20, 21, 109] die Ergebnisse mehrerer Codeklondetektoren und nutzten das Ergebnis zum Evaluieren ihrer eigenen Tools. Allerdings bleibt hier die Frage offen, inwieweit Menschen diesen Ergebnissen der Tools zustimmen würden. Im Allgemeinen wird eine Kombination aus textueller und tokenbasierter syntaktischer Übereinstimmung genutzt, um ähnliche syntaktische Codeklone zu identifizieren. Allerdings gibt es bisher kein Maß, um die Ähnlichkeit zweier semantisch ähnlichen, aber syntaktisch komplett verschiedenen Codefragmente zu messen. Auch Roy et al. [127] sagen bei Ihrer Definition eines Codeklons, dass zwei Codefragmente ein Codeklon bilden, wenn sie durch *irgendeine gegebene Definition von Ähnlichkeit* als ähnlich anzusehen sind.

Ein anderes Problem bei Codeklonen besteht darin, in welchem Größenbereich sich Codefragmente bewegen müssen, um in einem Klonerkennungsprozess Beachtung zu finden. In objektorientierten Programmiersprachen gibt es beispielsweise sogenannte *getter*- und *setter*-Methoden, die eine Instanzvariable schreiben oder zurückgeben, da kein direkter Zugriff auf die Instanzvariable erlaubt sein soll. Diese *getter*- und *setter*-Methoden bestehen häufig aus einer Zeile im Methodenrumpf. Natürlich sind fast alle *getter*-Methoden und fast alle *setter*-Methoden zueinander Codeklone und be-

stimmt gibt es Anwendungsfälle, wo genau diese Art der Codeklone gesucht werden sollen. Aufgrund der syntaktischen Übereinstimmung sind eben solche Codeklone relativ leicht zu erkennen und da es meist in objektorientierten Programmiersprachen recht viele dieser Methoden gibt, würde die Liste der gefundenen Klone sehr groß werden. Natürlich ist man gerade bei der Entwicklung eines Klonerkennungstools mehr an größeren und komplexeren Codefragmenten interessiert, die schwerer zu erkennen sind. Genau wegen dieser und ähnlicher Problematiken besteht also die Frage, ab welcher Codegröße sollten Codefragmente bei der Codeklondetektion Beachtung finden.
Roy und Cordy [128] haben dieses Themengebiet bereits behandelt und der folgende Abschnitt basiert auf den Berichten von Roy und Cordy. Einige Studien [69, 72, 79] kommen zu dem Schluss, dass für Token-basierte Klondetektoren (Abschnitt 3.2) Codefragmente in Betracht gezogen werden sollten, die mindestens 30 Tokens besitzen. Andere Studien argumentieren hingegen, dass Codefragmente in Abhängigkeit zu den enthaltenen Zeilen in Betracht für eine Klonerkennung gezogen werden sollten. Verschiedene Veröffentlichungen schlagen aber auch hier verschiedene Werte für die minimale Zeilenanzahl eines Codefragments vor. Baker [14] nutzt beispielsweise 15 Codezeilen und Johnson [65] 50 Codezeilen. Bellon [20] nutzte in seinen Arbeiten hingegen nur 6 Zeilen. In den Veröffentlichungen [82, 99] einiger syntaktischer Verfahren (Abschnitt 3.3) wird mit der Anzahl von Knoten in den aus dem Quelltext generierten Graphstrukturen gearbeitet. Andere Veröffentlichungen [90, 105] arbeiten hingegen nur mit Funktionen anstatt mit beliebigen Codefragmenten. Bei diesen werden also nur ganze Funktionen als Codefragmente genutzt, wobei dann keine Mindestgröße vorgegeben ist. Im Allgemeinen werden bei der Evaluation eines Klondetektors je nach verwendeter Technik und gegebener Problemstellung verschiedene Mindestgrößen für Programmcodefragmente festgelegt.

2.2 Wie entstehen Klone?

Klone können beabsichtigt oder auch unbeabsichtigt entstehen. Einerseits kann bewusst eine Kopie eines Programmfragments angelegt und benutzt werden. Andererseits können zwei verschiedene Entwicklerinnen/Entwickler bei ähnlicher Problemstellung zwei sehr ähnliche Programmfragmente erzeugen. Roy und Cordy [128] haben eine Beschreibung für mögliche Gründe zur Entstehung von Codeklonen verfasst. Zusammenfassend lassen sich folgende Gründe zur Entstehung von Codeklonen aus der Arbeit von Roy und Cordy [128] nennen.

Der **Wiederverwendungsansatz** in der Softwareentwicklung ist einer der Hauptgründe für Codeklone. Das Kopieren und Wiederverwenden von einem Programmcode, der beispielsweise auf StackOverflow[1] gefunden wird, ist eine häufige Praxis vor allem bei Programmieranfängern. Natürlich werden häufig auch ganze *Forks* von Projekten angelegt. Damit ist gemeint, dass ein Softwareprojekt komplett kopiert und anders weiterentwickelt wird als das Original. Roy und Cordy [128] führen hier das Beispiel eines Betriebsystem-Treibers an, der für eine bestimmte Hardware entwickelt wurde. Durch das Kopieren des kompletten Projektes und entsprechenden Anpassungen könnte er für eine andere Hardware weiterentwickelt werden. Da allerdings nicht jedes Programmfragment in der Software angepasst oder komplett verändert werden muss, entstehen auch zwischen dem originalen und kopierten Projekt Codeklone. Auch bei verschiedenen anderen Programmieransätzen können Klone entstehen. Beim *Mergen* von zwei Systemen, also dem Zusammenfügen von verschiedenen Projekten, mit dem Ziel, ein gemeinsames und mächtigeres Softwaresystem zu implementieren, können Klone entstehen. Wenn eine ähnliche Funktionalität in beiden Ausgangsprojekten enthalten war und diese Redundanz beim Zusammenfügen der Projekte nicht aufgelöst wurde, entstehen Klone. Auch bei der Verwendung von speziellen Tools, die einfachen Code generieren können, kommt es zu immer gleichem Code und damit zu Klonen. Diese Codegenerierungsverfahren benutzen oft ähnliche Vorlagen, wodurch eine große Menge an sehr ähnlichen oder gleichen Codes entsteht.
Auch die **Wartung von Softwareprojekten** kann ein Grund für die Verwendung von Softwareklonen sein. Als Beispiel führen Roy und Cordy [128] die Softwareentwicklung in der Finanzindustrie aus. Im Finanzsektor kann aufgrund einer Fehlfunktion oder eines Ausfalls der Software ein enorm hoher Schaden entstehen. Aus diesem Grund könnte es von Vorteil sein, gut getestete und bewährte Codefragmente zu kopieren und wiederzuverwenden. In vielen Bereichen der Softwareentwicklung wird strikt in bekannten Programmiermustern (wie beispielsweise dem *Model-View-Controller* Prinzip) entwickelt. Auch hier kann das Erzeugen von Klonen sinnvoll sein, um die Vorgaben der Programmiermuster nicht zu verletzen. Wenn Codeklone komplett unabhängig voneinander sind, können diese auch in einer unterschiedlichen Geschwindigkeit entwickelt werden, wobei ein erneuter Test dann nur für die angepasste Version nötig ist. Ein weiterer Grund für das bewusste Integrieren von Klonen könnte die robuste Redundanz in kritischen Systemen sein. So könnte es vorkommen, dass für den Menschen lebenswichtige Funktionalitäten einer Software von

[1] https://stackoverflow.com/

verschiedenen Teams implementiert und redundant nebeneinander ausgeführt werden. Sollte die eine Implementierung eine Schwachstelle oder einen Fehler enthalten, so könnte auf die andere Implementierung gewechselt werden, die diese Schwachstelle eventuell nicht besitzt. Codeklone können auch bewusst in Echtzeitprogrammen zum Einsatz kommen, wenn der genutzte Compiler kein *Inlining* unterstützt, aber Funktionsaufrufe insgesamt zu viel Zeit benötigen.
Weitere Gründe für Codeklone sind **Einschränkungen der genutzten Programmiersprache oder der Entwicklerin/ des Entwicklers**. Falls die genutzte Programmiersprache keine oder nicht ausreichende Abstraktionsmechanismen hat, wie beispielsweise generische Klassen in der Sprache Java, dann müssen Entwicklerinnen beziehungsweise Entwickler zwangsläufig die gleiche Funktionalität mit verschiedenen Datentypen implementieren. Eventuell wäre es mit sehr viel Aufwand möglich, dieses Problem zu umgehen, aber die gefundene Lösung wäre dann sehr viel komplexer und fehleranfälliger als das Benutzen zweier sehr ähnlicher Funktionen. Weiterhin wäre die Entwicklung einer generischen Funktionalität ohne passende Unterstützung der Programmiersprache ressourcenintensiver. Auch dies wären Gründe für das bewusste Nutzen von Codeklonen. Zudem können Codeklone entstehen, wenn Einwicklerinnen beziehungsweise Entwickler nur wenig Zeit für die Entwicklung von bestimmten Funktionalitäten eines Softwareprojektes zur Verfügung haben. Es werden dann bestehende Programmfragmente genutzt und kombiniert, um die Funktionalität schnellstmöglich zu implementieren. In einigen Fällen wird die Produktivität von Entwicklerinnen und Entwicklern aber auch an den implementierten Codezeilen gemessen. In einem solchen Szenario ist es für Entwicklerinnen und Entwickler verlockend, passende Codezeilen immer wieder zu kopieren und erneut zu verwenden, um möglichst viel Code zu produzieren, auch wenn das nicht einem „guten“ Entwicklungskonzept entspricht. Ein weiterer Grund für den bewussten Einsatz von Codeklonen könnte sein, dass eine Entwicklerin oder ein Entwickler nicht die Berechtigung besitzt, gewisse Teile des Quellcodes anzupassen. Das könnte darauf zurückzuführen sein, dass der Quelltext von einer anderen beteiligten Partei im Entwicklungsprojekt stammt. Hier wäre die einzige Möglichkeit, den Quelltext zu kopieren und entsprechend anzupassen.
Roy und Cordy [128] sagen weiterhin, dass auch **Klone durch Zufall** entstehen können. Wenn beispielsweise mit *Application-Programming-Interfaces* (APIs) gearbeitet wird, dann sind oft gleiche Logikschritte, wie das Anlegen von Datenstrukturen und das Aufrufen spezieller Bibliotheken oder Funktionen nötig, um mit den APIs zu arbeiten. Auf diese Weise können unbeabsichtigt Klone entstehen. Auch

wenn zwei Entwicklerinnen beziehungsweise Entwickler am gleichen Problem unabhängig voneinander oder sogar in verschiedenen Projekten arbeiten und zu einer ähnlichen Lösung des Problems kommen, können Klone entstehen. Weiterhin kann eine Entwicklerin beziehungsweise ein Entwickler ein bestimmtes Vorgehen bei der Lösung von Problemstellungen unbeabsichtigt erneut anwenden und so Codeklone erstellen.

2.3 Klontypen

Klone können in verschiedene Klontypen eingeteilt werden. Von exakten Kopien bis hin zu Teil- oder Unterklonen gibt es mehrere Kategorien, die von Roy und Cordy [128] zusammengefasst wurden und hier kurz vorgestellt werden. Dabei können sich die Klontypen auch überlagern und ein Klon kann zu verschiedenen Typen zählen.

Exakte Klone sind Programmfragmente, die exakte Kopien voneinander sind. Bei exakten Klonen können aber auch Kommentare, Leerzeichen oder Zeilenumbrüche verändert worden sein. **Umbenannte Klone** sind exakte Klone, bei denen aber zusätzlich noch Bezeichnernamen und die Werte von Literalen verändert sein können. Wenn bei den umbenannten Klonen die Umbenennungen konsistent durchgeführt wurden, sodass die Semantik komplett erhalten bleibt, so nennt man diese Klone auch **Parametrisierte Klone**. Wenn ein Klon zusätzlich zu den umbenannten und parametrisierten Kloneigenschaften noch kleine Veränderungen, Einfügungen oder Löschungen aufweist, ohne dass die Programmfragmente stark unterschiedliche Funktionalitäten haben, dann nennt man diese Klone ***Near-Miss* Klone**. Ein ***Gapped* Klon** ist hingegen ein Programmfragment, welches kopiert wurde und

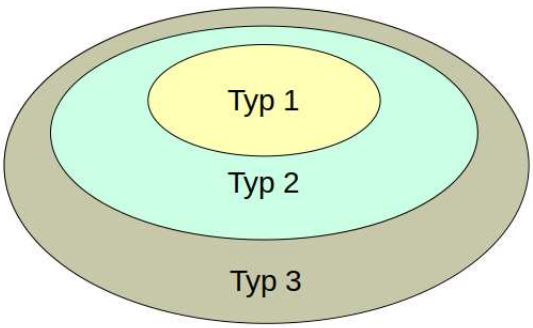


Abbildung 2.1: Übersicht von Typ-1- bis Typ-3-Klonen.

Instruktionen in der Mitte eingefügt wurden. Als **Strukturelle Klone** werden Programmfragmente bezeichnet, die eine ähnliche Designstruktur besitzen. Dabei lösen sie eine gleiche oder ähnliche Aufgabe, nutzen dafür aber nicht unbedingt die exakt gleichen Kontrollstrukturen. Weiterhin bewegen sich die Programmfragmente von strukturellen Klonen immer in syntaktischen Grenzen der jeweiligen Programmiersprache. Beispiele hierfür könnten ganze Klassen, Funktionen oder einfach Blöcke ({ ... }) sein. Wenn alle Programmsegmente jedoch aus einer gesamten Funktion oder Prozedur bestehen, dann nennt man entsprechende Klone auch **Funktionsklone**. Damit sind Funktionsklone eine Unterkategorie von strukturellen Klonen. Wenn zwei Programmfragmente die gleiche Funktionalität erfüllen, aber Anweisungen vertauscht werden, dann nennt man diese Klone **Umsortierte Klone**. Die vertauschten Anweisungen dürfen allerdings nicht voneinander abhängig sein, denn dann könnte sich die Semantik des Fragments ändern. **Allgegenwärtige Klone** bestehen meist aus kleinen Programmfragmenten, die in vielen Quellcodedateien auftauchen und jeweils Klonpaare bilden.
Neben den hier vorgestellten Klontypen stellen Roy und Cordy [128] auch noch Klontypen in Bezug auf den Entwicklungsfortschritt eines Softwareprojektes vor. Also Klontypen, die beschreiben, wie sich Klone innerhalb eines Softwareentwicklungszyklus verändern. Aufgrund der fehlenden Relevanz für die hier vorgelegte Arbeit wurden diese Klontypen an dieser Stelle nicht weiter behandelt.
Generell werden alle Klone in vier Typen eingeteilt [20, 86, 127]:

- **Typ-1-Klone** sind identische Codefragmente, bei denen Leerzeichen, Zeilenumbrüche und Kommentare verändert sein können. Ansonsten müssen alle Anweisungen exakt gleich sein.
- **Typ-2-Klone** können zusätzlich zu den Veränderungen bei Typ-1-Klonen noch veränderte Variablennamen, Literale und Typen besitzen.
- **Typ-3-Klone** haben zusätzlich zu den Eigenschaften von Typ-2-Klonen noch hinzugefügte, geänderte oder gelöschte Anweisungen.
- **Typ-4-Klone** bestehen aus Codefragmenten, die die gleiche Funktionalität erfüllen, aber auf syntaktisch komplett verschiedene Weise implementiert sind.

In Abbildung 2.1 kann man das Prinzip erkennen, dass beispielsweise Typ-3-Klone auch die Eigenschaften von Typ-2- und Typ-1-Klonen besitzen. In Abbildung 2.2 ist

```
1 int func(int[] array) {
2     int sum = 0;
3     for (int i = 0; i<array.length; i++)
4         if (array[i]>0)
5             sum *= array[i];
6     return sum;
7 }
```

Abbildung 2.2: Funktionsbeispiel.

```
1 int func( int[] array ) {
2     int sum = 0;
3     for ( int i = 0; i < array.length; i++ )
4         if ( array[i] > 0 )
5             sum *= array[i];
6     return sum;
7 }
```

Abbildung 2.3: Typ-1-Klon zu der Funktion in Abbildung 2.2.

```
1 int func(int[] arr) {
2     int s = 0;
3     for (int i = 0; i<arr.length; i++)
4         if (arr[i]>0)
5             s *= arr[i];
6     return s;
7 }
```

Abbildung 2.4: Typ-2-Klon zu der Funktion in Abbildung 2.2.

```
1 int func(int[] arr) {
2     int s = 0;
3     for (int i = 0; i<arr.length; i++)
4         if (arr[i]>0)
5             s *= (arr[i] * fu());
6     return s;
7 }
```

Abbildung 2.5: Typ-3-Klon zu der Funktion in Abbildung 2.2.

```
1 int func(int[] array) {
2     int sum = 0;
3     int i = 0;
4     while (i<array.length) {
5         sum = (array[i]>0) ? sum * array[i] : sum;
6         i = i+1;
7     }
8     return sum;
9 }
```

Abbildung 2.6: Typ-4-Klon zu der Funktion in Abbildung 2.2.

eine Beispielfunktion abgebildet, für die in Abbildung 2.3 ein Typ-1-Klon, in Abbildung 2.4 ein Typ-2-Klon, in Abbildung 2.5 ein Typ-3-Klon und in Abbildung 2.6 ein Typ-4-Klon zu finden ist.
Svajlenko et al. [147, 149] unterteilen Typ-3- und Typ-4-Klone zusätzlich weiter.

- **Very-Strongly Typ-3-Klone** haben eine syntaktische Übereinstimmung von mindestens 90%.
- **Strongly Typ-3-Klone** sind sich mindestens zu 70%, aber weniger als 90% syntaktisch ähnlich.
- Zwischen 50% und weniger als 70% syntaktischer Übereinstimmung von zwei Programmfragmenten spricht man von **Moderately Typ-3-Klonen**.
- Alle Programmfragmente mit weniger als 50% syntaktischer Übereinstimmung nennt man auch **Weakly Typ-3-** oder **Typ-4-Klone**.

2.4 Strukturen bei der Codeklonerkennung

Bei der Entwicklung von Codeklondetektoren werden die verschiedensten Verfahren und Herangehensweisen genutzt, um möglichst viele Klone zu finden. Einige der wichtigsten Strukturen werden in diesem Abschnitt beschrieben. Eine grundlegende Struktur ist der *gerichtete Graph*, dessen Definition angelehnt an Cormen et al. [37, S. 1179] wie folgt lautet:

> Ein ***gerichteter Graph*** oder *Digraph* G besteht aus Knoten von G in einer Menge $N = N(G)$. Weiterhin besteht G aus einer Menge $E = E(G)$ von geordneten Paaren $E \subseteq N \times N$, die Kanten von G genannt werden. Ein *gerichteter Graph* wird als $G = (N, E)$ beschrieben.

In der Visualisierung des Kontrollflussgraphen in Abbildung 2.7, der auch ein gerichteter Graph ist, lässt sich gut erkennen, dass Knoten miteinander verbunden sind und dadurch *Wege* bilden. Die Definition für einen Weg ist aus Cormen et al. [37, S. 1180] genommen.

> Ein **Weg oder Pfad** der Länge k von einem Knoten n zu einem Knoten n' in einem Graphen $G = (N, E)$ ist eine Folge $\langle n_0, ..., n_k \rangle$ von Knoten, für die $n = n_0$, $n' = n_k$ und $(n_{i-1}, n_i) \in E$ für $i = 1, ..., k$ gilt. Der Pfad

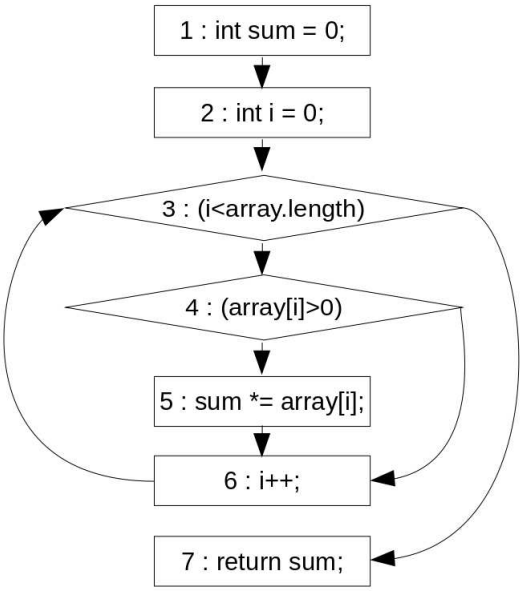


Abbildung 2.7: Kontrollflussgraph zur Funktion aus Abbildung 2.2.

enthält die Knoten $n_0, ..., n_k$ und die Kanten $(n_0, n_1), ..., (n_{k-1}, n_k)$. Wenn es einen Pfad p von n nach n' gibt, so sagen wir, dass n' von n aus über p erreichbar ist.
Ein **Teilpfad** des Pfades $p = \langle n_0, ..., n_k \rangle$ ist eine Teilsequenz von Knoten aus p, die zusammenhängend sein muss. Für jedes Paar i und j mit $0 \leq i \leq j \leq k$ ist die Teilsequenz $p' = \langle n_i, ..., n_j \rangle$ also ein Teilpfad von p.

Pfade oder Pfadmengen können bei der Darstellung in dieser Arbeit zur besseren Lesbarkeit auch mit einem Pfeil $\rightarrow$ als Trennzeichen der Knoten anstatt der korrekten Schreibweise mit einem Komma dargestellt werden. Eine Besonderheit bei Wegen ist es, wenn diese **Zyklen beziehungsweise Schleifen** enthalten. Eine Schleife in einem Weg W liegt vor, wenn eine Kante doppelt enthalten ist. Ein Graph enthält dann mindestens eine Schleife, wenn es mindestens einen Pfad im Graphen gibt, der eine Schleife enthält. Ein besonders wichtiger Graph für die Codeklonerkennung ist der *Kontrollflussgraph (CFG)*, der besonders bei den syntaktischen Verfahren im Abschnitt 3.3 und bei den maschinellen Lernverfahren im Abschnitt 3.6 zur Codeklonerkennung eingesetzt wird. Die nachfolgende Definition legt fest, dass ein Kontrollflussgraph ein gewöhnlicher gerichteter Graph mit nur einem Start- bzw. Endknoten ist und entstammt dem Buch von Zima [188, S. 57].

Ein **Kontrollflussgraph** $CFG = (N, E)$ ist ein gerichteter Graph (N,E) mit folgenden Eigenschaften:

1. Von einem Knoten $n \in N$, der ein Startknoten n_{Start} in G ist, gibt es nur genau einen.

   $$|\{n_{\text{Start}}\}| = 1$$

   Dieser Startknoten hat genau eine ausgehende Kante e.

   $$|n_{\text{Start}} \triangleleft| = |\{e_{\text{Start}}\}| = 1$$

2. Von einem Knoten $n \in N$, der ein Endknoten n_{End} in G ist, gibt es nur genau einen.

   $$|\{n_{\text{End}}\}| = 1$$

   Dieser Endknoten hat genau eine eingehende Kante e.

   $$|\triangleright n_{\text{End}}| = |\{e_{\text{End}}\}| = 1$$

3. Eine jede Kante $e \in E$ liegt auf einem Weg von e_{Start} zu e_{End}.

   $$\forall(e \in E)\exists(W \in W_{\text{e}_{\text{Start}}} \rightarrow W_{\text{e}_{\text{End}}}) : e \in W$$

Wenn ein Kontrollflussgraph mehr als einen Start- oder Endknoten hat, dann wird ein expliziter Start- bzw. Endknoten eingefügt. Die Knoten in dem Kontrollflussgraphen in Abbildung 2.7 sind nummeriert und enthalten aufgrund der besseren Übersicht zusätzlich noch die jeweiligen Instruktionen als Text. In den folgenden Abbildungen werden entweder die Nummerierung und die Instruktionen oder nur der Instruktionstext als Beschriftung genutzt. Es existiert eine Funktion, für die man durch die Übergabe der Knotennummer den entsprechenden Instruktionstext des Knotens erhält. Weiterhin ist in Abbildung 2.7 klar zu erkennen, dass ein Knoten jeweils genau eine Instruktion abbildet.
Die in einem Kontrollflussgraphen enthaltenen Knoten haben gewisse Abhängigkeiten zueinander, die durch die Dominatorrelation beschrieben werden. Die Definition der Dominatorrelation stammt von Zima [188, S. 59].

Die **Dominatorrelation** auf dem Kontrollflussgraph $G = (N, E)$ mit den Knoten $n, n' \in N$ ist folgend definiert.

1. n dominiert n' ($n \leqslant n'$), wenn jeder Pfad von n_{Start} zu n' auch n enthält.
2. n dominiert n' streng ($n < n'$), wenn $n \leqslant n'$ und $n \neq n'$ gilt.
3. n dominiert n' direkt ($n <_{\text{d}} n'$), wenn $n < n'$ gilt und es kein $n'' \in N$ gibt, für das $n < n'' < n'$ gilt.

4. $DOM(n) := \{n' : n' \leqslant n\}$ nennt man die Dominatorenmenge von n.

Diese Dominatorrelationen haben einige wichtige Eigenschaften, wie von Zima [188, S. 59] ausgeführt wird.

Sei $G = (N, E)$ ein Kontrollflussgraph mit der Dominanzrelation $\leqslant$, dann gelten folgende **Eigenschaften**:

1. $DOM(n_{\text{Start}}) = \{n_{\text{Start}}\}$
2. Ein jeder Knoten $n \in N$ hat als Dominator den Startknoten n_{Start}. $\forall\, n \in N : n_{\text{Start}} \in DOM(n)$
3. $\leqslant$ ist eine reflexive Teilordnung von N.
4. Für einen jeden Knoten $n \in N$, ist $DOM(n)$ linear sortiert durch $\leqslant$.
5. Der direkte Dominator eines jeden Knotens $n' \in N$ ist eindeutig.
6. Wenn $n <_{\text{d}} n'$ gilt, dann ist n der letzte Dominator von n' in jedem Pfad von n_{Start} zu n'. Für jeden Knoten n'' gilt $n'' < n$, wenn $n'' < n'$ und $n'' \neq n$ gelten.

Es ist wichtig an dieser Stelle anzumerken, dass ein direkter Dominator eines Knotens nicht der direkte Vorgängerknoten in einem Kontrollflussgraphen sein muss. Die eben genannte 5. und die 6. Eigenschaft implizieren, dass die Dominatorrelation als Baum dargestellt werden kann. Diesen Baum nennt man Dominatorbaum. Dominatorbäume werden in der hier vorliegenden Arbeit zum Erkennen von Codeklonen genutzt. Im Kapitel 4 wird hierzu ein auf klassischen Analysemethoden basierender Weg vorgestellt und im Kapitel 7 werden Dominatorbäume als Eingabe in ein maschinelles Lernverfahren zum Aufspüren von Codeklonen genutzt.

Der **Dominatorbaum** eines Kontrollflussgraphen ist formal definiert als ein gerichteter Graph $G = (N, E^*)$, wobei $E^* = \{(n, n') | n <_{\text{d}} n' \text{ in } G\}$.

Der Dominatorbaum des Kontrollflussgraphen aus Abbildung 2.7 kann in der Abbildung 2.8 betrachtet werden. Der wesentliche Unterschied eines Dominatorbaums zu einem Kontrollflussgraphen liegt darin, dass er niemals Zyklen enthält.

Eine weitere wichtige Struktur, die zur Codeklonerkennung eingesetzt wird, ist der Syntaxbaum. Beim Durchlaufen des Codes wird zur nachfolgenden Weiterverarbeitung (zum Beispiel in Compilern) der Syntaxbaum aufgebaut, der die wesentliche

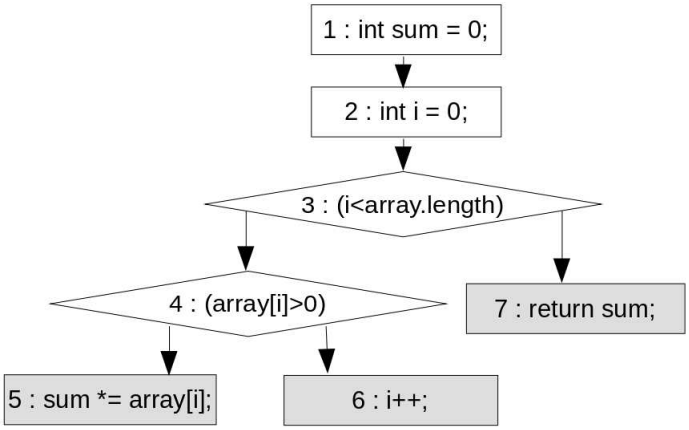


Abbildung 2.8: Dominatorbaum des Kontrollflussgraphen aus Abbildung 2.7.

syntaktische Struktur des Quelltextes beibehält, wie Wirth [172, S. 146] beschreibt. Nach Priese und Erk [117, S. 110] ist ein Ableitungs-, Parse- oder Syntaxbaum wie folgt beschrieben:

> Ein **Ableitungs-, Parse- oder Syntaxbaum** zu einer Grammatik V ist ein gerichteter Graph $G = (N, E)$, für den das Folgende gilt:
>
> 1. Jeder Knoten $n \in N$ ist mit einem Symbol aus Variablen V, Terminalen T oder dem leeren Wort belegt. $(V \cup T \cup \{\epsilon\})$
> 2. Jeder innerer Knoten (Knoten mit mindestens einem Sohnknoten) ist mit V belegt.
> 3. Jeder Blattknoten (Knoten ohne einen Sohnknoten) ist mit einem Symbol aus $T \cup \{\epsilon\}$ belegt.
> 4. Ist $n \in N$ ein innerer Knoten mit Söhnen $n_1, ..., n_k$ in dieser Anordnung, ist A die Belegung von n und A_i die Belegung von n_i, so ist $A \rightarrow A_1...A_k \in R$.
> 5. Ein mit ϵ belegter Blattknoten hat keinen Bruderknoten (Zwei Knoten mit dem gleichen Vaterknoten).

Bei einem **Abstrakten Syntaxbaum (AST)** handelt es sich dann um eine „Vergröberung“ des eben beschriebenen Syntaxbaums. Dabei wird versucht, möglichst viele Variablen (auch Nichtterminale genannt) aus dem Baum zu entfernen. Da das Vorgehen zur Erstellung und damit auch die Eigenschaften des *Abstrakten Syntaxbaums* jedoch sehr stark von der angestrebten Verwendung abhängt, wird zu einem

Abstrakten Syntaxbaum an dieser Stelle keine einheitliche formale Definition getroffen.
Natürlich gibt es neben diesen hier vorgestellten noch weitere Strukturen, die bei der Codeklonerkennung eingesetzt werden. Beispiele hierfür wären Tokensequenzen, Suffixbäume, Programmabhängigkeitsgraphen oder Bytecodeabhängigkeitsgraphen. Diese sollen jedoch an dieser Stelle nicht formal definiert werden, da in dieser Arbeit nicht mit solchen Strukturen gearbeitet wird. Dennoch arbeiten andere Klonerkennungsansätze, wie man im Kapitel 3 nachlesen kann, mit eben solchen Strukturen.

2.5 Evaluation von Klonerkennungsverfahren

In diesem Abschnitt werden sowohl Bewertungsmaßstäbe als auch Benchmarks vorgestellt, die die Güte von einem Klonerkennungsverfahren messen. Dabei gilt, dass die Bewertungsmaße aussagen, wie effektiv ein Verfahren arbeitet. Diese Bewertungsmaße werden mithilfe von Benchmarks ermittelt.

Bewertungsmaße

Zur Evaluation von Klonerkennungsverfahren, die beispielsweise auf dem AST oder dem CFG basieren, braucht es Messmethoden, um die Güte der Verfahren zu bestimmen. Wenn ein Verfahren Werte durch Berechnungen oder andere Verfahren ermittelt, dann gibt es richtig ermittelte Werte **(Wahr Positiv (TP))**, Werte, die fälschlicherweise ermittelt wurden **(Falsch Positiv (FP))**, Werte, die zurecht nicht ermittelt wurden **(Wahr Negativ (TN))** und Werte, die nicht ermittelt wurden, obwohl sie relevant sind **(Falsch Negativ (FN))**. Wenn ein Klonerkennungsverfahren Klone detektiert, dann bilden alle gefundenen Paare von Codefragmenten die *ermittelten Paare* ***(N_{erm}=TP+FP)***. Die tatsächlich in dem Datensatz enthaltenen Klone nennt man die tatsächlichen Klone ***(N_{tat}=TP)***. Besonders wichtig sind in diesem Zusammenhang der *Recall-Wert* und der *Precision-Wert*, deren allgemeine Definition nach Buckland und Grey [26] wie folgt lautet:

> Der **Recall-Wert** oder auch der **Erkennungsfaktor** beschreibt für eine berechnete oder ermittelte Elementmenge N_{erm}, die Anzahl an tatsächlich wichtigen Elementen in den ermittelten Elementen $N_{\text{erm}} \cap N_{\text{tat}}$, im Verhältnis zu allen tatsächlich wichtigen Elementen N_{tat}.
>
> $\text{Recall} = \frac{N_{\text{erm}} \cap N_{\text{tat}}}{N_{\text{tat}}} = \frac{TP}{TP+FN}$

Auf die Codeklonerkennung bezogen würde das bedeuten, wie viele tatsächliche Klone ($N_{\text{erm}} \cap N_{\text{tat}}$ oder TP) von in einem Datensatz insgesamt enthaltenen Klonen (N_{tat} oder $TP+FN$) findet ein Klonerkennungsverfahren. Ein Problem hierbei liegt nun darin, dass theoretisch alle in einem Datensatz enthaltenen Klone bekannt sein müssen, um einen Recall-Wert berechnen zu können.
Um einen Recall-Wert von 100% zu erhalten, könnte ein Tool einfach jede beliebige Kombination aus Programmfragmenten als Klon identifizieren. Das wäre aber natürlich für keinen Anwendungszweck sinnvoll. Aus diesem Grund bedarf es zusätzlich zu einer *Recall-Messung* auch immer noch der Messung des *Precision-Werts*.

> Der **Precision-Wert** oder der **Genauigkeitsfaktor** beschreibt für eine berechnete oder ermittelte Elementmenge N_{erm} die Anzahl an tatsächlich relevanten Elementen N_{tat} in der ermittelten Elementmenge $N_{\text{erm}} \cap N_{\text{tat}}$ im Verhältnis zu der Gesamtanzahl an berechneten oder ermittelten Elementen N_{erm}.
>
> Precision-Wert $= \frac{N_{\text{erm}} \cap N_{\text{tat}}}{N_{\text{erm}}} = \frac{TP}{TP+FP}$

Auf die Codeklonerkennung bezogen würde der Precision-Wert beschreiben, wie viele tatsächliche Klone ($N_{\text{erm}} \cap N_{\text{tat}}$ oder TP) in der Menge aller von einem Klonerkennungsverfahren ermittelten potentiellen Klone (N_{erm} oder $TP + FP$) sind. Idealerweise strebt jedes Verfahren danach, sowohl beim Recall-Wert als auch beim Precision-Wert 100% zu erreichen. Jedoch ist es bei der Codeklonerkennung meist so, dass wenn der Recall-Wert steigt, dann der Precision-Wert fällt und umgekehrt. Um diesen Kompromiss zwischen dem Recall-Wert und dem Precision-Wert abzubilden, gibt es den *F1-Wert*.

> Der **F1-Wert** bildet einen Mittelwert aus Recall-Wert und Precision-Wert, wobei die Recall- und die Precisison-Werte immer zwischen 0 und 1 liegen.
>
> F1 $= 2 \times \frac{Recall \times Precision}{Recall + Precision}$

Der F1-Wert wird oft bei maschinellen Lernverfahren als Gütemaß genutzt. Neben dem F1-Wert gibt es auch noch andere Maße wie beispielsweise die ***Area Under the Receiver-Operating-Characteristic Curve***, mit denen in dieser Arbeit nicht gearbeitet wird. Aus diesem Grund werden sie an dieser Stelle auch nicht weiter beschrieben oder definiert.

Benchmarks

Nachdem nun grundlegende Begriffe zur Beschreibung der Güte von Klonerkennungsverfahren besprochen wurden, soll nun aufgearbeitet werden, wie diese Güte in älteren Arbeiten bestimmt wurde und in neueren bestimmt werden kann. In der Arbeit von Svajlenko und Roy [151] wurden die verschiedenen Möglichkeiten zur Gütemessung und die veröffentlichten Benchmarks aufgearbeitet. Angelehnt an diese Arbeit werden die verschiedenen Möglichkeiten zur Validierung von Klonerkennungsverfahren hier vorgestellt.

Einige Experimente haben in der Vergangenheit den Recall-Wert ignoriert und lediglich den Precision-Wert gemessen [47, 63, 83, 87, 99]. Dazu wurden die vom Klonerkennungsverfahren als Klone identifizierte Fragmentpaare manuell überprüft. Um das Problem mit der Recall-Messung zu lösen, wurde in einigen Experimenten die Vereinigung der gefundenen Klone von mehreren anderen Klonerkennungsverfahren als Referenz genutzt [21, 27, 43, 111, 159].

Mithilfe der *Levenshtein-Distanz* [95] haben Lavoie und Merlo [92] automatisiert einen Typ-3-Klonbenchmark erstellt. Durch das Nutzen der Levenshtein-Distanz mussten keine manuellen Klonvalidierungen durchgeführt werden und ein kompletter Benchmark sowohl für Recall- als auch für Precision-Messungen war das Ergebnis. Der Nachteil ist allerdings, dass die Levenshtein-Distanz wiederum selbst einen Recall- und einen Precision-Wert besitzt. Aber zumindest kann mithilfe dieses Benchmarks ein Vergleich zwischen dem zu testenden Klonerkennungsverfahren und der Levenshtein-Distanz durchgeführt werden.

Krutz und Le [89] verfolgten den Ansatz, mithilfe von Expertinnen und Experten einen Datensatz zu validieren. Dazu haben mehrere Expertinnen und Experten manuell 1.536 zufällig gewählte Methodenpaare ausgewertet, um einen Klonbenchmark mit 66 Klonpaaren bestehend aus allen 4 Klontypen zu erstellen. Der Benchmark hat durch die vielen Expertinnen und Experten eine hohe Zuverlässigkeit, jedoch hat er nicht genügend Klone und damit auch nicht die Klonvielfalt, die ein guter Benchmark benötigt.

Yuki et al. [183] wählen im Gegensatz wieder einen Ansatz zum Erstellen eines Benchmarks, bei dem es keine manuellen Validierungen braucht. Dabei wurden automatisch Versionshistorien von Softwareprojekten analysiert, um nach Methodenzusammenführungen zu suchen, bei denen beide Ausgangsmethoden Codeklone zueinander waren. Der Nachteil dieses Verfahrens liegt darin, dass nur Codeklone aufgespürt werden, die den Entwicklerinnen und Entwicklern bekannt sind beziehungsweise waren.

Der sogenannte **Google Code Jam** Datensatz wurde von Wagner et al. [162] erstellt. Die Autoren nutzten bei Google Code Jam eingereichte Abgaben von Programmieraufgaben, um einen Benchmark mit semantisch ähnlichen Programmen zu erstellen. Dabei verließen sich die Autoren auf den Prozess von Google Code Jam, um die semantische Ähnlichkeit der Lösungen zu überprüfen und validierten sie nicht selbst. In einem ersten Schritt wurden 58 Codeklonpaare als eine erste Version des Benchmarks veröffentlicht.

Bellon's Benchmark [21] ist wohl einer der bekanntesten Benchmarks zur Codeklonerkennung, für den zudem mehrere Erweiterungen erstellt wurden [111, 148]. Bellon et al. validierten 2% aller von sechs verschiedenen Codeklondetektoren gefundene Klone. Dazu wurden Codeklondetektoren von vor dem Jahr 2002 genutzt und es wurden Klone in acht Softwareprojekten gesucht, wobei vier von ihnen Java-Projekte waren. Es wurden die vereinigten Ergebnisse der verschiedenen Tools genutzt, um den Benchmark zu erstellen. Zwar könnten so gute relative Ergebnisse für Recall- und Precision-Werte entstehen, jedoch gibt es keine Garantie, dass die genutzten Tools insgesamt alle Klone detektiert haben [13]. Auch die manuelle Validierung der Klone ist mit Problemen verbunden. Das manuelle Validieren von Codeklonen sollte möglichst von mehreren Expertinnen und Experten durchgeführt werden, was allerdings ziemlich zeitaufwendig ist. Den Expertinnen und Experten können aber auch vereinzelt Fehler unterlaufen, was dann den Datensatz korrumpiert. Weiterhin sagen Walenstein et al. [163], dass sich auch Expertinnen und Experten bei der Bewertung von Codeklonen nicht immer einig sind. Auch Charpentier et al. [31] fanden Meinungsverschiedenheiten über die jeweiligen Typen mehrerer Klone, wenn diese von verschiedenen unabhängigen Expertinnen und Experten bewertet wurden.

Der **BigCloneBench** [147, 149] von Svajlenko und Roy ist einer der wichtigsten Benchmarks bei der Erstellung von Codeklondetektoren. Er besteht aus 55.499 Sourcecodedateien in der Programmiersprache Java, die aus dem *IJaDataset 2.0*[2] extrahiert wurden, der wiederum aus echten Softwareprojekten besteht. BigCloneBench wurde erstellt, indem Funktionen von Expertinnen und Experten in insgesamt 43 verschiedene Kategorien eingeteilt wurden. Beispiele für Kategorien sind das Sortieren einer Liste oder der Zugriff auf eine Datenbank. Diese Funktionen einer Kategorie bilden dann zueinander Klone, welche allerdings nicht manuell überprüft wurden. Auf diese Weise konnte ein Benchmark mit über 8-Millionen Codeklonen erstellt werden. Die Recall-Messungen mit dem Benchmark sind nur in den 43 verschiedenen Kategorien möglich, die Messung des Precision-Wertes ist hingegen mit dem

[2]https://github.com/clonebench/BigCloneBench

Benchmark gar nicht möglich. Das liegt daran, dass nicht alle in den Dateien des Benchmarks enthaltenen Klone bekannt sind, sondern nur solche aus den 43 verschiedenen Kategorien. Mithilfe des Tools **BigCloneEval** [150], welches die Auswertungen auf dem BigCloneBench automatisiert, ist ein einfaches und bequemes Arbeiten mit BigCloneBench möglich. Auch aufgrund dieses Tools ist BigCloneBench einer der wichtigsten Klonerkennungsbenchmarks für die Programmiersprache Java, auch wenn leider nicht der Bytecode zur Verfügung steht. Denn dieser Fakt macht es Bytecode-basierten Klonerkennungstools unmöglich, mit dem BigCloneBench evaluiert zu werden.

Al-Omari et al. haben den **SemanticCloneBench** [3] veröffentlicht. Sie versuchen das Problem der manuellen Validierung durch Expertinnen und Experten zu umgehen, indem sie Antworten von StackOverflow automatisiert gefiltert und validiert haben. Im Anschluss mussten trotzdem einige manuelle Validierungen stattfinden, bei denen Fehler nicht auszuschließen sind. Mit dieser Herangehensweise konnte ein Benchmark mit über 4.000 Klonpaaren für die Programmiersprachen Java, C, C# und Python erstellt werden. Aufgrund der geringen Codeklonanzahl und der möglichen Validierungsfehler konnte sich der Benchmark bisher noch nicht durchsetzen.

Svajlenko und Roy veröffentlichten das **Mutation and Injection Framework** [151] zur Validierung von Codeklonen. Bei diesem Framework werden durch spezielle Vorgaben Ausgangsprogrammfragmente durch Hinzufügen, Weglassen oder Manipulieren von Anweisungen in Codeklone umgewandelt. Diese Codeklone sind aber nicht aus echten Projekten extrahiert, sondern künstlich erstellt worden. Programmfragmente, die aus einer ganzen Funktion oder nur aus einzelnen Blöcken bestehen, können zur Evaluierung genutzt werden. Von diesem Benchmark werden bisher die Programmiersprachen Java, C und C# unterstützt.

Generell gilt für diese hier vorliegende Arbeit folgendes. Ein von einem Klondetektor gemeldeter Kandidatenklon muss mindestens 70% eines Referenzklons abdecken, damit der Referenzklon als entdeckt gilt. Diese Vorgehensweise ist bereits aus dem BigCloneEval [150] bekannt.

Kapitel 3

Stand der Forschung

Viele verschiedene Techniken wurden zur Codeklonerkennung vorgestellt. Von rein textuellen Methoden bis hin zur emulierten Ausführung gibt es sehr unterschiedliche Herangehensweisen, um Codeklone zu detektieren. In den folgenden Abschnitten werden einige der bedeutendsten Arbeiten in dem Bereich der Codeklonerkennung vorgestellt. Für eine bessere Übersicht wurden die verschiedenen Verfahren in eine der sechs Kategorien eingeteilt. *Textuelle Verfahren* im Abschnitt 3.1 nutzen zur Klondetektion ausschließlich den Quelltext als Grundlage. Es werden fast immer ähnliche Zeichensequenzen gesucht, wobei diese dann als Klone identifiziert werden. Der Vorteil von textuellen Verfahren liegt darin, dass diese meist sehr effizient sind und mit wenig Aufwand auf jede beliebige Programmiersprache angewendet werden können. *Lexikalische Verfahren* im Abschnitt 3.2 wandeln den Quelltext oder den Binärcode in eine Folge von *Tokens* um. Ein Token hat einen Typ und einen Wert, wobei der Token-Typ beschreibt, zu welcher lexikalischen Kategorie der Token gehört. Der Token-Wert ist hingegen das textuelle Erscheinungsbild des Tokens. Diese Tokenfolgen werden dann auf unterschiedliche Weise miteinander verglichen, um ähnliche Programmfragmente zu identifizieren. Lexikalische Verfahren sind meistens weniger anfällig für kleinere Codeveränderungen. Die Effizienz ist hingegen meist etwas schlechter als bei textuellen Verfahren, da die Tokens erzeugt und Ersetzungsoperationen angewendet werden müssen. *Syntaktische Verfahren* im Abschnitt 3.3 nutzen häufig eine abstrahierte Form des Quell- oder Binärcodes. Häufig bilden *Abstrakte Syntaxbäume* (ASTs), *Programmabhängigkeitsgraphen* (PDGs), *Kontrollflussgraphen* (CFGs) oder andere Graphen- sowie Baumstrukturen die Grundlage zur Detektion von ähnlichem Code. Methoden dieser Kathegorie sind besonders robust gegen Codeänderungen und können daher Typ-1-, Typ-2- und auch

Typ-3-Klone gut erkennen. Da jedoch diese Graphen aus dem Quelltext oder dem Binärcode erzeugt werden müssen, braucht es für jede Programmiersprache einen eigenen Parser. Zudem ist das Parsen und Umwandeln in die Graphstrukturen sowie der Vergleich von Graphen oder Bäumen zeitaufwendig. Im Abschnitt 3.4 werden die *Semantischen Verfahren* vorgestellt, die im Vergleich zu den anderen Verfahren die Codeabschnitte virtuell/emuliert ausführen, um so semantische Informationen zu gewinnen. Das emulierte Ausführen der Programme ist durchaus komplex und zeitaufwendig. Die im Abschnitt 3.5 vorgestellten *Hybriden Verfahren* nutzen eine Mischform aus den bisher vorgestellten Kategorien. Da sich diese nicht klar in eine der bisherigen vorgestellten Kategorien einordnen ließen, wurde eine zusätzliche Kategorie gebildet. Zumeist werden die langsamen aber genaueren syntaktischen Verfahren mit den schnelleren textuellen oder lexikalischen Verfahren verknüpft. Die meisten neueren Arbeiten wurden im Bereich der *Maschinellen Lernverfahren* veröffentlicht, die im Abschnitt 3.6 beschrieben sind. Diese Methoden nutzen größtenteils *Neuronale Netzwerke*, um die Eigenschaften von Klonen zu lernen und auf ungesehene Codebeispiele zu übertragen. Auch wenn diese Verfahren großes Potential bieten, gibt es doch wenige gute Datensätze zum Trainieren der Lernverfahren. Falsch zusammengestellte oder unsaubere Datensätze können zu verfälschten Ergebnissen bei den maschinellen Lernverfahren führen.

3.1 Textuelle Verfahren

Bereits 1993 begann Johnson [66] damit, Codeklone mithilfe von „Fingerabdrücken“ zu identifizieren und später auch zu visualisieren [67]. Roy und Cordy entwickelten *NiCad* [36, 126], eines der bekanntesten Tools zur textuellen Codeklonerkennung für verschiedene Programmiersprachen. *NiCad* benutzt dabei drei interne Verfahren, um die Detektion von potentiellen Klonen zu verbessern. Erstens nutzt es unter Verwendung von einer *Agilen Syntaktischen Analyse* benutzerspezifisches flexibles *Pretty-Printing*, um die Formatierung zu vereinheitlichen, Rauschen zu entfernen, und Programmanweisungen in Teile zu zerlegen, so dass potentielle Änderungen als einfache zeilenweise Textunterschiede erkannt werden können. Zweitens bietet es eine flexible und effiziente Extraktion potentieller Klone mit verschiedenen Granularitäten. Drittens bietet es unter Verwendung von Transformationsregeln eine flexible Code-Normalisierung, um lokale Editierunterschiede zwischen ähnlichen Codesegmenten und das Herausfiltern uninteressanter Teile potentieller Klone zu ermöglichen. Chen et al. [32] nutzten *NiCad* beispielsweise dazu, Malware in Android-Anwendungen zu

identifizieren. Weiterhin gibt es einige Tools wie *Clone Swarm* [16] oder *VisCad* [9], welche die gefundenen Klone von *NiCad* visualisieren. Sudhamani und Rangarajan [145, 146] entwickelten eine Methode zum Erkennen von ähnlichen Codefragmenten mithilfe einer Metrik basierend auf den im Code enthaltenen Kontrollanweisungen. *Vuddy* [80] ist ein Tool von Kim et al. und erkennt Klone mithilfe von Fingerabdrücken, die nach einer Abstraktion und Normalisierung von Methoden gewonnen werden. *Vuddy* ist auf eine hohe Skalierbarkeit sowie auf das Erkennen von Schwachstellen und Sicherheitslücken spezialisiert. Yu et al. [179] entwickelten eine Codeklonerkennungsmethode, die den Java-Bytecode anstatt des Quellcodes als Grundlage zur Codeklonerkennung nutzt. Bei diesem Verfahren werden die Bytecode-Dateien in Textdateien umgewandelt, diese dienen anschließend als Grundlage für die textuelle Klonerkennung. Tukaram und Maheswari [154] entwickelten ein Klonerkennungstool, welches auf textueller Grundlage Codeklone identifiziert und auch grafisch aufbereitet anzeigen kann. Reddivari und Khan stellten das Tool *CloneTM* [120] vor, welches mithilfe eines generativen Wahrscheinlichkeitsmodells Klone identifiziert und auch visuell darstellen kann.

3.2 Lexikalische Verfahren

Baker hat bereits im Jahr 1992 das Programm *dup* [12] vorgestellt und somit eine der ersten Arbeiten mit Token-basierten Ansätzen zur Codeklonerkennung geliefert. Im Jahr 2002 veröffentlichten Kamiya et al. das mehrere Programmiersprachen unterstützende Klonerkennungstool *CCFinder* [69]. Dieses transformiert den Quellcode in eine Folge von Tokens und wendet darauf zusätzlich Optimierungen an, um schließlich mithilfe von entwickelten Metriken Codeklone zu identifizieren. Li et al. stellten das Tool *CP-Miner* [100] vor, das mithilfe der Token-basierten Klonerkennung Bugs in großen Softwarepaketen identifizieren kann. Göde und Koschke entwickelten den inkrementellen Klondetector *iClones* [54], der Klone auf Basis vorheriger Suchvorgänge identifiziert. iClones erzeugt eine Abbildung von Klonen einer Version des Quelltextes zu Klonen in einer weiterentwickelten Version, dadurch wird erhebliche Zeit bei der wiederkehrenden Klonsuche gespart. Als Grundlage zur Klonsuche wird innerhalb von iClones der generalisierte *Suffixbaum* genutzt. Ein *Suffixbaum* ist eine Datenstruktur, die alle Endungen von einer Zeichenkette speichert und viele nützliche Eigenschaften bei der Textverarbeitung besitzt. Rehman et al. entwickelten das Tool *LSC Miner* [121], welches die erzeugten Tokens des Quellcodes in zweidimensionalen Arrays speichert, um so die Laufzeit einer Klonsuche

zu beschleunigen. Das Tool *SourcererCC* [131] wurde von Sajnani et al. vorgestellt und nutzt einen optimierten invertierten Index, um schnell potentielle Klone eines bestimmten Codeblocks zu identifizieren. Es werden auf der Token-Reihenfolge basierende Filterheuristiken verwendet, um die Anzahl der Codeblockvergleiche sowie die Anzahl der erforderlichen Tokenvergleiche zum Identifizieren eines Klones zu reduzieren. Aufgrund der sehr guten Skalierbarkeit von *SourcererCC* zum Zeitpunkt der Veröffentlichung des Tools, nutzten Lopes et al. [103] bei ihrer Studie zu Klonen auf GitHub auch *SourcererCC*. *IDCCD* [108] ist ein Tool von Misu und Sakib, welches nach der Erzeugung der Tokens von einer Funktion die Schnittstelleninformation herleitet. Anschließend werden ähnliche Schnittstellen, also Methoden mit ähnlicher Signatur und Rückgabeparametern gesucht, um so Klone zu identifizieren. Wang et al. stellten das Tool *CCAligner* [166] vor, welches Editierdistanzen mithilfe von Codefenstern berechnet. Es wurde ein neuartiger *e-mismatch-Index* entwickelt, der asymmetrische Ähnlichkeitskoeffizienten als Ähnlichkeitsmaß verwendet. Liu et al. [102] stellten ein Tool vor, bei dem sie das Problem der Erkennung von Codeklonen in eine Frage des biologischen Sequenzabgleichs umwandelten. Sie schlagen einen neuen Ansatz vor, der den Code-Fingerabdruck mit dem Sequenzabgleich kombiniert. Der Sequenzabgleich basiert auf dem *Smith-Waterman-Algorithmus* [142], mit dem jedoch erhebliche Verbesserungen durch die dynamische Parametererfassungsstrategie erzielt werden konnten. Ragkhitwetsagul und Krinke entwickelten die Klonsuch-Maschine *Siamese* [118], welche mehrere Programmrepräsentationen, sowie die Reduktion von Abfragen und eine angepasste Ranking-Funktion zur Verbesserung der Codeklonerkennung verwendet. *LVMapper* [173] ist ein Tool von Wu et al., welches, ähnlich zu dem Tool von Liu et al. [102], das Problem der Codeklonerkennung als ein Sequenzvergleich wie in der Bioinformatik, auf die DNA-Analyse bezogen, angeht. Dabei nutzen sie aber dynamische Schwellenwerte, vermeiden zeitaufwendiges Indizieren von Seeds, um so effizienter Klone zu identifizieren und bessere Ergebnisse zu erzielen. Das Tool von Ullah et al. [156] ist darauf spezialisiert, Plagiate bei der Abgabe von Übungsaufgaben von Studentinnen/Studenten oder Schülerinnen/Schülern zu erkennen. Dabei nutzt es die *Latente Semantische Analyse* [178] aus dem Bereich der Textanalyse und -kategorisierung.

3.3 Syntaktische Verfahren

1998 stellten Baxter et al. [19] ein Ansatz zur Codeklonerkennung vor, bei dem der *Abstrakte Syntaxbaum* (AST) als Vergleichsgrundlage dient. Da es jedoch relativ

aufwendig ist, komplette Bäume miteinander zu vergleichen, nutzten Baxter et al. ein Hashverfahren, das von verschiedenen Teilbäumen Hashwerte berechnet. Somit konnten diese Hashwerte miteinander verglichen werden und erst bei einem gleichen Hashwert wurden die Bäume näher untersucht. Somit konnte die Komplexität reduziert werden. Komondoor und Horwitz [84] benutzen *Programmabhängigkeitsgraphen* (PDG) und *Program-Slicing*, um auch nicht zusammenhängende Codeteile als Klone zu identifizieren. Der PDG wurde auch von Krinke [87] genutzt, um Klone zu identifizieren. Durch den Graphvergleich ist bei diesem Ansatz die Komplexität hoch. Das Tool *JDiff* [7] von Apiwattanapong et al. ist speziell darauf ausgelegt, Klone in objektorientierten Programmiersprachen zu finden. Dazu nutzt das Tool einen *Kontrollflussgraph* (CFG), um das objektorientierte Verhalten eines Programms zu modellieren und damit Unterschiede und Gemeinsamkeiten zwischen zwei Programmen zu identifizieren. Liu et al. entwickelten das Plagiatserkennungstool GP_{LAG} [101], welches den PDG zum Aufspüren der Plagiate nutzt. Um die Skalierbarkeit zu gewährleisten, nutzten die Autoren einen statistisch verlustbehafteten Filter, um den Plagiatssuchraum zu bereinigen und somit zu verkleinern. Sager et al. [129] entwickelten eine Methode zum Identifizieren von ähnlichen objektorientierten Klassen. Dabei nutzten die Autoren die ASTs der Funktionen, die auf verschiedene Arten miteinander verglichen werden. *cpdetector* [86] ist ein Tool von Koschke et al. und nutzt aus dem AST generierte *Suffixbäume* zum Detektieren von Klonen. Verglichen wurde *cpdetector* mit vielen Token- und AST-basierten Tools. Das Tool Deckard [63] von Jiang et al. identifiziert ähnliche Teilbäume des AST, indem es für jeden Teilbaum einen Vektor erstellt und anschließend lokales sensitives Hashing [34] nutzt. Ähnliche Hashwerte von Teilbäumen werden dann als Klone identifiziert. Der Ansatz von Gabel et al. [47] kann als eine Erweiterung von Deckard gesehen werden, bei der zusätzlich zu dem AST noch semantische Informationen aus dem PDG extrahiert und mit in die Vektoren aufgenommen werden. Das lokale sensitive Hashing und die Klondetektion bleiben annähernd unverändert im Vergleich. Das Tool *Scorpio* [56] von Higo und Kusumoto nutzt auch PDGs zur Erkennung von Codeklonen. Anstatt der intraprozeduralen PDGs werden hierbei aber die interprozeduralen PDGs genutzt. Weiterhin wird *Program-Slicing* zur Verbesserung der Erkennungsrate verwendet. Die Methode und die prototypische Implementierung des Verfahrens von Higo et al. [58] ist die erste auf dem PDG basierende inkrementelle Codeklonsuche. Dabei werden die Kanten der PDGs in einer Datenbank gespeichert, um bei einer weiteren Codeklonsuche auf einer weiterentwickelten Version der Codebasis wieder genutzt werden zu können. So wurde der zeitliche Aufwand für wiederholte Klon-

detektion reduziert. Koschke [85] hat ein Klonerkennungsverfahren zum Aufspüren von Lizenzverstößen beim Quellcode vorgestellt. Dieses basiert auf dem *Suffixbaum* und nutzt Filterverfahren basierend auf Metriken und maschinellen Lernverfahren, um den Precision-Wert zu verbessern. Weiterhin wurde bei der Entwicklung Wert auf eine geringe Zeitkomplexität gelegt. Das Tool *Codeblast* [23] von Bhattacharjee und Jamil nutzt ein neuartiges *Graph-Matching Verfahren*, um Codeklone mit einem ähnlichen PDG zu identifizieren. Dazu identifiziert das Verfahren gewisse Datenflussmuster innerhalb der PDGs und lässt diese bei der Funktionspaarüberprüfung mit einfließen. Avetisyan et al. [10] nutzen in ihrem vorgestellten Ansatz Subgraphen des PDGs zur Codeklondetektion in Bitcode. Sargsyan et al. [132] nutzen ebenso PDGs, um Klone im kompilierten Code zu identifizieren. Dazu werden zunächst Subgraphen des PDGs erzeugt. Sollen zwei PDGs miteinander verglichen werden, schaut ein erster Algorithmus, ob ein Graphvergleich sinnvoll ist oder ob die Graphen eine zu große unterschiedliche Länge besitzen. Wenn die Größen der Graphen ähnlich sind, kommt es zu einem Graphvergleich, um Klone zu identifizieren. Kim et al. [78] stellten einen Algorithmus zur Messung der Ähnlichkeit zwischen zwei Codeabschnitten durch Lösung eines Subgraphen-Isomorphismus-Problems auf Abhängigkeitsgraphen vor. Die Autoren schlagen eine neue Kostenfunktion für dieses Problem vor, die die Eigenschaften des Quellcodes widerspiegelt. Zur Lösung des Problems wird ein inkrementeller Algorithmus verwendet. Wang et al. führten das Tool *CCSharp* [165] zur Codeklonerkennung auf der Basis des PDGs ein. Um das Problem der zeitlichen Komplexität beim Vergleich von PDGs zu lösen, änderten sie die PDG-Struktur. In der PDG-Strukturmodifikation schlagen Wang et al. eine neuartige Technik vor, um Knoten für Prozeduraufrufe zu verschmelzen, wodurch das Tool den Einfluss von Prozedurparametern beseitigen und die PDG-Struktur verschleiern sowie verkleinern kann. *GroupDroid* [104] von Marastoni et al. kodiert den Kontrollflussgraphen einer jeden Methode in einem Vektor und sucht mit einem Abstandsmaß nach ähnlichen Methoden, um so Schadsoftware in Android-Anwendungen auf Grundlage bereits bekannter Malware-Methoden zu identifizieren.

3.4 Semantische Verfahren

In der Arbeit von Jiang und Su [64] wurde versucht semantisch ähnliche Klone zu finden, indem Funktionen mit zufälligen Eingaben getestet wurden. Anhand der entsprechenden Ausgaben und den dazugehörigen Eingaben konnten so ähnliche Klone identifiziert werden. Das Tool *MeCC* [77] von Kim et al. identifiziert Klo-

ne anhand ihrer abstrakten Speicherbelegungen und ist somit unabhängig von der Syntax. *SPAPE* [24] ist ein Werkzeug von Bian et al., das ebenfalls nicht die Syntax zur Codeklonerkennung nutzt, sondern strukturelle semantische Informationen, die mithilfe von Transformationen generiert werden. In der Arbeit von Tekchandani et al. [153] werden aus den *Syntaxbäumen* die formalen Grammatiken abgeleitet, auf deren Grundlage dann ähnliche Funktionen gesucht werden. Kamiya et al. [70] detektieren Klone mithilfe einer Ausführungssemantik. Dabei sind zwei Codefragmente gleich, wenn ihre Programmsequenzen die gleichen Funktionsaufrufe in der gleichen Reihenfolge tätigen. In dieser Arbeit wurde außerdem eine *Dominatoranalyse* genutzt. Um Schwachstellen in Programmen zu finden, für die kein Quellcode zur Verfügung steht, haben Pewny et al. [116] ein Klonerkennungsverfahren entwickelt, welches ähnliche Bugs zwischen verschiedene CPU-Architekturen in kompilierten Programmen entdeckt. Dazu wird der Binärcode zunächst in eine Zwischencoderepräsentation gebracht, anschließend wird das Ein- und Ausgabeverhalten des Codes untersucht. Codeklone werden dann entdeckt, indem andere Programme mit ähnlichem Ein- und Ausgabeverhalten gefunden werden. Das Tool *DyCLINK* [144] von Su et al. führt die Programme beispielhaft aus, um so wichtige Sequenzen im Instruktionslevel zu finden. So werden *Abhängigkeitsgraphen* im Instruktionslevel aufgebaut, zwischen denen dann ähnliche Subgraphen und somit Klone gesucht werden. Hu et al. entwickelten das Tool *CACompare* [59], welches Argumente und indirekte Sprünge von binären Funktionen erkennt und so eine Ausführung emuliert. Auf diese Art werden semantische Signaturen zur Ähnlichkeitsmessung extrahiert. Die Codeklon-Suchmaschine *Clone-Seeker* [55] von Hammad et al. sucht nach Klonen mithilfe von Metadaten in Form eines Textdokuments, in dem Metainformationen, wie beispielsweise genutzte Variablennamen, stehen. Auf diese Weise kann eine Entwicklerin beziehungsweise ein Entwickler nach Codeklonen mithilfe einer Abfrage in natürlicher Sprache oder aber mit einer Abfrage in einer Programmiersprache suchen.

3.5 Hybride Verfahren

Das Tool *BOREAS* [2] von Agrawal und Yadav nutzt zum Detektieren von Klonen sowohl textuelle als auch tokenbasierte Techniken. Mithilfe der Tokens lassen sich äußerst schnell Klone vom Typ-1 oder Typ-2 finden, anschließend kommt ein textueller Vergleich für Typ-3-Klone zum Einsatz. Durch diese hybride Herangehensweise konnten die Autoren ein effizientes Tool implementieren. In der Arbeit von Rahman

und Roy [119] wurde untersucht, ob Code mit vielen Codeklonen oder mit weniger Codeklonen mehr Bugs enthält. Weiterhin wurde überprüft, in welchem der beiden Quellcodes der Wartungsaufwand höher ist. Dazu nutzen sie das Tool *ChangeDistiller* [46], das ASTs nutzt, und das Tool *NiCad* [126], welches ein textuelles Verfahren ist. Higo und Kusumoto [57] stellten in ihrer Arbeit ein Vergleich zwischen mehreren Verfahren zur Codeklonerkennung vor. Dabei benutzten sie strukturelle Ähnlichkeit, Ähnlichkeit der genutzten Variablennamen beziehungsweise Schlüsselwörter und die Ähnlichkeit von Methodennamen. *CloneManager* [81] von Kodhai und Kanmani nutzt sowohl einfache textuelle Analyse als auch angepasste Metriken zur Codeklonerkennung. Das von Wang et al. [167] entwickelte Verfahren nutzt zur Codeklonerkennung sowohl einen erweiterten *Systemabhängigkeitsgraphen* als auch eine Metrik-basierte Herangehensweise. Ein Metrik-basierter Ansatz im Zusammenspiel mit PDGs ist der Ansatz von Roopam und Singh [124]. Nasiloo und Azimzadeh [113] nutzen abstrakte Speicherzustände der Funktionen zusammen mit deren PDGs, um schneller Klone mit einem besseren Recall-Wert zu detektieren. Yang et al. [177] erzeugen von Funktionen zunächst den AST. Anschließend durchlaufen sie den AST mit einer Tiefensuche und erhalten so einen Vektor, der die Funktion repräsentiert. Solche Vektoren werden dann mit dem *Smith-Waterman-Algorithmus* [142] verglichen, um Codeklone zu identifizieren. Das Tool *BinMatch* [60] von Hu et al. identifiziert Codeklone im Binärcode mithilfe einer emulierten Ausführung von Funktionen. Dabei werden Laufzeitinformationen von den Funktionen gesammelt, die dann unter Verwendung einer Metrik verglichen werden. Tekchandani et al. [152] stellen eine Methode zur Codeklonerkennung vor, die Kontroll- und Datenflussanalysen auf den ASTs vornimmt. Dabei wurden *Dead-Code-Elimination* und *Reaching-Definition* Optimierungen vorgenommen. Besonders beim *Internet-of-Things* konnten so in einer ersten Evaluation einige Codeklone detektiert werden. Bei der vorgestellten Arbeit von Yang et al. [176] werden Funktionen zunächst vorverarbeitet. Anschließend wird ein Filter auf die extrahierten Funktionen angewendet und diese werden dann mit der *Longest-Common-Subsequenz* Algorithmus [126] sowie dem *Suffix Array* Algorithmus [18] verglichen.

3.6 Maschinelle Lernverfahren

In den letzten Jahren wurde damit begonnen, maschinelle Lernverfahren und *Deep Learning* Verfahren zum Auffinden von Codeklonen zu benutzen. In der Arbeit von Gascon et al. [49] wurden Kodierungen (*Embeddings*) als Vektoren von *Funkti-*

onsaufrufgraphen erzeugt und zum Klassifizieren in einer *Support-Vector-Maschine* ausgewertet, um so Schadsoftware in Android-Anwendungen zu identifizieren. Jadon [62] benutzt *Support-Vector-Maschinen* [38] zum Auffinden von ähnlichen Programmteilen in Quelltexten der Programmiersprache C. White et al. [171] benutzen einen *Autoencoder* [17], um automatisch unüberwacht lexikalische Informationen mit syntaktischen Mustern zu verbinden, um so Codeklone zu identifizieren. Das *CDLH* Tool [170] wurde von Wei und Li entwickelt und nutzt eine Fehlerfunktion basierend auf Hashwerten, um die Ähnlichkeit zweier Codefragmente zu analysieren. In dieser Arbeit wurde zudem erstmals eine eingeschränkte Version des *BigCloneBench* vorgestellt, mit der in der Folge viele *Machine Learning Tools* evaluiert wurden.[1] *CCLearner* [97] von Li et al. extrahiert Tokens aus Codeklonen und nicht Codeklonen, die aus Methoden bestehen, um einen Klassifizierer auf Grundlage von *Tiefen Neuronalen Netzen* (DNN) zu trainieren. Anschließend kann dieser Klassifizierer Klone identifizieren. Wei und Li veröffentlichten aufbauend auf dem *CDLH* Tool das *CDPU* Tool [169]. Dabei definierten sie die Klonerkennungsaufgabe als Positive-Unlabeled Lernproblem. Um dieses zu lösen, wurde neben dem klassischen maschinellen Lernansatz auch ein *Adversarial Training* verwendet. Dabei werden bewusst Störungen in die bereits ermittelten Features während des Trainings integriert, um die Robustheit des gelernten Modells gegenüber extrem ähnlich aussehenden Nicht-Klonpaaren zu verbessern, die sich aber unterschiedlich verhalten. Saini et al. beschreiben ein *Siamesisches Netz* [15], das aus zwei neuronalen Netzen besteht, die zusammenfließen und gemeinsam eine Ausgabe produzieren. Von Menschen definierte Softwaremetriken werden als Input für *Oreo* [130] verwendet. Der Klondetektor wurde mit vielen Java-Projekten von GitHub trainiert. Zhao und Huang [187] stellten das Deep-Learning-basierte Tool *DeepSim* vor, das semantische Metriken mit manuell extrahierten semantischen Merkmalen von CFGs als Input für ein neuronales Netz verwendet. Leider sind die extrem guten Ergebnisse nicht mit dem veröffentlichten Code reproduzierbar. Sheneamer hat das Tool *CCDLC* [139] vorgestellt, welches Klone im Java-Bytecode mithilfe von Informationen aus dem AST, dem PDG und dem *Bytecode Dependency Graphs* sowie mithilfe eines *Convolutional-Neural-Network* (CNN) [93] detektiert. Zeng et al. [185] analysieren die ASTs der Programmfragmente mittels eines gewichteten rekursiven *Autoencoders*. So extrahieren sie die Programmmerkmale und kodieren die Funktionen in Vektoren. Während des Modellierungsprozesses berücksichtigen die Auto-

[1] Inwieweit dieser Benchmark wirklich dazu geeignet ist, um *Machine Learning Tools* für die Codeklonerkennung zu evaluieren, wird näher in Kapitel 7 besprochen.

ren Knotengewichtungsinformationen in ASTs, um den Informationsanteil wichtiger Knoten in der endgültigen Vektordarstellung eines Programms zu erhöhen. Funktionen mit ähnlichen Vektoren werden anschließend als Codeklone identifiziert. Büch und Andrzejak [29] benutzen ein *Siamesisches Netzwerk* bestehend aus *Rekursiven Neuronalen Netzwerken* (RNN) [96], welches sie mit Informationen aus dem AST trainieren. Eine wichtige Aussage dieser Veröffentlichung ist, dass die Trainings-, Evaluations- und Testdaten klar voneinander getrennt sein müssen, damit die Generalisierbarkeit des neuronalen Netzes getestet werden kann.[2] Chen et al. [33] nutzen in ihrer Methodik ASTs mit *Application-Programming-Interfaces* (API) erweiterten Informationen zur Erkennung semantischer Klone. Zu den Anwendungen des *Siamesischen Netzwerks* können auch die semantische Codesuche und die Code-Zusammenfassung gezählt werden. Perez und Chiba [115] haben mithilfe von ASTs und einem *Long-Short-Term-Memory* (LSTM) Netzwerk [141] ein Tool gebaut, welches Codeklone zwischen verschiedenen Programmiersprachen identifizieren kann. Weiterhin haben die Autoren einen Datensatz, bestehend aus Klonen in den Programmiersprachen Java und Python erstellt. Das Tool *TBCCD* [181] wurde von Yu et al. entwickelt und nutzt sowohl strukturelle Eigenschaften des AST sowie lexikalische Informationen von Tokens, um diese mit einem Baum-basierten LSTM zu kodieren. Weiterhin wurde in der Arbeit von Yu et al. eine positionsbeachtende Zeichenkodierung (PACE) entwickelt, mit der sich deutlich bessere Ergebnisse erzielen lassen. *ASTNN* [186] von Zhang et al. kodiert AST-Teilbäume von Anweisungen mithilfe eines RNN. Diese Kodierungen werden dann als Eingabe für ein RNN verwendet, um eine Repräsentation für ein Programm zu lernen. In der Arbeit von Li et al. [98] wurden erstmals *Graph-Matching-Networks* (GMN) vorgestellt. Diese haben zusätzlich zu früheren *Siamesischen Netzwerken* einen *Cross Graph Attention Mechanismus*. Das bedeutet, dass bei der Kodierung eines Graphen auch der zweite simultan eingegebene Graph Einfluss auf die Kodierung nimmt. Um die neuartige Netzwerkarchitektur zu evaluieren, wurden von Li et al. ähnliche Kontrollflussgraphen gesucht. Wie genau diese GMNs funktionieren, kann auch im Abschnitt 7 nachgelesen werden. Das Tool *TECCD* [48] wurde von Gao et al. entwickelt und erzeugt zunächst einen Vektor für jeden Knoten des ASTs mit Unterstützung eines *Skip-Gram Modells*, bevor es alle Vektoren zu einem Vektor zusammenführt, der dann den gesamten Baum repräsentiert. Mithilfe der *Euklidischen Distanz* werden so letztendlich Klone detektiert. *CLCDSA* [112] ist ein Tool zum Erkennen von Klo-

[2] Im Kapitel 7 geht es auch genau um die Generalisierbarkeit von maschinellen Lernverfahren zur Codeklonerkennung mit dem CDLH Datensatz.

nen zwischen verschiedenen Programmiersprachen. Es nutzt einerseits syntaktische Merkmale des Quellcodes und andererseits API-Aufrufe, um Klone mittels eines *Deep Learning Netzwerkes* zu identifizieren. Das Tool *RSharer+* [51] wurde von Guo et al. entwickelt, um ähnliche Paare aus gut und schlecht reviewten aber ähnlichen Projekten zu suchen, um so die schlechter reviewten Projekte mit den besser reviewten zu verbessern. Dazu wird ein CNN und ein *Autoencoder* Verfahren verwendet. Um die Codedetektion zu testen, wurde der *BigCloneBench* genutzt. Da die Detektion von semantischen Klonen äußerst schwierig ist, haben Yuan et al. [182] den CFG als eine Zwischenrepräsentation des Quellcodes genutzt. Die Autoren kombinierten das *Classical Dynamic Time Warping* [22] mit den zwei tiefen neuronalen Netzwerktypen, dem bidirektionalen *RNN Autoencoder* und einem *Graph Convolutional Netzwerk* [25], zur Detektion semantischer Klone. Wang et al. [168] nutzen die bereits weiter oben beschriebenen GMNs zur Detektion von Codeklonen. Als Eingabe nutzen die Autoren ASTs mit zusätzlich eingefügten Kontroll- und Datenflusskanten. Dieser *FA-AST+GMN* genannte Ansatz findet weitere Erwähnung im Abschnitt 7. Fang et al. [44] betrachten in ihrem Ansatz zur Codeklonerkennung die Beziehungen zwischen aufrufender und aufgerufener Funktion mithilfe der Kontrollflussgraphen bei Programmen in der Programmiersprache C++. Das Tool *SCDetector* [174] von Wu et al. behandelt den CFG als eine Art soziales Netzwerk und nutzt spezielle *Social-Network* Analysen, um den zentralen Aspekt eines jeden Basisblocks zu bestimmen. Anschließend wird dieser spezielle CFG als Eingabe in ein *Siamesisches Netzwerk* genutzt, um einen Klondetektor zu trainieren. Meng und Liu haben mit *At-biLSTM* [106] ein Modell vorgestellt, welches auf einem bidirektionalen LSTM mit *Self-Attention* basiert. Zunächst wird der AST in Anweisungsbäume aufgesplittet, die dann mithilfe eines Tiefensuche-Algorithmus speziell kodiert werden. Anschließend wird das neuronale Netz zur Kodierung aller Anweisungssequenzen genutzt und erzeugt als Ausgabe einen Vektor. *CodeBERT* [45] von Feng et al. ist ein mächtiges *Transformer Modell* [160] für natürliche Sprachen und Programmiersprachen, welches vortrainiert zur Verfügung steht. Durch einen als *Finetuning* bezeichneten Vorgang kann das neuronale Netz dann auf Aufgaben spezialisiert werden, wie beispielsweise die Codeklonerkennung. Darauf aufbauend wurde *GraphCodeBERT* [52] von Guo et al. vorgestellt, welches ganz ähnlich zu *CodeBERT* funktioniert, allerdings explizit Graphen als Eingabeform nutzt. Auch hier ist die Codeklonerkennung eine von vielen möglichen Spezialaufgaben, die das Netz übernehmen kann. Das Tool *FCAA* [61] von Hua et al. nutzt viele Codeeigenschaften zum Trainieren eines neuronalen Netzes mit einem *Attention-Mechanismus*. Verar-

beitet werden Informationen von aufeinanderfolgenden Token, dem AST und des CFG. Sheneamer [140] nutzt in seiner Arbeit nach der Vorverarbeitung des Quelltextes viele Ähnlichkeitswerte zwischen zwei Funktionen als Eingabe in ein neuronales Netz und erzielt dabei sehr gute Ergebnisse auf dem *CDLH* Datensatz. *DEGraph-CS* [184] von Zeng et al. modelliert aus dem Quellcode einen Variablen-basierten Flussgraphen, der als Eingabe in ein *Gated Graph Neural Network* genutzt wird und mit Programmen in der Programmiersprache C evaluiert wurde.

Kapitel 4

Dominatorbasierte Klonerkennung

In diesem Kapitel wird die Klonerkennung mithilfe von Dominatorinformationen detailliert vorgestellt. Dazu wird im ersten Abschnitt der Ansatz im Allgemeinen beschrieben. Im zweiten Abschnitt wird das Finden von ähnlichen Funktionen und Unterfunktionen besprochen. Mit dem hier vorgestellten Verfahren können auch ähnliche Unterblöcke entdeckt werden, dies wird im dritten Abschnitt vorgestellt. Im vierten Abschnitt wird dann der Vergleich zweier Pfade im Detail dargelegt. Abschließend wird im letzten Abschnitt aufgezeigt, wie ganze Pfadmengen miteinander verglichen werden.

4.1 Der grundlegende Ansatz zur Codeklonerkennung

Zur Detektion von Klonen können die verschiedensten Ansätze gewählt werden, wie bereits im Kapitel 3 dargelegt wurde. Bei den syntaktischen Verfahren werden häufig Graphen wie beispielsweise der *Abstrakte Syntaxbaum* oder der *Kontrollflussgraph* als Vergleichsgrundlage von Programmfragmenten genutzt. Die Verwendung paarweiser Graphenvergleichsalgorithmen zum Auffinden ähnlicher Funktionen und somit von Codeklonen wäre ziemlich ineffizient und würde zu teuren Rechenkosten führen [175]. Aus diesem Grund wurde in dieser Arbeit beschlossen, keinen graphbasierten Vergleich durchzuführen. Stattdessen werden Pfadmengen aus einem Dominatorbaum abgeleitet, die dann für den Vergleich verwendet werden. Auch in der Arbeit von Alon et al. [4] werden Pfade genutzt, die allerdings aus einem abstrakten Syntaxbaum und unter der Verwendung eines anderen Ableitungsverfahren abgelei-

tet werden.
Klondetektoren umfassen im Allgemeinen mehrere Schritte zur Code-Vorverarbeitung und -Transformation sowie einen Ähnlichkeitsvergleich auf Grundlage der berechneten Codedarstellung [127]. Der in dieser Arbeit verfolgte Ansatz zur Erkennung von Codeklonen, wie er in dem eigenständigen und voll funktionsfähigen Tool *StoneDetector* für Java-Quellcode, -Stackcode und -Registercode implementiert ist, erwartet Programmdateien als Eingabe, die in mehreren Schritten verarbeitet werden, wie in Abbildung 4.1 dargestellt ist.
In einem ersten Schritt führt StoneDetector eine lexikalische und syntaktische Analyse von Sourcecodedateien durch, die diese in Codefragmente einer bestimmten Granularität zerlegt. Es gilt zu beachten, dass in dieser Arbeit hauptsächlich ganze Funktionen als Codefragmente und damit zur Klonerkennung genutzt werden. Weiterhin werden die Codefragmente in diesem Schritt in Tokens umgewandelt und in abstrakte Syntaxbäume überführt. Im nächsten Schritt werden die abstrakten Syntaxbäume mithilfe einer leichtgewichtigen Kontroll- und Datenflussanalyse in Kontrollflussgraphen und Dominatorbäume umgewandelt. Ist die Eingabe jedoch kein Sourcecode, sondern Bytecode, dann wird dieser direkt in ein internes Format überführt und der abstrakte Dominatorbaum erzeugt.
Die Ähnlichkeit von Codefragmenten wird nicht anhand von Dominatorbäumen berechnet, sondern die Pfade der Dominatorbäume werden im folgenden Schritt in textuelle Darstellungen, sogenannte Beschreibungsmengen, kodiert. Dieses Vorgehen ermöglicht anschließend effizientere Vergleiche. Um die Codeklonerkennung noch effizienter umzusetzen, werden die Beschreibungsmengen durch ein Hashverfahren in einer kompakteren Form, sogenannten Fingerabdrücken des Dominatorbaums, umgewandelt. Der beschriebene Vorverarbeitungs- und Transformationsschritt umfasst auch verschiedene Normalisierungen und Abstraktionen, z. B. die Umbenennung von Bezeichnern und Konstanten, wie sie in anderen Klondetektoren wie beispielsweise NiCad [36,126] durchgeführt werden. Der Abgleich ähnlicher Codefragmente auf der

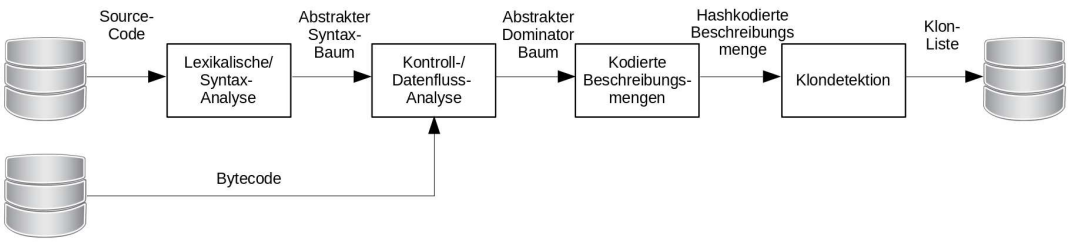
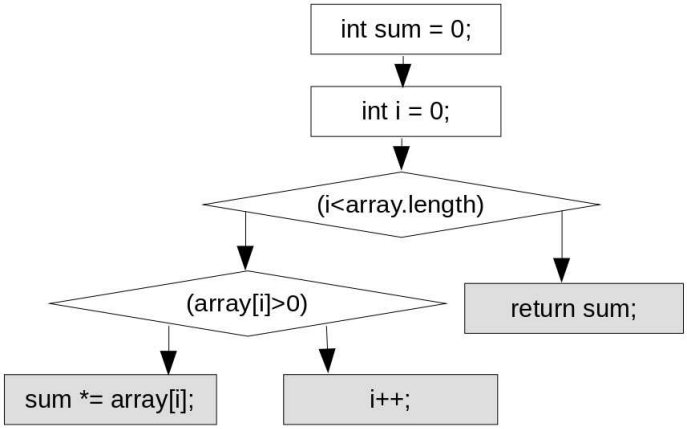


Abbildung 4.1: Allgemeiner Aufbau des Systems zur Codeklonerkennung.

Grundlage der generierten Fingerabdrucksmengen erfolgt im nächsten Schritt. Zu diesem Zweck unterstützt StoneDetector verschiedene Metriken zur Berechnung der Ähnlichkeit von Fingerabdrücken, es sind die Hamming-Distanz, die Levenshtein-Distanz und eine auf der Longest-Common-Subsequenz- basierende Metrik implementiert. Die so identifizierten, ähnlichen Codefragmente werden schließlich zu einer Liste von Codeklonen zusammengefasst.
Die hier vorgeschlagene Methode ist eine textorientierte Klonerkennungstechnik, bei der Codefragmente, d. h. Funktionen, nicht nur durch ihre Syntax, sondern auch durch ihren Kontrollfluss charakterisiert werden. Im Gegensatz zu anderen Techniken werden Funktionen nicht durch abstrakte Syntaxbäume oder Kontrollflussgraphen dargestellt, sondern durch ein spezielleres Format, den Dominatorbaum, welcher im Abschnitt 2.4 eingeführt wurde. Ein Dominatorbaum wird in Übersetzern häufig zur Modellierung der Ausführungsreihenfolge von Anweisungen verwendet. Die Verwendung von Dominatorbäumen hat den Vorteil, dass im Gegensatz zu Kontrollflussgraphen jeder Knoten genau über einen einzigen eindeutigen Pfad vom Startknoten aus erreicht werden kann und es keine Zyklen gibt, was eine effiziente Kodierung von Funktionen ermöglicht.

Die Darstellung von Programmfragmenten durch Dominator- bzw. Teildominatorbäume unterscheidet die vorgestellte Methode von den meisten anderen Techniken zur Erkennung von Klonen, da sie den Kontrollfluss und nicht die alleinige Syntax berücksichtigt. Im Gegensatz zu anderen Techniken, die eine Ähnlichkeitsprüfung auf dem gesamten Programmfragment durchführen, wird die Ähnlichkeitsprüfung mit der entwickelten Technik auf Basis von Pfaden durchgeführt. Durch eine zusätzliche Mittelung der Pfadabstände wird ein sensitives und fehlertoleranteres Abstandsmaß definiert, das lokal vorhandene Unterschiede in den Programmfragmenten verzeiht, sodass bessere Ergebnisse bei der Erkennung von Klonen mit größerer syntaktischer Varianz, aber ähnlicher Semantik erzielt werden können. Insbesondere strukturelle Klone, d. h. Codefragmente, in denen derselbe Kontrollfluss durch unterschiedliche Programmkonstrukte ausgedrückt wird, und Subklone, d. h. Klone, bei denen ein Codefragment den Kontrollfluss des anderen einbettet, können mit der Methode effektiv mit gleichbleibend hoher Präzision erkannt werden.
Als Hauptanwendung der entwickelten Methodik kann das Auffinden von Fehlern in homolog entwickeltem Programmcode angesehen werden. Ein homolog entwickelter Programmcode entsteht durch das Duplizieren und Modifizieren von bereits existierendem Code, weist syntaktische Ähnlichkeiten auf und kann verschiedenen



(a) Dominatorbaum

```
Dset=
[ [int sum = 0;] -> [int i = 0;] -> [(i<array.length)] -> [(array[i]>0)]
-> [sum *= array[i];],
[int sum = 0;] -> [int i = 0;] -> [(i<array.length)] -> [(array[i]>0)]
-> [i++;],
[int sum = 0;] -> [int i = 0;] -> [(i<array.length)] -> [return sum;] ]

Dsplit=
[ [int sum = 0;] -> [int i = 0;],
[(array[i]>0)] ,
[(array[i]>0)] -> [sum *= array[i];],
[(array[i]>0)] -> [i++;],
[(i<array.length)] -> [return sum;] ]
```

(b) Beschreibungsmengen

Abbildung 4.2: Dominatorbaum und die Beschreibungsmengen der Funktion aus Abbildung 2.2.

Mustern folgen: Forking etwa für das risikolose Experimentieren mit vorhandenem Code, Templating zur Realisierung von Code-Varianten, die nicht direkt in der Programmiersprache unterstützt werden oder auch Customization zur Anpassung von bestehendem Code an neue Problemstellungen [73]. Ein Klondetektor wie StoneDetector kann insbesondere für homolog entwickelten Code nützlich sein. Das Kopieren und Anpassen von Code findet üblicherweise auf Funktions- oder sogar auf Blockebene statt, was durch die Erkennung von ähnlichen Teilblöcken in StoneDetector unterstützt wird (vgl. auch Klone mit großen Lücken [102, 166, 173]). Als weitere Anwendung der entwickelten Methode kann die statische, aber auch dynamische Erkennung von Schadsoftware angesehen werden, wobei im letzteren Fall eine Klonüberprüfung erst auf den zur Laufzeit tatsächlich ausführbaren Maschinencode durchgeführt wird.

4.2 Vorgehen beim Klonerkennungsansatz

In diesen Unterkapiteln wird das detaillierte Vorgehen bei dem Klonerkennungsansatz mit Dominatorinformationen erläutert. Dabei wird allerdings nur das Herangehen beschrieben und nicht die jeweilige Implementierung.

4.2.1 Finden ähnlicher Funktionen und Unterfunktionen

Es werden Pfade aus einem Dominatorbaum D abgeleitet, die am Startknoten n_{start} beginnen und in einem Blattknoten in D enden. Die so ermittelten Pfade werden dann als eine Beschreibungsmenge zusammengefasst, die zur eindeutigen Darstellung eines Dominatorbaums verwendet wird. Ein Pfad kann durch eine textuelle Verkettung der (abstrakten) Knotenbelegungen kodiert werden. In dem Dominatorbaum in Abbildung 4.2 (a) kann der Pfad vom Startknoten zum linken Blattknoten beispielsweise wie folgt dargestellt werden:

$$[int\ sum = 0;] \rightarrow [int\ i = 0;] \rightarrow [(i < array.length)] \rightarrow$$
$$[(array[i] > 0)] \rightarrow [sum *= \ array[i];]$$

Im Folgenden sei D ein Dominatorbaum, $paths_D$ die Menge seiner Pfade und $leaves_D$ die Menge seiner Blattknoten. Die Menge der Pfade, die den Dominatorbaum eindeutig beschreiben, ist definiert als:
$D_{set} = \{(n_{start}, \cdots, n_k) \,|\, n_{\text{start}} \cdots n_k \in paths_D, n_k \in leaves_D\}$

Diese Menge D_{set} wird *Beschreibungsmenge* des Dominatorbaums D genannt. Für den Dominatorbaum in Abbildung 4.2 (a) ist die Beschreibungsmenge D_{set} in Abbildung 4.2 (b) dargestellt. Wie zu erkennen ist, sind die drei Pfade des Dominatorbaums, die am Startknoten beginnen und jeweils im linken, mittleren und rechten Blattknoten enden, in der Beschreibungsmenge enthalten.
Zwei Funktionen f und g mit Dominatorbäumen D bzw. D' heißen dann *streng ähnlich*, wenn sie die gleiche Kardinalität ihrer Beschreibungsmengen D_{set} und D'_{set} haben und wenn es für jeden Pfad $p \in D_{set}$ einen ähnlichen Pfad $p' \in D'_{set}$ gibt. Ist nur die letzte Bedingung erfüllt, sodass die Beschreibungsmengen D_{set} und D'_{set} nicht die gleiche Anzahl an Elementen haben, heißen f und g *partiell ähnlich*.
Die Ähnlichkeit von Pfaden in Dominatorbäumen D und D' wird formal durch eine Funktion $\delta : paths_D \times paths_{D'} \rightarrow \mathbb{N}$ definiert, die den Abstand zweier Pfade berechnet. Im Allgemeinen kann die Funktion δ je nach Anwendungsfall frei gewählt

werden. Voraussetzung ist jedoch, dass die Funktion bei identischen Pfaden den Wert Null und bei nicht-identischen Pfaden einen Wert größer Null liefert. Der Wert der Funktion wird als Metrik für die Ähnlichkeit von Pfaden in Dominatorbäumen und später auch zur Berechnung der Ähnlichkeit von Beschreibungsmengen und damit von ganzen Codeklonen verwendet (vgl. Abschnitt 4.2.4).

4.2.2 Entdeckung ähnlicher Unterblöcke

Mit der bisher vorgestellten Methodik kann die Ähnlichkeit von Funktionen f und g analysiert werden. Andererseits kann mit diesem Ansatz aber noch nicht geprüft werden, ob eine Funktion f bestimmte Unterblöcke einer Funktion g enthält, was aber für das Auffinden von Unter- oder Teilklonen unerlässlich ist. Unabhängige Unterblöcke in Funktionen können aufgrund der Art und Weise, wie die Pfade in Beschreibungsmengen definiert wurden, nicht identifiziert werden. Da die Pfade immer am Startknoten beginnen und in einem Blattknoten enden, können ähnliche Unterblöcke einer Funktion, die sich z. B. im Zentrum einer anderen Funktion befinden, nicht erkannt werden. Eine geeignetere Definition und Auswahl von Pfaden für die Darstellung von Dominatorbäumen, die auch die Suche nach Unterblöcken unterstützt, wird durch die Einführung von Splitknoten erreicht. Ein Knoten n in einem Dominatorbaum D heißt *Splitknoten*, wenn n mehr als einen Nachfolgeknoten in D hat. Die Splitknoten eines Dominatorbaums D werden im Folgenden mit der Menge $splits_D$ bezeichnet. Im Dominatorbaum des Beispiels aus Abbildung 4.2 sind die Splitknoten die Bedingungsknoten der Schleife und der if-Verzweigung, die als Rhombus dargestellt sind.
Mit der Einführung von Splitknoten ist die Menge der Pfade zur eindeutigen Darstellung eines Dominatorbaums D definiert als die Menge aller Pfade, die am Startknoten n_{start} oder einem Splitknoten beginnen und an einem Blattknoten bzw. dem Vorgänger eines Splitknotens enden und außer dem initialen Splitknoten keinen weiteren Splitknoten enthalten:

$$D_{split} = \{(n_1, \cdots, n_k) \,|\, n_1 \cdots n_k \in paths_D, (n_1 = n_{start} \vee n_1 \in splits_D), \\ (n_k \in leaves_D \vee n_k \in preds(splits_D), \nexists i > 1 : n_i \in splits_D)\}$$

Die so abgeleitete Beschreibungsmenge D_{split} für den Dominatorbaum unseres Beispiels ist ebenfalls in Abbildung 4.2 (b) dargestellt.
Im Übersetzerbau wird der Begriff Basisblock für eine Folge von Anweisungen verwendet, die nacheinander ausgeführt werden und bei denen der Kontrollfluss immer in der ersten Anweisung eintritt und aus der letzten Anweisung austritt. Im Prinzip

stellt also jeder Pfad in der Beschreibungsmenge mit Splitknoten einen Basisblock des Kontrollflussgraphen dar. Der einzige Unterschied ist, dass im Falle einer Bedingung als letzte Anweisung des Basisblocks die Bedingung zu den Beschreibungsmengen der nachfolgenden Basisblöcke hinzugefügt wird. Folglich kann in der hier vorgestellten Methodik das Auffinden von Unterblöcken auf der Ebene der modifizierten Basisblöcke erfolgen.
Für einen Unterblock b und eine Funktion f mit den entsprechenden Dominatorbäumen D und D' gilt, dass die Funktion f einen Unterblock enthält, der dem Unterblock b *streng ähnlich* ist, wenn es einen ähnlichen Pfad $p' \in D'_{split}$ für jeden Pfad $p \in D_{split}$ gibt. Ist die letztgenannte Bedingung nur für eine Teilmenge von D_{split} erfüllt, so enthält die Funktion f einen Teilblock, der b *teilweise ähnlich* ist.

4.2.3 Vergleich von Pfaden

Eine einfache Möglichkeit, zwei Pfade p und q von Dominatorbäumen zu vergleichen, besteht darin, ihre Knoten in der Reihenfolge ihres Auftretens auf textuelle Gleichheit zu überprüfen. Dieser Vergleich würde jedoch bedeuten, dass nur die Pfade als ähnlich angesehen werden, deren Knoten exakt gleich und somit textuell identisch sind. Bei Klonen des Typs 3 und 4 muss der Vergleich jedoch nicht so streng sein und Pfade auch dann als ähnlich bewertet werden, wenn sich im Pfad enthaltene Knoten unterscheiden. Darüber hinaus muss der Vergleich in der Lage sein, Unterpfade und Pfade, die Teil anderer Pfade sind, zu identifizieren. Metriken, die auf Zeichenketten definiert sind, eignen sich besonders gut für den Vergleich von Pfaden und sind daher hervorragende Kandidaten für eine Vergleichsfunktion δ. Um diese Techniken zu verwenden, können Knoten als Zeichen dargestellt werden und zwei Knoten werden dann als das gleiche Zeichen betrachtet, wenn sie für die gleiche Anweisung stehen. Ein einzelner Pfad wird auf diese Weise als Zeichenkette kodiert, sodass Zeichenketten-Metriken ohne weiteres für den Vergleich verwendet werden können. Eine grundlegende String-Metrik ist die *Hamming-Distanz* [155], die für Zeichensequenzen mit fester Länge definiert ist und als Ergebnis die Anzahl der unterschiedlichen Zeichen liefert. Eine Einschränkung der Hamming-Distanz-Metrik besteht darin, dass die Zeichen nur bis zur Länge der kürzeren Zeichenfolge verglichen werden. Die Anzahl der verbleibenden Zeichen der längeren Zeichenkette wird dann zu dem bisher berechneten Ergebnis addiert, was in unserem Fall jedoch zu einem unzulässigen Vergleich für unterschiedlich lange Pfade führt. Eine weitere grundlegende String-Metrik ist die *Levenshtein-Distanz* [95], die auch in der Bioinformatik

zum Vergleich von DNA-Sequenzen verwendet wird. Der Levenshtein-Abstand zwischen zwei Zeichenketten ist die minimal erforderliche Anzahl an Einfüge-, Lösch- und Ersetzungsoperationen, die erforderlich sind, um eine Zeichenkette in die andere umzuwandeln. Der Levenshtein-Abstand kann als eine Alternative der Hamming-Distanz angesehen werden, die in diesem Fall jedoch auf Ersetzungsoperationen und Zeichenketten gleicher Länge beschränkt ist.

Zur Überprüfung, ob zwei Pfade p und q ähnlich sind, wurde bereits im Abschnitt 4.2.1 ein Abstandsmaß $\delta(p, q) \rightarrow \mathbb{N}$ eingeführt. Ein Aufruf von $\delta(p, q)$ gibt also an, wie unterschiedlich die Pfade p und q sind. Die Funktion δ ist so gewählt, dass bei identischen Pfaden der Wert Null und bei nicht-identischen Pfaden ein Wert größer als Null zurückgegeben wird. Neben der Hamming- und der Levenshtein-Distanz könnte eine solche Funktion auch auf der **Longest-Common-Subsequenz (LCS)** [161] basieren. Dabei wird die längste gemeinsame Teilsequenz an Zeichen beider Pfade ermittelt. Da die in dieser Arbeit vorgestellten Ergebnisse mithilfe einer LCS-Metrik erstellt wurden, soll diese ausführlich vorgestellt werden.
Um den LCS-Algorithmus zu erläutern, wird dieser im Folgenden zunächst mit den zwei Zeichenketten *D-B-C* und *B-D-C-B* ausgeführt. Bei der Verwendung des Verfahrens besteht jeder Knoten nicht nur aus einem einzigen Zeichen, sondern aus einer Zeichensequenz. Das detaillierte Vorgehen des LCS kann in der jeweiligen Tabelle nachvollzogen werden. R_0 bis R_3 steht dabei jeweils für Reihe 0 bis Reihe 3 und S_0 bis S_4 steht für Spalte 0 bis Spalte 4. In den Zeichenketten wurde jeweils am Anfang ein leeres Element ∅ eingefügt. In der letzten Tabelle sind dann alle Zwischenschritte enthalten. Das generelle Vorgehen lautet wie folgt. Stimmen die Markierungen der Zeichen nicht überein, so wird die längste Sequenz aus der rechten und/oder oberen Zelle übernommen. Stimmen die Werte überein, so wird das Zeichen an die Sequenzen in der linken oberen Zelle angehängt.

- $\boldsymbol{R_0}$ und $\boldsymbol{S_0}$ wurden jeweils mit ∅ gefüllt, da der Vergleich von einer leeren Sequenz mit einer nicht-leeren Sequenz wieder eine leere Sequenz ergibt.

		S_0 ∅	S_1 **B**	S_2 **D**	S_3 **C**	S_4 **B**
R_0	∅	**∅**	**∅**	**∅**	**∅**	**∅**
R_1	**D**	**∅**				
R_2	**B**	**∅**				
R_3	**C**	**∅**				

- Der $\boldsymbol{LCS(R_1, S_1)}$ vergleicht die Zeichen D und B miteinander, die nicht gleich sind. Da der Wert von $LCS(R_1, S_0)$ und der Wert von $LCS(R_0, S_1)$ beide $\varnothing$ enthalten, erhält der $LCS(R_1, S_1)$ auch die leere Menge $\varnothing$.

		S_0 $\varnothing$	S_1 **B**	S_2 **D**	S_3 **C**	S_4 **B**
R_0	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
R_1	**D**	$\varnothing$	**←↑ ∅**			
R_2	**B**	$\varnothing$				
R_3	**C**	$\varnothing$				

- Bei $\boldsymbol{LCS(R_1, S_2)}$ werden die Zeichen D und D miteinander verglichen. Da sie gleich sind, wird D an die Sequenz der linken oberen Zelle der Tabelle angehängt. Das ergibt als Sequenz für $LCS(R_1, S_2)$ die leere Menge $\varnothing$ und D, wobei die leere Menge weggelassen werden kann.

		S_0 $\varnothing$	S_1 **B**	S_2 **D**	S_3 **C**	S_4 **B**
R_0	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
R_1	**D**	$\varnothing$	←↑ ∅	**↖ (D)**		
R_2	**B**	$\varnothing$				
R_3	**C**	$\varnothing$				

- Bei $\boldsymbol{LCS(R_1, S_3)}$ wird D mit C verglichen, die nicht gleich sind. Die darüberliegende Sequenz ist leer und die Sequenz der links daneben liegenden Zelle ist D. Die längere der beiden Sequenzen wird beibehalten, also zeigt der Pfeil nach links und D wird genutzt.

		S_0 $\varnothing$	S_1 **B**	S_2 **D**	S_3 **C**	S_4 **B**
R_0	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
R_1	**D**	$\varnothing$	←↑ ∅	↖ (D)	**← (D)**	
R_2	**B**	$\varnothing$				
R_3	**C**	$\varnothing$				

- Bei $\boldsymbol{LCS(R_1, S_4)}$ erhält man auf die gleiche Weise wieder D.

		S_0 $\varnothing$	S_1 **B**	S_2 **D**	S_3 **C**	S_4 **B**
R_0	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
R_1	**D**	$\varnothing$	←↑ ∅	↖ (D)	← (D)	**← (D)**
R_2	**B**	$\varnothing$				
R_3	**C**	$\varnothing$				

- Bei $\boldsymbol{LCS(R_2, S_1)}$ wird B mit B verglichen. Somit wird B an die leere Menge $\varnothing$ von links oben angehängt und es bleibt B.

		S_0 $\varnothing$	S_1 **B**	S_2 **D**	S_3 **C**	S_4 **B**
R_0	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
R_1	**D**	$\varnothing$	$\leftarrow\uparrow \varnothing$	$\nwarrow$ (D)	$\leftarrow$ (D)	$\leftarrow$ (D)
R_2	**B**	$\varnothing$	**$\nwarrow$ (B)**			
R_3	**C**	$\varnothing$				

- Bei $\boldsymbol{LCS(R_2, S_2)}$ stimmen B und D nicht überein. Somit wird der $LCS(R_1, S_2)$=(D) und der $LCS(R_2, S_1)$=(B) genutzt. Da in beiden Zellen jeweils ein Element enthalten ist, ist der $LCS(R_2, S_2)$=(B)&(D).

		S_0 $\varnothing$	S_1 **B**	S_2 **D**	S_3 **C**	S_4 **B**
R_0	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
R_1	**D**	$\varnothing$	$\leftarrow\uparrow \varnothing$	$\nwarrow$ (D)	$\leftarrow$ (D)	$\leftarrow$ (D)
R_2	**B**	$\varnothing$	$\nwarrow$ (B)	**$\leftarrow\uparrow$ (B) & (D)**		
R_3	**C**	$\varnothing$				

- Bei $\boldsymbol{LCS(R_2, S_3)}$ stimmt B nicht mit C überein. $LCS(R_2, S_2)$ enthält (B)&(D) und $LCS(R_1, S_3)$ enthält (D). Da (D) ja bereits in (B)&(D) enthalten ist, ergibt $LCS(R_2, S_3)$=(B)&(D).

		S_0 $\varnothing$	S_1 **B**	S_2 **D**	S_3 **C**	S_4 **B**
R_0	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
R_1	**D**	$\varnothing$	$\leftarrow\uparrow \varnothing$	$\nwarrow$ (D)	$\leftarrow$ (D)	$\leftarrow$ (D)
R_2	**B**	$\varnothing$	$\nwarrow$ (B)	$\leftarrow\uparrow$ (B) & (D)	**$\leftarrow\uparrow$ (B) & (D)**	
R_3	**C**	$\varnothing$				

- Bei $\boldsymbol{LCS(R_2, S_4)}$ stimmt B mit B überein, was an die linke obere Zelle angehängt wird. Also ergibt $LCS(R_2, S_4)$=(DB).

		S_0 $\varnothing$	S_1 **B**	S_2 **D**	S_3 **C**	S_4 **B**
R_0	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
R_1	**D**	$\varnothing$	$\leftarrow\uparrow \varnothing$	$\nwarrow$ (D)	$\leftarrow$ (D)	$\leftarrow$ (D)
R_2	**B**	$\varnothing$	$\nwarrow$ (B)	$\leftarrow\uparrow$ (B) & (D)	$\leftarrow\uparrow$ (B) & (D)	**$\nwarrow$ (DB)**
R_3	**C**	$\varnothing$				

- Bei $\boldsymbol{LCS(R_3, S_1)}$ stimmt C nicht mit B überein. Somit wird der darüberliegende Wert von $LCS(R_2, S_1)$=(B) für $LCS(R_3, S_1)$ genutzt.

		S_0 ∅	S_1 **B**	S_2 **D**	S_3 **C**	S_4 **B**
R_0	∅	∅	∅	∅	∅	∅
R_1	**D**	∅	←↑ ∅	↖ (D)	← (D)	← (D)
R_2	**B**	∅	↖ (B)	←↑ (B) & (D)	←↑ (B) & (D)	↖ (DB)
R_3	**C**	∅	**↑ (B)**			

- Bei $\boldsymbol{LCS(R_3, S_2)}$ stimmen C und D nicht überein. Es werden also die Elemente von der darüberliegenden Zelle und der linken Nachbarzelle genutzt und kombiniert. Der $LCS(R_3, S_2)$ ergibt $(B)\&(D)$.

		S_0 ∅	S_1 **B**	S_2 **D**	S_3 **C**	S_4 **B**
R_0	∅	∅	∅	∅	∅	∅
R_1	**D**	∅	←↑ ∅	↖ (D)	← (D)	← (D)
R_2	**B**	∅	↖ (B)	←↑ (B) & (D)	←↑ (B) & (D)	↖ (DB)
R_3	**C**	∅	↑ (B)	**←↑ (B) & (D)**		

- Bei $\boldsymbol{LCS(R_3, S_3)}$ stimmt C mit C überein, wobei C an die Sequenzen der linken oberen Zelle angehängt wird. Somit entsteht für $LCS(R_3, S_3)$ gleich $(BC)\&(DC)$.

		S_0 ∅	S_1 **B**	S_2 **D**	S_3 **C**	S_4 **B**
R_0	∅	∅	∅	∅	∅	∅
R_1	**D**	∅	←↑ ∅	↖ (D)	← (D)	← (D)
R_2	**B**	∅	↖ (B)	←↑ (B) & (D)	←↑ (B) & (D)	↖ (DB)
R_3	**C**	∅	↑ (B)	←↑ (B) & (D)	**↖ (BC) & (DC)**	

- Bei $\boldsymbol{LCS(R_3, S_4)}$ wird C mit B verglichen, was nicht übereinstimmt. Da die darüberliegende Zelle eine Sequenz mit zwei Elementen enthält und auch die linke Nachbarzelle nur Sequenzen mit zwei Elementen enthält, werden diese kombiniert, was für $LCS(R_3, S_4)$=$(BC)\&(DC)\&(DB)$ ergibt.

		S_0 ∅	S_1 **B**	S_2 **D**	S_3 **C**	S_4 **B**
R_0	∅	∅	∅	∅	∅	∅
R_1	**D**	∅	←↑ ∅	↖ (D)	← (D)	← (D)
R_2	**B**	∅	↖ (B)	←↑ (B) & (D)	←↑ (B) & (D)	↖ (DB)
R_3	**C**	∅	↑ (B)	←↑ (B) & (D)	↖ (BC) & (DC)	**←↑ (BC) & (DC) & (DB)**

Nach diesem Beispiel lässt sich auch die formale Definition des LCS-Algorithmus besser nachvollziehen. Das Beispiel wurde aufgrund einer besseren Nachvollziehbarkeit auf Zeichenketten ausgeführt. Die formale Definition des LCS-Algorithmus ist auf Pfaden/Sequenzen, die aus Knoten bestehen, aufgebaut.
$p = (x_1, x_2, ..., x_k)$ und $q = (y_1, y_2, ..., y_l)$ sollen zwei zu vergleichende Pfade sein. p besteht dabei aus den Teilpfaden $p_0, p_1, ..., p_k$ und q aus den Teilpfaden $q_0, q_1, ..., q_l$. Die Belegungen der Knoten ($x_1, ..., x_k$ und $y_1, ..., y_l$) kann durch eine allgemeine Belegungsfunktion $f_{label}(n_i)$ erhalten werden, wobei ein Knoten n als Eingabe dient und die entsprechende Belegung des Knotens zurückliefert. Die Definition des LCS-Algorithmus nach Wagner und Fischer [161] lautet wie folgt:

$$LCS(p_i, q_j) = \begin{cases} \varnothing & \text{wenn } i = 0 \text{ oder } j = 0 \\ LCS(p_{i-1}, q_{j-1}) \hat{} x_i & \text{wenn } i, j > 0 \text{ und } f_{\text{label}}(x_i) = f_{\text{label}}(y_j) \\ max\{LCS(p_i, q_{j-1}), LCS(p_{i-1}, q_j)\} & \text{wenn } i, j > 0 \text{ und } f_{\text{label}}(x_i) \neq f_{\text{label}}(y_j) \end{cases}$$

Um den LCS-Wert von p_{i} und q_{j} zu finden, werden $f_{\text{label}}(x_i)$ und $f_{\text{label}}(y_j)$ miteinander verglichen. Wenn sie gleich sind, dann wird die Sequenz von $LCS(p_{i-1}, q_{j-1})$ mit x_i erweitert. Wenn x_i und y_j nicht gleich sind, dann wird die längere der beiden Sequenzen aus $LCS(p_i, q_{j-1})$ und $LCS(p_{i-1}, q_j)$ genutzt. Sind diese beiden Sequenzen gleich lang und verschieden, so werden beide Sequenzen genutzt.
Dies ist das Vorgehen, um die längste gemeinsame Teilsequenz aus zwei Zeichenketten/Pfaden zu bestimmen. Die längste gemeinsame Teilsequenz der Pfade *D-B-C* und *B-D-C-B* hat die Länge 2, da alle drei (längsten) Teilsequenzen (BC & DC & DB) zwei Elemente enthalten. Aber natürlich erfüllt der LCS-Algorithmus nicht die Anforderungen an die δ-Funktion. Denn diese soll Null bei exakt gleichen Pfaden zurückgeben und einen Wert größer als Null, wenn die Pfade unterschiedlich sind. Zunächst bestimmt der in dieser Arbeit genutzte LCS-Algorithmus die Länge, der größten gemeinsamen Teilsequenz und liefert diese zurück. Weiterhin muss der Wert, der vom LCS-Algorithmus berechnet wird, noch so angepasst werden, dass er die Bedingung der δ-Funktion erfüllt. Die längste gemeinsame Teilsequenz von zwei Zeichenketten kann immer nur so lang sein, wie die kürzere der beiden Zeichenketten. Also wurde in dieser Arbeit von der Länge der kürzeren Zeichenkette der vom LCS-Algorithmus berechnete Wert abgezogen. Stimmen die Zeichenketten exakt überein, dann ergibt das den Wert Null, ansonsten einen Wert größer als Null.
$\delta_{LCS}(p, q) = (Length_p - LCS(p, q))$ mit $Length_p \leq Length_q$

4.2.4 Vergleich von Pfadmengen

Die im letzten Abschnitt 4.2.3 vorgestellte δ-Funktion kann zum Vergleich von Beschreibungsmengen erweitert werden, die zur Identifizierung von Codeklonen verwendet wird. Zu diesem Zweck wird beim Vergleich zweier Codefragmente mit Beschreibungsmengen DF_{set} bzw. DG_{set} für jeden Pfad p von DF_{set} der ähnlichste Pfad q von DG_{set} identifiziert, also q mit dem minimalen Wert $\delta(p, q)$ und anschließend der Mittelwert über alle diese Werte berechnet. Formal sei DSET das Universum der Beschreibungsmengen, dann sei der Vergleich von $DF_{set}, DG_{set} \in$ DSET mit $|DF_{set}| \leq |DG_{set}|$ als Funktion Δ definiert, sodass:

$$\Delta : \text{DSET} \times \text{DSET} \to \mathbb{N}, \text{ mit}$$

$$\Delta(DF_{set}, DG_{set}) = \frac{\sum_{p \in DF_{set}} min(\{\delta(p,q) \mid q \in DG_{set}\})}{|DF_{set}|}$$

Wie beim Vergleich von Pfaden ist es weniger interessant, ob zwei Beschreibungsmengen exakt identisch sind. Aus diesem Grund wird ein Schwellenwert τ eingeführt, der zur Definition einer Grenze für die maximale Differenz ähnlicher Beschreibungsmengen verwendet wird. Wenn $\Delta(D_{set}, D'_{set}) < \tau$ für zwei Beschreibungsmengen D_{set}, D'_{set} erfüllt ist, gelten die Beschreibungsmengen als ähnlich und die entsprechenden Codefragmente bilden einen Codeklon.

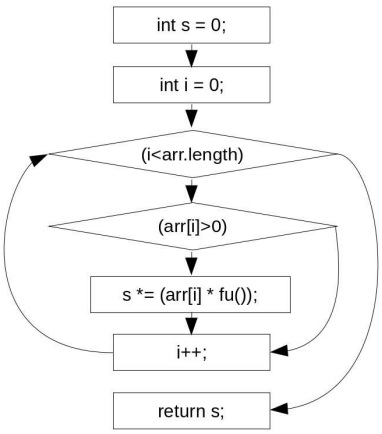
Kapitel 5

Dominatorbasierte Klonerkennung für Sourcecode

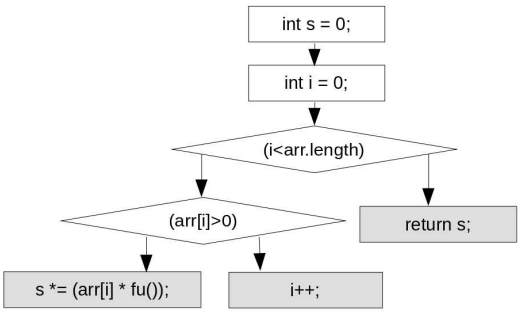
Der im vorherigen Abschnitt vorgestellte allgemeine Ansatz zur Codeklonerkennung wird in diesem Abschnitt auf den Java-Sourcecode angewendet. Man könnte dies als eine Instanziierung des allgemeinen Ansatzes verstehen, denn im Prinzip lässt sich der Ansatz für jede Programmiersprache oder Coderepräsentation umsetzen, für die gültige Programme in Form eines Kontrollflussgraphen modelliert werden können. In Abbildung 5.1 ist der Überführungsprozess für eine Java-Funktion hin zu den Beschreibungsmengen beschrieben. Zunächst wird für die Funktion aus Abbildung 5.1 (a) der Kontrollflussgraph in Abbildung 5.1 (b) erzeugt. Aus diesem wird dann, wie im vorherigen Abschnitt beschrieben, der Dominatorbaum in Abbildung 5.1 (c) erstellt. Aus diesem allgemeinen Dominatorbaum wird nun der *Abstrakte Dominatorbaum* in Abbildung 5.1 (d) erzeugt. Anweisungen und ihre Operanden sind darin in der Preorder-Notation kodiert. Des Weiteren wird nicht zwischen Variablennamen, Feldnamen und Literalen unterschieden. Stattdessen werden die Variablennamen in ein einziges Symbol V umbenannt, Literale in ein einzelnes Symbol L und Funktionsaufrufe in das Symbol C, um drei Beispiele zu nennen. Ein ähnliches Vorgehen wurde auch im Tool NiCad [36,126] umgesetzt. Zum Beispiel steht `[=,V,L]` im abstrakten Dominatorbaum für die Anweisung `int s = 0;`. Funktionsnamen werden speziell behandelt, indem ein Konstantenpool genutzt wird. Durch das Verwenden der unqualifizierten Funktionsnamen konnten in Experimenten verbesserte Precision-Ergebnisse ermittelt werden. Ein Zugriff auf einen Konstantenpool wird durch das Zeichen `#` verschlüsselt. So steht als Eintrag im Konstantenpool `#1 =`

```
1 int func(int[] arr) {
2     int s = 0;
3     for (int i = 0; i<arr.length; i++)
4         if (arr[i]>0)
5             s *= (arr[i] * fu());
6     return s;
7 }
```

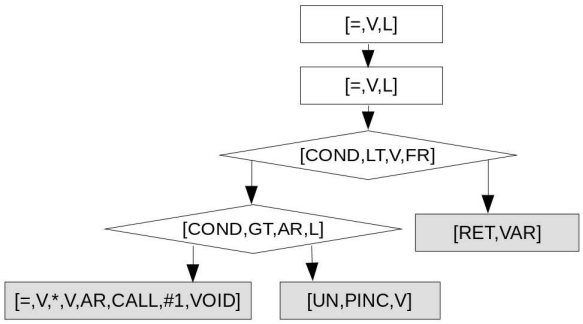
(a) Funktion aus Abbildung 2.5



(b) Kontrollflussgraph



(c) Dominatorbaum



(d) Abstrakter Dominatorbaum

⇓

```
Dset=
[[=,V,L]->[=,V,L]->[COND,LT,V,FR]->[COND,GT,AR,L]->
[=,V,*,V,AR,CALL,#1,VOID], [=,V,L]->[=,V,L]->[COND,LT,V,FR]
->[COND,GT,AR,L]->[UN,PINC,V],
 [=,V,L]->[=,V,L]->[COND,LT,V,FR]->[RET,VAR]]
```

0:	...
1:	this.fu()
2:	...
3:	...

Konstantenpool

⇓ Abbildung auf Hashwerte

```
[574292532->574292532->-241626780->-986283828->1899560438, 574292532->574292532->
-241626780->-986283828->619574584, 574292532->574292532->-241626780->602921683]

Dsplit=
[[=,V,L]->[=,V,L], [COND,GT,AR,L], [COND,GT,AR,L]->[=,V,*,V,AR,CALL,#1,VOID],
[COND,GT,AR,L]->[UN,PINC,V], [COND,LT,V,FR]->[RET,VAR]]
```

⇓ Abbildung auf Hashwerte

```
[574292532->574292532, -986283828, -986283828->1899560438, -986283828->619574584,
-241626780->602921683]
```

(e) Beschreibungsmengen

Abbildung 5.1: Funktion, Kontrollflussgraph, Dominatorbaum, Abstrakter Dominatorbaum und die Beschreibungsmengen der Funktion aus Abbildung 2.5.

```
1 private static void zipDir(final File zipDir,
2     final ZipOutputStream zos, final String path) {
3   final String [] dirList = zipDir.list();
4   final byte [] readBuffer = new byte[chunk];
5   int bytesIn = 0;
6   for (int i = 0; i < dirList.length; i ++) {
7     final File file = new File(zipDir, dirList[i]);
8     final String filePath =
9         ((path != null && path.length() > 0)
10        ? (path+File.separator) : "") + file.getName();
11    if (file.isDirectory()) {
12      zipDir(file, zos, filePath);
13      continue;
14    }
15    final FileInputStream fis =
16        new FileInputStream(file);
17    try {
18      final ZipEntry anEntry = new ZipEntry(filePath);
19      zos.putNextEntry(anEntry);
20      while ((bytesIn=fis.read(readBuffer)) != -1) {
21        zos.write(readBuffer, 0, bytesIn);
22      }
23      zos.flush();
24    } finally {
25      fis.close();
26    }
27  }
28 }
```

```
1 public static void zipDir(File zipDir,
2     ZipOutputStream zos, String path) {
3   String [] dirList = zipDir.list();
4   byte [] readBuffer = new byte[2156];
5   int bytesIn = 0;
6   for (int i = 0; i < dirList.length; i ++) {
7     File f = new File(zipDir, dirList[i]);
8
9
10
11    if (f.isDirectory()) {
12      zipDir(f, zos, path + f.getName() + "/");
13
14    } else {
15      FileInputStream fis = new FileInputStream(f);
16
17      ZipEntry anEntry =
18          new ZipEntry(path + f.getName());
19      zos.putNextEntry(anEntry);
20      while ((bytesIn=fis.read(readBuffer)) != -1) {
21        zos.write(readBuffer, 0, bytesIn);
22      }
23
24
25      fis.close();
26    }
27  }
28 }
```

Abbildung 5.2: Schwer zu erkennendes strukturelles Klonpaar.

`this.fu()`, wobei dann `[=,V,*,V,AR,CALL,#1,VOID]` für die Anweisung `sum *= (array[i] * fu());` steht. In der Kodierung steht $CALL$ für den Funktionsaufruf und mit #1 wird auf den Konstantenpool zugegriffen, um den Funktionsnamen zu erhalten. $VOID$ steht in diesem Aufruf dafür, dass kein Parameter übergeben wird. Mit dieser abstrakten Darstellung lassen sich Codefragmente finden, die syntaktisch nicht vollständig übereinstimmen, insbesondere Klone vom Typ-2 und Teile von Typ-3/4, aber natürlich auch zuverlässig Typ-1- und Typ-2-Klone.
Die Beschreibungsmengen werden dann aus dem abstrakten Dominatorbaum abgeleitet, wie es bereits im Abschnitt 4.2.1 beschrieben wurde. Aufgrund der schnelleren Verarbeitung werden die Beschreibungsmengen aber dann noch durch Hashwerte kodiert. Zur Kodierung mit Hashwerten wurde ein 4-Byte Hashverfahren genutzt, wobei die auf diese Weise erstellten Beschreibungsmengen in 5.1 (e) abgebildet sind. Mithilfe dieses Vorgehens können Beschreibungsmengen effizient erzeugt und verglichen werden. Damit eingeschätzt werden kann, wie gut ein solcher Klonerkennungsansatz für Java-Sourcecode mit Dominatorinformationen im Vergleich zu anderen Klonerkennungsverfahren arbeitet, braucht es einen modernen Klonerkennungsbenchmark. Der größte und am weitesten verbreitete Benchmark für Java-Programme ist der BigCloneBench [147–149], welcher bereits im Abschnitt 2.5 vorgestellt wurde. In dem Benchmark bestehen alle Klone aus ganzen Funktionen und nicht auch aus Teilfunktionen (Teilblöcke von Funktionen), weshalb Programmfragment im Weiteren immer für eine ganze Funktion steht. Mithilfe der vorgestell-

```
1 String substitute(String input, String var,
2     String [] fields, String [] fieldData) {
3   StringBuffer out = new StringBuffer();
4   int varlen = var.length();
5   int oidx = 0;
6   for (;;) {
7     if (DEBUG) System.out.println(
8         "checking for occurrence of " +
9         var + " in :" + input.substring(oidx));
10    int idx = input.indexOf(var, oidx);
11    if (idx == -1) break;
12    out.append(input.substring(oidx, idx));
13    idx += varlen;
14    if (input.charAt(idx) != '.')
15      throw new IOException("no field");
16    idx ++;
17    int idx_save = idx;
18    for (int i = 0; i < fields.length; i ++) {
19      String fld = fields[i];
20      int flen = fld.length();
21      if (DEBUG) System.out.println(
22          "checking if it is field " + fld);
23      if (input.regionMatches(idx, fld, 0, flen)) {
24        String value = fieldData[i];
25        if (DEBUG) System.out.println (
26            "field matches. outputting data:" + value);
27        out.append (value);
28        idx += flen;
29        break;
30      }
31    }
32    if (idx == idx_save)
33      throw new IOException("unknown field");
34    oidx = idx;
35  }
36  if (DEBUG) System.out.println(
37      "no more variables left on this line");
38  out.append (input.substring (oidx));
39  return out.toString ();
40 }
```

```
1 String substitute(String input, String var,
2     String value) {
3   StringBuffer out = new StringBuffer();
4   int varlen = var.length();
5   int oidx = 0;
6   for (;;) {
7
8
9
10    int idx = input.indexOf(var, oidx);
11    if (idx == -1) break;
12    out.append (input.substring(oidx, idx));
13    idx += varlen;
14
15
16
17
18
19
20
21
22
23
24
25
26
27    out.append(value);
28
29
30
31
32
33
34    oidx = idx;
35  }
36
37
38  out.append(input.substring(oidx));
39  return out.toString();
40 }
```

Abbildung 5.3: Schwer zu erkennendes Klonpaar: Subklon.

ten Methodik lassen sich insbesondere *Strukturelle Klone* und *Subklone* erkennen. In Abbildung 5.2 ist ein strukturelles Klonpaar abgebildet, welches im BigCloneBench enthalten ist. Dieses Klonpaar kann mit der hier vorgestellten Methodik mit einem Schwellenwert von 0.26 identifiziert werden. Andere Tools haben aufgrund der großen Anzahl an syntaktischen Unterschieden Schwierigkeiten, diesen Klon zu identifizieren. Das Klonerkennungssystem NiCad identifiziert beispielsweise dieses Klonpaar erst bei einem Schwellenwert von 0.58, also einem syntaktischen Unterschied der beiden Codefragmente von 58%. Bei einer genaueren Betrachtung kann man erkennen, dass die Unterschiede durch die Verwendung von verschiedenen syntaktischen Kontrollsequenzen stammen, die jedoch die gleiche Bedeutung haben. Das *continue*-Statement auf der linken Seite in Zeile 13 wird beispielsweise durch den *else*-Zweig auf der rechten Seite ersetzt. Der *try-finally*-Block auf der linken Seite in den Zeilen 17-26 wurde auf der rechten Seite durch eine einfache Instruktionssequenz mit dem Inhalt des *finally*-Blocks in der Zeile 25 ersetzt. Solche Codeklone, die einen ähnlichen oder gleichen Kontrollfluss abbilden, aber dafür verschiedene syntaktische Konstrukte benötigen, nennt man auch **Strukturelle Klone**, die bereits im Abschnitt 2.3 vorgestellt wurden. In Abbildung 5.3 ist ein weiteres Funktionspaar dargestellt, das von dem in der Arbeit vorgestellten

	Konfiguration
Deckard	Funktionsgranularität, 70% Ähnlichkeit, 2 Tokenschritte
SourcererCC	Funktionsgranularität, 70% Ähnlichkeit
NiCad v6.2	blind renaming, literal abstraction, 70% Schwellenwert
iClones	Single Eingabeformat, 50 Token min. Klonlänge, 15 Minblock
CCAligner	60% Schwellenwert, 6 Fenstergröße, 1 Editierdistanz
Oreo	55% action filter Schwellenwert, 60% partition Schwellenwert

Tabelle 5.1: Tooleinstellungen

Ansatz als Klonpaar identifiziert wird. Die beiden Funktionen stammen erneut aus dem BigCloneBench und unterscheiden sich nicht nur stark in der Syntax, sondern auch in der Länge. Der dominatorbasierte Ansatz findet diesen Klon mit einem Schwellenwert von 0.26, wohingegen NiCad den Klon erst bei einem Schwellenwert von erneut 0.58 findet. Interessanterweise findet auch das Tool CCAligner dieses Klonpaar nicht, obwohl es speziell für Klone mit großen Lücken [166] erstellt wurde. Die Funktion auf der linken Seite erweitert das Verhalten der Methode auf der rechten Seite mit zusätzlichen Kontrollflusspfaden, jedoch sind alle Elemente der rechten Funktion in der linken Seite enthalten. Klone, bei denen der Kontrollfluss eines Programmfragmentes vollständig in einem anderen Fragment enthalten ist und das gegebenenfalls zusätzliche Instruktionen und/oder Kontrollflusselemente beinhaltet, nennt man **Subklone**. Diese sind vergleichbar zu denen in Abschnitt 2.5 vorgestellten *Gapped Klonen*. Der Unterschied ist jedoch, dass *Gapped Klone* oder auch *Large-Gapped Klone* [166] durch die Syntax definiert werden, wohingegen *Subklone* durch den Kontrollfluss definiert sind. In den folgenden Unterabschnitten werden der Recall-Wert auf dem BigCloneBench gemessen, der Precision-Wert ermittelt,

Einstellungen
Variablengleichsetzung
Konstantengleichsetzung
Nutzung von Funktionsnamen
16 Threads
δ_{LCS}-Metrik
4-Byte-Hashing Kodierung
30% Schwellenwert
Mindestgröße 15 Zeilen
50% / 70% Beschreibungsmengenunterschied
Pfadentfernung 3 / 1 Knoten
Pfadunterschied 7 Pfade

Tabelle 5.2: Genutzte Einstellungen bei StoneDetector.

	Deckard		SourcererCC		NiCad		iClones		CCAligner		StoneDetector Normal		StoneDetector Split		Oreo	
Typ-1	18.318	87,8%	20.827	99,9%	20.829	99,9%	20.842	99,9%	20.829	99,9%	20.831	99,9%	20.831	99,9%	20.844	100%
Typ-2	3.317	95,4%	3.444	99,1%	3.460	99,5%	3.453	99,3%	3.456	99,4%	3.476	100%	3.476	100%	3.475	99,9%
Very-Strongly Typ-3	2.139	65,0%	3.209	97,6%	3.243	98,6%	3.253	98,9%	3.218	97,8%	3.272	99,4%	3.269	99,3%	3.283	99,8%
Strongly Typ-3	1.806	24,4%	5.784	78,1%	6.246	84,3%	6.531	88,2%	5.943	80,2%	6.702	90,5%	6.808	91,9%	6.584	88,9%
Moderately Typ-3	2.311	8,6%	1.462	5,5%	169	0,6%	4.051	15,1%	1.784	6,6%	4.335	16,1%	5.481	20,4%	5.187	19,4%
Typ-4	57.063	1,3%	250	0,0%	7	0,0%	6.554	0,1%	2.133	0,0%	2.691	0,0%	5.433	0,1%	6.821	0,1%
Klone insgesamt	13.239.074		804.315		754.022		987.659		1.244.160		1.006.823		1.275.539		1.015.425	

Tabelle 5.3: Recall-Werte mit Klontypen verschiedener Klonerkennungstools auf dem BigCloneBench.

die Skalierbarkeit festgestellt und ein detaillierter Vergleich mit anderen Klonerkennungstools vorgenommen. Alle hier vorgestellten Ergebnisse und Messungen wurden mit den Tooleinstellungen ermittelt, die in Tabelle 5.1 eingesehen werden können. Dies sind entweder die Standardkonfigurationen für die entsprechenden Tools oder aber durch Experimente ermittelte verbesserte Konfigurationen. Die Einstellungen für StoneDetector sind in der Tabelle 5.2 abgebildet. An Stellen, bei denen zwei Werte angegeben sind, wurde der erste Wert für ganze Pfade und der zweite Wert für gesplittete Pfade genutzt. Alle Experimente wurden auf einer Linux-Workstation mit einem Intel Core i7-10700K Prozessor und mit 32 GB RAM durchgeführt.

5.1 Recall-Wert

In diesem Abschnitt wird ermittelt, wie die Recall-Werte des vorgestellten Ansatzes im Vergleich zu den Werten der anderen Tools auf dem BigCloneBench abschneiden. Dazu wurde BigCloneEval [150] genutzt, um jeweils einen Recall-Test auf dem BigCloneBench durchzuführen. Die bereits im Kapitel 3 vorgestellten Tools Deckard [63], SourcererCC [131], NiCad [36,126], iClones [54], CCAligner [166] und Oreo [130] wurden dabei unter den gleichen Bedingungen wie der StoneDetector getestet. In dem BigCloneBench sind Programmfragmente mit verschiedenen Längen und damit auch Klonpaare mit den verschiedensten Längen enthalten. Wie bereits im Abschnitt 2.1 erklärt, werden in verschiedenen Arbeiten verschiedene Mindestlängen

für Programmfragmente angenommen, die als Programmfragmente für die Codeklonerkennung genutzt werden können. Da die hier vorgestellte dominatorbasierte Technik besonders verschiedene bzw. gleiche Kontrollflüsse berücksichtigt, wurde entschieden, dass nur Funktionen mit mindestens 15 Zeilen in Betracht gezogen werden. Bei 15 Zeilen ist es wahrscheinlicher, dass mehr Kontrollstrukturen im Quelltext zum Einsatz kommen, die den Kontrollfluss aufteilen. Kleinere Funktionen mit beispielsweise 6 Zeilen sind meist leichter (auch für andere Tools) zu detektieren, da sie weniger Strukturelemente enthalten können, die den Kontrollfluss aufteilen. Zudem bestehen 3 der 6 Zeilen aus der Funktionssignatur sowie aus der öffnenden und schließenden geschweiften Klammer. Daher sind mindestens 15 Zeilen ein anspruchsvoller Wert, bei dem die Tools miteinander verglichen werden können.
Viele der genutzten Tools sind darauf ausgelegt, auch Klone zwischen Programmfragmenten zu detektieren, die nicht aus ganzen Funktionen bestehen. Damit ein fairer Vergleich gezogen werden konnte, mussten die Ergebnisse der Tools daher überarbeitet werden. Das typische Vorgehen hierbei soll an dem Beispiel von iClones kurz beschrieben werden. So wurde mit BigCloneEval eine Klonsuche mit iClones gestartet, wobei iClones alle Programmfragmente (nicht nur Funktionen) zur Verarbeitung genutzt hat, die aus mindestens 50 Tokens bestehen. Anschließend wurde eine Liste mit allen vorhandenen Funktionen im BigCloneBench erstellt. Wenn ein von iClones detektiertes Klonpaar ein Programmfragment enthielt, welches keine ganze Funktion darstellt, so wurde die umliegende Funktion in der Liste aller Funktionen gesucht und genutzt. Nachdem alle nötigen Programmfragmente ersetzt wurden, wurden nun solche Klonpaare ausgeschlossen, bei denen mindestens eine Funktion weniger als 15 Zeilen Code besitzt. Auf diese Weise konnte ein fairer Vergleich der Recall-Werte durchgeführt werden.
In der Tabelle 5.3 sind die Recall-Werte für die verschiedenen Tools nach den im Abschnitt 2.3 vorgestellten Klontypen aufgelistet. Dabei sind sowohl die absolute Anzahl als auch der prozentuale Anteil an gefundenen Klonen abgebildet. Weiterhin sind die insgesamt gefundenen Klone für jedes Tool aufgeführt, bei denen alle Dopplungen an gefundenen Klonen herausgefiltert wurden. Dopplungen können entstehen, da einige Sourcecode-Dateien mehrfach im Benchmark enthalten sind. Die insgesamt gefundenen Klone geben aber keine Auskunft darüber, wie die Precision-Werte für die entsprechenden Tools sind. Dieser Wert dient lediglich als ein weiterer Vergleichswert zwischen den Klonerkennungstools.
StoneDetector wurde sowohl mit ganzen Pfaden als auch mit den aufgeteilten/gesplitteten Pfaden gemessen und erzielt in beiden Konfigurationen mindestens gleich

gute oder bessere Recall-Werte im Vergleich zu den anderen Klonerkennungsanwendungen. Ab Typ-3-Klonen sinkt der Prozentsatz der gefundenen Klone bei allen Tools, was auf die allgemeinen Schwierigkeiten beim Auffinden von Codeklonen mit größerer syntaktischer Varianz hinweist. Außerdem wird der Unterschied zwischen den Near-Miss-Klondetektoren, also NiCad und SourcererCC, und den Tools, die sich auch mit Klonen des Typs 3 und 4 befassen, also CCAligner, iClones, Oreo, StoneDetector und Deckard, deutlich. Erwartungsgemäß schneidet StoneDetector bei den schwieriger zu findenden moderaten Typ-3- und Typ-4-Klonen deutlich besser ab als die Near-Miss-Klondetektoren und liefert auch im Vergleich zu CCAligner bessere Recall-Werte. StoneDetector mit gesplitteten Pfaden erreicht im Allgemeinen die beste Auffindbarkeit aller Tools für stark und mäßig starke Typ-3-Klone, wird aber von Oreo und Deckard bei Typ-4-Klonen übertroffen. Die hohe Anzahl von Typ-4-Klonen bei Deckard ist mit Vorsicht zu genießen, wenn man die im nächsten Abschnitt 5.2 und in der Literatur beschriebene schlechte Genauigkeit von Deckard bedenkt [131,166]. Zusammenfassend lässt sich sagen, dass StoneDetector bei Codeklonen mit größerer syntaktischer Varianz besser abschneidet als die anderen textuellen, lexikalischen und syntaktischen Tools und sogar eine fast ebenso gute Trefferquote erzielt wie das maschinelle Lernverfahren Oreo. Anders als Oreo braucht aber StoneDetector keine aufwendige Trainingsphase und *gelabelte* Trainingsdaten. Betrachtet man nur StoneDetector, so kann man feststellen, dass die Verwendung von gesplitteten Pfaden insgesamt die besseren Recall-Werte erzielt. Dies ist zu erwarten, da die Variante mit geteilten Pfaden toleranter gegenüber syntaktischer Varianz ist und die Erkennung von Subklonen unterstützt (vgl. Abschnitt 4.2.2).

5.2 Precision-Wert

Neben der Anzahl der Codeklone, die ein Klondetektor finden kann, muss auch bewertet werden, wie viele davon tatsächlich Codeklone sind. Es muss also der im Abschnitt 2.5 beschriebene Precision-Wert des Tools bestimmt werden, um den Gesamtnutzen des Klondetektors beurteilen zu können. Leider gibt es für die Messung des Precision-Wertes keinen Benchmark und BigCloneBench kann ebenso wenig genutzt werden, da BigCloneBench nur Informationen zu einem kleinen Teil aller möglichen Klonkandidaten enthält [149]. Aus diesem Grund und in Ermangelung eines gemeinsamen Maßstabs für die Messung des Precision-Wertes musste auf eine manuelle Bewertung der gefundenen Codeklone zurückgegriffen werden, wie sie in der Praxis und in verwandten wissenschaftlichen Veröffentlichungen üblich ist [130, 131, 166]. Für

	Deckard		SourcererCC		NiCad		iClones		CCAligner		StoneDetector Normal		StoneDetector Split		Oreo	
Typ-1	67	16,75%	214	53,50%	195	48,75%	182	45,50%	183	45,75%	174	43,50%	163	40,75%	108	27,00%
Typ-2	4	1,00%	34	8,50%	38	9,50%	26	6,50%	32	8,00%	33	8,25%	29	7,25%	19	4,75%
Typ-3	20	5,00%	151	37,75%	165	41,25%	160	40,00%	102	25,50%	160	40,00%	151	37,75%	211	52,75%
Typ-4	95	23,75%	1	0,25%	2	0,50%	21	5,25%	76	19,00%	29	7,25%	46	11,50%	58	14,50%
Keine Klone	214	53,50%	0	0,00%	0	0,00%	11	2,75%	7	1,75%	4	1,00%	11	2,75%	4	1,00%
Precision	0,46		1,00		1,00		0,97		0,98		0,99		0,97		0,99	

Tabelle 5.4: Precision-Wert und Klonklassifikation für 400 Klonpaarbeispiele verschiedener Klonerkennungstools.

jedes Tool aus der Recall-Messung im vorherigen Abschnitt 5.1 wurde eine Stichprobe von 400 entdeckten Klonpaaren nach dem Zufallsprinzip ausgewählt. Die Klonpaare wurden blind, d. h. ohne Offenlegung des Werkzeugs, drei „Juroren“ vorgelegt. Die Juroren klassifizierten die Klonpaare dann einzeln als richtig oder falsch positiv und lösten anschließend jede unklare Bewertung in einer Diskussion untereinander auf. Dabei wussten die Juroren zu keinem Zeitpunkt, von welchem Tool die potentiellen Klonpaare stammten. Darüber hinaus ordneten die Juroren den Klonen ihre entsprechenden Klontypen anhand der gemeinsamen Definitionen aus 2.3 zu. Die Juroren hatten die Instruktion, Funktionspaare als Klone anzusehen, wenn sie auch so im BigCloneBench vorkommen könnten. Für die Ermittlung des Precision-Wertes werden die Informationen über die Klontypen nicht gebraucht, jedoch bietet es einen besseren Einblick in die gefundenen Klonarten und -typen. Die Ergebnisse sind in Tabelle 5.4 dargestellt. In der Tabelle ist die Anzahl der Codeklone, die als *Wahr Positiv*, aufgeschlüsselt nach dem zugewiesenen Typ (Typ-1/Typ-2/Typ-3/Typ-4), *Falsch Positiv* (keine Klone) sowie die resultierende Genauigkeit für jedes Tool aufgeführt. Die Anzahl der Codeklone wird in absoluten Zahlen und relativ zum Anteil in der Zufallsstichprobe angegeben. Mit Ausnahme von Deckard erreichen alle Tools einschließlich StoneDetector sehr gute Precision-Werte und nur eine kleine Anzahl der von den Tools als Klone markierten Funktionspaare wurde als falsch positiv bewertet. In Übereinstimmung mit anderen Veröffentlichungen [130, 166] über die Precision-

Werte von Klonerkennungstools, hat Deckard einen vergleichsweise schlechten Precision-Wert. Nur 46% der entdeckten Codeklone wurden von den drei Juroren als tatsächliche Codeklone eingestuft, verglichen mit 97-100% bei den anderen Tools. Im Vergleich zu ähnlichen Auswertungen wurden bei dieser Auswertung im Allgemeinen bessere Precision-Werte durch die Tools erzielt. Dies ist auf die gewählte Mindestgröße von 15 Zeilen für Programmfragmente, im Vergleich zu der Mindestgröße von 6 Zeilen, die oft in anderen Arbeiten verwendet wird, zurückzuführen. Eine höhere Mindestgröße der Klone führt zu verbesserten Precision-Werten der Klondetektoren, wie bereits von Wang et al. [166] beschrieben wurde.

5.3 Detaillierter Vergleich mit Klonerkennungssystemen

In diesem Abschnitt sollen die gefundenen Klone von StoneDetector noch genauer mit den Klonlisten von einigen ausgewählten anderen Tools verglichen werden. Tabelle 5.4 gibt bereits einen Hinweis zu den Eigenschaften von StoneDetector. Betrachtet man die unterschiedlichen Anteile der Klontypen in den Stichproben der Tools, so kann man erneut den Unterschied zwischen den Near-Miss-Klondetektoren, z. B. NiCad, und den Klondetektoren, die auch Klone des Typs-3 und -4 ansprechen, einschließlich StoneDetector, beobachten. Interessanterweise hat Deckard im Gegensatz zu den anderen Tools nur einen geringen Anteil von Typ-3-Klonen. Jedoch muss auch eingeräumt werden, dass die Zahlen aufgrund der Unausgewogenheit von BigCloneBench [149] teilweise verzerrt sind. Aus diesem Grund wurden weitere Untersuchungen zu den Eigenschaften von StoneDetector durchgeführt. Zu diesem Zweck wurde StoneDetector paarweise mit NiCad, einem *Near-Miss-Klondetektor*, CCAligner, einem *Large-Gap-Klondetektor*, iClones, einem tokenbasierten Ansatz, und Oreo als Vertreter eines maschinellen Lernansatzes verglichen. Es wurde die StoneDetector-Variante mit gesplitteten Pfaden verwendet, weil sie eine bessere Trefferquote aufweist. Für jeden paarweisen Vergleich wurde die Gesamtzahl der Klone von BigCloneBench analysiert, die von beiden Tools ($A \cap B$), nur von StoneDetector ($A \setminus B$) und nur von dem anderen Tool ($B \setminus A$) erkannt wurden. Die Codeklone wurden anhand der Dateiposition und der Zeilennummern identifiziert. Die Ergebnisse der paarweisen Vergleiche sind in Tabelle 5.5 dargestellt. NiCad, CCAligner, iClones und Oreo haben jeweils eine große Überschneidung mit StoneDetector von bemerkenswert ähnlicher Größe. In Kombination mit der relativ geringen Anzahl

A	B	A∩B	A\B	B\A
StoneDetector	NiCad	719.379	556.160	34.643
	CCAligner	775.214	500.325	468.946
	iClones	753.297	522.242	234.362
	Oreo	774.672	500.867	240.753

Tabelle 5.5: Gemeinsame und exklusive Codeklone für Klonerkennungstools.

von Klonen, die nur von NiCad erkannt wurden, deutet dies hauptsächlich auf die Teilmenge der exakten und Near-Miss-Klone hin, die von allen vier Tools erkannt wurden. Darüber hinaus gibt es sowohl für CCAligner, iClones als auch für Oreo einen signifikanten Anteil an Codeklonen, die nur von StoneDetector erkannt werden. Es kann daher angenommen werden, dass StoneDetector eine große Menge von Codeklonen mit größerer syntaktischer Varianz erkennt und dass sich diese von denen unterscheiden, die von CCAligner, dem Klondetektor mit großen Lücken, iClones, dem tokenbasierten Klondetektor, und dem maschinellen Lernwerkzeug Oreo erkannt werden. Auf dieser Grundlage bestand das Interesse, die ausschließlich von den einzelnen Tools identifizierten Codeklone genauer zu untersuchen. Dazu wurde erneut auf eine manuelle Analyse von Stichproben zurückgegriffen und die Juroren untersuchten unabhängig voneinander Stichproben von 100 Klonen aus den Mengen $A \setminus B$ und $B \setminus A$ für jeden paarweisen Vergleich.

NiCad vs StoneDetector

Bei der Überprüfung von 100 zufälligen Klonpaaren, die von StoneDetector, nicht aber von NiCad erkannt wurden, wurden die Vorteile der Dominatorbaum-basierten Methode deutlich. StoneDetector findet mehr Klone mit größerer syntaktischer Varianz, insbesondere strukturelle Klone (55) und Subklone (35), wie in Abschnitt 4 vorgestellt. Die übrigen Klone wurden als andere Typen (4) und falsch-positive Klone (6) klassifiziert. Die Analyse von 100 Klonen, die nur von NiCad gefunden wurden, hat ergeben, dass StoneDetector diese aus zwei Gründen nicht finden konnte. Der erste Grund ist auf die Code-Vorverarbeitung und das *Pretty-Printing* in NiCad zurückzuführen. Dies führt insbesondere dazu, dass die Größe von Codefragmenten, d. h. von Funktionen und Methoden, oft durch das Einfügen zusätzlicher Zeilen erhöht und damit der relative Unterschied kleiner wird. Außerdem erhöht sich die Länge von Codefragmenten in NiCad, da Variablendeklarationen in die

Längenberechnung einbezogen werden. In StoneDetector hingegen werden Variablendeklarationen bei der lexikalischen und syntaktischen Analyse eliminiert. Der zweite Grund, der bei der Untersuchung der ausschließlich von NiCad gefundenen Klone deutlich wurde, zeigt eine Schwäche des dominatorbasierten Ansatzes auf. Die Pfade im Dominatorbaum sind oft kürzer als im Kontrollflussgraphen oder sogar kürzer als in den von NiCad verwendeten normalisierten Quellcode-Sequenzen. Treten jedoch Unterschiede lokal in kürzeren Pfaden auf, bedeutet dies ein Problem für die Berechnung der Ähnlichkeit von Pfaden, was in diesen Fällen zu größeren Distanzwerten im Vergleich zu NiCad führt.

CCAligner vs StoneDetector

Ein besonderes Interesse lag auf dem Vergleich mit dem Large-Gap-Klondetektor CCAligner. Insgesamt lassen sich die vom StoneDetector, aber nicht vom CCAligner entdeckten Codeklone in zwei Kategorien einteilen. Erstens in ganz einfache Klone, die auch von einem einfachen Klondetektor gefunden werden. Tatsächlich befanden sich unter den gefundenen Klonen viele Klonpaare, bei denen die Funktionen aus kleinen einzelnen Grundbausteinen bestanden. Außerdem war auffällig, dass das Auftreten von Anweisungen, die nicht in beiden Funktionen enthalten waren, über die gesamte Länge der Funktionen verteilt waren. Der Grund dafür, dass CCAligner solche einfachen Klone nicht finden kann, ist vermutlich damit zu begründen, dass der CCAligner zur Vermeidung eines hohen Speicherverbrauchs eine Blockgröße von 6 und eine Editierdistanz von 1 für den Vergleich von überlappenden Blöcken verwendet. Beim Auftreten von mehreren Anweisungen, die nicht in beiden Funktionen über die gesamte Funktionslänge enthalten sind, führt dies zu wenig Gleichheit bei der Untersuchung der überlappenden Blöcke. StoneDetector, aber auch NiCad führen häufig Pfad- oder Textvergleiche über längere Strecken durch, sodass ein Auftreten nicht übereinstimmender Anweisungen durch die gewählte Metrik leichter kompensiert werden kann. Die zweite Art von Klonen, die ausschließlich von StoneDetector, nicht aber vom CCAligner gefunden wurden, waren stark unstrukturierte Subklone, die so unstrukturiert waren, dass das Auffinden von überlappenden Blöcken nicht erfolgreich war und daher solche Funktionen vom CCAligner nicht als Klonpaar erkannt werden konnten. Der StoneDetector hingegen beschreibt solche unstrukturierten Klonpaare durch einen strukturierten Satz von Grundblöcken, was zu einer leichteren Erkennung ähnlicher Funktionen führt. Bei der manuellen Untersuchung einer Zufallsstichprobe von 100 Klonpaaren, die nur von Stonedetec-

tor, nicht aber von CCAligner erkannt wurden, zeigt sich, dass 52 der gefundenen Codeklone Subklone von einfachen einzelnen Basisblöcken sind und 21 davon Klone von Funktionen, die starke unstrukturierte Klone bilden. 9 der von StoneDetector entdeckten Klonpaare wurden als keine Codeklone eingestuft. Insgesamt zeigen die durchgeführten Untersuchungen im Vergleich zum CCAligner, dass der StoneDetector, vergleichbar mit dem CCAligner, oft auch Large-Gap-Klone erkennen kann, jedoch trotz alledem und im Gegensatz zum CCAligner auch als kompletter allgemein verwendbarer Klondetektor eingesetzt werden kann.
Eine Untersuchung der Klone, die ausschließlich von CCAligner, nicht aber von StoneDetector gefunden wurden, ergab insbesondere zwei Arten von Klonen. Bei der ersten Art von Klonen handelt es sich um Subklone, die für einzelne Grundblöcke stehen, die sich in der Blocklänge stark unterscheiden. Der CCAligner kann solche Grundblöcke finden, weil er den kleineren Grundblock in überlappende Blöcke unterteilt und dann prüft, ob die so entstandenen Blöcke im größeren Grundblock zu finden sind. Wenn eine vordefinierte Anzahl ähnlicher Blöcke gefunden wird, werden beide Grundblöcke als Klon klassifiziert. StoneDetector kann in der aktuellen Version solche unterschiedlichen Grundblöcke nicht als Klonpaare klassifizieren. Der zweite Klontyp, den CCAligner erkennt, Stonedetector aber nicht, sind Funktionen, die gemeinsame Subklone enthalten. Wenn für die kleinere Funktion gilt, dass durch die Aufteilung in überlappende Blöcke die Funktion in Subklone aufgeteilt wird und CCAligner einen davon in der größeren Funktion findet, klassifiziert der CCAligner die Funktionen als Klonpaar. Solche Funktionen werden vom StoneDetector nicht als Klone identifiziert, da StoneDetector im Gegensatz zum CCAligner nicht blockorientiert, sondern auf Funktionsebene arbeitet. Die Analyse weiterer Klonpaare, die ausschließlich vom CCAligner gefunden wurden, zeigte zudem, dass der CCAligner - anders als der StoneDetector und die anderen Tools - Klone von Funktionen, die in internen Klassendefinitionen definiert sind, immer auch als Klone zu der Funktion betrachtet, die die interne Klassendefinition enthält. Diese Klassifizierung konnte tatsächlich bei 31 der 100 zufällig erzeugten Klonpaare beobachtet werden. Der StoneDetector ließe sich zwar leicht um eine solche Klassifizierung erweitern, jedoch ist die Zuordnung eines einzelnen Klonpaares zur internen Funktion ausreichend und genauer. Bei der manuellen Untersuchung einer Zufallsstichprobe von 100 Klonpaaren, die nur von CCAligner, nicht aber von StoneDetector erkannt wurden, zeigt sich wie bereits erwähnt, dass 21 der gefundenen Codeklone Subklone von Basisblöcken und 5 davon Klone von Funktionen sind, die gemeinsame Subklone enthalten. Neben 31 Klonen, die dadurch entstehen, dass der CCAligner Klone zu inneren Klassen

auch als Klone zur definierenden Funktion behandelt, wurden 10 der vom CCAligner entdeckten Klonpaare als keine Codeklone eingestuft.

iClones vs StoneDetector

Bei der Überprüfung von 100 Klonpaaren, die von iClones gefunden wurden, jedoch von StoneDetector nicht, zeigte sich, dass die gefundenen Klone vom Aussehen und Aufbau her dem ähneln, was auch exklusiv von CCAligner gefunden wurde. Aufgrund des gleichen Ansatzes, dass jeweils einzelne (Unter)-Blöcke miteinander verglichen werden, war es auffällig, dass ein Teil der Funktionen übereinstimmte, jedoch der Rest der Funktionen unterschiedlich war. 60 der 100 untersuchten Klone waren auf die Übereinstimmung kleinerer Teilfunktionen zurückzuführen. Bei weiteren 29 Funktionen war sehr auffällig, dass meist der untere Teil der Funktionen, der in diesen Fällen aus *catch-* und *finally-Blöcken* besteht, exakt übereinstimmt, jedoch der Rest der Funktionen unterschiedlich war. 11 der untersuchten Funktionspaare wurden als keine Klone identifiziert.
Bei den Funktionspaaren, die von StoneDetector, nicht jedoch von iClones entdeckt wurden, ergab sich erneut ein ähnliches Bild zu dem Vergleich mit CCAligner. Insgesamt konnten erneut die Funktionspaare in 2 Arten eingeteilt werden. 67 der untersuchten Funktionspaare waren Unterfunktionen, wobei die ähnlichen Instruktionen über die ganzen Funktionen verteilt waren. 18 weitere untersuchte Funktionspaare waren strukturelle Klone und 2 Funktionspaare konnten als keine Klone identifiziert werden.

Oreo vs StoneDetector

Im Gegensatz zu StoneDetector und den anderen Tools, die bisher näher betrachtet wurden, versucht Oreo Klone nicht durch den Vergleich von Text oder Token zu erkennen, sondern leitet Eigenschaften von Funktionen ab, um sie dann zum Vergleich zu verwenden. Die Eigenschaften, die Oreo ableitet, sind Informationen wie die Anzahl der Feld- und Methodenzugriffe, die Anzahl der Anweisungen, die Anzahl der Operationen und die Anzahl der Ausdrücke, um einige zu nennen. Insbesondere spielt beim Vergleich von Funktionen in Oreo die Länge von Pfaden oder Textbereichen keine Rolle. Oreo führt nach Ableitung der Eigenschaften einen direkten Vergleich der Funktionen durch. Wenn dies nicht zum Erfolg führt, dann werden die Vergleiche mit einem neuronalen Netz durchgeführt. Eine manuelle Überprüfung der

von Oreo gefundenen Klone, nicht aber der von StoneDetector gefundenen Klone, spiegelt die unterschiedliche Arbeitsweise der beiden Werkzeuge wider. Auch hier gibt es zwei Arten von Klonen, die von Oreo erkannt werden, von StoneDetector aber nicht. Dies sind zum einen - ähnlich wie beim CCAligner - Klonpaare, die für Funktionen stehen, die aus einzelnen, in der Länge sehr unterschiedlichen Basisblöcken bestehen. Weiterhin konnte beobachtet werden, dass von Oreo strukturelle Klone gefunden werden, die StoneDetector nicht erkennt, weil die Längen der Basisblöcke in den zu überprüfenden Funktionen zu unterschiedlich sind. StoneDetector erkennt solche strukturellen Klone wegen des Einflusses der unterschiedlichen Längen nicht. Oreo hingegen hat keine Probleme mit der Erkennung dieser Klone. Bei der manuellen Untersuchung einer Stichprobe von 100 Klonpaaren, die nur von Oreo, nicht aber von StoneDetector erkannt wurden, zeigt sich, dass es sich bei 48 der gefundenen Codeklone um Subklone von Basisblöcken, bei 14 Klonpaaren um Klone mit generell sehr unterschiedlichen Längen und bei 15 Klonpaaren um strukturelle Klone von Funktionen mit starken Unterschieden in der Länge ihrer Basisblöcke handelt. 2 der von Oreo entdeckten Klonpaare wurden als keine Codeklone eingestuft. Eine Untersuchung der von StoneDetector, aber nicht von Oreo gefundenen Klone zeigte, dass StoneDetector auch eine große Anzahl zusätzlicher Struktur- und Subklone entdeckt, die Oreo nicht findet. Besonders auffällig war, dass darunter eine große Anzahl struktureller Klone war, die sich nur durch das Vorhandensein zusätzlicher *if*-Anweisungen oder *for*- beziehungsweise *while*-Schleifen unterschieden. Der Grund dafür ist vermutlich, dass die Anzahl solcher Anweisungen von Oreo gezählt und zur Schätzung von Klonen verwendet wird. StoneDetector hingegen erkennt solche Klone, da bei der Erkennung von Klonen nur ein Vergleich der in den Funktionen enthaltenen Grundblöcke eine Rolle spielt. Bei der manuellen Untersuchung einer Stichprobe von 100 Klonpaaren, die nur von StoneDetector, nicht aber von Oreo erkannt wurden, zeigt sich, dass 49 der gefundenen Codeklone Subklone mit einer unterschiedlichen Anzahl von *if*-Anweisungen oder *for*- beziehungsweise *while*-Schleifen sind, 14 Klonpaare waren reine Subklone. 6 der von StoneDetector entdeckten Klonpaare wurden als keine Codeklone eingestuft.

5.4 Laufzeitverhalten

Eine wichtige Anforderung an ein Klonerkennungstool ist, dass es in der Lage sein muss, große Programmdatensätze zu verarbeiten. Um die Skalierbarkeit vom StoneDetector und den anderen Tools zu testen, wurde der gleiche Experimentenaufbau

	1-Millionen LOC	10-Millionen LOC	100-Millionen LOC
NiCad	57s	18,49min	2,45h
CCAligner	18s	6,58min	-
SourcererCC	4,17min	28,09min	46,52h
iClones	24s	7,26min	-
StoneDetector	24s	6,17min	7,13h
Oreo	2,42min	9,54min	7,10h

Tabelle 5.6: Skalierbarkeitsexperiment auf dem IJaDataset 2.0.

wie in der Arbeit von Sajnani et al. [131] verwendet. Es wurden drei Datensätze mit zufälligen Dateien aus dem *IJaDataset 2.0*[1] erstellt, die 1, 10 und 100-Millionen Codezeilen (LOC) umfassen. Anschließend wurden die Tools auf diese Datensätze angewendet und ihre Laufzeit ermittelt (siehe Tabelle 5.6). Deckard wurde dabei ausgelassen, da sein Speicherbedarf die verfügbaren Ressourcen übersteigt, wie auch in [131] berichtet wird. Eine Einschränkung von CCAligner ist, dass er nicht auf 100-Millionen Codezeilen skaliert [166]. Auch iClones skaliert nicht auf 100-Millionen Codezeilen, aber erreicht trotzdem eine bessere Skalierbarkeit als in dem Test von Sajnani et al. [131]. Im Gegensatz zu [131] war die aktuelle Version von NiCad v6.2, wie sie in dieser Arbeit verwendet wurde, in der Lage, auf 100-Millionen Codezeilen zu skalieren und zusätzlich noch das schnellste Tool. StoneDetector kann in seiner schnellsten Konfiguration gut mit den anderen Klondetektoren mithalten und evaluierte die 100-Millionen Codezeilen innerhalb von 7 Stunden und 13 Minuten. Damit ist StoneDetector etwa so schnell wie Oreo, aber langsamer als das schnellste Tool NiCad.
Aufgrund der guten Ergebnisse wurde ein weiterer Skalierungstest von StoneDetector mit über 2,5-Millionen Java-Quelldateien und über 328-Millionen Codezeilen, also auf dem kompletten *IJaDataset 2.0*, durchgeführt. StoneDetector konnte auch auf diesen sehr großen Datensatz skaliert werden und benötigte dafür 79 Stunden.

Ein noch nicht integrierter Optimierungsschritt wäre es, die *Exemplar-Strategie* [35, 126] von NiCad in abgewandelter Form zu integrieren. Die Grundidee der Strategie besteht darin, immer dann, wenn ein Klonpaar f und g gefunden wird, die weiteren Prüfungen nur für eine der Funktionen, d. h. f oder g, fortzusetzen. Angenommen, die Prüfungen werden für f fortgesetzt und für g beendet, dann werden alle Klone zu f am Ende der Überprüfung auch Klone zu g. So spart man sich die Überprüfung der Funktion g. Im Gegensatz zur NiCad-Implementierung werden in der StoneDetector-Variante der *Exemplar-Strategie* zwei Funktionen nur dann als Klonpaare betrachtet und die *Exemplar-Strategie* angewendet, wenn sie identisch

[1] https://github.com/clonebench/BigCloneBench

zueinander sind, d. h. wenn sie einen Distanzwert von Null haben. Klonpaare, die nur aufgrund des Schwellenwerts als Klone angesehen werden, aber einen anderen Abstandswert als Null haben, werden nicht bei dieser Strategie berücksichtigt.
Ein weiterer Optimierungsschritt wäre, alle Pfade aus der Beschreibungsmenge zu entfernen, für die es bereits einen ähnlichen Pfad gibt, wobei der Unterschied zwischen den Pfaden maximal ein Knoten sein darf. Dies würde zu einer erheblich verbesserten Laufzeit führen, da so weniger Pfade vorhanden wären, die miteinander verglichen werden müssten.

Kapitel 6

Dominatorbasierte Klonerkennung für nicht Sourcecode-basierte Darstellungen

Zur Überprüfung, ob das vorgestellte Dominatorbaum-basierte Klonerkennungsverfahren auch auf nicht textuelle Darstellungsformen angewendet werden kann, wird in diesem Abschnitt eine weitere Implementierung der allgemeinen Herangehensweise zur dominatorbasierten Codeklonerkennung aus Kapitel 4 vorgestellt. Dabei werden Codeklone gesucht, indem der Java-Bytecode als Stackcode genutzt wird. In einer weiteren Implementierung wird der Java-Bytecode in Registercode umgewandelt, um zu zeigen, dass das Verfahren auch für Registercode-basierten Code verwendet werden kann. Bevor die Implementierung und die Ergebnisse dieser Ansätze vorgestellt werden können, muss jedoch zunächst ein geeigneter Benchmark gefunden werden, mit dem man ein Klonerkennungsverfahren auf Basis von Java-Bytecode evaluieren kann.

6.1 Java-Bytecode-basierter Klonerkennungsbenchmark

Um die dominatorbasierte Codeklonerkennung auf Basis des Java-Bytecodes bewerten zu können, braucht es einen geeigneten Benchmark. Der BigCloneBench kann jedoch ohne weitere Anpassungen nicht genutzt werden, da bei der Erstellung des

BigCloneBench Java-Dateien aus vielen verschiedenen Open-Source-Projekten genutzt wurden. Diese Dateien wurden ohne die entsprechenden Abhängigkeiten aus den Projekten entnommen, weshalb sie nicht kompilierbar sind.
Im Folgenden werden die wenigen Java-Bytecode-basierten Klondetektoren und deren Vorgehen bei der Bewertung der Ergebnisse vorgestellt, um einen Überblick über die in der Literatur genutzten Benchmarks zu bekommen. Das am weitesten verbreitete Framework zur Evaluierung von Java-Bytecode-basierten Klondetektoren ist *Bellon's Benchmark* [21], für das sowohl der Java-Quellcode als auch der Java-Bytecode verfügbar ist. Zur Erstellung des Benchmarks validierten Bellon et al. manuell 2% aller Klone, die von sechs zeitgenössischen Klondetektoren aus der Zeit vor 2002 in acht Softwareprojekten, darunter vier Java-Projekten, gefunden wurden. Bei den vier Java-Projekten in Bellon's Benchmark wurden etwa 3,3-Tausend Klone von Bellon bewertet, wobei die Klone nicht aus ganzen Funktionen bestehen mussten. Murakami et al. [111] und Svajlenko et al. [148] argumentieren jedoch, dass Bellon's Benchmark veraltet und für die Bewertung moderner Klondetektoren nicht geeignet ist. Als Alternative werden andere öffentliche Softwareprojekte, die Java-Quellcode und -Bytecode zur Verfügung stellen, für die Bewertung verwendet. Dabei wird ein Java-Sourcecode-basiertes Klonerkennungsverfahren auf den Java-Sourcecode angewendet und die Ergebnisse mit dem Java-Bytecode-basierten Verfahren verglichen. Allerdings fehlen Informationen über die verschiedenen Klontypen und auch der Recall insgesamt kann nur mit erheblichem Aufwand ermittelt werden. Zum Beispiel wurde in Selim et al. [138] der Java-Quellcode in das Zwischenformat Jimple [157] von Soot [91, 158] übersetzt. Anschließend wurden Quellcode-Klondetektoren auf die Jimple-Codedarstellung angewendet, um Codeklone zu identifizieren, wobei Bellon's Benchmark für diese Evaluierung verwendet wurde. Im Gegensatz dazu wurde die Evaluierung des Bytecode-basierten Klondetektors SeByte [74–76] auf nur vier Java-Projekten aus verschiedenen Domänen durchgeführt. Yu et al. [179] präsentieren einen Klondetektor, der mit Sequenzen von Methodenaufrufen und Bytecode-Anweisungen arbeitet. Die Autoren evaluierten ihren Ansatz anhand des Bellon-Benchmarks und weiterer Open-Source-Java-Projekte. Ähnlich wie Selim et al. [138] verwendet das Tool SootDiff [40] das Jimple-Format von Soot und den Diff-Algorithmus von Myer, um Codeklone zu detektieren. Ein kleiner Satz von Java-Beispielprogrammen und der Bytecode, wie er von verschiedenen Java-Compilern generiert wird, wurde für diese Evaluierung verwendet [1]. In der Arbeit von Yu et al. [180] wird die Erkennung von Codeklonen auf Java-Bytecode unter Verwendung von Sequenzvergleichen für Methodenaufrufe und Befehlssequen-

zen implementiert. Die Evaluierung wurde wiederum mit dem Bellon-Benchmark und einer kleineren Anzahl von Open-Source-Java-Projekten durchgeführt. Bei der Evaluierung der maschinellen Lernmethode DeepSim [187] zur Erkennung von Codeklonen standen die Autoren ebenso vor dem Problem, dass die Abhängigkeiten der Java-Dateien im BigCloneBench nicht verfügbar waren. Sie generierten den Kontrollflussgraphen und den Datenflussgraphen aus dem Java-Quellcode mithilfe von Wala[1]. Dazu ersetzten sie alle externen Klassentypen durch einen eindeutigen Typ `java.lang.Object` und konnten so 95% aller Klonpaare in BigCloneBench erhalten. Das weitere Vorgehen wurde von den Autoren jedoch nicht beschrieben und auch nicht veröffentlicht. Auch auf eine Anfrage wurde keine Antwort gegeben.

6.2 Stubber

Es kann festgehalten werden, dass es zurzeit keinen geeigneten und modernen Benchmark für Java-Bytecode-basierte Klondetektoren gibt. Auch andere Autoren [180] haben für die Notwendigkeit repräsentativer Benchmarks zur Bewertung von Java-Bytecode-basierten Klondetektoren plädiert. Aus diesem Grund wurde ein Verfahren entwickelt, mit dessen Hilfe sich die Dateien des BigCloneBench kompilieren lassen, obwohl die Abhängigkeiten nicht vorhanden sind. Dieses Verfahren wurde in dem `Stubber-Tool`[2] implementiert. Mithilfe des Stubber-Tools können nun mit dem großen und modernen BigCloneBench auch Java-Bytecode-basierte Klondetektoren evaluiert und mit Java-Sourcecode-basierten Klondetektoren verglichen werden. Bevor das detaillierte Vorgehen und die Ergebnisse des Stubber-Tools vorgestellt werden, sollen verschiedene Ansätze zum Kompilieren der Dateien im BigCloneBench besprochen werden.

Suche nach fehlenden Abhängigkeiten

Eine Möglichkeit, Java-Quelldateien mit fehlenden Abhängigkeiten zu kompilieren, besteht darin, nach den fehlenden Abhängigkeiten in Software-Forges wie GitHub oder mavenCentral zu suchen. Zu diesem Zweck könnten Methoden des *Repository-Mining* verwendet werden, um eine skalierbare Suche durchzuführen, die z. B. auf referenzierten Java-Paketnamen basiert. Leider sind die voll qualifizierten Paketnamen in den Quelldateien des BigCloneBench nicht immer vorhanden. Weiterhin

[1] https://github.com/wala/WALA
[2] https://github.com/andre-schaefer-94/Stubber

könnte es von den zu suchenden Abhängigkeiten verschiedene Versionen geben, wobei nur eine spezielle Version zum Kompilieren kompatibel ist. Es gibt allerdings auch über diese Versionsnummern keine Informationen, was zu aufwändigen Tests führen würde.
Im BigCloneBench [149], haben alle Java-Quelldateien ihren Ursprung in Open-Source-Java-Projekten aus dem *IJaDataset 2.0* und ihr Projektname ist in der BigCloneBench Datenbank aufgeführt. Daher wäre es im Allgemeinen möglich gewesen, diese Projekte und ihre Abhängigkeiten zu suchen. Da jedoch insbesondere BigCloneBench mehrere Versionen desselben Projekts enthalten kann, wären verschiedene Versionen der Abhängigkeiten erforderlich gewesen. Außerdem erfordert die Abhängigkeitssuche und -speicherung einen beträchtlichen Zeitaufwand und viele Rechenressourcen. Aus diesen Gründen wurde sich gegen dieses Vorgehen entschieden.

Platzhalter und Typinferenz

Alternativ können für fehlende Abhängigkeiten Dummy-Klassen generiert werden, d. h. Platzhalter, welche die Signaturtypen bereitstellen, die in den gegebenen Java-Quelldateien aufgerufen wurden. Die Generierung der Dummy-Klassen erfordert die Signatur-Ableitung der referenzierten Typen, z. B. Parameter- und Rückgabetypen von aufgerufenen Methoden sowie Typen von zugegriffenen Feldern und vieles mehr entsprechend ihrer Verwendung in den Quelldateien. Dies ist eine anspruchsvolle Aufgabe und erfordert die Anwendung von (globaler) Typinferenz aufgrund der unvollständigen Klassenhierarchie und des unbekannten Verhaltens der fehlenden Abhängigkeiten. Dieser Ansatz wurde so weit implementiert, dass ca. 59% der Java-Quelldateien im BigCloneBench kompiliert werden konnten. Aufgrund des erforderlichen Aufwands wurde diese Herangehensweise jedoch aufgegeben und es wurde sich für eine einfachere Lösung entschieden.

Platzhalter und Typabstraktion

Der dritte alternative Ansatz, der auch im Stubber-Tool implementiert wurde, führt eine abstrakte Platzhalterklasse `StubClass` ein, die referenzierte Typen subsumiert. Anstatt die Signaturen der Typen entsprechend ihrer Verwendung in den gegebenen Quelldateien abzuleiten, wird dann `java.lang.Object` oder `StubClass` in den

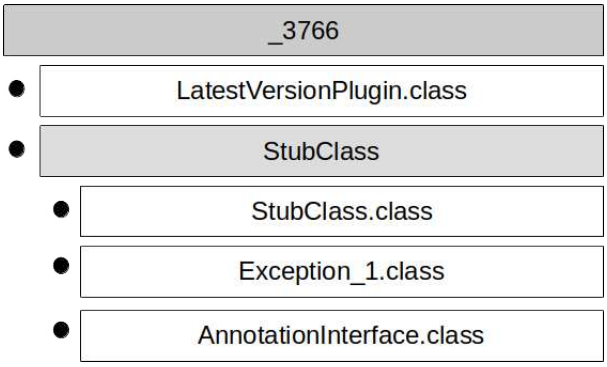


Abbildung 6.1: Ordnerstruktur und Jar-Dateien, die vom Stubber-Tool erzeugt werden.

Signaturen verwendet und die Quelldateien werden entsprechend angepasst. Die Einzelheiten dieses Ansatzes werden im nächsten Abschnitt beschrieben.

6.2.1 Funktionsweise

In diesem Abschnitt wird die generelle Vorgehensweise des Stubber-Tools vorgestellt. Dabei werden einige interessante Aspekte und Implementierungen kurz angesprochen.

6.2.1.1 Generelle Anpassungen

Es sind einige allgemeine Anpassungen an dem Quellcode notwendig, die im Folgenden vorgestellt werden. Im Prinzip erzeugt Stubber ein JAR-Archiv für eine gegebene Java-Quelldatei. Abbildung 6.1 zeigt die allgemeine Paketstruktur für eine Eingabedatei `3766.java` mit einer einzelnen Klassendefinition `LatestVersionPlugin` als Beispiel. Die grauen Kästen stehen für Ordner und die weißen Kästen für Klassendateien. Damit Klondetektoren und Analyse-Frameworks wie Soot zwischen den in den ursprünglichen Quelldateien definierten Klassen und den von Stubber erzeugten Artefakten unterscheiden können, befinden sich erstere im Hauptpaket bzw. -ordner und letztere im Unterpaket bzw. -ordner `StubClass`. Wie man sieht, wurde der Name des Top-Level-Pakets für das Beispiel angepasst, da Bezeichner in Java nicht mit einer Ziffer beginnen dürfen. Außerdem werden alle ursprünglichen Import-Anweisungen durch einen einzigen Import des Unterpakets `StubClass` in den Java-Quelldateien ersetzt.

Klassendefinitionen

Eine einzelne Quelldatei kann mehrere Klassendefinitionen enthalten, was zu mehreren Klassendateien im generierten JAR-Paket führt. Superklassen von Klassendefinitionen werden durch `StubClass` ersetzt, wobei alle Typen zusammengefasst werden. Der Einfachheit halber werden alle Schnittstellen aus der anzupassenden Java-Sourcecode-Datei entfernt. Außerdem werden Felder und Variablen, die in einer Klasse, aber nicht in einer Methode definiert sind, entfernt. Da Interesse an der Erkennung von Codeklonen auf Methodenebene besteht, werden diese Klassen- oder Instanzvariablen nicht benötigt. Aus demselben Grund werden auch generische Typargumente entfernt.

Methodensignatur

Ein Beispiel für eine Java-Methode und ihre von Stubber für die Kompilierung angepasste Version ist in Abbildung 6.2 dargestellt. Wie man sieht, enthält die angepasste Version eine neue Java-Annotation, die verwendet wird, um die Quellcode-Lokalisierung der Methode im kompilierten Bytecode zu erhalten. Die so erhaltenen Informationen können später verwendet werden, um in der Java-Bytecode-basierten-Version eines Klondetektors gefundene Codeklone mit Informationen über Quellcode-Klone zu verknüpfen. Aufgrund dieser Intention folgen die enthaltenen Informationen auch dem Format von BigCloneBench/BigCloneEval, d. h. Unterordner, Dateiname, Start- und Endzeile der Methode sind in der Annotation enthalten. Im Zuge der Integration dieser neuen Annotation werden alle anderen Annotationen entfernt. Für diese müssten sonst zusätzliche Annotationsschnittstellen erstellt werden, jedoch spielen Annotationen bei der Erkennung von Codeklonen oft keine Rolle. Bei einem Methodenrückgabetyp, der nicht `void` ist, und bei Methodenargumenttypen werden die Typen durch `java.lang.Object` oder `StubClass` ersetzt, je nach ihrer Verwendung. Dieser konservative Ansatz ist zwar nicht unbedingt erforderlich, wenn der Typ ein Standarddatentyp der Programmiersprache Java ist, aber er ist eine Voraussetzung, um unbekannte externe Typen zu ersetzen und ermöglicht außerdem eine einheitliche und einfache Transformation. Es können jedoch Probleme auftreten, wenn Methoden überladen werden. So würden beispielsweise zwei Methoden `Integer m(Integer p) {...}` und `String m(String p) {...}` auf dieselbe Signatur `Object m(Object)` abgebildet werden. Aus diesem Grund werden die Methodennamen umbenannt, z. B. `Objekt m1(Objekt)` und `Objekt m2(Objekt)`.

```
1
2
3 public void init() {
4   running_as_applet = true;
5   acontext = getAppletContext();
6   String s = getParameter("jorbis");
7   playlistfile = s;
8   s = getParameter("jorbis.player.icestats");
9   if (s != null && s.equals("yes")) {
10    icestats = true;
11  }
12  loadPlaylist();
13  initUI();
14  if (playlist.size() > 0) {
15    s = getParameter("jorbis");
16    if (s != null && s.equals("yes")) {
17      playonstartup = true;
18    }
19  }
20  setBackground(Color.lightGray);
21  getContentPane().setLayout(new BorderLayout());
22  getContentPane().add(panel);
23 }
```

```
1 @AnnotationInterface(SubFolder="default",
2 FileName="3566.java",StartLine=81,EndLine=101)
3 public void init_localFunction1() {
4   running_as_applet_2 = true;
5   acontext_2 = getAppletContext_1();
6   Object s_1 = getParameter_1("jorbis");
7   playlistfile_2 = s_2;
8   s_3 = getParameter_2("jorbis.player.icestats");
9   if (s_4 != null && s_5.equals_1("yes")) {
10    icestats_2 = true;
11  }
12  loadPlaylist_1();
13  initUI_1();
14  if (playlist_1.size_1() > 0) {
15    s_6 = getParameter_3("jorbis");
16    if (s_7 != null && s_8.equals_2("yes")) {
17      playonstartup_2 = true;
18    }
19  }
20  setBackground_1(Color_1.lightGray_2);
21  getContentPane_1().setLayout_1(new StubClass());
22  getContentPane_2().add_1(panel_1);
23 }
```

Abbildung 6.2: Originale Java-Methode und durch Stubber angepasste Methode zur Kompilierung.

Methodenkörper

Wie bereits erwähnt, werden Rückgabe- und Parametertypen von Methoden durch `java.lang.Object` bzw. `StubClass` ersetzt. Um typisierbare und kompilierbare Klassen zu erhalten, müssen auch die Typen der lokalen Variablen angepasst werden, wie in Zeile 6 in Abbildung 6.2 zu sehen ist, wo `String` aufgrund der Zuweisung des Rückgabewerts der Methode durch `java.lang.Object` ersetzt wurde. Während bei diesem Ansatz sicherlich Informationen über die verwendeten Typen verloren gehen, abstrahieren viele Klondetektoren ohnehin von diesen Informationen, um bessere Ergebnisse bei Codeklonen des Typs-2 und höher zu erzielen [36]. Um die Transformation weiter zu vereinfachen, wird ein neuer Variablenname für jedes Auftreten von Variablen eingeführt. So tritt beispielsweise die Variable `s` von der linken Seite von Abbildung 6.2 in Form der verschiedenen Variablen `s_1`,...,`s_8` auf der rechten Seite von Abbildung 6.2 auf, woraus folgt, dass keine aufwendige Typinferenz erforderlich ist. Stattdessen kann der Typ jeder Variable, d. h. `java.lang.Object` oder `StubClass`, anhand des lokalen Kontexts der Anweisung oder des Ausdrucks, in dem die Variable verwendet wird, bestimmt werden. So wird der Typ der Variablen `s_1` auf `java.lang.Object` gesetzt, während der Typ der Variablen `s_5`, die den Aufrufer, für den Methodenaufruf `s_5.equals(...)` in Zeile 9 des Beispiels ist, auf `StubClass` gesetzt wird. Die Deklarationen der zusätzlich eingeführten Methoden und Variablen, z. B. `s_5`, werden zu `StubClass` hinzugefügt, die durch die Definition von `StubClass` als Oberklasse sichtbar sind. Da alle Java-Dateien mit der gleichen Vorgehensweise im Stubber-Tool angepasst werden, sind die Änderungen dementsprechend auch in allen generierten Java-Dateien und dem generierten Java-

Bytecode konsistent.
Eine ähnliche Strategie wird für die Ableitung von Methodenrückgabe- und Argumenttypen verwendet, d. h. für die Entscheidung, ob `java.lang.Object` oder `StubClass` gewählt werden soll. Dazu gehört auch die Einführung mehrerer Methodendefinitionen derselben Methode in der StubClass-Datei, also eine neue Methode in StubClass für jede Aufrufstelle von `getParameter` (vgl. `getParameter_1`, ...,`getParameter_3` in den Zeilen 6, 8, 15 von Abbildung 6.2). Für den Spezialfall der Objektinitialisierung (vgl. Zeile 21) werden Konstruktoren durch Konstruktoren von `StubClass` mit entsprechender Anzahl von Argumenten vom Typ `java.lang.Object` ersetzt. Ein subtilerer Ansatz ist jedoch erforderlich, wenn *boolesche* oder *arithmetische Ausdrücke* vorhanden sind. In solchen Fällen werden statt der Typen `java.lang.Object` oder `StubClass` die entsprechenden primitiven Typen abgeleitet und für die Typisierung von Variablen und Methoden verwendet, wobei wiederum der lokale Kontext berücksichtigt wird. So wird beispielsweise in Zeile 14 in Abbildung 6.2 der Rückgabewert der Methode `size_1` mit 0 verglichen, was den Rückgabetyp der Methode auf `int` festlegt. Wie in den Zeilen 4, 6 und 14 in Abbildung 6.2 zu sehen ist, werden Konstanten und Literale beibehalten und unverändert gelassen. Die Struktur des Kontrollflusses wird in der Transformation beibehalten, und Typ-Casts werden aufgrund der unvollständigen Typinformationen entfernt.

Ausnahmebehandlungen

Die Behandlung von Ausnahmen erfordert eine genauere Betrachtung. Es ist nicht möglich, alle Ausnahmen auf eine einzige Klasse abzubilden. Wenn zum Beispiel mehrere Ausnahmen in einem try-catch-Block abgefangen werden, müssen diese von unterschiedlichen Typen sein. Folglich wird bei der Transformation für jede in den Java-Quelldateien auftretende Ausnahme eine eigene Klasse erstellt. Der Einfachheit halber erben die eingeführten Ausnahmetypen von `java.lang.RuntimeException`, d. h. sie sind als *unchecked* Ausnahmen definiert. Würde man stattdessen *checked* Ausnahmen verwenden, müsste man prüfen, ob jeder try-catch-Block, in dem eine solche Ausnahme abgefangen wird, mindestens eine Anweisung enthält, die die Ausnahme auslösen kann, da sonst die Kompilierung fehlschlägt.

```
RuntimeInvisibleAnnotations:
    _3566.StubClass.AnnotationInterface(
      SubFolder="default"
      FileName="3566.java"
      StartLine=81
      EndLine=101
    )
```

Abbildung 6.3: Disassemblierte Annotation mit den Informationen.

6.2.1.2 BigCloneBench/BigCloneEval Integration

In diesem Abschnitt wird die Verwendung des Tools Stubber zur Erkennung von Codeklonen für Java-Bytecode mit BigCloneEval und BigCloneBench vorgestellt. Das Tool Stubber wurde für eine einfache Integration mit BigCloneEval entwickelt. Um das Tool zu verwenden, muss zunächst BigCloneEval installiert und eingerichtet werden, wie in der entsprechenden Anleitung beschrieben.[3] Die Teilmenge des *IJaDataset 2.0*, wie sie in BigCloneEval verwendet wird, ist auch die Eingabe für das Stubber-Tool. Das Tool kompiliert über 95% aller Java-Quelldateien, die im BigCloneBench enthalten sind, zu JAR-Dateien und behält die BigCloneBench-Ordnerstruktur bei, die aus 43 Ordnern und den Unterordnern `sample`, `default` und `selected` besteht. Dateien, die nicht kompiliert werden können, werden zur manuellen Überprüfung und Verarbeitung protokolliert. Es können nicht alle Java-Quelltext-Dateien kompiliert werden, da das verwendete Framework zur Quelltext-Anpassung nicht alle neuen Java-Features verarbeiten kann. Kompilierte Dateien werden in den Ordner von BigCloneEval abgelegt, wo BigCloneEval die Dateien erwartet und in dem nach Klonen gesucht werden soll. Das Tool Stubber enthält auch Skripte zum Anpassen der Benchmark-Datenbank von BigCloneEval, z. B. zum Entfernen von Codeklonpaaren, bei denen mindestens eine Methode aus einer Datei stammt, die nicht zu Java-Bytecode kompiliert werden konnte. Anschließend kann das BigCloneEval-Framework wie gewohnt verwendet werden, um Java-Bytecode-basierte Klondetektoren zu evaluieren.
Methoden in Klonpaaren werden von BigCloneEval anhand des Unterordners und der Java-Quelldatei, in der sie sich befinden, sowie anhand ihrer jeweiligen Start- und Endzeile in der Quelldatei identifiziert. Offensichtlich können die Zeilennummern der vom Stubber-Tool erzeugten Dateien nicht für diesen Zweck verwendet werden. Stattdessen bietet das Tool eine Möglichkeit, die Start- bzw. Endzeilen der generierten Java-Bytecode-Methoden auf die ursprünglichen Java-Quellcode-Methoden

[3] https://github.com/jeffsvajlenko/BigCloneEval

zurückzuführen.
Wie in Abbildung 6.2 zu sehen ist, erstellt das Tool Stubber Annotationen für die Methoden während der Kompilierung zu Java-Bytecode, die die von BigCloneEval geforderten Lage-Informationen der Methoden enthalten. Darüber hinaus können die Annotationen verwendet werden, um zwischen Methoden zu unterscheiden, die in den ursprünglichen Java-Quelldateien enthalten waren, und Methoden, die künstlich durch das Tool Stubber eingeführt wurden. Die Definition der verwendeten Annotationsschnittstelle ist Teil des `StubClass` Unterpakets, das in den generierten JAR-Dateien enthalten ist (vgl. Abbildung 6.1). Die Annotation ist im Java-Bytecode sichtbar und es kann auf die Informationen zugegriffen werden. Zum Beispiel zeigt Abbildung 6.3 den Ausschnitt der Ausgabe, die die Annotation enthält, wenn der Disassembler *javap* auf den Java-Bytecode des Beispiels aus Abbildung 6.2 zugreift. Zur bequemen Verwendung, enthält das Stubber-Tool ein Skript, das die Methoden im Java-Bytecode zu den entsprechenden Originalmethoden im Java-Quellcode zuordnet. Zu diesem Zweck analysiert das Skript die Annotationen in den generierten Bytecode-Klassendateien und erzeugt eine CSV-Datei `ByteCodeInfo.csv`, in der für jede Methode der Klassendatei die Informationen über den Ort des Quellcodes gespeichert ist. Zusätzlich enthält die CSV-Datei auch Informationen über den Methodennamen und die Referenz der Methode im Konstantenpool des Bytecodes. Das Skript kann auch als Blaupause für den Zugriff auf die Annotationen verwendet werden.

6.2.2 Evaluation

Mithilfe des Stubber-Tools können von den 55.499 Java-Quelldateien in BigCloneBench 52.816 erfolgreich zu Java-Bytecode kompiliert werden, was mehr als 95% des

	NiCad			
	Originale Java-Dateien		Java-Dateien nach Stubber	
Typ-1	24.271	99,7%	24.288	99,8%
Typ-2	3.660	99,6%	3.678	99,9%
Very-Strongly Typ-3	3.897	98,8%	3.947	99,3%
Strongly Typ-3	7.672	75,0%	8.225	80,1%
Moderatly Typ-3	301	0,4%	826	1,1%
Typ-4	11	0,0%	30	0,0%
Klone insgesamt	1.240.857		1.279.866	

Tabelle 6.1: NiCads Recall-Werte auf den originalen und den durch das Stubber-Tool angepassten BigCloneBench-Dateien $\geq$ 10 Zeilen.

	Klone	Typ-1		Typ-2		Very-Strongly Typ-3		Strongly Typ-3		Moderately Typ-3		Typ-4	
BCB	8.584.153	48.116	0,56%	4.234	0,04%	4.763	0,05%	17.203	0,20%	87.480	1,01%	8.422,357	98,11%
BCB $\geq$ 15 Zeilen	4.560.492	20.858	0,45%	3.236	0,07%	3.672	0,08%	7.560	0,16%	27.393	0,60%	4.497.773	98,62%
Stubbed BCB	7.945.129	47.999	0,60%	4.134	0,05%	4.615	0,05%	16.505	0,20%	79.755	1,00%	7.792.121	98,07 %
Stubbed BCB $\geq$ 15 Zeilen	4.203.486	20.807	0,49%	3.230	0,07%	3.621	0,08%	7.461	0,17%	26.084	0,62%	4.142.283	98,54%

Tabelle 6.2: Zusammensetzung des BigCloneBench vor und nach der Anwendung des Stubber-Tools.

Benchmarks ausmacht. Die kompilierten Dateien decken 92,5% oder mehr als 7,9-Millionen der über 8,5-Millionen bekannten Klonpaare im BigCloneBench ab. Von den insgesamt 279-Tausend falsch positiven Beispielen in BigCloneBench konnten 261-Tausend Paare erhalten werden, was über 93% entspricht. Das Stubber-Tool benötigte etwa 10,5 Stunden auf einem Laptop mit einem Intel i7 1.8GHz und 8 Kernen CPU mit 16GB RAM.

Die Zusammensetzung des BigCloneBench aller Klone und nur der Klone, die $\geq$ 15 Zeilen aufweisen, kann vor und nach der Anwendung des Stubber-Tools in Tabelle 6.2 eingesehen werden. Es zeigt sich, dass die Anwendung des Stubber-Tools praktisch keinen Einfluss auf die Klonverteilung im Benchmark hat.

Um zu untersuchen, was für einen Einfluss die Anpassungen an dem Java-Sourcecode auf die Codeklonerkennung haben, wurde der Sourcecode-basierte Klondetector NiCad [36] verwendet und zur Klondetektion auf den originalen Soucecode sowie auf den durch das Stubber-Tool angepassten Sourcecode angewendet. NiCad wurde mit den Optionen *blind-renaming*, *literal-abstraction*, 0,3 Schwellenwert und einer Mindestgröße der Codeklone von 10 Zeilen verwendet. Um einen validen Vergleich zu ermöglichen, haben wir jedoch Klone, die zu Dateien gehören, die vom Stubber-Tool nicht kompiliert werden konnten, aus der Bewertung herausgenommen.

Die von BigCloneEval gemessenen Recall-Werte für beide Datensätze von Java-Quelldateien sind in Tabelle 6.1 dargestellt. Während die Zahlen für alle Klontypen recht ähnlich sind, können

	StoneDetector (Sourcecode)			
	Normal		Split	
Typ-1	20.696	99,5%	20.751	99,7%
Typ-2	3.466	99,8%	3.463	99,7%
Very-Strongly Typ-3	3.217	99,1%	3.215	99,0%
Strongly Typ-3	6.608	90,5%	6.648	91,0%
Moderatly Typ-3	3.968	15,5%	4.816	18,8%
Typ-4	2.716	0,0%	4.286	0,1%
Klone insgesamt	621.169		911.352	
Zeit auf dem BCB	16,26min		18,27min	

Tabelle 6.3: Recall-Werte des Sourcecode-basierten StoneDetector-Ansatzes auf den durch das Stubber-Tool angepassten Sourcecode $\geq$ 15 Zeilen.

wir feststellen, dass NiCad bei den transformierten Java-Quelldateien etwas bessere Werte erreicht. Dies ist teilweise auf die Tatsache zurückzuführen, dass in den transformierten Dateien Typ-Casts entfernt werden. Da außerdem alle Typen auf `java.lang.Object` und `StubClass` abgebildet werden, gehen generische Typargumente in den transformierten Dateien verloren, die jedoch von NiCad für den Quellcodevergleich verwendet werden. Man beachte, dass NiCad ansonsten auch von identifizierten Namen, Literalen und Typen durch seine Strategie *blind-renaming* und der Abstraktion von Literalen abstrahiert.
Der Sourcecode-basierte Ansatz von StoneDetector aus Abschnitt 5 erzielt auf dem durch Stubber angepassten Sourcecode annähernd die gleichen Ergebnisse, wie auf dem originalen Sourcecode, wie man in Tabelle 6.3 sehen kann. Dabei wurden alle Funktionen genutzt, die mindestens 15 Zeilen lang sind. Die gemessenen Zeiten geben an, wie lange StoneDetector auf dem BigCloneBench (BCB) mithilfe des BigCloneEval-Tools gerechnet hat.
Zusammenfassend lässt sich sagen, dass die Quellcode-Transformation keine starke Verzerrung für die Bewertung von Klondetektoren auf BigCloneBench darstellt. Es ist jedoch zu empfehlen, die transformierten Java-Quelldateien für den Vergleich zu verwenden, wenn Quellcode- und Bytecode-basierte Klondetektoren gemischt evaluiert werden.

6.3 Java-Bytecode-basierte Klonerkennungsansätze

Mit dem Stubber-Tool steht ein moderner und großer Klonerkennungsbenchmark zur Verfügung, der auch zur Evaluierung von Java-Bytecode-basierten Klondetektoren genutzt werden kann. In den folgenden Abschnitten sollen deshalb zwei weitere

```
1 int func(int[] arr) {
2     int s = 0;
3     for (int i = 0; i<arr.length; i++)
4         if (arr[i]>0)
5             s *= (arr[i] * fu());
6     return s;
7 }
```

(a) Sourcecode

```
1  push 0
2  store.i i5
3  push 0
4  store.i i6
5  load.i i6
6  load.r r0
7  arraylength
8  ifcmpge.i goto 24
9  load.r r0
10 load.i i6
11 arrayread.i
12 ifle goto 22
13 load.i i5
14 load.r r0
15 load.i i6
16 arrayread.i
17 load.r r1
18 virtualinvoke fu()
19 mul.i
20 mul.i
21 store.i i5
22 inc.i i6 1
23 goto 5
24 load.i i5
25 return.i
```

(b) Stackcode

```
1  i5 = 0
2  i6 = 0
3  $i0 = lengthof r0
4  if i6 >= $i0 goto 13
5  $i1 = r0[i6]
6  if $i1 <= 0 goto 11
7  $i2 = r0[i6]
8  $i3 = virtualinvoke fu()
9  $i4 = $i2 * $i3
10 i5 = i5 * $i4
11 i6 = i6 + 1
12 goto 3
13 return i5
```

(c) Registercode

Abbildung 6.4: Funktion aus Abbildung 2.5 mit dem entsprechenden Stackcode und dem Registercode.

Instanziierungen des allgemeinen Dominator-basierten Klonerkennungsansatzes aus Kapitel 4 vorgestellt und evaluiert werden, zunächst ein Stackcode-basierter Ansatz und danach ein Registercode-basierter Ansatz.

Die Analyse des Java-Bytecodes sowie die Transformation desselben wird mithilfe des Soot-Frameworks [91, 158] erreicht. Soot ist ein Framework zur Java-Bytecode-

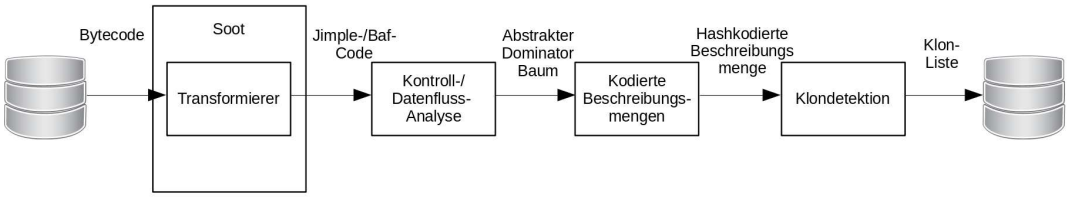


Abbildung 6.5: Allgemeiner Aufbau des Systems zur Codeklonerkennung auf Basis von Java-Bytecode.

Manipulation und -Optimierung und besitzt dazu insgesamt vier verschiedene Code-Repräsentationen. Der Stackcode-basierte Ansatz nutzt die *Baf*-Repräsentation, die ungefähr dem Java-Bytecode entspricht und somit über 200 verschiedene Operationen besitzt. Bei dem Registercode-basierten Ansatz wird die *Jimple*-Repräsentation [157] von Soot genutzt, die ein *Drei-Adressen-Code* ist. Es wurde sich für die *Baf*-Repräsentation entschieden, da diese dem tatsächlichen Java-Bytecode sehr nahe ist. Des Weiteren wurde sich für den *Jimple*-Code entschieden, da dieser sich gut für Optimierungen eignet und somit durch entsprechende Optimierungsschritte bessere Ergebnisse auf dem BigCloneBench erzielt werden können. In Abbildung 6.4 (a) ist die Beispielsfunktion aus der Abbildung 2.5 mit dem entsprechenden Stackcode sowie dem Registercode abgebildet. Aus diesen lassen sich Kontrollflussgraphen und daraus auch Dominatorbäume bilden, aus denen dann Pfade und Pfadmengen abgeleitet werden können. Es lässt sich in der Abbildung 6.4 (b) gut erkennen, dass der Stackcode sehr viel mehr Instruktionen enthält als beispielsweise der Registercode in Abbildung 6.4 (c). Der Grund dafür ist, dass jede Instruktion des Sourcecodes in mehrere Stackcode-Instruktionen aufgeteilt wird.

Im Soot-System werden zur Erzeugung des Registercodes dem Programm zwei Arten von Registern zugewiesen:

- alle im Bytecode enthaltenen lokalen Variablen werden auf virtuelle Register abgebildet
- alle durch Bytecode-Instruktionen auf dem Operandenstapel abgelegte Werte werden implizit in temporären Registern gespeichert

Die Registercode-Generierung wird dann durch eine abstrakte Interpretation realisiert, bei der auf dem Stack nicht die Werte selbst, sondern die Register, in denen sich die Werte befinden, oder Operandenkonstanten abgelegt werden. Dabei werden bei der Ausführung der abstrakten Befehle zeitgleich die Registerinstruktionen

Einstellungen	
Stackcode	Registercode
Variablengleichsetzung	
Konstantengleichsetzung	
Nutzung von Funktionsnamen	
16 Threads	
δ_{LCS}-Metrik	
4-Byte-Hashing Kodierung	
ganze / gesplittete Pfade	
15% Schwellenwert	30% Schwellenwert
Mindestgröße 15 Zeilen	
30% Beschreibungsmengenunterschied	
Pfadentfernung 30%	Pfadentfernung 40%
Pfadunterschied 30%	

Tabelle 6.4: Genutzte Einstellungen der Bytecode-basierten Ansätze.

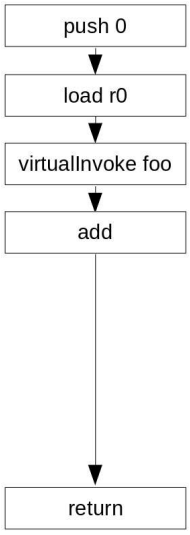
erzeugt, wobei die Operanden jeweils vom aktuellen Stack genommen werden. Temporäre Register kann man in der Abbildung 6.4 (c) im Registercode in den Zeilen 3, 5, 7, 8 und 9 erkennen. Die temporären Register beginnen dabei immer mit dem $-Zeichen.

In Abbildung 6.5 kann der grundsätzliche Ablauf der Ansätze eingesehen werden. Der Bytecode wird durch Soot geladen und in die jeweilige interne Repräsentation durch einen Transformierer umgewandelt. Mithilfe dieses internen Formates können dann die Kontrollflussgraphen gebildet werden. Der restliche Ansatz entspricht dem, was bereits im Kapitel 4 vorgestellt wurde.

Alle in den nächsten Abschnitten vorgestellten Messungen haben die in der Tabelle 6.4 aufgelisteten Einstellungen verwendet. Dabei wurden beispielsweise durch die *Variablengleichsetzung*, die verschiedenen Identifizierer aller Variablen auf einen einheitlichen Namen abgebildet. Nutzung von Funktionsnamen beschreibt, dass die Namen von aufgerufenen Funktionen mit kodiert und somit beim Vergleich zweier Pfade auch mit genutzt werden. Der Beschreibungsmengenunterschied gibt an, dass nur Beschreibungsmengen und damit Funktionen verglichen werden, die sich um nicht mehr als 30% in der Anzahl der Pfade unterscheiden. Der Pfadunterschied gibt an, dass nur Pfade auf Gleichheit geprüft werden, wenn diese sich um nicht mehr als 30% beziehungsweise 40% voneinander unterscheiden. Die Pfadentfernung beschreibt, dass alle Pfade, die kleiner als 30% beziehungsweise 40% im Vergleich zum längsten Pfad in der Beschreibungsmenge sind, aus der Beschreibungsmenge entfernt werden.

```
1public int function1(){
2    int i=0;
3    i=i+foo();
4
5    return i;
6}
```

```
1public int function2(){
2    int i=0;
3    i=i+foo();
4    i=i+bar();
5    return i;
6}
```



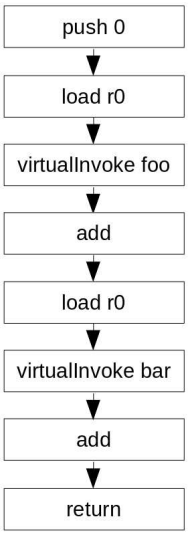


Abbildung 6.6: Beispielfunktionen und Dominatorbäume mit Stackcode-Instruktionen.

6.3.1 Stackcode-basierte Klonerkennung

Der in diesem Absatz vorgestellte Ansatz nutzt als Grundlage zur Klonerkennung einen aus dem Java-Bytecode abgeleiteten Stackcode. Das Vorgehen bei der Erstellung des Dominatorbaums, des abstrakten Dominatorbaums, der Extraktion der Pfade und die Erstellung der Beschreibungsmengen wurde auf die gleiche Art implementiert, wie es in den Kapiteln 4 und 5 erläutert wurde. Auch der Vergleich der kodierten Pfadmengen wurde beibehalten.
In Abbildung 6.6 sind zwei Funktionen und die dazugehörigen Dominatorbäume mit vereinfachten Stackcode-Instruktionen zu sehen. Da in den beiden Funktionen keine Instruktionen enthalten sind, die den Kontrollfluss aufteilen, gibt es auch keine Knoten im Dominatorbaum, die mehr als einen Nachfolger haben. Wenn nun zunächst nur die beiden Funktionen betrachtet werden, lässt sich ziemlich leicht ermitteln, dass fünf von sechs Zeilen übereinstimmen. Betrachtet man hingegen die Stackcode-basierten Dominatorbäume, die durch das Fehlen von kontrollflussaufteilenden Instruktionen auch gleichzeitig die jeweiligen Pfade und Pfadmengen für die entsprechenden Funktionen sind, dann sieht man, dass nur fünf von acht Knoten

	StoneDetector (Stackcode) mit ganzen Pfaden							
	Normal		EäP		EvSI		EäP + EvSI	
Typ-1	20.792	99,9%	20.792	99,9%	20.792	99,9%	20.792	99,9%
Typ-2	3.467	99,9%	3.467	99,9%	3.461	99,7%	3.463	99,7%
Very-Strongly Typ-3	3.112	95,9%	3.140	96,7%	3.112	95,9%	3.157	97,2%
Strongly Typ-3	4.888	66,9%	5.272	72,2%	4.868	66,6%	5.035	68,9%
Moderatly Typ-3	2.354	9,2%	3.327	13,0%	2.266	8,8%	2.804	10,9%
Typ-4	746	0,0%	1.642	0,0%	1.787	0,0%	3.178	0,0%
Klone insgesamt	427.090		503.218		397.202		439.767	
Zeit auf dem BCB	558,15min		886,10min		54,52min		77,20min	
	StoneDetector (Stackcode) mit gesplitteten Pfaden							
	Normal		EäP		EvSI		EäP + EvSI	
Typ-1	20.792	99,9%	20.792	99,9%	20.792	99,9%	20.792	99,9%
Typ-2	3.467	99,9%	3.467	99,9%	3.463	99,7%	3.466	99,8%
Very-Strongly Typ-3	3.060	94,2%	3.131	96,4%	3.039	93,6%	3.098	95,4%
Strongly Typ-3	4.665	63,9%	4.966	68,0%	4.704	64,4%	5.047	69,1%
Moderatly Typ-3	3.088	12,1%	4.111	16,1%	3.468	13,5%	4.184	16,4%
Typ-4	4.836	0,1%	7.445	0,1%	5.796	0,1%	7.774	0,2%
Klone insgesamt	457.284		565.277		508.486		582.397	
Zeit auf dem BCB	75,45min		90,47min		20,38min		21,29min	

Tabelle 6.5: Recall-Werte des Stackcode-basierten StoneDetector-Ansatzes für ≥ 15 Zeilen.

übereinstimmen. Stimmt wie in diesem Beispiel nur eine Zeile im Sourcecode nicht überein, führt das zu einem größeren syntaktischen Unterschied zwischen den Pfaden im Stackcode, da eine Zeile im Sourcecode mehrere Instruktionen im Stackcode erzeugen kann. Diese Tatsache spielt bei der Erkennung von Typ-1- und Typ-2-Klonen keine Rolle, da die Instruktionen im Sourcecode zweier Funktionen gleich sein müssen, wenn sie einen Typ-1- oder Typ-2-Klon bilden. Bei Typ-3- und Typ-4-Klonen hingegen führt der Umstand, dass beim Fehlen von Sourcecode-Instruktionen in einer Funktion der Stackcode eine größere syntaktische Differenz hat, dazu, dass es mithilfe des Stackcodes schwieriger wird, die entsprechenden Klone zu identifizieren.

Mithilfe des Stubber-Tools konnten die Quellcode-Dateien des BigCloneBench zu großen Teilen kompiliert werden. Dadurch ist es nun möglich, den hier vorgestellten Stackcode-basierten Klonerkennungsansatz auf dem BigCloneBench mithilfe des BigCloneEval auszuwerten. Die Tabelle 6.5 zeigt verschiedene Messwerte. In der oberen Hälfte der Tabelle sind die Stackcode-basierten Messergebnisse aufgezeigt, bei denen ganze Pfade verwendet wurden. In der unteren Hälfte der Tabelle wurden zur Ermittelung der Werte gesplittete Pfade genutzt. Bei jeder Messung sind die Recall-Werte der verschiedenen Klontypen, die insgesamt gefundenen Klone und die zur Messung benötigte Zeit auf dem BigCloneBench abgebildet.

Bei den Messungen, über denen `Normal` abgebildet ist, wurden bei den Pfadvergleichen alle Pfade mit allen enthaltenen Instruktionen genutzt. Bei den Messungen, bei denen `Eliminierung ähnlicher Pfade (EäP)` die Überschrift ist, wurden in einer Beschreibungsmenge alle Pfade bis auf einen entfernt, die sich um nicht mehr als zwei Knoten voneinander unterscheiden. Das hat zur Folge, dass die Beschreibungsmengen kleiner werden. Der Parameter Beschreibungsmengenunterschied aus Tabelle 6.4 gibt an, um wie viel Prozent sich Beschreibungsmengen in ihrer Anzahl an Pfaden unterscheiden dürfen, damit sie verglichen werden. Durch das Entfernen von Pfaden aus Beschreibungsmengen werden die Beschreibungsmengen insgesamt kleiner und es zeigte sich, dass mehr Beschreibungsmengen und damit Funktionen miteinander verglichen werden. Das führt dazu, dass sich die Recall-Werte leicht verbessern, die insgesamte Anzahl an Klonen leicht steigt und die benötigte Zeit erheblich ansteigt. Bei den Messungen, bei denen `Eliminierung von Stackcode-Instruktonen (EvSI)` darüber abgebildet ist, werden die Anpassungen von den `EäP` Messungen nicht übernommen. Stattdessen wurden einige Instruktionen, wie beispielsweise *new*-, *Feldzugriffs*-, *Konstantenzugriffs*-, *Load*- und *Store*-Anweisungen aus allen Pfaden herausgenommen. Das führt zu einer Beschleunigung der Vergleiche zwischen Beschreibungsmengen, da nun die zu vergleichenden Pfade deutlich kleiner sind. Die Recall-Werte sind allerdings leicht gefallen. Bei den Messungen, bei denen `EäP + EvSI` darüber abgebildet ist, wurden die Anpassungen aus `EäP` und `EvSI` gemeinsam angewendet. Durch die Kombination aus den verbesserten Recall-Werten von EäP und der schnellen Laufzeit von der `EvSI`-Messung konnte so ein insgesamt besseres Ergebnis erreicht werden, was den besten Kompromiss aus hoher Laufzeit mit guten Recall-Werten und geringer Laufzeit mit niedrigeren Recall-Werten bildet.

Interessant ist jedoch, dass diese Ergebnisse mit einem Schwellenwert von 15% erreicht wurden, wie in Tabelle 6.4 zu sehen ist. Dieser niedrige Schwellenwert musste genutzt werden, da ansonsten die insgesamte Anzahl an potentiell gefundenen Klonen sehr groß gewesen wäre. Insbesondere Funktionspaare, die einen Typ-3-Klon bilden, unterscheiden sich dadurch, dass eine Instruktion in einer der Funktionen verändert wurde. Stellt man sich eine ähnliche Funktion zu der aus Abbildung 6.4 vor, bei der jedoch das `int s = 0;` in Zeile 2 mit einem `int s = 1;` ersetzt worden wäre, dann würde sich im Stackcode nur die Zeile 1 ändern. Die Zeile 2 mit `store.i i5`, die auch aus der genannten Sourcecode-Zeile gebildet wird, bleibt unverändert. Bei einer Veränderung einer Instruktion zwischen zwei Funktionen ist also wider Erwartend die syntaktische Differenz sehr gering, weshalb ein niedriger Schwellenwert

```
1public int function1(){
2     int i=0;
3     i=i+foo();
4
5     return i;
6}
```

```
1public int function2(){
2     int i=0;
3     i=i+foo();
4     i=i+bar();
5     return i;
6}
```

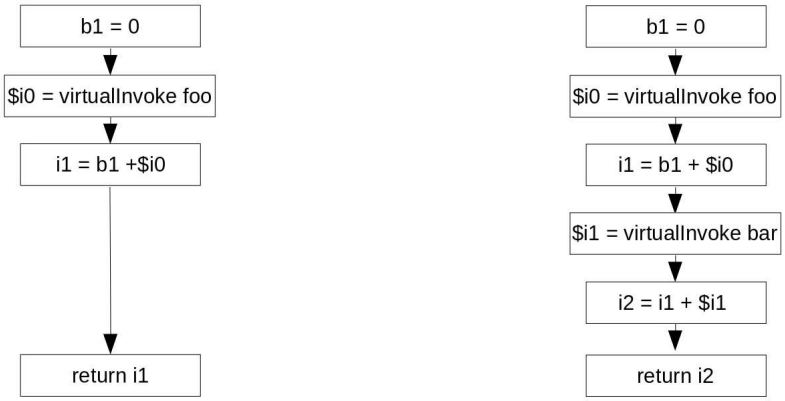


Abbildung 6.7: Beispielfunktionen und Dominatorbäume mit Registercode-Instruktionen.

genutzt werden muss, damit nicht zu viele potentiell falsch positive Klone ermittelt werden. Verallgemeinernd lässt sich festhalten, dass im Stackcode insgesamt sehr viele gleichartige Knoten in den Pfaden enthalten sind und somit beim Pfadvergleich die Pfade sehr ähnlich zueinander sind. Enthält ein Pfad beim Vergleich zweier Pfade jedoch zusätzliche Instruktionen, dann ist der syntaktische Unterschied größer als beim Sourcecode-basierten Vergleich, da eine zusätzliche Sourcecode-Instruktion zu mehreren Stackcode-Instruktionen übersetzt wird.

Insgesamt lässt sich zusammenfassen, dass durch die Kombination aus `EäP` und `EvSI` eine Verbesserung der Recall-Werte mit einer besseren Laufzeit gegenüber der Ausgangsmessung erzielt werden konnte. Alles in allem kann der Stackcode-basierte Ansatz jedoch nicht an die Ergebnisse des Sourcecode-basierten Ansatzes aus Tabelle 6.3 heranreichen, da gerade die *Strongly Typ-3* Recall-Werte viel geringer sind.

6.3.2 Registercode-basierte Klonerkennung

Bei dieser Instanziierung des allgemeinen Ansatzes aus Kapitel 4 wird anstatt des Quellcodes oder des Stackcodes als Grundlage der Codeklonerkennung ein Registercode verwendet. Aus diesem Register-basierten Jimplecode können erneut der

	StoneDetector (Registercode)							
	Ganze Pfade				Gesplittete Pfade			
	Normal		Kombi. Knoten		Normal		Kombi. Knoten	
Typ-1	20.792	99,9%	20.792	99,9%	20.792	99,9%	20.792	99,9%
Typ-2	3.467	99,9%	3.467	99,9%	3.467	99,9%	3.467	99,9%
Very-Strongly Typ-3	3.227	99,4%	3.228	99,4%	3.182	98,0%	3.157	97,2%
Strongly Typ-3	6.244	85,5%	6.334	86,7%	5.562	76,2%	5.768	79,0%
Moderatly Typ-3	8.994	35,2%	5.340	20,9%	4.705	18,4%	3.648	14,2%
Typ-4	20.594	0,5%	5.726	0,1%	11.025	0,2%	4.870	0,1%
Klone insgesamt	967.443		721.903		940.900		905.553	
Zeit auf dem BCB	229,34min		41,22min		26,12min		16,15min	

Tabelle 6.6: Recall-Werte des Registercode-basierten StoneDetector-Ansatzes für $\geq$ 15 Zeilen mit und ohne kombinierte Knoten.

Kontrollflussgraph, der (abstrakte) Dominatorbaum und die Beschreibungsmengen gebildet werden. In Abbildung 6.7 lassen sich die im vorherigen Abschnitt beschriebenen Beispielfunktionen einsehen. Nun sind jedoch die vereinfachten Dominatorbäume auf Registercode-Basis dargestellt. Fünf von sechs Zeilen stimmen bei den Sourcecode-Funktionen überein und die Stackcode-Pfade hatten lediglich fünf von acht Knoten mit syntaktischer Übereinstimmung. Die Registercode-Pfade, die direkt aus dem Dominatorbaum abgelesen werden können, haben vier von sechs gleiche Knoten. Die syntaktische Übereinstimmung ist beim Registercode also größer als beim Stackcode, jedoch nicht so groß wie beim Sourcecode, wenn eine ganze Zeile hinzugefügt beziehungsweise gelöscht wurde. Um die syntaktische Übereinstimmung der Pfade weiter zu erhöhen, wurden sogenannte *Kombinierte Knoten* eingeführt. Existiert eine Instruktion, deren Ergebnis im Sourcecode nicht direkt einer Variable zugewiesen wird, so wird sie im Registercode in den temporären Registern zwischengespeichert und nicht direkt in ein virtuelles Register geschrieben. Dieses Verhalten kann in Abbildung 6.7 gut an dem Beispiel des *foo*-Funktionsaufrufes nachvollzogen werden. Das Ergebnis des Funktionsaufrufs wird nicht direkt in einer Variable gespeichert, sondern zunächst mit dem Wert der Variable i addiert. Aus diesem Grund wird das Ergebnis des Funktionsaufrufs in beiden Beispielen der Abbildung in dem temporären Register *$i0* zwischengespeichert. Erst das Ergebnis der Addition von *b1* und *$i0* wird in dem virtuellen Register *i1* fest gespeichert.
Bei der Verwendung von *Kombinierten Knoten* konnten die temporären Knoten zusammengefasst und mit der nächsten Instruktion kombiniert werden, die einer Zuweisung im Sourcecode entspricht. Da auf diese Weise im Vergleich zum Sourcecode die zusätzlich eingefügten Knoten zusammengefasst wurden, konnte auch anders als im Stackcode wieder ein Schwellenwert von 30% genutzt werden. Diese zusammen-

gefassten Knoten werden *Kombinierte Knoten* genannt. Erzeugt und ersetzt man in dem Beispiel aus Abbildung 6.7 gedanklich die verschiedenen Instruktionen durch kombinierten Knoten, so sind nach der Ersetzung drei von vier Knoten syntaktisch übereinstimmend. Das Ergebnis des Sourcecodes kann nur nicht erreicht werden, da in dem hier vorgestellten Ansatz die Funktionsköpfe und Klammern nicht mit betrachtet werden können.

In Tabelle 6.6 sind nun die StoneDetector-Ergebnisse des Registercode-basierten Ansatzes aufgezeigt, wobei sowohl die Ergebnisse mit ganzen Pfaden, mit gesplitteten Pfaden und jeweils mit und ohne der Nutzung der kombinierten Knoten dargestellt sind. Durch die Nutzung der kombinierten Knoten konnten insbesondere die benötigten Zeiten für die Klonsuche auf dem BigCloneBench verringert werden. Auch die insgesamt gefundene Anzahl an potentiellen Klonen ist gefallen. An den Recall-Werten haben sich durch die Nutzung der kombinierten Knoten wider Erwartend keine großen Veränderungen ergeben. Da jedoch die niedrigere Anzahl an insgesamt gefundenen Klonen und die verringerte Rechenzeit positive Ergebnisse sind, wurden die weiteren Messungen in Tabelle 6.7 mit den kombinierten Knoten als Grundeinstellung durchgeführt. Alle dort abgebildeten Ergebnisse nutzen also die kombinierten Knoten. Wie bereits im vorherigen Abschnitt beschrieben, wurden bei den Messungen unter `Eliminierung ähnlicher Pfade (EäP)` alle Pfade außer einem aus einer Beschreibungsmenge entfernt, die sich in weniger als zwei Knoten voneinander unterscheiden. Bei der Variante mit den gesplitteten Pfaden waren es beispielsweise insgesamt über 200.000 Pfade, die aus den Beschreibungsmengen entfernt werden konnten. Bei den Messungen unter `Eliminierung von Registercode-Instruktionen (EvRI)` wurden erneut Instruktionen, wie beispielsweise *new*-, *Feldzugriffs*-, *Konstantenzugriffs*-, und *println*-Anweisungen aus allen Pfaden herausgenommen. Bei den Messungen von `EäP + EvRI` wurden die Anpassungen von `EäP`, von `EvRI` und natürlich die Verwendung der kombinierten Knoten angewendet.
Auffällig ist die große Anzahl an gefundenen potentiellen Klonen bei dem Test mit gesplitteten Pfaden unter der Verwendung von `EvRI`. Dies ist dadurch zu erklären, dass aufgrund des Löschens vieler Instruktionen aus den bereits kurzen Pfaden, sehr viele sehr kurze Pfade entstehen, die sich dann ausgesprochen ähnlich sind und das Ergebnis manipulieren. Aufgrund des Löschens von ähnlichen Pfaden konnte das Problem dann im letzten Experiment wieder aufgehoben werden.
Insbesondere die Ergebnisse der Variante mit ganzen Pfaden und allen Parametern

	StoneDetector (Registercode) mit ganzen Pfaden					
	EäP		EvRI		EäP + EvRI	
Typ-1	20.792	99,9%	20.792	99,9%	20.792	99,9%
Typ-2	3.467	99,9%	3.466	99,8%	3.466	99,8%
Very-Strongly Typ-3	3.234	99,6%	3.237	99,7%	3.238	99,7%
Strongly Typ-3	6.409	87,8%	6.298	86,2%	6.430	88,0%
Moderatly Typ-3	5.611	21,9%	3.887	15,2%	4.311	16,8%
Typ-4	5.924	0,1%	3.383	0,0%	3.754	0,0%
Klone insgesamt	667.158		619.170		600.522	
Zeit auf dem BCB	38,13min		38,35min		35,36min	
	StoneDetector (Registercode) mit gesplitteten Pfaden					
	EäP		EvRI		EäP + EvRI	
Typ-1	20.792	99,9%	20.792	99,9%	20.792	99,9%
Typ-2	3.467	99,9%	3.466	99,8%	3.466	99,8%
Very-Strongly Typ-3	3.116	96,0%	3.198	98,5%	3.197	98,5%
Strongly Typ-3	5.437	74,4%	6.232	85,3%	5.847	80,1%
Moderatly Typ-3	3.330	13,0%	5.041	19,7%	4.793	18,7%
Typ-4	5.610	0,1%	8.996	0,2%	6.544	0,1%
Klone insgesamt	661.674		2.267.643		878.069	
Zeit auf dem BCB	16,05min		17,25min		16,24min	

Tabelle 6.7: Recall-Werte des Registercode-basierten StoneDetector-Ansatzes für $\geq$ 15 Zeilen.

sind sehr nah an dem, was die Quellcode-basierte Variante des StoneDetectors (Tabelle 6.3) findet. Lediglich die Laufzeit ist etwas höher, was daran liegt, wie die Kontrollflussgraphen mithilfe des Soot-Frameworks aus dem Java-Bytecode erzeugt werden. Insgesamt lässt sich jedoch festhalten, dass die Registercode-basierte Version des StoneDetectors sehr gut Codeklone identifizieren kann, die ursprünglich im Quelltext als solche markiert wurden.

6.3.3 Vergleich StoneDetector mit iClones

Von denen im Abschnitt 6.1 genannten Java-Bytecode-basierten Klonerkennungstools wurde der Quelltext oft nicht veröffentlicht [74, 179, 180] oder er ist nun nicht mehr verfügbar [76]. Grundsätzlich lassen sich wohl zwei Gründe dafür ausmachen. Der erste Grund ist, dass es bisher keinen geeigneten und modernen Java-Bytecode-basierten Klonerkennungsbenchmark gab, weshalb ein Vergleich zwischen verschiedenen Tools schwierig war und deshalb eine Veröffentlichung des Quelltextes als überflüssig erachtet wurde. Der zweite Grund ist, dass es im Vergleich zur Quelltext-basierten Klonerkennung ein geringeres Interesse an Forschungen in dem Bereich der Java-Bytecode-basierten Klonerkennung gab, weshalb insgesamt weniger Arbeiten

	StoneDetector (Stackcode)				iClones (Stackcode)		StoneDetector (Registercode)			
	Normal		Split				Normal		Split	
Typ-1	20.792	99,9%	20.792	99,9%	20.777	99,9%	20.792	99,9%	20.792	99,9%
Typ-2	3.463	99,7%	3.466	99,8%	3.373	97,2%	3.466	99,8%	3.466	99,8%
Very-Strongly Typ-3	3.157	97,2%	3.098	95,4%	3.043	93,7%	3.238	99,7%	3.197	98,5%
Strongly Typ-3	5.035	68,9%	5.047	69,1%	3.559	48,7%	6.430	88,0%	5.847	80,1%
Moderatly Typ-3	2.804	10,9%	4.184	16,4%	1.345	5,2%	4.311	16,8%	4.793	18,7%
Typ-4	3.178	0,0%	7.774	0,2%	1.087	0,0%	3.754	0,0%	6.544	0,1%
Klone insgesamt	439.767		582.397		362.403		600.522		878.069	
Zeit auf dem BCB	77,20min		21,29min		37,59min		35,36min		16,24min	
Precision	0,99		0,97		1,00		0,99		0,96	

Tabelle 6.8: Vergleich StoneDetector und iClones auf Bytecode-/Registercode-Basis für ≥ 15 Zeilen.

und Tools in dem Bereich veröffentlicht wurden.

Da kein geeignetes Java-Bytecode-basiertes Tool mit veröffentlichtem Quelltext gefunden werden konnte, wurde das im Abschnitt 3.2 vorgestellte Quelltext-basierte Klonerkennungstool iClones [54] weiterentwickelt, sodass mit iClones nun auch im Java-Bytecode nach Codeklonen gesucht werden kann. Freundlicherweise wurde der Quelltext von iClones von den Autoren zur Verfügung gestellt. Anders als im StoneDetector wurde die Bytecode-Verarbeitung nicht mit dem Soot-Framework vorgenommen, sondern mit javassist[4]. javassist ist eine leichtgewichtige Bibliothek, die schnell und unkompliziert Java-Bytecode verarbeiten und manipulieren kann. Allerdings ist es mit javassist nicht möglich, die Jimple-Repräsentation und damit den Registercode zu erzeugen.

Bei der Implementierung des Java-Bytecode-basierten Ansatzes in iClones wurden, wie bereits im Abschnitt 6.3.1 beschrieben, *Load-*, *Store-*, *new-*, *Konstantenzugriffs-* und *Feldzugriffs*-Anweisungen nicht berücksichtigt, um einen fairen Vergleich zu StoneDetector zu ermöglichen. Die `Eliminierung von ähnlichen Pfaden` im StoneDetector ist jedoch eine spezielle Optimierung, die nur bei Pfad-basierten Ansätzen vorgenommen werden kann. Aus diesem Grund war die Umsetzung dieser Optimierung in iClones nicht möglich. Für iClones wurden die Parameter bestimmt, mit denen die besten Ergebnisse erzielt werden konnten. Die minimale Tokenanzahl, die ein Codefragment haben muss, wurde auf 80 Tokens festgelegt. Die minimale Anzahl an übereinstimmenden Tokenreihungen, die zum Zusammenfassen von Klonen genutzt werden, hat einen optima-

[4]https://www.javassist.org/

len Wert von 20.
In der Tabelle 6.8 ist der Stackcode-basierte und der Registercode-basierte StoneDetector-Ansatz jeweils mit ganzen und gesplitteten Pfaden abgebildet. Bei dem Stackcode-basierten Ansatz wurden die Recall-Werte aus der Tabelle 6.5 mit *EäP + EvSI* und bei dem Registercode-basierten Ansatz wurden die Werte aus der Tabelle 6.7 mit *EäP + EvRI* genutzt. Weiterhin sind in der Tabelle 6.8 die Ergebnisse vom Stackcode-basierten iClones-Tool abgebildet. Ein Vergleich zwischen iClones und StoneDetector ist jedoch nur fair, wenn die Stackcode-basierte StoneDetector-Variante genutzt wird, da anzunehmen ist, dass iClones mit Registercode ebenso bessere Ergebnisse erzielen würde.
Die Recall-Werte wurden erneut mithilfe des BigCloneBench beziehungsweise BigCloneEval nach der Anwendung des Stubber-Tools gemessen, wobei nur Funktionen untersucht wurden, die mindestens 15 Zeilen im Sourcecode besitzen (vgl. Abschnitt 5.1). Aus der Liste aller identifizierten potentiellen Klone eines Tools wurden alle Einträge entfernt, die mehr als einmal enthalten waren. Weiterhin wurde die Zeit gemessen, die das entsprechende Tool benötigt hat, um die Klone im BigCloneBench zu identifizieren. Anschließend wurde für jedes Tool ein Precision-Wert bestimmt. Dazu wurden von jedem Verfahren 400 zufällig ausgewählte Klonpaare von einem menschlichen „Juror“ überprüft, wobei jedoch nicht bekannt war, welches Tool das jeweilige Funktionspaar als Klon identifiziert hat (vgl. Abschnitt 5.2). Diese Ergebnisse sind für jedes Tool und jede Toolvariante in der Tabelle 6.8 abgebildet.
Die Recall-Werte des Stackcode-basierten StoneDetector-Ansatzes sind dabei sowohl bei der Nutzung von ganzen als auch bei gesplitteten Pfaden etwas besser als bei iClones. Dazu muss aber angemerkt werden, dass iClones insgesamt am wenigsten potentielle Klone identifiziert und auch mit 1,0 den besten Precision-Wert besitzt. Die Laufzeit von iClones liegt mit 37,59 Minuten genau zwischen den beiden Werten des Stackcode-basierten StoneDetector-Ansatzes.

In Tabelle 6.9 können gemeinsame und exklusive Codeklone zwischen der Stackcode-basierten StoneDetector-Variante und iClones eingesehen werden. Dabei ist besonders interessant, dass iClones jeweils nur etwas weniger als 80.000 exklusive Klone besitzt. Der Stackcode-basierte StoneDetector-Ansatz findet also mit 282.510 beziehungsweise 287.665 die meisten Codeklone, die auch von iClones detektiert werden. Die von iClones exklusiv detektierten Klone sind abermals Funktionspaare, bei denen einige Funktionszeilen gleich sind, wohingegen sich die restlichen Zeilen der Funktionen unterscheiden. Dies ist also ähnlich zu dem Quelltext-basierten

A	B	A∩B	A\B	B\A
StoneDetector (Stackcode, Normal)	iClones	282.510	157.257	79.893
StoneDetector (Stackcode, Split)	iClones	287.665	294.732	74.738
StoneDetector (Stackcode, Normal)	StoneDetector (Registercode,Normal)	392.331	47.436	208.191
StoneDetector (Stackcode, Split)	StoneDetector (Registercode,Split)	388.160	194.237	489.909

Tabelle 6.9: Gemeinsame und exklusive Codeklone für Java-Bytecode-basierte Klonerkennungstools.

Vergleich von StoneDetector und iClones im Abschnitt 5.3. In der Tabelle 6.9 ist weiterhin ein Vergleich zwischen dem Stackcode-basierten und dem Registercode-basierten StoneDetector-Ansatz zu sehen. Dabei ist zu erkennen, dass mit 392.331 beziehungsweise 388.160 für die normalen und gesplitteten Pfade jeweils etwas mehr als die Hälfte der gefundenen Klone von beiden Ansätzen gleich detektiert wurden. Das deutet darauf hin, dass es deutliche Unterschiede zwischen dem Stackcode und dem Registercode gibt, sodass doch sehr verschiedene Funktionspaare als Klone detektiert werden und aus diesem Grund die Schnittmenge nicht größer ist.
Insgesamt lässt sich zusammenfassen, dass aus allen Ansätzen die Registercode-basierte StoneDetector-Variante mit ganzen Pfaden die besten Recall-Werte besitzt und dabei eine gute Laufzeit und einen sehr guten Precision-Wert aufweist. Zur Nutzung eines Java-Bytecode-basierten Klonerkennungsansatzes kann also die Registercode-basierte StoneDetector-Variante mit ganzen Pfaden empfohlen werden.

Kapitel 7

Maschinelles Lernen zur Klonerkennung

Einige der in Abschnitt 3.6 vorgestellten maschinellen Lernverfahren [45,53,168,170] geben an, zusätzlich zu den Typ-1- bis Typ-3-Klonen auch Typ-4-Klone zu finden. Dabei sollen sie, wie in Tabelle 7.1 zu sehen, sehr gute Recall- und Precision-Werte auf dem CDLH-Teildatensatz des BigCloneBench erzielen. Zum Trainieren und zum Evaluieren wurden Klone aus allen Typkategorien verwendet. FA-AST+GMN [168] erzielt beispielsweise einen Precision-Wert von 0,96 mit einem Recall-Wert von 0,94. Da hierbei auch Typ-4-Klone genutzt wurden und BigCloneBench, wie in Tabelle 6.2 zu sehen, zu 98% aus Typ-4-Klonen besteht, wäre das Verfahren deutlich besser im Detektieren von Codeklonen als der StoneDetector-Ansatz.
Um diese guten Ergebnisse der maschinellen Lernverfahren zu überprüfen, wurde in diesem Kapitel das maschinelle Lernverfahren aus der Arbeit von FA-AST+GMN genutzt und mit Dominatorbäumen trainiert. Die Dominatorbäume wurden aus dem

Modell	Precision	Recall	F1
RtvNN	0,95	0,01	0,01
CDLH	0,92	0,74	0,82
ASTNN	0,92	0,94	0,93
FA-AST+GMN	**0,96**	**0,94**	**0,95**
RoBERTa	0,960	0,955	0,957
CodeBERT	0,964	0,966	0,965
GraphCodeBERT	0,973	0,968	0,971

Tabelle 7.1: Ergebnisse [53] verschiedener maschineller Lernverfahren auf dem BigCloneBench (CDLH).

Java-Bytecode erzeugt. Es wurden insgesamt 7 Experimente durchgeführt, die zeigen sollen, dass diese guten Ergebnisse keine praktische Relevanz haben, da die Ergebnisse nicht auf ungesehenen Daten erzielt werden können.

7.1 CDLH-Teildatensatz

Man kann die Dateien des BigCloneBench nicht ohne weitere Vorverarbeitung zum Trainieren und Evaluieren von maschinellen Lernverfahren einsetzen. Maschinelle Lernverfahren benötigen Trainings- und Evaluationsdatensätze, die wiederum aus positiven und negativen Beispielen bestehen. Positive Beispiele sind tatsächliche Klonpaare und negative Beispiele Codefragmentpaare, die keine Klone zueinander sind. Wei und Li [170] waren die ersten, die eine solche spezielle BigCloneBench-Teilmenge für maschinelle Lernverfahren erstellten, als sie ihr Framework für tiefes *Feature-Learning (CDLH)* vorstellten, das seitdem auch von vielen anderen Tools [45, 53, 168] für das Trainieren und Evaluieren anderer maschineller Lernverfahren verwendet wird. Dieser CDLH-Datensatz wurde aus einer älteren Version des BigCloneBench erstellt, in der der Benchmark „nur“ Klone aus 10 verschiedenen Funktionalitäten enthielt. Der Benchmark hatte in dieser älteren Version etwa 6-Millionen Codeklone. Wei und Li entfernten alle Codefragmente ohne ein bekanntes wahres und falsches Klonpaar. So blieben nur 9.134 Codefragmente übrig, aus denen dann der CDLH-Datensatz aufgebaut wurde. Dazu werden die bekannten Klone der 9.134 Codefragmente als positive Beispiele und alle anderen Funktionspaare als negative Beispiele verwendet. Die Annahme, dass zwei Funktionen, die im Benchmark nicht als Klon enthalten sind, kein Klonpaar bilden, ist jedoch falsch [88]. Es könnte trotzdem ein Klonpaar sein, dass jedoch nicht in die Kategorien des Benchmarks fällt.
Aus dem grundsätzlichen Ansatz konnten jedoch ausgewogene Trainings-, Validierungs- und Evaluationsdatensätze gebildet werden. Die Ergebnisse einiger Klondetektoren unter Verwendung des CDLH-Datensatzes sind in Tabelle 7.1 zu sehen.

7.2 Graph-Matching-Network

Herkömmliche tiefe neuronale Netze wie *rekursive neuronale Netze (RNN)* [96] oder *convolutional neuronale Netze (CNN)* [93] haben bei sequentiellen Daten wie natürlicher Sprache oder euklidischen Daten wie Bildern große Erfolge erzielt. Im Gegensatz dazu sind Graphen jedoch sehr viel komplexer. Text kann als eine Abfolge

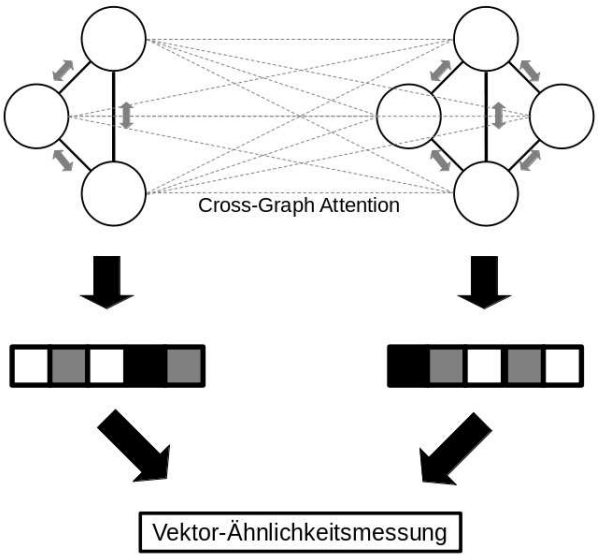


Abbildung 7.1: Architektur-Illustration eines *Graph-Matching-Networks*.

von Wörtern und Bilder als eine Abfolge von Pixeln betrachtet werden, während Graphen aus Knoten und Kanten und somit aus zwei verschiedenen Informationen bestehen. Daher wurden neue Architekturen für neuronale Netze benötigt, die in der Lage sind, Modelle von Graphen zu lernen. In einem ersten Schritt wurden *Graph Neural Networks (GNN)* [133] eingeführt. Ein GNN lernt ein *Zustandsembedding* für jeden Knoten eines Graphen, das Informationen aus der Nachbarschaft des Knotens enthält und erstellt auf dieser Grundlage ein *Embedding* des gesamten Graphen. Die meisten bestehenden GNN-Modelle können als *Message Passing Neural Networks (MPNN)* [50] klassifiziert werden. In MPNNs werden zwei Phasen durchgeführt: Nachrichtenübermittlung und Auslesung. Während der Nachrichtenübermittlung aktualisiert das Netz den verborgenen internen Zustand jedes Knotens, indem es Informationen von den Nachbarn des Knotens über die Kanten sammelt und verarbeitet. Dieser Aktualisierungsprozess wird in mehreren Schritten durchgeführt. In der Auslesungsphase wird eine Vektordarstellung des gesamten Graphen durch das Modell bestimmt.

In der Arbeit von Li et al. [98] werden *Graph-Matching-Networks (GMNs)* vorgestellt, die ganz ähnlich wie GNNs funktionieren und Embeddings für ein Graphenpaar auf die fast gleiche Weise berechnen. Der Unterschied zu GNNs ist jedoch, dass GMNs zusätzlich sogenannte *Cross-Graph-Attentions* zwischen den Knoten der Graphen berechnen. Bei GNNs ist das für einen Graphen erzeugte Embedding immer dasselbe, unabhängig davon, welcher zusätzliche Graph als Eingabe dient. Bei GMNs hingegen wird das Embedding eines Graphen nun in Abhängigkeit von dem anderen

Graphen berechnet.
Es wird also sowohl bei GNNs als auch bei GMNs jeweils ein Embedding für die Graphen berechnet, die dann mit einer Ähnlichkeitsfunktion verglichen werden können. Abbildung 7.1 zeigt eine Illustration des GMN-Ansatzes mit gestrichelten Verbindungen zwischen den Knoten der zwei Graphen, welche die *Cross-Graph-Attention* darstellen sollen.

7.3 Experimente zu GMNs und Dominatorbäumen

Um eine Abgrenzung zu dem bestehenden System FA-AST+GMN [168] zu schaffen, wurden nicht wie bei FA-AST+GMN abstrakte Syntaxbäume mit zusätzlichen Kontroll- und Datenflusskanten als Eingabe in das GMN genutzt, sondern komplette Dominatorbäume, ohne dass Pfade aus diesen abgeleitet werden. Der Grund dafür ist, dass GMNs darauf ausgelegt sind, ganze Graphenstrukturen zu lernen. Zudem wurden die Dominatorbäume nicht aus dem Sourcecode extrahiert, sondern aus dem Java-Bytecode abgeleitet. Die Verwendung von GMN mit Dominatorbäumen wird im Folgenden *DT+GMN* genannt.
Die Erzeugung von Trainingsdatensätzen erfolgt mithilfe des BigCloneBench. Natürlich werden Funktionen und zugehörige Codeklone, die vom Stubber-Tool nicht kompiliert werden können, vorher aus der Datenbank des Benchmarks entfernt.
Alle nachfolgenden Ergebnisse wurden mit der *Early-Stop-Methode* ermittelt. Die Ergebnisse jeder Zwischenauswertung wurden nur gespeichert, wenn sie besser waren als die Ergebnisse davor. So konnte das jeweils beste Ergebnis ermittelt werden. Mit Ausnahme des direkten Vergleichs von DT+GMN mit FA-AST+GMN in Experiment 1 wurden erneut Funktionen mit weniger als 15 Zeilen im Quellcode übersprungen. Dies geschah, um den Einfluss von leicht zu findenden Klonen auf das Gesamtergebnis zu vermeiden, wie bereits im Abschnitt 5.1 erläutert wurde. In den Experimenten wurde zunächst gezeigt, dass die Ergebnisse der Klonerkennung mit GMN auf abstrakten Syntaxbäumen mit zusätzlichen Kontroll- und Datenflusskanten, generiert aus dem Java-Quellcode, unter Verwendung des CDLH-Datensatzes, mit GMN auf Dominatorbäumen, generiert aus dem Java-Bytecode, reproduziert werden können (Experiment 1). Anschließend wird der gesamte BigCloneBench betrachtet und nicht nur die 9.134 Codefragmente des CDLH-Datensatzes, um zu überprüfen, wie gut DT-GMN auf dem gesamten Benchmark abschneidet (Experiment 2). Diese beiden Experimente bilden die Grundlage für die nachfolgende Auswertung. Mit den darauffolgenden Experimenten soll ermittelt werden, wie gut sich

der BigCloneBench und der CDLH-Datensatz für das Training und die Evaluierung von auf maschinellem Lernen basierenden Klondetektoren eignen. In den Experimenten werden zwei Hauptprobleme von dem BigCloneBench und dem CDLH-Datensatz aufgedeckt:

1. Die Trainings- und Evaluationsdatensätze sind nicht strikt voneinander getrennt

2. Es gibt kein *ground-truth* darüber, was kein Codeklon ist

Die Experimente im Abschnitt 7.3.2 untersuchen die Auswirkungen einer strikteren Trennung von Trainings- und Evaluationsdatensätzen (Experiment 3 und 4). Die Vermutung bestand, dass einige Codeklone sowohl in den Trainings- als auch in den Evaluationsdaten vorkommen. Dies hätte zur Folge, dass das neuronale Netz diese Beispiele auswendig lernt und aus diesem Grund die guten Ergebnisse in Tabelle 7.1 entstehen. Eine Folge daraus wäre, dass diese Netze nicht gut auf ungesehene Daten abstrahieren können und somit die Ergebnisse wenig praktische Relevanz hätten.
Die Experimente in Abchnitt 7.3.3 sollen die Frage beantworten, ob die Art und Weise, wie die negativen Beispiele ausgewählt werden, einen positiven Einfluss auf das Ergebnis der maschinellen Lernverfahren hat, wie es in anderen Anwendungsfeldern von maschinellen Lernverfahren bekannt ist [11]. Im CDLH-Datensatz kann jedes Funktionspaar als negatives Beispiel genutzt werden, wenn es nicht im Benchmark als Klon enthalten ist, auch wenn nicht alle Klone im Benchmark enthalten sind, wie die Autoren des BigCloneBench [149] schreiben. Aus diesem Grund wird StoneDetector aus Abschnitt 5 verwendet, um zumindest für Typ-1- bis Typ-3-Klone auf dem BigCloneBench einen fundierteren *ground-truth* zu liefern (Experiment 5, 6 und 7).

7.3.1 Experimente zum Vergleich von DT+GMN mit anderen Tools

In diesem Unterabschnitt wird ein Vergleich von DT+GMN und FA-AST+GMN durchgeführt. Darüber hinaus wird eine weitere Bewertung mit dem vollständigen BigCloneBench-Benchmark anstelle des begrenzten CDLH-Datensatzes beschrieben. Das Verfahren zur Erstellung der Datensätze bleibt jedoch gleich.

Experiment	Precision	Recall	F1
Experiment 1	0,936	0,961	0,948
Experiment 2	0,857	0,891	0,873
Experiment 3	0,804	0,822	0,813
Experiment 4	0,720	0,725	0,723
Experiment 5	0,857	0,878	0,867
Experiment 6	0,989	0,976	0,983
Experiment 7	0,986	0,930	0,957

Tabelle 7.2: Ergebnisse der Experimente für DT+GMN.

Experiment 1: CDLH Teildatensatz

Im ersten Experiment wird der CDLH-Teildatensatz verwendet. Wie bereits beschrieben, enthält der CDLH-Teildatensatz insgesamt 9.134 Funktionen, von denen 8.787 mit dem Stubber-Tool kompiliert werden können. Somit können 96,2% aller Funktionen im CDLH-Datensatz für das Experiment verwendet werden. Wie in Tabelle 7.2 zu erkennen ist, wird in diesem Experiment ein *F1*-Wert von 0,948 erreicht, der nur knapp unter der Referenzarbeit FA-AST+GMN von Wang et al. [168] liegt. Dabei wurden 90% der Daten zum Training und 10% der Daten zum Evaluieren genutzt. DT+GMN fügt allerdings keine zusätzlichen Daten- oder Kontrollflusskanten in den Dominatorbaum ein. Zum besseren Vergleich sind die Ergebnisse von Wang et al. sowie anderen Tools für den CDLH-Teildatensatz in Tabelle 7.1 dargestellt. Die neueren und sehr viel komplexeren BERT-Transformatormodelle liefern sogar noch bessere Ergebnisse für diesen Datensatz. Die Ergebnisse in Tabelle 7.1 stammen aus Guo et al. [53].

Experiment 2: Kompletter BigCloneBench

In diesem Experiment werden alle Dateien und alle Ordner des neuesten BigCloneBench-Datensatzes verwendet, um die Trainings- und Evaluationsdatensätze zu erzeugen, nachdem nicht kompilierbare Dateien entfernt wurden (siehe Abschnitt 6.2). Das *F1*-Ergebnis von 0,873 ist etwas schlechter als im vorherigen Experiment, wie in Tabelle 7.2 zu sehen ist. Dies ist auf die größere Anzahl verschiedener Funktionen zurückzuführen, die jetzt zur Generierung der Datensätze verwendet werden. Im CDLH-Datensatz werden nur 8.787 kompilierbare Funktionen verwendet. Im Gegensatz dazu werden in diesem Experiment, das auf dem aus 43 verschiedenen Funktionalitäten bestehenden vollständigen BigCloneBench basiert, über 150.000 verschiedene Funktionen verwendet, um die Trainings- und Evaluationsdatensätze

zu erstellen. Negative Trainings- und Evaluationsbeispiele, d. h. Codefragmente, die nicht als Codeklone betrachtet werden, werden wie im vorherigen Experiment mit jedem Funktionspaar, das nicht als Codeklon in BigCloneBench annotiert wurde, erzeugt.

7.3.2 Experimente zur Separation von Trainings- und Evaluationsdatensatz

In den folgenden Experimenten kann gezeigt werden, dass die Recall-Ergebnisse des GMN-Klondetektors umso schlechter werden, je strikter die Trennung von Trainings- und Evaluationssdaten ist. Um dies zu zeigen, wurden zwei Experimente durchgeführt. Im ersten Experiment (Experiment 3) darf dieselbe Funktion nur entweder in den Trainings- oder in den Evaluationsdaten vorkommen. Dieser Ansatz ist jedoch offensichtlich nicht ausreichend, da eine Funktion und ihr Codeklon sowohl in den Trainings- als auch in den Auswertungsdaten vorhanden sein können. Aus diesem Grund wird in einem weiteren Experiment (Experiment 4) eine striktere Trennung vorgenommen, die auf den Funktionalitäten des BigCloneBench basiert.

Experiment 3: Funktionsseparation

In Experiment 1 und 2 werden Trainings- und Evaluationsdaten nicht strikt getrennt, wie man an dem folgenden Beispiel sehen kann:

- A ist ein Klon zu B
- A ist ein Klon zu C
- B ist ein Klon zu C
- Klonpaar A-B ist im Trainings- und A-C im Evaluationsdatensatz

Wenn A, B und C Funktionen sind, dann sind die Dominatorbäume des Funktionspaares A-B im Trainingsdatensatz genau die gleichen wie die Dominatorbäume des Funktionspaares A-C im Evaluationsdatensatz. Dieser Fall würde das Endergebnis verfälschen. Aus diesem Grund werden in diesem Experiment die Datensätze so konstruiert, dass eine Funktion nur entweder im Trainings- oder im Evaluationsdatensatz enthalten sein kann. Interessanterweise sinkt der Wert von *F1* auf 0,813, wie in Tabelle 7.2 zu sehen ist. Dies deutet darauf hin, dass einige Funktionspaare zuvor

sowohl in den Trainings- als auch in den Auswertungsdatensätzen vorkamen und das neuronale Netz diese auswendig gelernt hat, wodurch ein verfälschtes Ergebnis entstand.

Experiment 4: Separation von Funktionalitäten

Leider ist der Ansatz in Experiment 3 nicht rigoros genug, wie das folgende Beispiel zeigt:

- A ist ein Typ-1-Klon zu B
- C ist ein Typ-1-Klon zu D
- A ist ein Typ-4-Klon zu C
- B ist ein Typ-4-Klon zu D
- Klonpaar A-C ist im Trainings- und B-D im Evaluationsdatensatz enthalten

Auch in diesem Beispiel sind dieselben Codefragmente sowohl im Trainingsdatensatz als auch im Evaluationsdatensatz enthalten, allerdings mit unterschiedlichen Namen. BigCloneBench besteht aus 43 verschiedenen Funktionalitäten. In diesem Experiment werden 4 Funktionalitäten mit den meisten Codeklonen verwendet, um den Trainingsdatensatz zu erstellen, und der Evaluierungsdatensatz wird aus den verbleibenden 39 Funktionalitäten erstellt, wodurch Szenarien wie das oben genannte besser vermieden werden können. Obwohl nur die Klone aus 4 Funktionalitäten zur Erstellung des Trainingsdatensatzes verwendet wurden, existieren viel mehr Trainings- als Evaluationsdaten. Das Ergebnis dieses Experiments 4 ist in Tabelle 7.2 zu sehen. Mit einem *F1*-Wert von 0,723 schneidet das GMN deutlich schlechter ab als in den vorherigen Experimenten.

Weiterhin wurden auch die DT+GMN-Recall-Ergebnisse für die verschiedenen Klontypen in BigCloneBench genauer betrachtet und mit den Ergebnissen von StoneDetector verglichen. Dies ist möglich, da anders als in den vorherigen Experimenten, die Datensätze nicht aus dem gesamten Benchmark erstellt wurden, sondern nach Funktionalitäten aufgeteilt wurden. Also konnten aus dem BigCloneBench die Klone der 4 Funktionalitäten entfernt werden, die zur Erstellung des Trainingsdatensatzes genutzt wurden. Wie in Tabelle 7.3 zu sehen ist, erkennt StoneDetector bis zu *Strongly Type-3* deutlich mehr Klone als DT+GMN. DT+GMN findet mehr Klone

Typ	DT+GMN		StoneDetector	
Typ-1	4.348 / 5.408	80,39%	5.309 / 5.408	98,16%
Typ-2	185 / 259	71,42%	256 / 259	98,84%
Very-Strongly Typ-3	1.311 / 2.245	58,39%	2.187 / 2.245	97,41%
Strongly Typ-3	1.291 / 3.458	37,33%	2.816 / 3.458	81,43%
Moderately Typ-3	1.736 / 9.102	19,07%	1.084 / 9.102	11,90%
Typ-4	85.271 / 1.333.866	6,39%	438 / 1.333.866	0,00%
Klone insgesamt	2.458.907		265.076	

Tabelle 7.3: Evaluation von DT+GMN auf 4 Funktionalitäten trainiert.

des *Moderately Typs-3* und des *Typs-4*. Allerdings ist die Gesamtzahl der Funktionspaare, die von DT+GMN als gefundene Klone eingestuft wurden, auch deutlich höher als bei StoneDetector. Diese besseren Ergebnisse von DT+GMN bei *Moderately Typ-3* und *Typ-4* sind zunächst überraschend. Der Grund dafür ist, dass der Precision-Wert von DT+GMN in diesem Experiment nur 0,720 beträgt und somit die zusätzlich gefundenen potenziellen Klone eine große Anzahl von Nicht-Klonen umfassen, sodass die zusätzlich gefundenen Typ-4-Klone als Zufallsfunde eingestuft werden können. Insgesamt wird die Analyse durch den geringeren Precision-Wert ungenauer.

7.3.3 Experimente mit genaueren Kloninformationen

Experimente zu maschinellen Lerntechniken mit BigCloneBench leiden unter einem ungenauen *ground-truth*, d. h. Informationen darüber, welche Paare von Codefragmenten ein Klonpaar darstellen und welche nicht. Um dieses Problem zu analysieren, wird in den Experimenten dieses Abschnitts versucht, bessere Informationen darüber zu erhalten und zu nutzen, welche Funktionspaare Klone und welche keine Klone sind. In einem ersten Schritt wurden die bisher nur zufällig zugewiesenen negativen Trainingsbeispiele zusätzlich mit StoneDetector überprüft und als negative Trainingsbeispiele ausgeschlossen, wenn StoneDetector sie als Codeklone identifiziert (Experiment 5). StoneDetector findet jedoch nur wenige Typ-4-Klone, sodass viele der negativen Trainingsbeispiele trotzdem noch Codeklone sein könnten, jedoch von StoneDetector nicht als solche identifiziert wurden. Um DT+GMN mit einem noch genaueren *ground-truth* zu trainieren und zu bewerten, werden in einem weiteren Experiment (Experiment 6) Training und Evaluation ausschließlich mit den von StoneDetector bereitgestellten Informationen über Codeklone durchgeführt. Obwohl tatsächlich immer noch falsche Kloninformationen verwendet werden, spiegelt dieser Ansatz den Einfluss von Klon- und Nicht-Klon-Informationen genauer wider.

In einem letzten Experiment werden die Ansätze kombiniert, um Trainings- und Evaluationsdaten zu trennen und einen qualitativ besseren *ground-truth* zu erhalten (Experiment 7).

Experiment 5: Beispiele durch StoneDetector überprüft

In diesem Experiment werden die Trainings- und Evaluationsdaten auf die gleiche Weise wie in Experiment 2 aufgebaut, aber jetzt werden die negativen Trainingsbeispiele mit StoneDetector überprüft. Insgesamt wurden 229 Codeklone gefunden und aus den negativen Trainingsbeispielen entfernt. Leider findet StoneDetector nur einen kleinen Teil aller Klone vom Typ-4. Aus diesem Grund bleibt das Ergebnis fast das gleiche wie im vorherigen Experiment 2, wie man in Tabelle 7.2 sehen kann.

Experiment 6: StoneDetector als Orakel

Die Informationen zu den Codeklonen in BigCloneBench sind nicht vollständig, da nicht alle Klone, die tatsächlich existieren, im Benchmark enthalten sind. Aufgrund seiner Konstruktion gibt es, wie bereits beschrieben, nur Informationen zu den 43 verschiedenen Funktionalitäten. Um dieses Problem zu beheben, wird eine andere Quelle für Codeklon-Informationen genutzt. Für dieses Experiment wird StoneDetector als weiteres Codeklon-Orakel verwendet. Alle Paare von Codefragmenten, die von StoneDetector als Codeklone identifiziert wurden, dienen als positive Trainingsbeispiele und die verbleibenden Paare von Codefragmenten dienen als negative Trainingsbeispiele. Auf diese Weise wird sowohl der Trainings- als auch der Evaluationsdatensatz generiert. In Tabelle 7.2 ist für dieses Experiment ein extrem hoher *F1*-Wert von 0,983 zu finden. Es ist jedoch zu beachten, dass StoneDetector nur eine geringe Anzahl von Klonen des Typs-4 identifiziert und aus diesem Grund nur wenige Klone des Typs-4 in dieses Experiment einbezogen wurden. Die Fragestellung dieses Experimentes ist nun anders und lautet, ob DT+GMN alle Klone finden kann, die durch StoneDetector gefunden werden konnten.

Experiment 7: Separation von Funktionalitäten und StoneDetector als Orakel

Im letzten Experiment wird der Ansatz von Experiment 4 und Experiment 6 in Kombination verwendet: StoneDetector wird als Orakel für die positiven und negativen

Typ-1	5.401 / 5.408	99,87%
Typ-2	258 / 259	99,61%
Very-Strongly Typ-3	1.659 / 2.245	73,89%
Strongly Typ-3	822 / 3.458	23,77%
Klone insgesamt	543.668	

Tabelle 7.4: Evaluation von DT+GMN trainiert auf 4 Funktionalitäten und StoneDetector als Orakel.

Trainingsbeispiele verwendet und nur die 4 Funktionalitäten von BigCloneBench, deren Ordner die meisten Java-Dateien enthalten, wurden für das Training verwendet, während die restlichen 39 Funktionalitäten für die Evaluation verwendet wurden. Das Ergebnis ist wiederum in Tabelle 7.2 mit einem *F1*-Wert von 0,957 zu finden. In Tabelle 7.4 werden außerdem die Recall-Werte für die verschiedenen Arten von Codeklonen gezeigt. Typ-4 wurde in dieser Tabelle ausgelassen, da im Trainingsdatensatz auf der Grundlage von StoneDetector größtenteils nur Codeklone von Typ-1 bis Typ-3 enthalten waren. Insbesondere zeigt dieses Experiment, dass bei genauer Kenntnis des *ground-truth* eine gute Klonerkennung möglich ist, selbst wenn Lern- und Evaluationsdaten getrennt sind.

7.4 Diskussion

In diesem Kapitel wurde zunächst gezeigt, dass DT+GMN mit Dominatorbäumen aus dem Java-Bytecode die gleichen Ergebnisse wie FA-AST+GMN auf dem begrenzten CDLH-Datensatz erbringt. Dabei nutzt FA-AST+GMN aus dem Java-Sourcecode generierte abstrakte Syntaxbäume mit zusätzlichen Kontroll- und Datenflusskanten. In den folgenden Experimenten wurden die Trainings- und Evaluationsdatensätze mit zunehmender Strenge getrennt. Das bedeutet, dass immer weniger gleiche Funktionspaare sowohl in den Trainings- als auch in den Evaluationsdatensätzen vorkamen. Infolgedessen schnitt das trainierte Modell immer schlechter ab, je strikter Trainings- und Evaluationsdaten getrennt wurden. In weiteren Experimenten wurde die Rolle der Qualität des *ground-truth*, d. h. der genauen Informationen über Klone und Nicht-Klone, analysiert und je genauer diese Informationen sind, desto besser ist die Leistung des trainierten Modells. Da in der Praxis jedoch nicht alle Klone und Nicht-Klone bekannt sind, sind die in diesen Experimenten erzielten Ergebnisse eher theoretischer Natur und geben lediglich einen Hinweis darauf, welche Ergebnisse in einer idealen Umgebung möglich wären.
Bei der Bewertung eines Modells für maschinelles Lernen sollten Trainings- und

Evaluationsdaten so streng wie möglich voneinander getrennt werden, um festzustellen, wie gut das neuronale Netz Vorhersagen für ungesehene Daten tätigen kann. Daher ist das Experiment 4 mit der Funktionstrennung wahrscheinlich die geeignetste Methode zur Durchführung einer Bewertung, wenn auch Klone vom Typ-4 einbezogen werden. Es könnte jedoch der Einwand erhoben werden, dass „nur“ 4 Funktionalitäten für das Training verwendet werden. Aus diesem Grund sollte ein ähnlicher Ansatz wie der von Saini et al. [130] für ihr Tool Oreo gewählt werden. Sie verwendeten einen völlig anderen Datensatz zum Trainieren und evaluierten auf dem kompletten BigCloneBench. Vergleichbar mit dem hier beschriebenen Ansatz leitet Oreo das *ground-truth* für den Trainingsdatensatz ab, indem es einen bereits existierenden Klondetektor, den SourcererCC, verwendet, was die erzielten Ergebnisse ebenfalls einschränkt.
Die mit Oreo erzielten Ergebnisse zeigen auch, dass es eines weiteren Datensatzes bedürfte, der ebenfalls eine ausreichende Anzahl von Typ-4-Klonen enthält. Allerdings gibt es keinen vergleichbaren Datensatz zum BigCloneBench, der zum Training verwendet werden könnte. Auch der Google-Code-Jam-Datensatz ist anders strukturiert und daher nicht unbedingt gut geeignet. Um eine wirklich aussagekräftige Bewertung eines auf maschinellem Lernen basierenden Klonerkennungswerkzeugs durchführen zu können, wäre daher ein neuer Klonbenchmark erforderlich. Insbesondere der häufig verwendete CDLH-Teildatensatz ist nicht geeignet, um ein maschinelles Lernwerkzeug zur Klonerkennung zu evaluieren. Der Fakt, dass gleiche Klone sowohl im Trainings- als auch im Evaluationsdatensatz vorkommen, führt zu deutlich besseren Ergebnissen, als maschinelle Lernwerkzeuge auf ungesehenen Daten erreichen würden. Die auf dem CDLH-Teildatensatz trainierten maschinellen Lernverfahren produzieren auf ungesehenen Daten schlechtere Ergebnisse und können somit schlechter generalisieren, als es die Ergebnisse in Tabelle 7.1 vermuten lassen. Andere Arbeiten [8, 88, 143] unterstützen die Erkenntnisse aus diesem Kapitel.

Kapitel 8

Zusammenfassung

In der hier vorgelegten Arbeit wurde ein neues Klonerkennungsverfahren vorgestellt, das Codeklone mithilfe von Pfadmengen identifiziert, die aus den Dominatorbäumen der Funktionen abgeleitet werden. Das Verfahren wurde in der Klonerkennungssoftware StoneDetector implementiert. Die erste These dieser Arbeit lautet:

> Dominatorinformationen können zur Quelltext-Klonerkennung verwendet werden und erzielen bei hohen Precision-Werten vergleichbare oder bessere Recall-Ergebnisse als bisher bekannte Klonerkennungsverfahren.

Der im Kapitel 4 vorgestellte allgemeine Dominatorbaum-basierte Ansatz zur Codeklonerkennung wurde im StoneDetector-System umgesetzt und validiert. Dazu wurde im Kapitel 5 die Quelltext-basierte Variante genutzt, um die Recall-Werte auf dem BigCloneBench zu ermitteln. Deckard [63], SourcererCC [131], NiCad [36, 126], iClones [54], CCAligner [166] und Oreo [130] wurden dabei unter den gleichen Bedingungen wie der StoneDetector getestet und die entsprechenden Recall-Werte der vier verschiedenen Klontypen in dem BigCloneBench gemessen. Insgesamt konnte festgestellt werden, dass sowohl StoneDetector mit ganzen Pfaden als auch mit gesplitteten Pfaden gleiche oder bessere Recall-Werte erzielt, als die anderen Anwendungen. Einzige Ausnahme bildet hierbei Deckard, das bei Typ-4-Klonen deutlich mehr Klonpaare identifiziert hat, was allerdings in dem schlechten Precision-Wert und der hohen Anzahl an gefundenen Funktionspaaren des Werkzeugs begründet liegt. Für alle Klondetektoren wurden 400 identifizierte Klonpaare blind, d. h. ohne Offenlegung des Werkzeugs, von drei menschlichen „Juroren“ unabhängig voneinander überprüft. Dabei wurde nicht nur festgestellt, ob es sich tatsächlich um ein Klonpaar handelt, sondern auch, um welchen Klon-Typ es sich handelt. Insgesamt

erreichte StoneDetector mit ganzen Pfaden einen Precision-Wert von 0,99 und mit gesplitteten Pfaden einen Precision-Wert von 0,97. Also sind 99% beziehungsweise 97% der von StoneDetector als Klone identifizierte Funktionspaare tatsächlich Klone. Insgesamt lässt sich also feststellen, dass die erste These wahr ist.

Die zweite These lautet:

> Klonerkennungsverfahren unter Verwendung von Dominatorbäumen entdecken im Gegensatz zu bisherigen Verfahren vermehrt auch Klone vom Typ-3 und Typ-4, insbesondere Subklone und strukturelle Klone.

Wie die aus dem BigCloneBench und damit aus veröffentlichten quelloffenen Projekten entnommenen Beispiele aus Abbildung 5.2 und aus Abbildung 5.3 zeigen, kann StoneDetector strukturelle Klone und Subklone erkennen. Wie unter den Beispielen beschrieben wurde, benötigen andere Klonerkennungsverfahren deutlich abgeschwächte Schwellenwerte, um diese Klone zu erkennen. Ein abgeschwächter Schwellenwert führt zu einer Verschlechterung des Precision-Wertes. Da sich diese beiden Beispiele in die Kategorie der Typ-3- beziehungsweise Typ-4-Klone einordnen lassen, kann der zweiten These zugestimmt werden.

Die dritte These lautet wie folgt:

> Die Verwendung von Dominatorbäumen bei der Quelltext-Klonerkennung ist effizient durchführbar und zudem auf große Datenmengen anwendbar.

Das in der Tabelle 5.6 aufgeführte Skalierbarkeitsexperiment mit StoneDetector und einigen anderen Klonerkennungswerkzeugen zeigt, dass StoneDetector auf 1-Millionen, 10-Millionen und 100-Millionen Quellcodezeilen angewendet werden kann und dabei schneller oder ähnlich schnell im Vergleich zu den anderen Verfahren ist. Ein weiteres Experiment hat gezeigt, dass StoneDetector sogar auf dem kompletten *IJaDataset 2.0* mit 328-Millionen Codezeilen angewendet werden kann und die Klonsuche in 79 Stunden beendet hatte. Somit kann auch der dritten These zugestimmt werden.

Da mithilfe des Stubber-Tools der BigCloneBench zu großen Teilen kompiliert werden konnte und somit ein Bytecode-basierter Benchmark zur Verfügung steht, konnte

der Dominatorbaum-basierte Klonerkennungsansatz auch für Stackcode beziehungsweise Registercode implementiert und getestet werden. Es zeigte sich, dass sowohl der Stackcode-basierte als auch der Registercode-basierte Ansatz zur Klonerkennung geeignet sind. Mit dem Registercode-basierten Ansatz konnten sogar ähnliche Ergebnisse im Vergleich zu dem Sourcecode-basierten StoneDetector-Ansatz erzielt werden. Dabei sind die Laufzeiten zwar etwas schlechter, nicht jedoch die Recall- oder die Precision-Werte. Die längere Laufzeit entsteht dabei durch die Verwendung des Soot-Frameworks [91,158], welches zur Verarbeitung des Java-Bytecodes genutzt wurde. Durch diese vorgestellten Ergebnisse kann auch der folgenden vierten These zugestimmt werden.

> Für bereits kompilierte Programmbeschreibungen, die in Stackcode oder Registercode vorliegen, können effizient die Quelltext-Klone abgeleitet werden.

Da einige maschinelle Lernverfahren angeben, sehr hohe Recall- und Precision-Werte bei allen Klontypen zu haben, wurde das maschinelle Lernverfahren der *Graph-Matching-Networks* mit Dominatorbäumen trainiert. Es konnten ähnlich gute Ergebnisse im Vergleich zu den Referenzarbeiten erzielt werden. Die dritte These lautet:

> Dominatorbäume können zur Klonerkennung mit maschinellen Lernverfahren verwendet werden.

Aufgrund der Reproduktion der Ergebnisse von Vergleichsarbeiten mit *Graph-Matching-Natworks* und Dominatorbäumen kann dieser fünften These zugestimmt werden. Um zu überprüfen, ob diese maschinellen Lernverfahren auch ihre guten Ergebnisse auf ungesehenen Daten erzielen würden, wurden einige Experimente durchgeführt. Zum Trainieren und Evaluieren der maschinellen Lernverfahren wurde in der Literatur der CDLH-Datensatz genutzt. Dieser Datensatz ist eine eingeschränkte Teilmenge des BigCloneBench. Die Experimente haben gezeigt, dass die Recall- und Precision-Werte des maschinellen Lernverfahrens schlechter werden, je strenger der Trainings- und der Evaluationsdatensatz voneinander getrennt wurden. Bei dem häufig verwendeten CDLH-Datensatz waren Funktionspaare sowohl im Trainings- als auch im Evaluationsdatensatz vorhanden, was zu einem Auswendiglernen des neuronalen Netzes führt. Der folgenden sechsten These kann also zugestimmt werden:

Bisher beim maschinellen Lernen verwendete Benchmarkprogramme, wie beispielsweise der CDLH-Datensatz, sind nicht geeignet, um maschinelle Lernverfahren zur Klonerkennung gleichzeitig zu trainieren und zu evaluieren.

Weiterführende Arbeiten

Eine mögliche Erweiterung des StoneDetector-Systems wäre es, weitere Programmiersprachen und Programmcodeformen als Eingabe zu integrieren. So könnten Programmiersprachen wie C, C++ und C# in das System aufgenommen und implementiert werden. Auch kompilierte Programme, wie der Binärcode eines C-Programms, wären eine interessante Erweiterungsmöglichkeit. Dabei könnte es sich als praktisch erweisen, alle Eingabeformate auf ein einheitliches internes Format abzubilden, mit dem dann Codeklone gesucht werden. Spannende Arbeiten in diesem Zusammenhang könnten sein, Klone zwischen Programmiersprachen oder Eingabeformaten hinweg zu identifizieren. Dies wäre durch die einheitliche Abbildung auf ein internes Format möglich.
Eine zusätzliche Verbesserung des StoneDetector-Systems wäre es, wenn mehr Abstandsmaße zur Verfügung ständen. Beispielsweise sollen Metriken wie das *Program-Slicing* oder *Def-Use*-Ketten in Pfaden und Beschreibungmengen integriert werden, was zu einer Verbesserung der Recall-Werte führen könnte. Weiterhin könnten beispielsweise die Damerau-Levenshtein-Distanz [39] oder die gewichtete Levenshtein-Distanz [94] integriert und auch ausreichend getestet werden. Gerade wenn nach Klonen mit großen Lücken gesucht werden soll, würde eine gewichtete Levenshtein-Distanz von Vorteil sein. Die Lücken könnten dann ein deutlich geringeres Gewicht bei der Bewertung auf Übereinstimmung bekommen, wobei auf diese Art mehr Klone mit großen Lücken gefunden werden könnten.
StoneDetector ist theoretisch in der Lage, interessante strukturelle Klone und Subklone sowie Klone mit großen Lücken zu finden. Zurzeit gibt es jedoch keinen geeigneten Benchmark für diese Codeklonarten, denn auch im BigCloneBench kann keine Unterscheidung zwischen „normalen“ und Klonen mit großen Lücken, strukturellen Klonen oder Subklonen getroffen werden. Es müsste ein neuer Klonerkennungsbenchmark entwickelt werden, der eben auch Klone mit großen Lücken, strukturelle Klone und Subklone unterstützt, der aber weiterhin auch die Nachteile des BigCloneBench vermeidet. Mithilfe eines solchen Benchmarks könnten noch deutlich mehr Vorteile von StoneDetector herausgearbeitet werden. Es sollte dann mit diesem

Benchmark möglich sein, auch die Precision-Werte eines Klonerkennungswerkzeuges zu messen und er sollte auch zum Trainieren und Evaluieren von maschinellen Lernverfahren geeignet sein. In der Arbeit von Krinke und Ragkhitwetsagul [88] wurde zudem eine Evaluation von 100 Typ-4-Codeklonen aus dem BigCloneBench vorgenommen, von denen nach Aussage der Autoren 86 als keine Klone zu werten sind. Hier müsste es in einem neuen Benchmark genauere Definitionen zu Typ-4-Klonen geben. So wäre es möglich, weiterführende Evaluationen und Optimierungen am StoneDetector-System vorzunehmen.

Literaturverzeichnis

[1] Java Example Programs. https://www.cs.uic.edu/~sloan/CLASSES/java/.

[2] Akshat Agrawal and Sumit Kumar Yadav. A hybrid-token and textual based approach to find similar code segments. In *2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*, pages 1–4, July 2013.

[3] Farouq Al-Omari, Chanchal K. Roy, and Tonghao Chen. SemanticCloneBench: A Semantic Code Clone Benchmark using Crowd-Source Knowledge. In *2020 IEEE 14th International Workshop on Software Clones (IWSC)*, pages 57–63, February 2020. ISSN: 2572-6587.

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40:1–40:29, January 2019.

[5] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo. Modeling clones evolution through time series. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, pages 273–280, November 2001. ISSN: 1063-6773.

[6] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta. Analyzing cloning evolution in the Linux kernel. *Information and Software Technology*, 44(13):755–765, October 2002.

[7] T. Apiwattanapong, A. Orso, and M.J. Harrold. A differencing algorithm for object-oriented programs. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pages 2–13, Linz, Austria, 2004. IEEE.

[8] Saad Arshad, Shamsa Abid, and Shafay Shamail. CodeBERT for Code Clone Detection: A Replication Study. In *2022 IEEE 16th International Workshop on Software Clones (IWSC)*, pages 39–45, October 2022. ISSN: 2572-6587.

[9] Muhammad Asaduzzaman, Chanchal K. Roy, and Kevin A. Schneider. VisCad: flexible code clone analysis support for NiCad. In *Proceeding of the 5th international workshop on Software clones - IWSC '11*, page 77, Waikiki, Honolulu, HI, USA, 2011. ACM Press.

[10] Arutyun Avetisyan, Shamil Kurmangaleev, Sevak Sargsyan, Mariam Arutunian, and Andrey Belevantsev. LLVM-based code clone detection framework. In *2015 Computer Science and Information Technologies (CSIT)*, pages 100–104, September 2015.

[11] Blaine Ayotte, Mahesh K. Banavar, Daqing Hou, and Stephanie Schuckers. Group Leakage Overestimates Performance: A Case Study in Keystroke Dynamics. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 1410–1417, Nashville, TN, USA, June 2021. IEEE.

[12] Brenda S. Baker. A program for identifying duplicated code. In *Computer Science and Statistics: Proc. Symp. on the Interface*, pages 49–57, March 1992.

[13] Brenda S. Baker. Finding Clones with Dup: Analysis of an Experiment. *IEEE Transactions on Software Engineering*, 33(9):608–621, September 2007. Conference Name: IEEE Transactions on Software Engineering.

[14] B.S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of 2nd Working Conference on Reverse Engineering*, pages 86–95, July 1995.

[15] Pierre Baldi and Yves Chauvin. Neural Networks for Fingerprint Recognition. *Neural Computation*, 5(3):402–418, May 1993. Conference Name: Neural Computation.

[16] Venkat Bandi, Chanchal K. Roy, and Carl Gutwin. Clone Swarm: A Cloud Based Code-Clone Analysis Tool. In *2020 IEEE 14th International Workshop on Software Clones (IWSC)*, pages 52–56, February 2020. ISSN: 2572-6587.

[17] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders, April 2021. arXiv:2003.05991 [cs, stat].

[18] Hamid Abdul Basit and Stan Jarzabek. Efficient token based clone detection with flexible tokenization. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 513–516, New York, NY, USA, September 2007. Association for Computing Machinery.

[19] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377, November 1998. ISSN: 1063-6773.

[20] Stefan Bellon. Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Institute for Software Technology, September 2002. University of Stuttgart (Germany).

[21] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, September 2007. Conference Name: IEEE Transactions on Software Engineering.

[22] Donald J. Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, AAAIWS'94, pages 359–370, Seattle, WA, July 1994. AAAI Press.

[23] Anupam Bhattacharjee and Hasan M. Jamil. CodeBlast: a two-stage algorithm for improved program similarity matching in large software repositories. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing - SAC '13*, page 846, Coimbra, Portugal, 2013. ACM Press.

[24] Yixin Bian, Gunes Koru, Xiaohong Su, and Peijun Ma. SPAPE: A semantic-preserving amorphous procedure extraction method for near-miss clones. *Journal of Systems and Software*, 86(8):2077–2093, August 2013.

[25] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral Networks and Locally Connected Networks on Graphs, May 2014. arXiv:1312.6203 [cs].

[26] Michael Buckland and Fredric Gey. The relationship between Recall and Precision. *Journal of the American Society for Information Science*, 45(1):12–19, 1994. _eprint:

https://onlinelibrary.wiley.com/doi/pdf/10.1002/%28SICI%291097-4571%28199401%2945%3A1%3C12%3A%3AAID-ASI2%3E3.0.CO%3B2-L.

[27] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Second IEEE International Workshop on Source Code Analysis and Manipulation Proceedings*, pages 36–43, October 2002.

[28] E. Burd and M. Munro. Investigating the maintenance implications of the replication of code. In *1997 Proceedings International Conference on Software Maintenance*, pages 322–329, October 1997. ISSN: 1063-6773.

[29] Lutz Büch and Artur Andrzejak. Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 95–104, February 2019. ISSN: 1534-5351.

[30] Alejandro Calleja, Juan Tapiador, and Juan Caballero. The MalSource Dataset: Quantifying Complexity and Code Reuse in Malware Development. *IEEE Transactions on Information Forensics and Security*, 14(12):3175–3190, December 2019. Conference Name: IEEE Transactions on Information Forensics and Security.

[31] Alan Charpentier, Jean-Rémy Falleri, David Lo, and Laurent Réveillère. An empirical assessment of Bellon's clone benchmark. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, EASE '15, pages 1–10, New York, NY, USA, April 2015. Association for Computing Machinery.

[32] Jian Chen, Manar H. Alalfi, Thomas R. Dean, and Ying Zou. Detecting Android Malware Using Clone Detection. *Journal of Computer Science and Technology*, 30(5):942–956, September 2015.

[33] Long Chen, Wei Ye, and Shikun Zhang. Capturing source code semantics via tree-based convolution over API-enhanced AST. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pages 174–182, Alghero Italy, April 2019. ACM.

[34] A. Chin. Locality-preserving hash functions for general purpose parallel computation. *Algorithmica*, 12(2):170–181, September 1994.

[35] James R. Cordy, Thomas R. Dean, and Nikita Synytskyy. Practical language-independent detection of near-miss clones. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '04, pages 1–12, Markham, Ontario, Canada, October 2004. IBM Press.

[36] James R. Cordy and Chanchal K. Roy. The NiCad Clone Detector. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 219–220, June 2011. ISSN: 1092-8138.

[37] Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, and Clifford Stein. *Algorithmen - eine Einführung.* Oldenbourg, Deutschland, 4. edition, 2013.

[38] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, September 1995.

[39] Fred J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, March 1964.

[40] Andreas Dann, Ben Hermann, and Eric Bodden. SootDiff: bytecode comparison across different Java compilers. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis - SOAP 2019*, pages 14–19, Phoenix, AZ, USA, 2019. ACM Press.

[41] N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley. The development of a software clone detector. *International Journal of Applied Software Technology*, 1995. Accepted: 2007-09-06T15:33:29Z.

[42] Ekwa Duala-Ekoko and Martin P. Robillard. Tracking Code Clones in Evolving Software. In *29th International Conference on Software Engineering (ICSE'07)*, pages 158–167, May 2007. ISSN: 1558-1225.

[43] Raimar Falke, Pierre Frenzel, and Rainer Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Software Engineering*, 13(6):601–643, December 2008.

[44] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 516–527, Virtual Event USA, July 2020. ACM.

[45] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. Technical Report arXiv:2002.08155, arXiv, September 2020. arXiv:2002.08155 [cs] type: article.

[46] B. Fluri and H.C. Gall. Classifying Change Types for Qualifying Change Couplings. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 35–45, June 2006. ISSN: 1092-8138.

[47] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*, page 321, Leipzig, Germany, 2008. ACM Press.

[48] Yi Gao, Zan Wang, Shuang Liu, Lin Yang, Wei Sang, and Yuanfang Cai. TECCD: A Tree Embedding Approach for Code Clone Detection. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 145–156, September 2019. ISSN: 2576-3148.

[49] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54, Berlin Germany, November 2013. ACM.

[50] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for Quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 1263–1272, Sydney, NSW, Australia, August 2017. JMLR.org.

[51] Chenkai Guo, Hui Yang, Dengrong Huang, Jianwen Zhang, Naipeng Dong, Jing Xu, and Jingwen Zhu. Review Sharing via Deep Semi-Supervised Code Clone Detection. *IEEE Access*, 8:24948–24965, 2020. Conference Name: IEEE Access.

[52] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. GRAPHCODEBERT: PRE-TRAINING CODE REPRESEN- TATIONS WITH DATA FLOW. page 18, 2021.

[53] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, and Ming Zhou. GraphCodeBERT: Pre-training Code Representations with Data Flow. *arXiv:2009.08366 [cs]*, September 2020.

[54] Nils Göde and Rainer Koschke. Incremental Clone Detection. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 219–228, March 2009. ISSN: 1534-5351.

[55] Muhammad Hammad, Önder Babur, Hamid Abdul Basit, and Mark Van Den Brand. Clone-Seeker: Effective Code Clone Search Using Annotations. *IEEE Access*, 10:11696–11713, 2022. Conference Name: IEEE Access.

[56] Yoshiki Higo and Shinji Kusumoto. Enhancing Quality of Code Clone Detection with Program Dependency Graph. In *2009 16th Working Conference on Reverse Engineering*, pages 315–316, October 2009. ISSN: 2375-5369.

[57] Yoshiki Higo and Shinji Kusumoto. How should we measure functional sameness from program source code? an exploratory study on Java methods. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 294–305, New York, NY, USA, November 2014. Association for Computing Machinery.

[58] Yoshiki Higo, Ueda Yasushi, Minoru Nishino, and Shinji Kusumoto. Incremental Code Clone Detection: A PDG-based Approach. In *2011 18th Working Conference on Reverse Engineering*, pages 3–12, October 2011. ISSN: 2375-5369.

[59] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. Binary Code Clone Detection across Architectures and Compiling Configurations. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 88–98, May 2017.

[60] Yikun Hu, Yuanyuan Zhang, Juanru Li, Hui Wang, Bodong Li, and Dawu Gu. BinMatch: A Semantics-Based Hybrid Approach on Binary Code Clone Analysis. pages 104–114. IEEE Computer Society, September 2018.

[61] Wei Hua, Yulei Sui, Yao Wan, Guangzhong Liu, and Guandong Xu. FCCA: Hybrid Code Representation for Functional Clone Detection Using Attention Networks. *IEEE Transactions on Reliability*, 70(1):304–318, March 2021. Conference Name: IEEE Transactions on Reliability.

[62] Shruti Jadon. Code clones detection using machine learning technique: Support vector machine. In *2016 International Conference on Computing, Communication and Automation (ICCCA)*, pages 399–303, April 2016.

[63] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105, Minneapolis, MN, USA, May 2007. IEEE. ISSN: 0270-5257.

[64] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis - ISSTA '09*, page 81, Chicago, IL, USA, 2009. ACM Press.

[65] Johnson. Substring matching for clone detection and change tracking. In *Proceedings 1994 International Conference on Software Maintenance*, pages 120–126, September 1994.

[66] J. Howard Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering - Volume 1*, CASCON '93, pages 171–183, Toronto, Ontario, Canada, October 1993. IBM Press.

[67] J. Howard Johnson. Visualizing textual redundancy in legacy source. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '94, page 32, Toronto, Ontario, Canada, October 1994. IBM Press.

[68] J. Howard Johnson. Navigating the textual redundancy web in legacy source. In *Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '96, page 16, Toronto, Ontario, Canada, November 1996. IBM Press.

[69] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002. Conference Name: IEEE Transactions on Software Engineering.

[70] Toshihiro Kamiya. An execution-semantic and content-and-context-based code-clone detection and analysis. In *2015 IEEE 9th International Workshop on Software Clones (IWSC)*, pages 1–7, March 2015.

[71] C. Kapser and M.W. Godfrey. Aiding comprehension of cloning through categorization. In *Proceedings. 7th International Workshop on Principles of Software Evolution, 2004.*, pages 85–94, September 2004. ISSN: 1550-4077.

[72] Cory J. Kapser and Michael W. Godfrey. Supporting the analysis of clones in software systems. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):61–82, 2006. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.327.

[73] Cory J. Kapser and Michael W. Godfrey. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, December 2008.

[74] Iman Keivanloo, Chanchai K. Roy, and Juergen Rilling. Java bytecode clone detection via relaxation on code fingerprint and Semantic Web reasoning. In *2012 6th International Workshop on Software Clones (IWSC)*, pages 36–42, Zurich, Switzerland, June 2012. IEEE.

[75] Iman Keivanloo, Chanchal K. Roy, and Juergen Rilling. SeByte: A semantic clone detection tool for intermediate languages. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 247–249, June 2012. ISSN: 1092-8138.

[76] Iman Keivanloo, Chanchal K. Roy, and Juergen Rilling. SeByte: Scalable clone and similarity search for bytecode. *Science of Computer Programming*, 95:426–444, December 2014.

[77] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. MeCC: memory comparison-based clone detector. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 301–310, May 2011. ISSN: 1558-1225.

[78] Jinhyun Kim, HyukGeun Choi, Hansang Yun, and Byung-Ro Moon. Measuring Source Code Similarity by Finding Similar Subgraph with an Incremental Genetic Algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 925–932, Denver Colorado USA, July 2016. ACM.

[79] Miryung Kim, Vibha Sazawal, David Notkin, and Gail C Murphy. An Empirical Study of Code Clone Genealogies. page 10, 2005.

[80] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614, May 2017. ISSN: 2375-1207.

[81] Egambaram Kodhai and Selvadurai Kanmani. Method-level code clone detection through LWH (Light Weight Hybrid) approach. *Journal of Software Engineering Research and Development*, 2(1):12, December 2014.

[82] R. Komondoor and S. Horwitz. Effective, automatic procedure extraction. In *11th IEEE International Workshop on Program Comprehension, 2003.*, pages 33–42, May 2003. ISSN: 1092-8138.

[83] Raghavan Komondoor and Susan Horwitz. Using Slicing to Identify Duplication in Source Code. In Patrick Cousot, editor, *Static Analysis*, Lecture Notes in Computer Science, pages 40–56, Berlin, Heidelberg, 2001. Springer.

[84] Raghavan Komondoor and Susan Horwitz. Using Slicing to Identify Duplication in Source Code. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Patrick Cousot, editors, *Static Analysis*, volume 2126, pages 40–56. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. Series Title: Lecture Notes in Computer Science.

[85] Rainer Koschke. Large-Scale Inter-System Clone Detection Using Suffix Trees. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 309–318, March 2012. ISSN: 1534-5351.

[86] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. In *2006 13th Working Conference on Reverse Engineering*, pages 253–262, October 2006. ISSN: 2375-5369.

[87] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 301–309, October 2001. ISSN: 1095-1350.

[88] Jens Krinke and Chaiyong Ragkhitwetsagul. BigCloneBench Considered Harmful for Machine Learning. In *2022 IEEE 16th International Workshop on Software Clones (IWSC)*, pages 1–7, October 2022. ISSN: 2572-6587.

[89] Daniel E. Krutz and Wei Le. A code clone oracle. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 388–391, New York, NY, USA, May 2014. Association for Computing Machinery.

[90] B. Lague, D. Proulx, J. Mayrand, E.M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *1997 Proceedings International Conference on Software Maintenance*, pages 314–321, October 1997. ISSN: 1063-6773.

[91] Patrick Lam, Eric Bodden, Ondrej Lhotak, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. *http://www.sable.mcgill.ca/publications/papers/2011-6/11.cetus.soot.pdf*, 2011.

[92] Thierry Lavoie and Ettore Merlo. Automated type-3 clone oracle using Levenshtein metric. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 34–40, New York, NY, USA, May 2011. Association for Computing Machinery.

[93] Yann LeCun, Bernhard Boser, John Denker, Donnie Henderson, R. Howard, Wayne Hubbard, and Lawrence Jackel. Handwritten Digit Recognition with a Back-Propagation Network. In *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1989.

[94] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, February 1966. ADS Bibcode: 1966SPhD...10..707L.

[95] V. I. Levenshtein. On the minimal redundancy of binary error-correcting codes. *Information and Control*, 28(4):268–291, August 1975.

[96] Esther Levin. A recurrent neural network: Limitations and training. *Neural Networks*, 3(6):641–650, January 1990.

[97] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. CCLearner: A Deep Learning-Based Clone Detection Approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260, September 2017.

[98] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph Matching Networks for Learning the Similarity of Graph Structured Objects. *arXiv:1904.12787 [cs, stat]*, May 2019. arXiv: 1904.12787.

[99] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, March 2006. Conference Name: IEEE Transactions on Software Engineering.

[100] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA, December 2004. USENIX Association.

[101] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. GPLAG: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '06*, page 872, Philadelphia, PA, USA, 2006. ACM Press.

[102] Jinze Liu, Tao Wang, Chenhui Feng, Huaimin Wang, and Dongsheng Li. A Large-Gap Clone Detection Approach Using Sequence Alignment via Dynamic Parameter Optimization. *IEEE Access*, 7:131270–131281, 2019. Conference Name: IEEE Access.

[103] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. DéjàVu: a map of code duplicates on GitHub. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28, October 2017.

[104] Niccolò Marastoni, Andrea Continella, Davide Quarta, Stefano Zanero, and Mila Dalla Preda. GroupDroid: Automatically Grouping Mobile Malware by Extracting Code Similarities. In *Proceedings of the 7th Software Security, Protection, and Reverse Engineering / Software Security and Protection Workshop*, pages 1–12, Orlando FL USA, December 2017. ACM.

[105] Mayrand, Leblanc, and Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *1996 Proceedings of International Conference on Software Maintenance*, pages 244–253, November 1996. ISSN: 1063-6773.

[106] Yao Meng and Long Liu. A Deep Learning Approach for a Source Code Detection Model Using Self-Attention. *Complexity*, 2020:1–15, September 2020.

[107] E. Merlo, M. Dagenais, P. Bachand, J.S. Sormani, S. Gradara, and G. Antoniol. Investigating large software system evolution: the Linux kernel. In *Proceedings 26th Annual International Computer Software and Applications*, pages 421–426, August 2002. ISSN: 0730-3157.

[108] Md Rakib Hossain Misu and Kazi Sakib. Interface Driven Code Clone Detection. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 747–748, December 2017.

[109] B.S. Mitchell and S. Mancoridis. CRAFT: a framework for evaluating software clustering results in the absence of benchmark decompositions [Clustering Results Analysis Framework and Tools]. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 93–102, October 2001. ISSN: 1095-1350.

[110] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proceedings Eighth IEEE Symposium on Software Metrics*, pages 87–94, June 2002. ISSN: 1530-1435.

[111] Hiroaki Murakami, Yoshiki Higo, and Shinji Kusumoto. A dataset of clone references with gaps. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, pages 412–415, Hyderabad, India, 2014. ACM Press.

[112] Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. CLCDSA: Cross Language Code Clone Detection using Syntactical Features and API Documentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1026–1037, November 2019. ISSN: 2643-1572.

[113] Hamid Nasirloo and Fatemeh Azimzadeh. Semantic code clone detection using abstract memory states and program dependency graphs. In *2018 4th International Conference on Web Research (ICWR)*, pages 19–27, April 2018.

[114] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. Clone Management for Evolving Software. *IEEE Transactions on Software Engineering*, 38(5):1008–1026, September 2012. Conference Name: IEEE Transactions on Software Engineering.

[115] Daniel Perez and Shigeru Chiba. Cross-Language Clone Detection by Learning Over Abstract Syntax Trees. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 518–528, May 2019. ISSN: 2574-3864.

[116] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-Architecture Bug Search in Binary Executables. In *2015 IEEE Symposium on Security and Privacy*, pages 709–724, May 2015. ISSN: 2375-1207.

[117] Lutz Priese and Katrin Erk. *Theoretische Informatik - Eine umfassende Einführung*. Springer-Verlag, Berlin Heidelberg New York, 4. edition, 2018.

[118] Chaiyong Ragkhitwetsagul and Jens Krinke. Siamese: scalable and incremental code clone search via multiple code representations. *Empirical Software Engineering*, 24(4):2236–2284, August 2019.

[119] Md. Saidur Rahman and Chanchal K. Roy. A Change-Type Based Empirical Study on the Stability of Cloned Code. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 31–40, September 2014.

[120] Sandeep Reddivari and Mohammed Salman Khan. CloneTM: A Code Clone Detection Tool Based on Latent Dirichlet Allocation. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 930–931, July 2019. ISSN: 0730-3157.

[121] Saif Ur Rehman, Kamran Khan, Simon Fong, and Robert Biuk-Aghai. An efficient new multi-language clone detection approach from large source code. In *2012 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 937–940, October 2012. ISSN: 1062-922X.

[122] Matthias Rieger. *Effective Clone Detection Without Language Barriers*. dissertation, University of Bern, June 2005. Publication Title: Rieger, Matthias (2005). Effective Clone Detection Without Language Barriers. (Dissertation, University of Bern, Philosophisch-naturwissenschaftliche Fakultät).

[123] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. Function Merging by Sequence Alignment. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 149–163, February 2019.

[124] Roopam and Gurpreet Singh. To enhance the code clone detection algorithm by using hybrid approach for detection of code clones. In *2017 International Conference on Intelligent Computing and Control Systems (ICICCS)*, pages 192–198, June 2017.

[125] Chanchal K. Roy and James R. Cordy. An Empirical Study of Function Clones in Open Source Software. In *2008 15th Working Conference on Reverse Engineering*, pages 81–90, October 2008. ISSN: 2375-5369.

[126] Chanchal K. Roy and James R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *2008 16th IEEE International Conference on Program Comprehension*, pages 172–181, June 2008. ISSN: 1092-8138.

[127] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, May 2009.

[128] Chanchal Kumar Roy and James R Cordy. A Survey on Software Clone Detection Research. *School of Computing*, Queens University(Tech. Rep. TR 2007-541):115.

[129] Tobias Sager, Abraham Bernstein, Martin Pinzger, and Christoph Kiefer. Detecting similar Java classes using tree algorithms. In *Proceedings of the 2006 international workshop on Mining software repositories - MSR '06*, page 65, Shanghai, China, 2006. ACM Press.

[130] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina Lopes. Oreo: Detection of Clones in the Twilight Zone. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 354–365, October 2018. arXiv: 1806.05837.

[131] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. SourcererCC: Scaling Code Clone Detection to Big-Code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1157–1168, May 2016. ISSN: 1558-1225.

[132] S. Sargsyan, Sh. Kurmangaleev, A. Belevantsev, and A. Avetisyan. Scalable and accurate detection of code clones. *Programming and Computer Software*, 42(1):27–33, January 2016.

[133] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1):61–80, January 2009. Conference Name: IEEE Transactions on Neural Networks.

[134] André Schäfer, Wolfram Amme, and Thomas S. Heinze. Detection of Similar Functions Through the Use of Dominator Information. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, pages 206–211, August 2020.

[135] André Schäfer, Wolfram Amme, and Thomas S. Heinze. Stubber: Compiling Source Code into Bytecode without Dependencies for Java Code Clone Detection. In *2021 IEEE 15th International Workshop on Software Clones (IWSC)*, pages 29–35, October 2021. ISSN: 2572-6587.

[136] André Schäfer, Wolfram Amme, and Thomas S. Heinze. Experiments on Code Clone Detection and Machine Learning. In *2022 IEEE 16th International Workshop on Software Clones (IWSC)*, pages 46–52, October 2022. ISSN: 2572-6587.

[137] André Schäfer, Thomas S. Heinze, and Wolfram Amme. You Look so Different: Finding Structural Clones and Subclones in Java Source Code. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 70–80, September 2021. ISSN: 2576-3148.

[138] Gehan M.K. Selim, King Chun Foo, and Ying Zou. Enhancing Source-Based Clone Detection Using Intermediate Representation. In *2010 17th Working Conference on Reverse Engineering*, pages 227–236, October 2010. ISSN: 2375-5369.

[139] Abdullah Sheneamer. CCDLC Detection Framework-Combining Clustering with Deep Learning Classification for Semantic Clones. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 701–706, December 2018.

[140] Abdullah M. Sheneamer. Multiple Similarity-based Features Blending for Detecting Code Clones using Consensus-Driven Classification. *Expert Systems with Applications*, 183:115364, November 2021.

[141] Alex Sherstinsky. Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network. *Physica D: Nonlinear Phenomena*, 404:132306, March 2020.

[142] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, March 1981.

[143] Tim Sonnekalb, Bernd Gruner, Clemens-Alexander Brust, and Patrick Mäder. Generalizability of Code Clone Detection on CodeBERT. In *37th IEEE/ACM International Conference on Automated Software Engineering*, ASE22, pages 1–3, New York, NY, USA, January 2023. Association for Computing Machinery.

[144] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. Code relatives: detecting similarly behaving software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 702–714, Seattle WA USA, November 2016. ACM.

[145] M Sudhamani and Lalitha Rangarajan. Code clone detection based on order and content of control statements. In *2016 2nd International Conference on Contemporary Computing and Informatics (IC3I)*, pages 59–64, December 2016.

[146] M. Sudhamani and Lalitha Rangarajan. Code similarity detection through control statement and program features. *Expert Systems with Applications*, 132:63–75, October 2019.

[147] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480, September 2014. ISSN: 1063-6773.

[148] Jeffrey Svajlenko and Chanchal K. Roy. Evaluating Modern Clone Detection Tools. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 321–330, September 2014. ISSN: 1063-6773.

[149] Jeffrey Svajlenko and Chanchal K. Roy. Evaluating clone detection tools with BigCloneBench. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 131–140, September 2015.

[150] Jeffrey Svajlenko and Chanchal K. Roy. BigCloneEval: A Clone Detection Tool Evaluation Framework with BigCloneBench. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 596–600, October 2016.

[151] Jeffrey Svajlenko and Chanchal K. Roy. The Mutation and Injection Framework: Evaluating Clone Detection Tools with Mutation Analysis. *IEEE Transactions on Software Engineering*, 47(5):1060–1087, May 2021. Conference Name: IEEE Transactions on Software Engineering.

[152] Rajkumar Tekchandani, Rajesh Bhatia, and Maninder Singh. Semantic code clone detection for Internet of Things applications using reaching definition and liveness analysis. *The Journal of Supercomputing*, 74(9):4199–4226, September 2018.

[153] Rajkumar Tekchandani, Rajesh Kumar Bhatia, and Maninder Singh. Semantic code clone detection using parse trees and grammar recovery. In *Confluence 2013: The Next Generation Information Technology Summit (4th International Conference)*, pages 41–46, September 2013.

[154] Dethe Tukaram and Uma Maheswari B. Design and Development of Software Tool for Code Clone Search, Detection, and Analysis. In *2019 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA)*, pages 1002–1006, June 2019.

[155] Esko Ukkonen. On approximate string matching. In Marek Karpinski, editor, *Foundations of Computation Theory*, Lecture Notes in Computer Science, pages 487–495, Berlin, Heidelberg, 1983. Springer.

[156] Farhan Ullah, Junfeng Wang, Muhammad Farhan, Sohail Jabbar, Zhiming Wu, and Shehzad Khalid. Plagiarism detection in students' programming assignments based on semantics: multimedia e-learning based smart assessment methodology. *Multimedia Tools and Applications*, 79(13-14):8581–8598, April 2020.

[157] Raja Vallee-Rai and Laurie J Hendren. Jimple: Simplifying Java Bytecode for Analyses and Transformations. page 15.

[158] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: a Java bytecode optimization framework. In

CASCON First Decade High Impact Papers on - CASCON '10, pages 214–224, Toronto, Ontario, Canada, 2010. ACM Press.

[159] F. Van Rysselberghe and S. Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pages 336–339, September 2004. ISSN: 1938-4300.

[160] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[161] Robert A. Wagner and Michael J. Fischer. The String-to-String Correction Problem. *Journal of the ACM*, 21(1):168–173, January 1974.

[162] Stefan Wagner, Asim Abdulkhaleq, Ivan Bogicevic, Jan-Peter Ostberg, and Jasmin Ramadani. How are functionally similar code clones syntactically different? An empirical study and a benchmark. *PeerJ Computer Science*, 2:e49, March 2016. Publisher: PeerJ Inc.

[163] A. Walenstein, N. Jyoti, Junwei Li, Yun Yang, and A. Lakhotia. Problems creating task-relevant clone detection reference data. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.*, pages 285–294, November 2003. ISSN: 1095-1350.

[164] Andrew Walenstein and Arun Lakhotia. The Software Similarity Problem in Malware Analysis. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, volume 6301 of *Dagstuhl Seminar Proceedings (DagSemProc)*, pages 1–10, Dagstuhl, Germany, 2007. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISSN: 1862-4405.

[165] Min Wang, Pengcheng Wang, and Yun Xu. CCSharp: An Efficient Three-Phase Code Clone Detector Using Modified PDGs. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 100–109, December 2017.

[166] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy. CCAligner: A Token Based Large-Gap Clone Detector. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1066–1077, May 2018. ISSN: 1558-1225.

[167] Tiantian Wang, Kechao Wang, Xiaohong Su, and Peijun Ma. Detection of semantically similar code. *Frontiers of Computer Science*, 8(6):996–1011, December 2014.

[168] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting Code Clones with Graph Neural Networkand Flow-Augmented Abstract Syntax Tree. *arXiv:2002.08653 [cs]*, February 2020. arXiv: 2002.08653.

[169] Hui-Hui Wei and Ming Li. Positive and Unlabeled Learning for Detecting Software Functional Clones with Adversarial Training. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, pages 2840–2846, Stockholm, Sweden, July 2018. International Joint Conferences on Artificial Intelligence Organization.

[170] Huihui Wei and Ming Li. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 3034–3040, Melbourne, Australia, August 2017. International Joint Conferences on Artificial Intelligence Organization.

[171] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 87–98, Singapore Singapore, August 2016. ACM.

[172] Niklaus Wirth. *Grundlagen und Techniken des Compilerbaus*. Oldenbourg, Deutschland, 3. edition, 2011.

[173] Ming Wu, Pengcheng Wang, Kangqi Yin, Haoyu Cheng, Yun Xu, and Chanchal K. Roy. LVMapper: A Large-Variance Clone Detector Using Sequencing Alignment Approach. *IEEE Access*, 8:27986–27997, 2020. Conference Name: IEEE Access.

[174] Yueming Wu, Deqing Zou, Shihan Dou, Siru Yang, Wei Yang, Feng Cheng, Hong Liang, and Hai Jin. SCDetector: Software Functional Clone Detection Based on Semantic Tokens Analysis. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 821–833, September 2020. ISSN: 2643-1572.

[175] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, Dallas Texas USA, October 2017. ACM.

[176] Jun Yang, Yu Xiong, and Jinxin Ma. A function level Java code clone detection method. In *2019 IEEE 4th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, volume 1, pages 2128–2134, December 2019. ISSN: 2381-0947.

[177] Yanming Yang, Zhilei Ren, Xin Chen, and He Jiang. Structural Function Based Code Clone Detection Using a New Hybrid Technique. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 01, pages 286–291, July 2018. ISSN: 0730-3157.

[178] Bo Yu, Zong-ben Xu, and Cheng-hua Li. Latent semantic analysis for text categorization using neural network. *Knowledge-Based Systems*, 21(8):900–904, December 2008.

[179] Dongjin Yu, Jie Wang, Qing Wu, Jiazha Yang, Jiaojiao Wang, Wei Yang, and Wei Yan. Detecting Java Code Clones with Multi-granularities Based on Bytecode. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 317–326, July 2017. ISSN: 0730-3157.

[180] Dongjin Yu, Jiazha Yang, Xin Chen, and Jie Chen. Detecting Java Code Clones Based on Bytecode Sequence Alignment. *IEEE Access*, 7:22421–22433, 2019. Conference Name: IEEE Access.

[181] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. Neural Detection of Semantic Code Clones Via Tree-Based Convolution. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 70–80, May 2019. ISSN: 2643-7171.

[182] Yuan Yuan, Weiqiang Kong, Gang Hou, Yan Hu, Masahiko Watanabe, and Akira Fukuda. From Local to Global Semantic Clone Detection. In *2019 6th International Conference on Dependable Systems and Their Applications (DSA)*, pages 13–24, January 2020.

[183] Yusuke Yuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Generating clone references with less human subjectivity. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–4, May 2016.

[184] Chen Zeng, Yue Yu, Shanshan Li, Xin Xia, Zhiming Wang, Mingyang Geng, Linxiao Bai, Wei Dong, and Xiangke Liao. DEGraphCS: Embedding Variable-based Flow Graph for Neural Code Search. *ACM Transactions on Software Engineering and Methodology*, page 3546066, July 2022.

[185] Jie Zeng, Kerong Ben, Xiaowei Li, and Xian Zhang. Fast Code Clone Detection Based on Weighted Recursive Autoencoders. *IEEE Access*, 7:125062–125078, 2019. Conference Name: IEEE Access.

[186] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794, May 2019. ISSN: 1558-1225.

[187] Gang Zhao and Jeff Huang. DeepSim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 141–151, New York, NY, USA, October 2018. Association for Computing Machinery.

[188] Hans Zima. *Supercompilers for Parallel and Vector Computers* -. ACM Press, Reno, 1990.

Abbildungsverzeichnis

1.1 Schwer zu identifizierender struktureller Klon. 2

2.1 Übersicht von Typ-1- bis Typ-3-Klonen. 12
2.2 Funktionsbeispiel. 14
2.3 Typ-1-Klon zu der Funktion in Abbildung 2.2. 14
2.4 Typ-2-Klon zu der Funktion in Abbildung 2.2. 14
2.5 Typ-3-Klon zu der Funktion in Abbildung 2.2. 14
2.6 Typ-4-Klon zu der Funktion in Abbildung 2.2. 14
2.7 Kontrollflussgraph zur Funktion aus Abbildung 2.2. 16
2.8 Dominatorbaum des Kontrollflussgraphen aus Abbildung 2.7. 19

4.1 Allgemeiner Aufbau des Systems zur Codeklonerkennung. 38
4.2 Dominatorbaum und die Beschreibungsmengen der Funktion aus Abbildung 2.2. 40

5.1 Funktion, Kontrollflussgraph, Dominatorbaum, Abstrakter Dominatorbaum und die Beschreibungsmengen der Funktion aus Abbildung 2.5. 52
5.2 Schwer zu erkennendes strukturelles Klonpaar. 53
5.3 Schwer zu erkennendes Klonpaar: Subklon. 54

6.1 Ordnerstruktur und Jar-Dateien, die vom Stubber-Tool erzeugt werden. 73
6.2 Originale Java-Methode und durch Stubber angepasste Methode zur Kompilierung. 75

6.3 Disassemblierte Annotation mit den Informationen. 77

6.4 Funktion aus Abbildung 2.5 mit dem entsprechenden Stackcode und dem Registercode. 81

6.5 Allgemeiner Aufbau des Systems zur Codeklonerkennung auf Basis von Java-Bytecode. 82

6.6 Beispielfunktionen und Dominatorbäume mit Stackcode-Instruktionen. 84

6.7 Beispielfunktionen und Dominatorbäume mit Registercode-Instruktionen. 87

7.1 Architektur-Illustration eines *Graph-Matching-Networks*. 97

Tabellenverzeichnis

5.1 Tooleinstellungen 55

5.2 Genutzte Einstellungen bei StoneDetector. 55

5.3 Recall-Werte mit Klontypen verschiedener Klonerkennungstools auf dem BigCloneBench. 56

5.4 Precision-Wert und Klonklassifikation für 400 Klonpaarbeispiele verschiedener Klonerkennungstools. 59

5.5 Gemeinsame und exklusive Codeklone für Klonerkennungstools. . . . 61

5.6 Skalierbarkeitsexperiment auf dem IJaDataset 2.0. 66

6.1 NiCads Recall-Werte auf den originalen und den durch das Stubber-Tool angepassten BigCloneBench-Dateien $\geq$ 10 Zeilen. 78

6.2 Zusammensetzung des BigCloneBench vor und nach der Anwendung des Stubber-Tools. 79

6.3 Recall-Werte des Sourcecode-basierten StoneDetector-Ansatzes auf den durch das Stubber-Tool angepassten Sourcecode $\geq$ 15 Zeilen. . . 80

6.4 Genutzte Einstellungen der Bytecode-basierten Ansätze. 83

6.5 Recall-Werte des Stackcode-basierten StoneDetector-Ansatzes für $\geq$ 15 Zeilen. 85

6.6 Recall-Werte des Registercode-basierten StoneDetector-Ansatzes für $\geq$ 15 Zeilen mit und ohne kombinierte Knoten. 88

6.7 Recall-Werte des Registercode-basierten StoneDetector-Ansatzes für $\geq$ 15 Zeilen. 90

6.8 Vergleich StoneDetector und iClones auf Bytecode-/Registercode-Basis für $\geq$ 15 Zeilen. 91

6.9 Gemeinsame und exklusive Codeklone für Java-Bytecode-basierte Klonerkennungstools. 93

7.1 Ergebnisse [53] verschiedener maschineller Lernverfahren auf dem BigCloneBench (CDLH). 95

7.2 Ergebnisse der Experimente für DT+GMN. 100

7.3 Evaluation von DT+GMN auf 4 Funktionalitäten trainiert. 103

7.4 Evaluation von DT+GMN trainiert auf 4 Funktionalitäten und StoneDetector als Orakel. 105

Abkürzungsverzeichnis

CPU:	Central Processing Unit
RAM:	Random-Access Memory
CFG:	Kontrollflussgraph (Control Flow Graph)
AST:	Abstrakter Syntaxbaum (Abstract Syntax Tree)
API:	Application Programming Interface
JAR:	Java Archive
DNN:	Tiefes Neuronales Netz (Deep Neural Network)
CNN:	Convolutional Neural Network
RNN:	Rekurrentes Neuronales Netz (Recurrent Neural Network)
LSTM:	Long Short-Term Memory
FA-AST+GMN:	Flow-Augmented Abstract Syntax Tree mit Graph-Matching-Network
DT+GMN:	Dominatortree (Dominatorbaum) mit Graph-Matching-Network

Ehrenwörtliche Erklärung

Hiermit erkläre ich, dass

- mir die Promotionsordnung der Fakultät bekannt ist,
- ich die Dissertation selbst angefertigt habe,
- ich keine Textabschnitte oder Ergebnisse eines/einer Dritten oder eigene Prüfungsarbeiten ohne Kennzeichnung übernommen habe,
- alle von mir benutzten Hilfsmittel, persönliche Mitteilungen und Quellen in meiner Arbeit angegeben wurden,
- ich keine Hilfe einer kommerziellen Promotionsvermittlerin/ eines kommerziellen Promotionsvermittlers in Anspruch genommen habe,
- Dritte weder unmittelbar noch mittelbar geldwerte Leistungen von mir für Arbeiten erhielten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen,
- ich die Dissertation noch nicht als Prüfungsarbeit für eine staatliche oder andere wissenschaftliche Prüfung einreichte.

Bei der Auswahl und Auswertung des Materials sowie bei der Herstellung des Manuskripts haben mich folgende Personen unterstützt:

- Prof. Dr. habil. Wolfram Amme

Ich habe weder die gleiche, noch eine in wesentlichen Teilen ähnliche bzw. andere Abhandlung bereits bei einer anderen Hochschule als Dissertation eingereicht.

....

Ort, Datum Unterschrift