Erkennung semantischer Klone mittels Locality-Sensitive-Hashing charakteristischer Vektoren

Universität Stuttgart
Institut für Softwaretechnik, Abteilung Programmiersprachen
Diplomarbeit
2016

Jannik Zschiesche

Inhaltsverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
AST	Abstract Syntax Tree
BSD	Berkeley Software Distribution
CLI	Command Line Interface
FFI	Foreign Function Interface
(GNU) GPL	(GNU) General Public License
IML	InterMediate Language
IST	Interesting Semantic Thread
LLOC	Logical Lines Of Code
LSH	Locality Sensitive Hashing
MIT	Massachusetts Institute of Technology
PDG	Program Dependence Graph
SSA	Static Single Assignment

Abbildungsverzeichnis

Codebeispiele

1 Einleitung

Duplizierter Quellcode, Codeklone, sind ein alltägliches Nebenprodukt der regelmäßigen Programmierarbeit. Insbesondere in großen Teams ist es nicht möglich, den Detailblick für jede Komponente des Systems zu bewahren. An anderer Stelle ist aus es programmarchitektonischer Sicht nicht möglich, die vorliegende Abstraktion zu verwenden und sie muss absichtlich dupliziert werden. In einigen Fällen kann es sogar von Vorteil sein, duplizierten Code an einigen wenigen Stellen in seinem Programm einzusetzen.

Doch ungeachtet der Historie und des Bewusstseins, ist die Kenntnis über die Stellen des Programms wichtig, in denen duplizierter Code verwendet wird. Nur durch diese Information können überlegte Entscheidungen im Programmieralltag, oder in einem Forschungsumfeld größere Systemanalysen durchgeführt werden.

Es sind bereits einige Verfahren etabliert, diese beschränken sich in aller Regel jedoch auf die syntaktischen Klone. Um zusätzlich semantische Klone zu erkennen, sind bisher keine skalierbaren Verfahren bekannt. DECKARD von Jiang et al, und darauf aufbauend Gabel et al haben ein Klonerkennungsverfahren entworfen, das einerseits stark in der syntaktischen Klonerkennung ist, aber auch in einigen Teilen semantische Klone erkennen kann.

Diese Diplomarbeit führt zunächst in alle relevanten Themen dieses Verfahrens ein. Anschließend wird im Detail die Implementierung namens Bryant[1] vorgestellt, die abschließend analysiert und ausgewertet wird.

[1] In dieser Arbeit wird sowohl die Implementierung dieser Diplomarbeit, als auch die theoretischen Vorarbeiten durch Gabel et al einheitlich „Bryant“ genannt, da das Verfahren von Gabel et al keine eigene Bezeichnung besitzt.

2 Grundlagen und Motivation

Duplizierter Programmcode kann auf vielfältige Weise entstehen. Die offensichtlichste ist sicherlich Copy & Paste, das heißt der Autor hat den betroffenen Quellcodeabschnitt aus einem anderen Teil des Programms direkt kopiert. Aber auch mehrfach benötigte Funktionalität gepaart mit der Unkenntnis, dass das vorliegende Problem bereits im Programm an anderer Stelle gelöst ist, führt zu mehrfacher Implementierung ähnlicher Lösungen. Diese auf unterschiedlichem Wege erreichten Lösungen geben einen Einblick in die Struktur und die Wachstumshistorie des Programms und können für weiterführende Analysen verwendet werden. So können aus der Betrachtung großer Softwaresysteme und Programmierteams in Kombination mit der Auswertung von Codeklonen Rückschlüsse über die Arbeitsweise und Denkstrukturen typischer Softwareteams gezogen werden.

Neben der theoretischen Betrachtung ist duplizierter Code insbesondere für die Wartung von Software interessant. So sind Programme, die nicht die angemessenen Abstraktionswerkzeuge der Sprache verwenden, in aller Regel länger, was die Anzahl von potenziellen Programmfehlern erhöht. Außerdem müssen nun bei Änderungen am Programm die duplizierten Stellen manuell nachverfolgt und ebenfalls (und identisch) angepasst werden. Bei dieser manuellen Arbeit können sich Programmfehler einschleichen, die dann in der Anwendung nur auftreten, wenn man die Funktion genau auf diesem einen Wege ausführt[2].

Auch rechtliche Fragen können mit Codeklonerkennern beantwortet werden. Programmcode oder nur Teile von diesem können unter Copyright stehen. So kann ein Klonerkenner bei der Entdeckung von unerlaubten Kopien urheberrechtlich geschützten Programmcodes behilflich sein. Ein prominentes Beispiel ist die Klage von Oracle America, Inc. gegen Google, Inc., bei der es um die Frage ging, ob Google in Android ohne Genehmigung Java-APIs verwendet und implementiert hat[3].

In der Optimierung von Programmen kann duplizierter Code ebenfalls von Bedeutung sein. Webanwendungen transportieren ihr HTML, CSS und JavaScript bei jedem Seitenaufruf an den Client. Es ist daher vorteilhaft, wenn die Größe dieser Datenmenge so klein wie möglich gehalten wird, insbesondere auf Mobilgeräten, die oftmals mit einer langsameren Datenverbindung und höherer Latenz mit dem Internet verbunden sind. Hier führt die Deduplikation zu einer verringerten Ladezeit.[4]

Jedoch ist nicht jeder Klon negativ zu bewerten. Automatisch generierter Code ist in erheblichem Maße repetitiv, dies wird jedoch akzeptiert, da der generierte Code nicht aktiv bearbeitet wird (sondern im Zweifelsfall schlicht neu generiert wird). Darüber

[2] Ein häufiges Beispiel für duplizierten Code bei einer Shopsoftware ist das Einlösen von Coupons. So kann es bei Duplizierung zu unterschiedlichem Verhalten kommen, je nachdem ob man den Coupon im Warenkorb oder auf der Bestellabschlussseite einlöst.
[3] Siehe [10].
[4] Genau in diesem speziellen Fall ist eine übermäßige Deduplikation jedoch nicht immer von großem Nutzen. Da bei vielen Anwendungen die Daten vor Übertragung mit GZIP komprimiert werden, wird der Einfluss der Duplikate auf die Dateigröße bereits durch das Kompressionsverfahren minimiert.

hinaus gibt es Fälle, in denen Codeklone notwendig sind. Bei Fehlern in externen Bibliotheken wird oftmals der externe Code kopiert und angepasst. Außerdem können auch Performanceoptimierungen durch duplizierten Code erreicht werden. Unterstützt ein Compiler beispielsweise kein automatisches Aufrollen von Schleifen[5], kann es je nach Anwendungsfall notwendig sein, die Schleife selbst aufzulösen und die Blöcke manuell zu kopieren.

2.1 Ausprägungen von Codeklonen

Während bei der Analyse einzelner Klone feinstufige Unterscheidungen definiert werden können, teilt sich die Menge aller Codeklone zunächst in zwei Variationen. Auf der einen Seite die syntaktischen Klone und auf der anderen Seite die semantischen Klone.

Syntaktische Klone sind Programmfragmente, die bereits in textueller Form große Ähnlichkeit aufweisen. So sind die grundlegenden Strukturen identisch und nur einige Teilaspekte werden abstrahiert oder die Sortierung von Programmanweisungen vereinheitlicht.

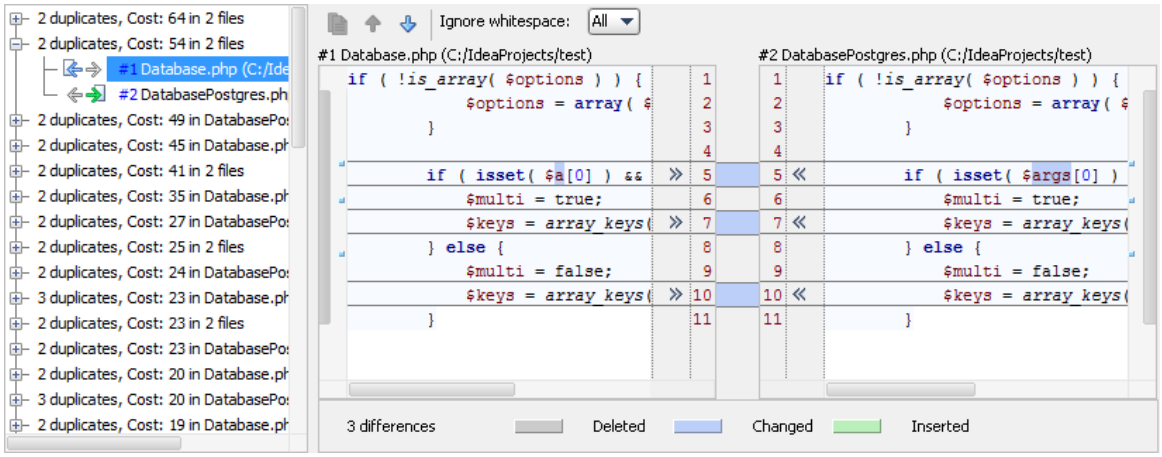

Abbildung 1: Syntaktische Klonerkennung in der Code Duplication Analysis der Programmierumgebung PhpStorm (Screenshot von [11]).

Syntaktische Klonerkennungsprogramme verwenden unterschiedliche Grade von Abstraktion[6]. Die *stringbasierten Tools* teilen zunächst das Programm in Zeilen auf und operieren anschließend auf Programmcodezeilen. Hierbei wird ein parametrisierter Stringvergleich verwendet, der gegen marginaler Unterscheidungen resistent ist[7]. Token-basierte Tools bauen auf den Lexer und vergleichen generierte Tokenstreams. Diese Tools sind in der Regel robuster gegenüber Codeformatierung und Spacing[8]. Die

[5] Bei dieser Optimierung wird die Schleife entweder komplett aufgelöst und durch passend untereinander kopierte Codeblöcke ersetzt oder zumindest soweit umgebaut, dass der Schleifenrumpf zugunsten einer geringeren Iterationszahl länger wird. Hierdurch wird die Anzahl der Sprünge in der Kontrollanweisung und dem Sprung zum Beginn des Schleifenrumpfes optimiert. Für Details siehe [11].
[6] Die Abgrenzung der Abstraktionsgrade ist aus [7].
[7] Beispielsweise die Arbeiten von Baker in [13] und [14].
[8] Beispielsweise CCFinder [8] und CP-Miner [9].

baumbasierten Tools arbeiten entweder auf dem Parsebaum oder AST und vergleichen Syntaxbäume oder Fingerprints[9] der Bäume[10].

Semantische Klone sind Programmcodeabschnitte, die auf einer syntaktischen Ebene möglicherweise unterschiedliche Struktur besitzen, jedoch die gleiche Funktion implementieren. Ein triviales Beispiel hierfür sind for- und while-Schleifen. Diese sehen syntaktisch zwar unterschiedlich aus, können in der Regel aber in die jeweils andere Form überführt werden. Programmcode der die eine Variante implementiert, kann also funktional identisch zu Programmcode sein, der die andere Variante verwendet.

Echte semantische Gleichheit im Allgemeinen (Programmäquivalenz) ist unentscheidbar[11]. Daher wurden einige Verfahren entwickelt, die zwar keine echte Äquivalenz entscheiden können, jedoch zumindest ein Teilwissen über die semantischen Details haben[12].

Fast alle referenzierten Verfahren der syntaktischen Klonerkennung haben gezeigt, dass sie auch für sehr große Programme skalieren können. Tools, die semantische Analysen anwenden konnten bisher noch nicht skalierbar implementiert werden.

2.2 Verwendetes Verfahren

Die Effektivität von Klonerkennungsverfahren ist maßgeblich abhängig von der zugrundeliegenden Definition eines Codeklons. Ausgehend von dieser wird das Verfahren modelliert. Die verwendete Definition muss also fundiert, theoretisch gesichert und berechenbar sein.

Die Implementierung in dieser Diplomarbeit baut auf der Arbeit von Gabel et al.[13] auf. Diese wiederum basiert auf dem im Klonerkenner DECKARD[14][15] verwendeten Verfahren, mit einer Erweiterung um eine semantische Analyse mithilfe des PDGs[16]. DECKARD ist ein gegenüber Quellcodeformatierung robuster syntaktischer Klonerkenner aus der Familie der baumbasierten Verfahren.

[9] Hierbei werden die Bäume in einen Hashwert überführt, der Rückschluss auf die Struktur der Bäume zulässt. Die Fingerprints sind erheblich effizienter zu vergleichen als die Baumstruktur.
[10] Die Arbeiten von Baxter et al ([15], [2]) und Wahler et al ([16]) vergleichen Parse- oder Syntaxbäume. Die Arbeiten [17] und [18] verwenden Fingerprinting.
[11] Dies geht direkt aus dem Satz von Rice hervor.
[12] Komondoor und Horwitz haben einen Codeklonerkenner entwickelt, der unter Zuhilfenahme des PDG und Program Slicing grundlegende semantische Analysen durchführen kann [19].
[13] Siehe [5].
[14] Siehe [7].
[15] Das Verfahren wurde benannt nach der Hauptfigur des Films Blade Runner (1982). Dieser muss ebenfalls „Klone“ (Replikanten) erkennen.
[16] Program Dependence Graphs (PDGs) werden in [4] näher beschrieben.

```
int f ()
{
   int a, b;

   a = 10;
   b = 5;

   return a + b;
}
```

Codebeispiel 1: Beispielfunktion mit Addition zweier konstanter Zahlen.

Daher sind die Definition und das Verfahren direkt aus den beiden Arbeiten übernommen, alle Details werden im Folgenden jedoch noch einmal ausführlich erklärt. DECKARD arbeitet auf Parsebäumen, Bryant arbeitet jedoch mit ASTs. Die folgenden Kapitel besprechen daher direkt ASTs.

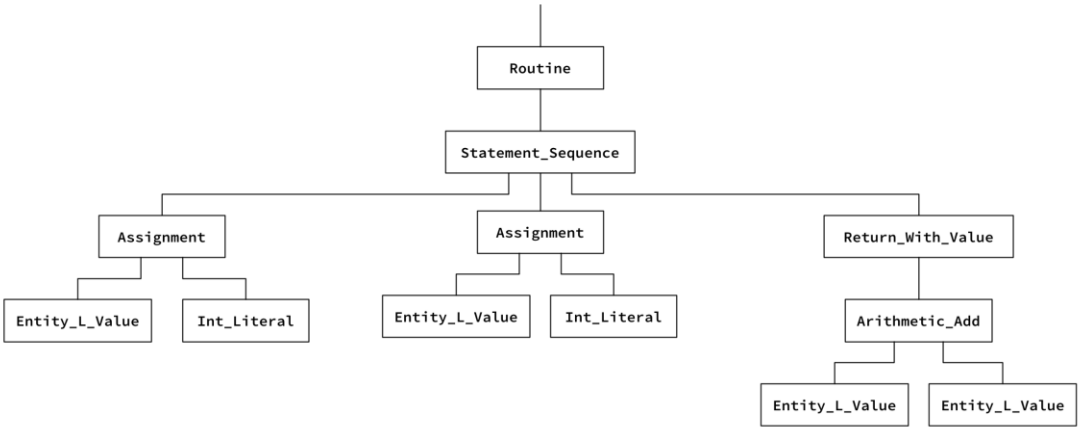

Abbildung 2: AST zum Codebeispiel 1, ab Beginn der Routine `f()`[17].

2.3 Definition von Codeklonen

Ein Programm wird bei der Übersetzung in aller Regel in einen Syntaxbaum überführt. Die Knoten dieses Baumes stehen für ein Sprachkonstrukt, das im Programmcode verwendet wird. Abstrakte Syntaxbäume unterscheiden sich von Parsebäumen dadurch, dass einige Bereiche bereits abstrahiert wurden, wie beispielsweise Klammersetzung, und die Daten in strukturierter Form vorliegen[18].

[17] Der AST ist eine vereinfachte Darstellung, daher wurden für das Verständnis unwichtige Knoten ausgeblendet.
[18] So sind in einer if-Anweisung beispielsweise die Aspekte Condition, If-Zweig und Else-Zweig bereits semantisch strukturiert und gruppiert in einem Knoten (mit Unterknoten) und nicht mehr wie im Parsebaum eine Abfolge von mehreren Knoten.

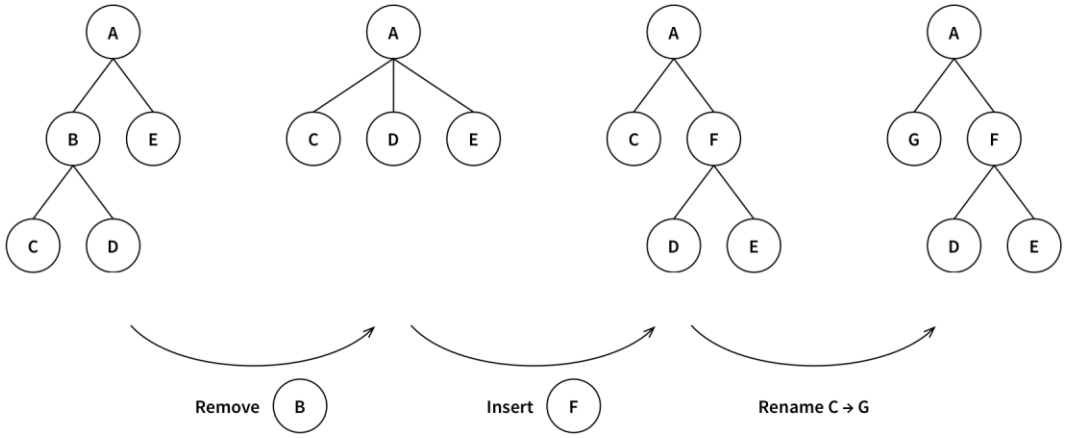

Abbildung 3: Beispielhafter Ablauf einer Tree-Edit-Sequenz.

Ein Codeklon wird ausgehend von ASTs wie folgt definiert: man wählt zwei Programmabschnitte aus, erstellt die ASTs (bzw. Unterbäume[19]) und vergleicht diese. Die verwendete Metrik des Vergleichs ist hierbei die *minimale Edit-Distanz* δ[20]. Die Edit-Distanz repräsentiert die Menge der Änderungen[21] (Einfügen, Umbenennen, Löschen) die nötig ist, um den einen in den anderen Baum zu überführen. Intuitiv kann daher festgestellt werden, dass zwei Bäume umso ähnlicher sind, je geringer die Edit-Distanz ist.

***Definition 1**: Zwei Bäume T_1 und T_2 sind σ-ähnlich für ein gegebenes σ, wenn für ihre Edit-Distanz $\delta(T_1, T_2) < \sigma$ gilt.*[22]

Ausgehend von dieser Definition können nun Codeklone definiert werden.

***Definition 2**: Zwei Code-Fragmente C_1 und C_2 gelten als Klonpaar, wenn ihre zugehörigen Baumrepräsentationen T_1 und T_2 σ-ähnlich für ein spezifiziertes σ sind.*[23]

Diese Definition ist zwar theoretisch valide und fundiert, birgt allerdings ein praktisches Problem. Die Berechnung der Edit-Distanz ist nicht effizient möglich[24]. Außerdem

[19] Die folgenden Abschnitte arbeiten nie auf „vollständigen" ASTs, da dies ein intraprozedurales Verfahren ist und daher nur ASTs innerhalb einer Routine bearbeitet werden. Im Folgenden wird zur Vereinfachung nur von „ASTs" gesprochen – die Verfahren funktionieren prinzipiell für jede Art von Syntaxbaum.
[20] Die Edit-Distanz ist nicht eindeutig, da es mehrere Abfolgen von Anweisungen geben kann, um einen Baum in einen anderen Baum zu überführen. Diese können beliebig groß werden (durch Einfügen und Löschen unbenutzter Knoten). Daher wird die minimale Edit-Distanz verwendet, die kurzmöglichste Abfolge darstellt.
[21] In den meisten Verfahren werden die „Kosten" der möglichen Änderungen (Einfügen, Umbenennen, Löschen) unterschiedlich gewichtet, so kann ein Einfügen teurer (in Bezug auf die Metrik) als ein Umbenennen sein. Dieses Detail ist in dieser Hinführung allerdings nicht relevant.
[22] Siehe [7].
[23] Siehe [7].
[24] Die Komplexität ist $O(|T_1| \times |T_2| \times d_1 \times d_2)$, wobei $|T_i|$ die Größe von T_i ist und d_i das Minimum aus der Tiefe von T_i und der Anzahl der Blätter von T_i. (siehe [7]).

müssen alle Bäume miteinander verglichen werden, was im schlechtesten Fall quadratischen Aufwand bedeutet.

2.4 Alternative Darstellung von Codefragmenten

Die Codeklon-Definition aus dem vorigen Kapitel ist theoretisch fundiert, besitzt aber ungünstige Laufzeiteigenschaften. Daher wird in diesem Kapitel zunächst eine alternative Repräsentation der ASTs eingeführt, die eine effizientere Implementierung ermöglichen. Anschließend wird das Berechnungsproblem der Baum-Edit-Distanz auf eine Distanzberechnung in der neuen Darstellung reduziert. Diese Reduktion ist wichtig, da nur dadurch gewährleistet werden kann, dass ein gefundenes Ergebnis in der neuen Darstellung auch ein korrektes Ergebnis in der Darstellung als Edit-Distanz ist. Und da im verwendeten Verfahren Codeklone über die Edit-Distanz definiert sind, kann ohne diese Rückführung keine Aussage getroffen werden.

2.4.1 Atomic Tree Patterns[25] und charakteristische Vektoren

Gegeben sei ein Binärbaum. Man definiert eine Familie an Atomic Tree Patterns, parametrisiert mit der Höhe des Binärbaums q.

Ein q-level Atomic Pattern ist ein vollständiger Binärbaum der Höhe q. Die Knoten dieses Binärbaums entstammen aus einer Knotenmenge $\mathcal{L}$ (diese Menge beinhaltet auch das leere Label ε). Es existieren insgesamt $|\mathcal{L}|^{2^q-1}$ mögliche Binärbäume der Höhe q, dies sind alle möglichen Permutationen der Knotenelemente auf die $2^q - 1$ Knoten eines vollständigen Binärbaumes der Höhe q. Diese Menge an Bäumen wird durchnummeriert[26]. Nun wird ein Vektor der Länge $|\mathcal{L}|^{2^q-1}$ erstellt, wobei das i-te Element die Anzahl der Vorkommen des i-ten Baumes im Ursprungsbaum darstellt.

[25] Siehe [7].
[26] Die tatsächliche Sortierung ist irrelevant, so lange sie identisch über die gesamte Vektorgenerierung ist.

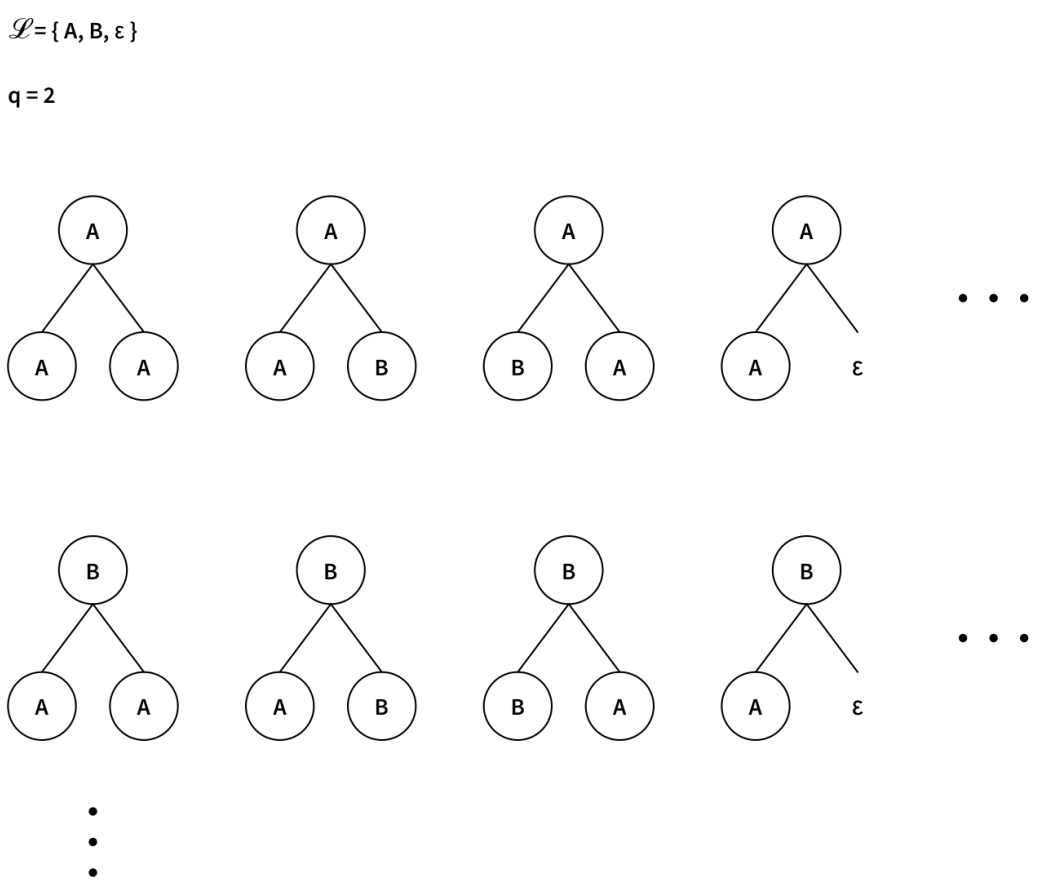

Abbildung 4: Beispielhafte Vorschau einiger q-Atomic Patterns der Höhe 2 für die angegebene Knotenmenge (insgesamt gibt für diese Knotenmenge und Höhe 27 Permutationen).

Dieser Vektor ist ein Fingerprint des ursprünglichen Baumes und wird der charakteristische Vektor des Baumes genannt (gekennzeichnet durch $v_q(T)$). Die Abbildung von Baum auf Vektor ist surjektiv: identische Bäume ergeben identische Vektoren, identische Vektoren wiederum müssen aber nicht identische Bäume ergeben. Dies ist allerdings eine erwünschte Eigenschaft, da im Hinblick auf ASTs die Reihenfolge der Knoten keine Rolle spielen soll. Das Verfahren soll robust gegenüber Umsortierung des Programmcodes sein, was durch diese Eigenschaft erfüllt wird.

Zur vereinfachten Berechnung der Ähnlichkeit von ASTs werden charakteristische Vektoren verwendet. Hierzu werden die ASTs in die charakteristischen Vektoren umgewandelt, die Höhe der Pattern ist $q = 1$, die Knotenmenge sind alle Knotentypen des ASTs. Durch diese Definition repräsentiert der charakteristische Vektor eines ASTs ein Konstrukt, in dem die Anzahl jedes Knotentypen innerhalb dieses ASTs aufgelistet ist.

Im nächste Schritt wird (nach der alternativen Darstellung der ASTs) die euklidische Distanz der Vektoren[27] zu berechnet. Wenn zwei Vektoren sehr nahe sind, soll die Edit-Distanz zwischen den Ursprungsbäumen gering sein. Nur durch diese Eigenschaft ist die Reduktion valide. Ihr Beweis wird daher im Folgenden anskizziert.

***Theorem 1**: Für zwei beliebige Bäume T_1 und T_2 mit Edit-Distanz $\delta(T_1, T_2) = k$ ist die l_1-Norm der zugehörigen q-level Vektoren für T_1 und T_2 $\mathcal{H}\left(v_q(T_1), v_q(T_2)\right)$ nicht größer als $(4q-3)k$.*[28]

Wie in obigem Theorem beschrieben, existiert eine Verbindung zwischen der l_1-Norm (Hamming-Distanz) der Vektoren und der Edit-Distanz der zugehörigen Bäume. Nun muss noch eine Verbindung zwischen der l_1 und l_2-Norm (euklidische Distanz)[29] gezeigt werden, dann ist durch den transitiven Abschluss die Reduktion theoretisch bestätigt.

***Theorem 2**: Für zwei beliebige Ganzzahl-Vektoren gilt $\sqrt{\mathcal{H}(v_1, v_2)} \leq \mathcal{D}(v_1, v_2) \leq \mathcal{H}(v_1, v_2)$.*[30]

Durch Verknüpfung von Theorem 1 mit Theorem 2 folgt direkt:

***Korollar 1**: Für zwei beliebige Bäume T_1 und T_2 mit Edit-Distanz $\delta(T_1, T_2) = k$ ist die l_2-Norm der q-level Vektoren dieser Bäume $\mathcal{D}(v_q(T_1), v_q(T_2))$ nicht größer als $(4q-3)k$ und nicht kleiner als die Quadratwurzel der l_1-Norm. Dies bedeutet:*

$$\sqrt{\mathcal{H}(v_q(T_1), v_q(T_2))} \leq \mathcal{D}\left(v_q(T_1), v_q(T_2)\right) \leq (4q-3)k$$

Da in dieser Arbeit die Höhe der Atomic Patterns $q = 1$ gilt, sind entweder die euklidische Distanz, oder die Quadratwurzel der Hamming-Distanz eine untere Schranke für die Edit-Distanz. So gilt: wenn die Distanz zweier Vektoren größer als ein vorher definiertes σ ist, können die zugehörigen ASTs nicht σ-ähnlich sein. Andersherum ist es sehr wahrscheinlich, dass wenn die untere Schranke kleiner als ein definiertes σ ist, der tatsächliche Wert ebenfalls kleiner als σ ist[31].

Durch diese Reduktion kann das Problem der Edit-Distanz zweier ASTs also auf die euklidische Distanz zweier charakteristischer Vektoren vereinfacht werden. Für die Implementierung ist noch wichtig, dass diese Herleitung auch für Abstract Syntax

[27] Die Vektoren können als Punkte dargestellt werden, wobei der Vektor die Strecke zwischen Punkt und dem Ursprung des Koordinatensystems darstellt. Im Folgenden wird allerdings weiterhin von der „Distanz zweier Vektoren" gesprochen, auch wenn im Grunde die Entfernung zweier Punkte berechnet wird.
[28] Siehe Theorem 3.3 in [19].
[29] Für genaue Definition zur l_1 und l_2-Norm sind im Appendix (Kapitel 7.1) zu finden.
[30] Siehe [6].
[31] Nach [6].

Forests[32] gilt, da diese durch Hinzufügen eines gemeinsamen Eltern-Knoten zu einem einzelnen Baum umgewandelt werden können.

2.4.2 Vektorgenerierung

Nachdem das theoretische Fundament gesichert ist, verschiebt sich nun der Fokus der Arbeit auf die tatsächliche Generierung der Vektoren. Die Phase der Vektorerstellung teilt sich in zwei Abschnitte auf: zunächst die direkt aus AST-Knoten erstellten Vektoren und anschließend in Kapitel 2.4.3 die zusammengesetzten Vektoren.

Mit Einstieg in eine Routine werden alle Knoten des AST in Postorder-Reihenfolge durchlaufen. Bei jedem Vektor werden zunächst die Vektoren der Kinder summiert. Dann wird der Eintrag an dem Index des AST-Knotens im Vektor um 1 erhöht, falls der Vektor *relevant* ist. Falls der Vektor zusätzlich *signifikant* ist, wird der Vektor in die globale Liste aller erstellten Vektoren hinzugefügt.

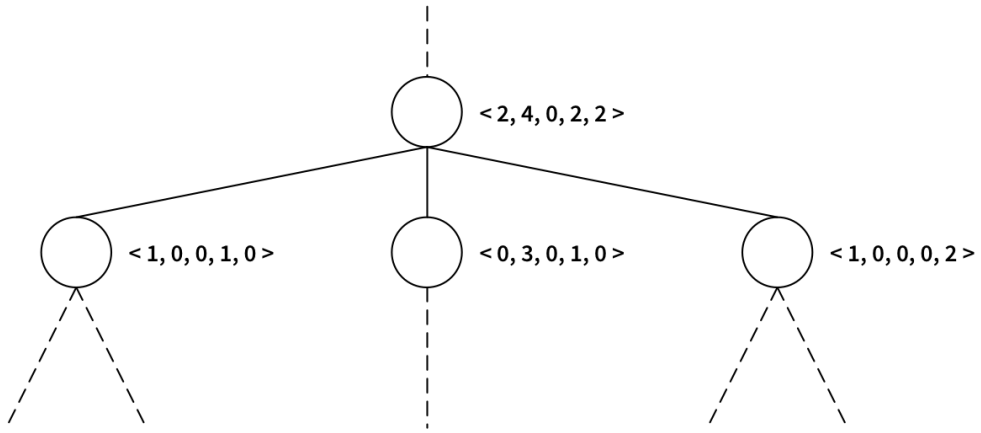

Abbildung 5: Annotation der AST-Knoten mit den charakteristischen Vektoren (der obere Knoten ist relevant und erhöht den zweiten Eintrag im Vektor um 1).

Die Relevanz eines Knoten gibt an, ob dieser Knoten in den charakteristischen Vektoren gezählt werden soll[33]. Durch die Verwendung des ASTs entfallen viele „technische Knoten“ eines Parsebaums wie Klammern und Semikola bevor der Algorithmus startet. Aber auch Casts oder andere strukturierende Knoten können für das Zählen nicht interessant sein. Insbesondere wird über die Relevanz die Granularität der Klonerkennung gesteuert. So können Unterschiede im AST „ignoriert“ oder absichtlich gleich gezählt werden. Ein Beispiel explizit zusammengefasster Knoten können Schleifen sein – so werden unterschiedliche Schleifenvarianten explizit auf den gleichen Eintrag im Vektor indexiert, wodurch der Klonerkenner diese als identisch erkennt.

[32] Eine Menge an ASTs.
[33] Die für diese Arbeit getroffene Auswahl an relevanten IML-Knoten findet sich in Kapitel 3.43.2.1.

Die Signifikanz eines Knotens gibt an, wie wahrscheinlich dieser ein Kandidat für die Klonerkennung ist. Der Hintergrund ist, dass die so gefundenen Kandidaten eine Mindestgröße überschreiten sollen. Ohne diese Mindestgröße würde sonst jede Initialisierung einer Variable als Klon erkannt:

```
int i = 1;
```

Codebeispiel 2: Beispielprogramm mit Variableninitialisierung

Vor dem Hinzufügen eines Knotens wird zunächst überprüft, ob dessen Vektor genügend für die Signifikanz interessante Knoten enthält. Nicht alle Knotentypen sind für diese Eigenschaft interessant, sondern nur eine Auswahl von Knotentypen, die sehr wahrscheinlich der Ausgangspunkt eines Codeklons sein könnten.

Den numerischen Mindestwert für die Signifikanzbestimmung festzulegen ist nicht trivial. Während in der Originalimplementierung von DECKARD 30 Token (der Standardwert) in der Regel zu etwa 3 Statements gehören, ist dies bei AST-Knoten nicht der Fall. Hier können die 30 Knoten entweder zu weniger (möglicherweise weniger als einem vollen Statement), oder erheblich mehr Statements gehören. Daher wird von einer rein numerischen Betrachtung des Vektors abgesehen, sondern das zusätzliche Wissen dahingehend miteinbezogen, dass eine gewichtete Summe des Vektors für die Signifikanzberechnung verwendet wird.

Die *Relevanz* gibt also an, welche Knoten gezählt werden und die *Signifikanz* gibt an, von welchen Knoten die Vektoren in die Klon-Kandidatenliste aufgenommen werden.

2.4.3 Vector Merging (Vektor-Kombinationen)

Mit den Klon-Kandidaten, die direkt aus einzelnen AST-Knoten erzeugt werden, sind im Programmcode einzelne Statements abgedeckt. Viele interessante Klone werden sich jedoch über mehrere Zeilen Programmcode, über mehrere Statements erstrecken. Daher werden im zweiten Abschnitt der Vektorgenerierung aus mehreren benachbarten AST-Knoten zusammengefasste Vektoren erzeugt.

Der Algorithmus für das Zusammenfassen der Knoten ist simpel. Es wird die Liste aller Kindknoten als Basis verwendet, anschließend wird ein sogenanntes Sliding Window erstellt und über die Liste der Kindknoten verschoben. Für jeden Knoten in der Liste wird entschieden, ob dieser Teil einer Fragmentkombination sein kann, oder nicht. Falls der Knoten nicht kombinierbar ist, wird er aus der Liste der Kindknoten herausgefiltert.

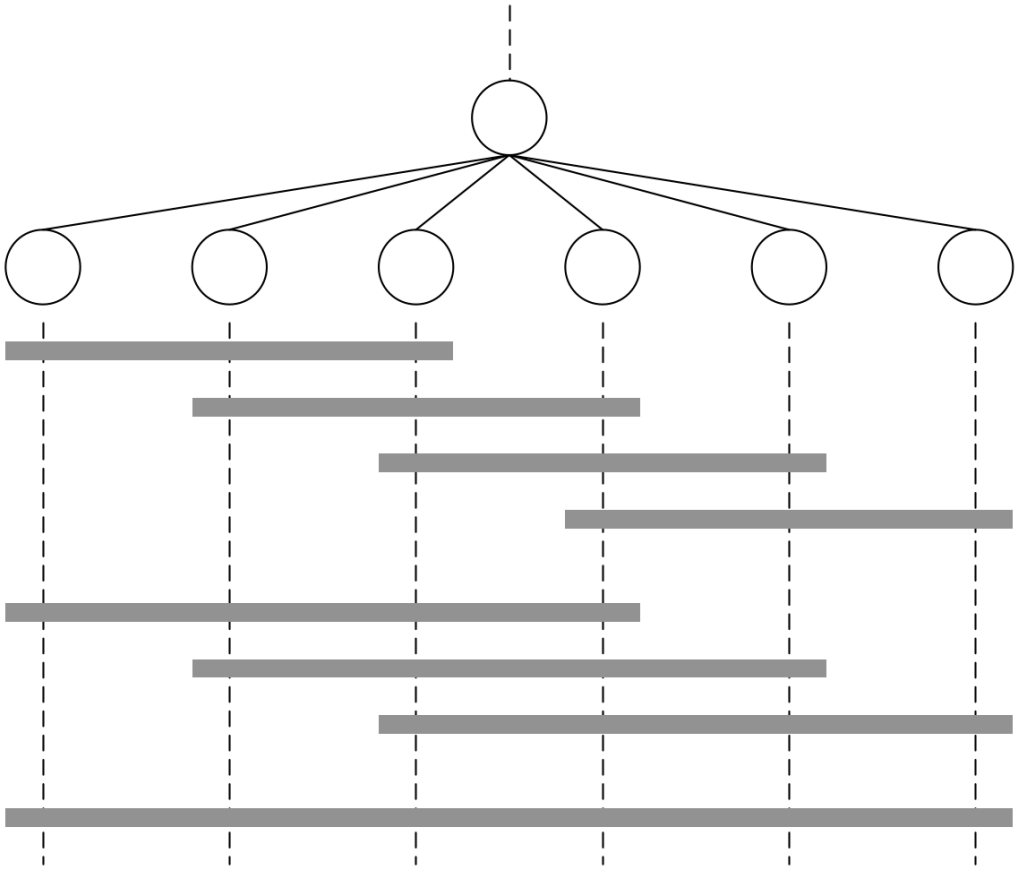

Abbildung 6: Veranschaulichung des iterativ verschobenen und vergrößerten Sliding Windows.

Das Sliding Window betrachtet einige aufeinanderfolgende Knoten. Aus diesen wird ein summierter Vektor erzeugt. Falls dieser signifikant ist, wird er der Liste der Klonkandidaten hinzugefügt. Anschließend wird das Sliding Window um eine Position weiter geschoben[34].

[34] Die Originalimplementierung von DECKARD (siehe [6]) hat keine automatische Vergrößerung des Sliding Windows verwendet, dafür aber die Schrittweite konfigurierbar gemacht. Dies ist bei der erweiterten Implementierung nicht mehr notwendig.

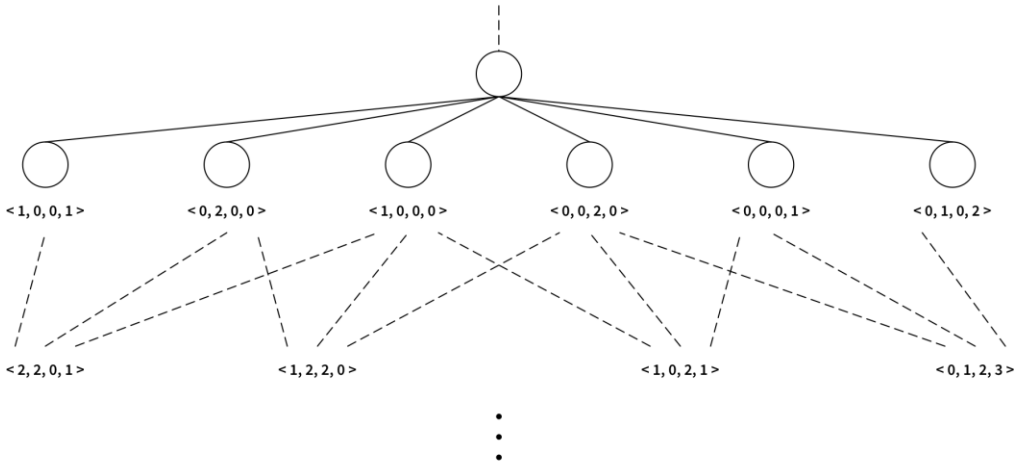

Abbildung 7: Neue Codefragmente aus Knoten mit zugehörigen Vektoren, die durch das Vector Merging erstellt wurden.

Die Größe des Sliding Windows wird iterativ vergrößert. Die Arbeit von Gabel et al[35] hat herausgefunden, dass es sinnvoll ist, die Größe des Sliding Windows initial auf 3 zu setzen und nach jedem Durchlauf multiplikativ um den Faktor 1,5 zu vergrößern. Dieser Abschnitt der Vektorgenerierung endet, wenn die Größe des Sliding Windows größer als die Anzahl der benachbarten Vektoren ist.

2.5 Locality-Sensitive-Hashing[36]

Nach der Generierung der Vektoren müssen nun effizient benachbarte Vektoren gefunden werden. Hierbei wird zunächst die Datenstruktur erstellt, anschließend wird für jeden Vektor in der Kandidatenliste die Liste der benachbarten Vektoren gesucht. Intuitiv hat dieses Verfahren in etwa quadratischen Aufwand, da jeder Vektor mit jedem anderen Vektor verglichen werden muss[37]. Deshalb wird ein Verfahren eingeführt, das die Vektoren in einer Vorverarbeitung clustert, sodass die Distanzberechnungen auf ein Minimum reduziert werden können[38].

Für eine Minimierung der Zahl der Vektorvergleiche wird Locality-Sensitive-Hashing[39] verwendet. In diesem Verfahren wird zu jedem Vektor ein Hash-Wert berechnet. Die zugrundeliegende Hashfunktion ist so konstruiert, dass für nahe Vektoren mit hoher Wahrscheinlichkeit gleiche Hash-Werte berechnet werden. Bei der Suche nach nahen Knoten werden von einem Anfragevektor aus alle Vektoren mit dem gleichen Hashwert

[35] Siehe [4].
[36] Die Definitionen und Beweise für diese Kapitel stammen aus [5].
[37] Dies ist also eine quadratische Menge an Vektorvergleichen, wobei die Vektorvergleiche (Distanzberechnungen) selbst ebenfalls einen Beitrag zur Laufzeit leisten.
[38] Die Zahl der Vektorvergleiche ist quadratisch in der Anzahl der Klonkandidaten. Ein einzelner Vektorvergleich (Distanzberechnung) selbst ist linear in der Anzahl der Dimensionen eines charakteristischen Vektors. LSH minimiert nur die Zahl der Vektorvergleiche.
[39] Wie beschrieben in [5].

gesucht und nur für diese kleine Teilmenge werden konkrete Distanzberechnungen durchgeführt.

2.5.1 Verwendete Hashfunktion

Die zugrundeliegende Hashfunktion muss die folgende Eigenschaft erfüllen, dass sie für eine LSH-Berechnung verwendet werden kann:

Definition 3[40]*: Eine Familie an Hashfunktionen* $h : \mathcal{V} \longrightarrow \mathcal{U}$ *heißt* (p_1, p_2, R, c)*-sensitiv (mit* $c \geq 1$*), falls* $\forall\, u, v \in \mathcal{V}$ *gilt:*

falls	$\mathcal{D}(u,v) < R$	*dann*	$Prob[h(u) = h(v)] \geq p_1$
falls	$\mathcal{D}(u,v) > cR$	*dann*	$Prob[h(u) = h(v)] \leq p_2$

Um als LSH-Hashfunktion sinnvoll zu sein, sollte für die gegebene Funktionsfamilie $p_1 > p_2$ und $c \geq 1$ gelten.

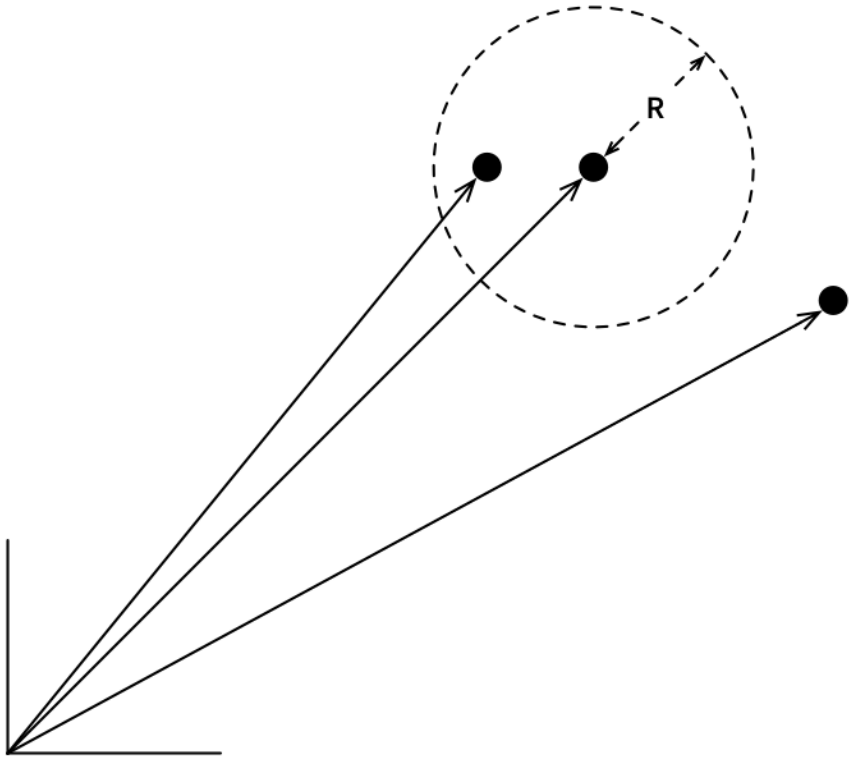

Abbildung 8: Der Raum wird in die Kugel um den Punkt und den restlichen Raum aufgeteilt.

[40] Aus [2], Definition 1 bzw. dessen Adaption in [6], Definition 3.9.

Datar et al[41] haben gezeigt, dass die folgende Hashfunktion diese Eigenschaft erfüllt:

$h_{\alpha,b} : \mathbb{R}^d \to \mathbb{N}$

$h_{\alpha,b}(v) = \left\lfloor \frac{\alpha \cdot v + b}{w} \right\rfloor$

mit $w \in \mathbb{R}$, $b \in [0, w]$ *und* $\alpha \in \mathbb{R}^d$ *als zufällig gewählten Vektor.*

2.5.2 Vorbereitung der LSH-Datenstruktur

Als grundlegende Datenstruktur bei der LSH-Berechnung wird eine Hashmap verwendet. Jeder Eintrag in dieser Hashmap („Bucket“) ist eine Liste von in diesen Bucket gehashten Vektoren.

Der Ablauf der LSH-Berechnung ist wie folgt: zunächst wird jeder Punkt nicht nur einmal gehasht, sondern es werden mehrere Hashfunktionen (insgesamt k Hashberechnungen) nacheinander ausgeführt. Dies verringert intuitiv mit jedem weiteren Hashing die Wahrscheinlichkeit, dass zwei Punkte die weit entfernt sind, den gleichen Hashwert erhalten. Diese Kette an h_i-Funktionen werden fortan g_j genannt. Anschließend wählt man eine Zahl L, die die Zahl der unterschiedlichen Hash-Buckets angibt, in die der Vektor gehasht wird.

Bildlich gesprochen projiziert die Hashfunktion den zufälligen Vektor α und den Vektor v auf einen Punkt auf der reellen Achse. Diese Projektion erhält die Distanzeigenschaft[42]. Dies bedeutet, dass Vektoren, die vorher weit entfernt waren, mit hoher Wahrscheinlichkeit nach der Projektion ebenfalls als Punkte auf der reellen Achse weit entfernt sind (und umgekehrt). Das wiederholte Hashing verringert die Wahrscheinlichkeit, dass zwei weit entfernte Vektoren über einen ungünstig liegenden α-Vektor den gleichen Hashwert erhalten[43].

[41] Siehe [2].
[42] Wie gezeigt in [2].
[43] Veranschaulicht vergrößert es die Kluft zwischen Punkten im Intervall $[0; R]$ und den Punkten im Intervall (R, ∞).

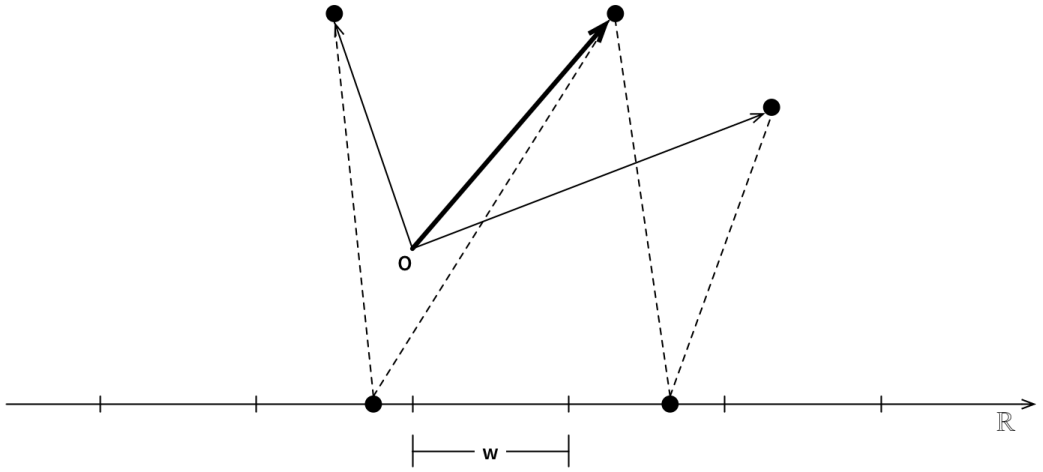

Abbildung 9: Veranschaulichung der Projektion der Vektoren auf die reelle Achse (der dickere Vektor ist a). Ebenfalls sieht man hier die gleichmäßige Aufteilung der reellen Achse in Segmente mit Breite w.

Nachdem der Vektor auf einen Punkt auf der reellen Achse projiziert wurde, wird die reelle Achse gleichmäßig in Segmente aufgeteilt, mit Breite w. Die Hashfunktion berechnet durch die Division und das Abrunden den Segmentindex. Der Segmentindex dient anschließend als Index für die Hashmap.

Nicht nur die Wahrscheinlichkeit, dass zwei weit entfernte Vektoren in den gleichen Bucket gehasht werden (falsch positive Treffer), sondern auch die Wahrscheinlichkeit, dass zwei nahe Vektoren *nicht* in den gleichen Bucket gehasht werden (falsch negative Treffer) ist interessant. Während die erste Variante nur das Laufzeitverhalten negativ beeinflusst, beeinträchtig die zweite Variante das Ergebnis. Wenn zwei nahe Vektoren in unterschiedliche Buckets gehasht werden, wird der Klonerkenner diese nicht als Klone erkennen.

Um dieses Problem zu umgehen, werden die Vektoren in mehrere Buckets gehasht. Insgesamt in L unterschiedliche Buckets.

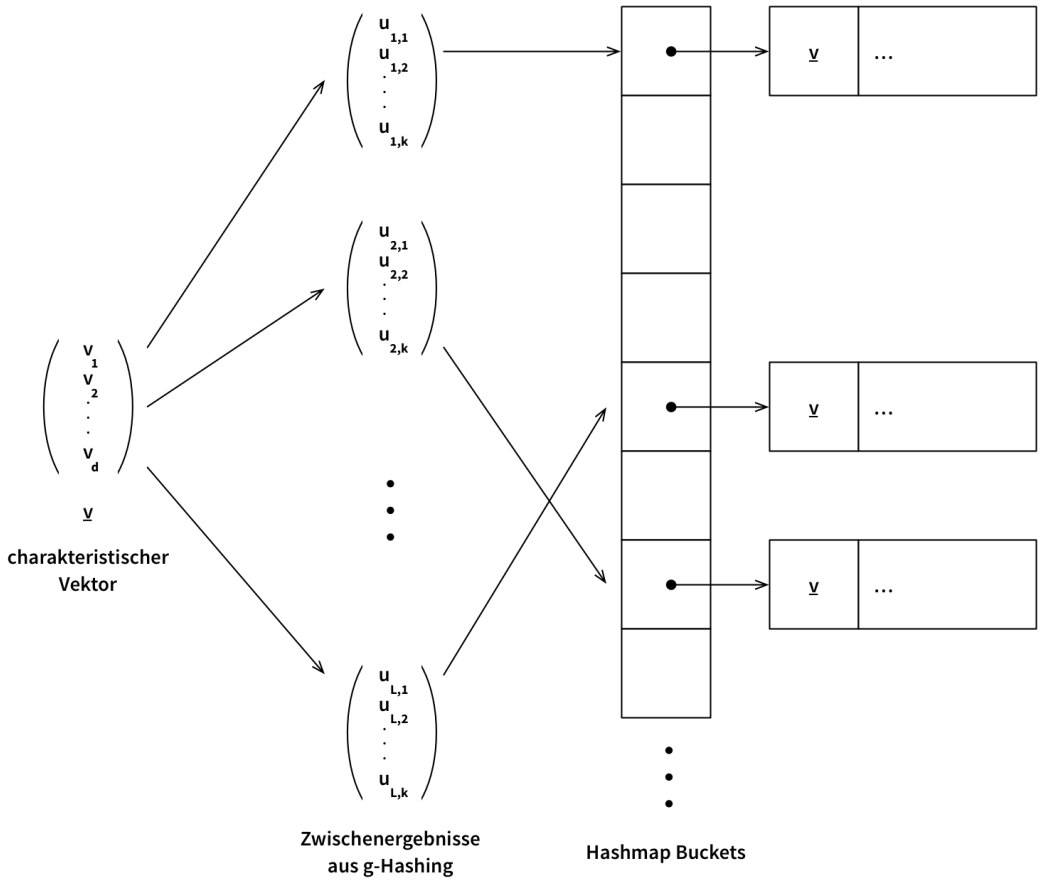

Abbildung 10: Gesamtüberblick aller Hashing-Vorgänge in LSH.

Der Ablauf in der Gesamtbetrachtung ist wie folgt: als Eingabe erhält LSH einen Vektor mit d Einträgen. Dieser Vektor wird k-mal mit unterschiedlichen Hashfunktionen gehasht (die Funktion, die k-mal mit unterschiedlichen Hashfunktionen hasht heißt g_i (wobei $i = 1 \dots L$)). Das Ergebnis ist ein neuer, k-dimensionaler Vektor, in dem im i-ten Eintrag das Ergebnis der i-ten Hashfunktion h_i steht. Dieser Vektor wiederum wird mit L regulären Hashfunktionen zu Bucketindizes der Hashmap umgerechnet. In jeden dieser Buckets wird der Vektor eingefügt.

Bei der Abfrage nach der Menge der nahen Nachbarn eines Punktes wird der Abfragepunkt ebenfalls gehasht und in Bucketindizes umgerechnet. Alle Vektoren in diesen Buckets werden anschließend direkt mit dem Abfragepunkt verglichen.

Relevante Parameter in diesem Verfahren sind die Zufallsfunktion, mit der die α- und b-Parameter der Hashingfunktionen generiert werden. Außerdem ist die Zahl der konkatenierten Hashfunktionen (k) und die Anzahl der Buckets, in die die Vektoren gespeichert werden (L), interessant. Der Parameter c aus Definition 3 wird auf 1 gesetzt, wodurch der Raum in Punkte innerhalb der Radiuskugel um den Punkt und alle restlichen Punkte aufgeteilt wird. Die Kollissionswahrscheinlichkeiten p_1 und p_2, sowie der Radius

R sind zwei Parameter, die nicht fest im System vorgegeben sein müssen, sondern bei denen der Benutzer der implementierten Klonerkennung die Genauigkeit gegenüber der Laufzeit abwägen kann.

Die wichtigste Eigenschaft der Hashfunktion ist, dass die Verteilung stabil über der l_2-Norm[44] ist. Dies ist zum Beispiel bei der Gauß-Verteilung gewährleistet. Darüber hinaus ist bekannt, dass man eine unter der l_2-Norm stabile Zufallsvariable aus zwei unabhängigen, normalverteilten Zufallsvariablen aus dem Intervall $[0; 1]$ erzeugen kann[45].

Die zwei undefinierten internen Parameter k und L werden für ein theoretisch optimales Ergebnis wie folgt definiert[46]:

$k = log_{\frac{1}{p_2}}(n)$

$L = n^{\rho}$ *mit* $\rho = \frac{ln(\frac{1}{p_1})}{ln(\frac{1}{p_2})}$

wobei n die Anzahl aller Vektoren ist.

Diese Definition hält die beiden Faktoren k und L so klein wie möglich (für ein theoretisch optimales Laufzeitverhalten), aber so groß wie nötig um die vorgegebenen Erfolgswahrscheinlichkeiten zu erhalten.

2.6 Erweiterung um PDG-Codefragmente[47]

DECKARD ist im Grunde eine Art Framework für Codeklonerkennung. Als Eingabe dienen unterschiedlichste Verfahren um Code-Fragmente (Mengen aus AST-Knoten und zugehörige charakteristische Vektoren) zu erzeugen, aus denen im weiteren Verlauf die Klone gefiltert werden. Diese Verfahren können variieren, es können neue hinzugefügt, existierende ausgetauscht oder entfernt werden.

Bryant[48] fügt ein neues Verfahren zur Erstellung von Code-Fragmenten hinzu. Da nur weitere Vektorerzeuger hinzugefügt werden, ist die Menge der mit dem abgewandelten Verfahren gefundenen Klone eine echte Obermenge von DECKARD. Das heißt es werden auf jeden Fall alle Klone gefunden, die DECKARD auch findet. Darüber hinaus fügt Bryant einen neuen Schritt der Vektorgenerierung hinzu, der aus dem PDG heraus Klonkandidaten erstellt. Durch die Zuhilfenahme des PDG bringt das Verfahren ein gewisses semantisches Bewusstsein in die Klonerkennung ein, da der PDG grundlegende Aussagen über die Semantik eines Programms treffen kann.

---

[44] Die Verteilung muss „2-stable“ sein, siehe [2].
[45] Siehe [20].
[46] Aus [2].
[47] Dieses Kapitel baut zu weiten Teilen auf dem Kapitel 3.3 aus [4] auf, da dort dieses Verfahren eingeführt wurde.
[48] Nach [4].

Zunächst wird die Definition eines Codeklons leicht erweitert. Die bisherige Arbeitsdefinition eines Codeklon ist wie folgt:

***Definition 4 (Syntaktische Codeklone)[49]**: Zwei disjunkte, zusammenhängende Programmsequenzen S_1 und S_2 sind Codeklone, genau dann wenn $\delta(S_1, S_2)$*

Wobei δ eine Ähnlichkeitsmetrik darstellt. In Bezug auf DECKARD ist das die in Kapitel 2.3 vorgestellte Edit-Distanz. Diese Definition wird nun um semantische Klone erweitert.

***Definition 5 (Semantische Codeklone)[50]**: Zwei disjunkte, möglicherweise nicht-zusammenhängende Programmsequenzen S_1 und S_2 sind semantische Codeklone, genau dann wenn S_1 und S_2 syntaktische Codeklone sind oder $\mu(S_1)$ isomorph ist zu $\mu(S_2)$.*

In dieser Funktion wird eine neue Abbildung μ verwendet, das sogenannte syntaktische Abbild. Diese Abbildung ist nötig, da bisher noch kein performanter und skalierbarer Algorithmus existiert, um semantische Klone zu erkennen. Die direkte Implementierung führt zu einer kombinatorischen Explosion an neuen möglichen Klonkandidaten, sowie zu einer sehr aufwändigen Berechnung der Graphisomorphie[51]. Bryant macht sich die Tatsache zunutze, dass sowohl der AST, als auch der PDG auf dem Programmcode basieren. Daher wird eine Funktion definiert, die zu einem gegebenen PDG den zugehörigen Quellcode findet.

***Definition 6 (Syntaktisches Abbild)[52]**: Das syntaktische Abbild eines PDG Teilgraphen G, genannt $\mu(G)$, ist die maximale Menge an AST-Teilbäumen, die zu der konkreten Syntax der Knoten in G gehören. Diese Menge ist dominierend, d.h. für jedes Paar von Bäumen $T, T' \in \mu(G)$ gilt $T \subseteq T'$.*

Durch die Abbildung von PDG auf AST-Knoten reduziert sich das komplexe Graphähnlichkeitsproblem zu einem Baumähnlichkeitsproblem. Weiterführend kann die Ähnlichkeit der Bäume bereits mit den charakteristischen Vektoren und dem LSH-Verfahren gelöst werden.

Damit schließen die durch PDG erzeugten Klonkandidaten nahtlos an die durch DECKARD erzeugten Klonkandidaten an. Den gesamten PDG als einen Kandidaten zu verwenden ist allerdings nicht sinnvoll, daher werden nur Teilbereich des PDG als Klonkandidat verwendet. Die richtige Auswahl dieser Kandidaten ist nicht trivial und wird im folgenden Kapitel beschrieben.

---

[49] Siehe [4].
[50] Siehe [4],
[51] Graphisomorphie liegt in NP und es ist keine effiziente Implementierung bekannt. Obwohl die Zahl der PDG-Knoten in aller Regel relativ gering sein wird, ist es trotzdem im Allgemeinen eine sehr aufwändige Berechnung.
[52] Siehe [4].

2.6.1 Auswahl der PDG-Teilgraphen[53]

Der Algorithmus muss für eine bestmögliche Klonerkennung eine (endliche) Menge an interessanten Teilgraphen des PDG erzeugen. Diese Teilgraphen gehen dann in die Klonerkennung als Kandidaten ein und werden im weiteren Verlauf mit anderen Kandidaten verglichen. Gabel et al stellen zwei Varianten der Teilgraph-Auswahl vor. Beide Varianten werden in diesem Kapitel besprochen.

```
int func (int i, int j) {
  int k = 10;

  while (i < k) {
    i++;
  }

  j = 2 * k;

  printf("i=%d, j=%d\n", i, j);
  return k;
}
```

Codebeispiel 3: Basis für den folgenden PDG[54].

[53] Das Verfahren wurde übernommen von [4], Kapitel 3.3. Da in dieser Arbeit alle Grundlagen für diese Diplomarbeit geschaffen wurden, hält sich die folgende Beschreibung eng an die Vorlage.
[54] Übernommen aus [4], Figure 1, Seite 322.

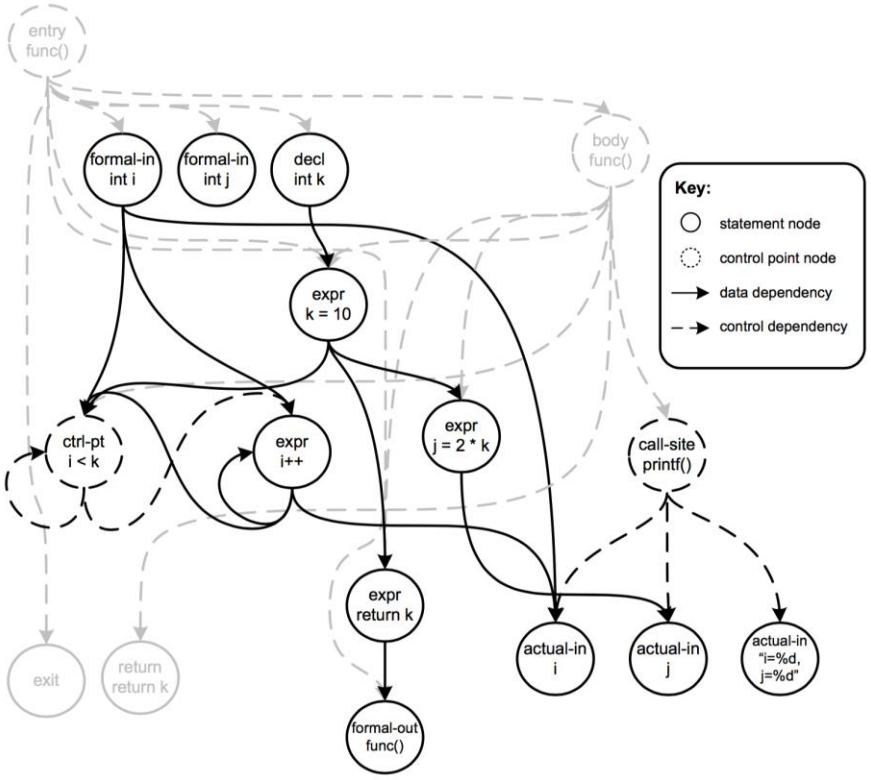

Abbildung 11: Beispielhafter PDG[55]

Ein PDG besitzt viele technische Basisknoten, wie in Abbildung 11 zu sehen ist („`body`“, „`exit`“, „`entry`“). Diese Knoten werden, wie in der Abbildung angedeutet, von der Erstellung der Codefragmente ausgenommen, da diese keine Relevanz für das Verfahren haben und wie im Beispiel des „`entry`“-Knoten zu einer unnötig hohen Überdeckung der Teilgraphen führen.

(Schwache) Zusammenhangskomponente[56]

Die einfachste und konservativste Implementierung ist sicherlich mittels Zusammenhangskomponenten. Hierbei werden alle gerichteten Kanten des PDG durch ungerichtete Kanten ersetzt und die Zusammenhangskomponenten gesucht.

Der Hintergrund ist, dass zwei voneinander unabhängige Statements, wie beispielsweise Zeile 5 (increment) und 8 (Zuweisung) in Codebeispiel 3, keine direkte Verbindung haben, durch den Aufruf von `printf` aber trotzdem verbunden sind. Zwei Statements sind nur dann unabhängig voneinander, wenn sie in unterschiedlichen Zusammenhangskomponenten stehen.

Die Implementierung der Erkennung von schwachen Zusammenhangskomponenten ist mit einer Tiefensuche mit linearem Aufwand möglich.

[55] Das Bild stammt aus [4], Figure 3, Seite 322.
[56] Siehe das Kapitel 7.2 im Appendix für eine kurze Definition von Zusammenhangskomponenten.

Semantic Threads

Obwohl Zusammenhangskomponenten bereits interessante Ergebnisse in der Klonerkennung liefern, wird eine feingranularere Graphselektion benötigt. Folgendes Codebeispiel veranschaulicht das allgemeine Problem mit einem Verbot jeglicher Überlappung der Teilgraphen[57]:

```
struct coordinates *generate_example_point () {
  struct coordinates *point = malloc(sizeof(struct coordinates));
  int x = 0;
  int y = 0;
  int z = 0;

  // aufwändige Berechnung von x
  // …

  // aufwändige Berechnung von y
  // …

  // aufwändige Berechnung von z
  // …

  point->x = x;
  point->y = y;
  point->z = z;
  return point;
}
```

Codebeispiel 4: Das Problem mit Zusammenhangskomponenten[58]

Durch die Zusammenführung der komplett separaten Berechnungen von `x`, `y` und `z` am Ende der Funktion wird in den letzten vier Zeilen die gesamte Funktion zu einer einzigen, großen Zusammenhangskomponente – und das obwohl die Funktion offensichtlich drei separate Berechnungszweige hat.

Ein Weg dieses Problem zu umgehen, ist die Verwendung von Forward Slices[59]. Ein Forward Slice ist das Ergebnis einer speziellen Form des Program Slicing[60]. Die Arbeit von Gabel et al verwendet eine leicht vereinfachte Definition. Hierbei werden von einer Variablen an einem Punkt in einem Programm aus alle folgenden Programmabschnitte gesammelt, die direkt oder indirekt von dem Wert dieser Variablen abhängen. Wenn bereits ein PDG vorliegt, ist die Berechnung eines Forward Slices trivial:

[57] Zusammenhangskomponenten verbieten per Definition jegliche Überlappung.
[58] Frei adaptiert von [4], Figure 6, Seite 325.
[59] Beschrieben in [21].
[60] Beschrieben in [22].

Definition 7 (Forward Slice)[61]*: Sei G ein PDG der Prozedur P, und sei s ein Statement in P. Der statische intraprozedurale Forward Slice von s über P ist definiert als die Menge aller von s in G erreichbaren Knoten.*

Im obigen Beispiel der Koordinatenberechnung ergeben die drei Initialisierungen von `x`, `y` und `z` drei separate Forward Slices, die erst in den letzten vier Zeilen eine Überlappung haben. Diese Datenflüsse werden fortan als semantische Threads bezeichnet.

Definition 8 (Semantischer Thread)[62]*: Ein semantischer Thread einer Prozedur P ist entweder ein Forward Slice oder die Vereinigung mehrerer Forward Slices.*

Diese semantischen Threads haben in der Regel einen Grad an Überlappung mit anderen semantischen Threads. Im obigen Beispiel ist die Überlappung relativ gering, daher ist anzunehmen, dass die unterschiedlichen Datenflüsse unterschiedliche Berechnungen und Programmaspekte darstellen. Ist die Überlappung zweier semantischer Threads allerdings zu hoch, ist davon auszugehen, dass die zwei einzelnen Threads zu einer höheren, größeren Berechnung gehören. Daher werden diese stark überlappenden Threads zu einem zusammenfassenden Thread gruppiert.

```
int example () {
  int a = 1;
  int b = 2;

  for (int j = 0; j < b; j++) {
    // umfangreiche Berechnung mit j, a und b
  }

  // …
}
```

Codebeispiel 5: Beispiel für stark überlappende semantische Threads.

In Codebeispiel 5 ist direkt ersichtlich, dass obwohl die Initialisierungen von `a` und `b` mehrere semantische Threads erzeugen, diese in Wirklichkeit zu einer größeren Berechnung im Rumpf der Schleife gehören. Dies als getrennte Threads anzusehen wird nicht nur redundante Codeklone erzeugen, sondern könnte auch die Sicht auf größere Zusammenhänge verwehren. Daher werden die interessanten γ-überlappenden semantischen Threads[63] $IST(P, \gamma)$ definiert, die eine Auswahl von semantischen Threads treffen, die möglichst gute Kandidaten für semantische Klone erzeugen.

[61] Aus [4], Definition 3.4.
[62] Aus [4], Definition 3.5.
[63] Fortan kurz „ISTs“.

Definition 9 (interessante semantische Klone)[64]*: die Menge der interessanten, γ-überlappenden semantischen Threads ist eine endliche Menge von semantischen Threads mit folgenden Eigenschaften:*

1. Die Menge ist vollständig, ihre Vereinigung entspricht dem PDG.

2. Die Menge darf keine Threads enthalten, die vollständigen in anderen Threads enthalten sind.
$\nexists sl, sl' \in IST(P, \gamma): sl' \subseteq sl$

3.Zwei beliebige Threads in der Menge dürfen sich in maximal γ Knoten überschneiden:
$\forall sl, sl' \in IST(P, \gamma): |sl \cap sl'| \leq \gamma$

4.$IST(P, \gamma)$ ist maximal, d.h. es hat die maximale Größe aller Mengen, die Eigenschaft 1-3 erfüllen.

Mit einer maximalen Überdeckung von $\gamma = 1$ werden für das Codebeispiel 5 drei separate, semantische Threads erzeugt. Ein Spezialfall stellt $\gamma = 0$ dar, denn dann werden genau die Zusammenhangskomponenten erzeugt.

Einen Algorithmus für die Berechnung der $IST(P, \gamma)$ stellen Gabel et al vor[65]. Der vorgestellte Algorithmus ist simpel in der Implementierung und hat im schlechtesten Fall kubische Laufzeit. Dies stellt allerdings kein Problem dar, da in der Regel die Anzahl der Knoten in den semantischen Threads gering ist. Darüber hinaus skaliert der Algorithmus auch gut für größere Programme, da dort eher die Zahl der Prozeduren steigt, aber nicht zwangsläufig deren Länge.

2.7 Ergebnisfilterung

Die in den vorangegangenen Schritten erzeugten Codefragmente können, je nach Anforderung an die zugehörigen Generator mit unterschiedlichen Ansätzen erstellt worden sein. Eine Variante ist, dass möglichst viele Fragmente erstellt werden, die anschließend gefiltert werden, eine anderen Alternative ist ein aufwändigeres Erzeugungsverfahren, das feiner hinsichtlich der erzeugten oder ausgelassenen Vektoren entscheidet.

Die Ergebnisfilterung ist dahingehend trivial, dass das Ergebnis der LSH-Queries eine Liste an Klonpaaren sind, die paarweise verglichen werden müssen. Diese Paare bestehen aus zwei Codefragmenten, die die unterschiedlichen Stellen im Quellcode markieren.

[64] Aus [4], Definition 3.6.
[65] Siehe [4], Algorithm 1.

Die redundanten Einträge der Liste der Klonpaare müssen für ein aussagekräftiges Ergebnis ausgefiltert werden. Dies schließt ein:

- Klonpaare, bei denen beide Fragmente auf den gleichen Code zeigen.
- Klonpaare, bei denen das eine Fragment ein Teilstück des anderen Fragments ist.
- Und abschließend die Filterung zwischen den unterschiedlichen Klonpaaren, mit der Überprüfung, ob die beinhalteten Fragmente entweder äquivalent oder in den anderen Fragmenten enthalten sind.

Diese Überprüfung muss performant umgesetzt werden, da die Zahl der erstellten Klonkandidaten sehr groß werden kann.

3 Implementierung

Nachdem das theoretische Fundament gesichert ist, wird die Implementierung mit Anbindung an das Bauhaus-Framework besprochen. Hierbei wurde ein Programm namens Bryant[66] implementiert, dass an das Framework andockt und die Codeklonerkennung implementiert. Bryant, sowie der größte Teil von Bauhaus, sind in Ada geschrieben[67].

3.1 Bauhaus[68]

Das Projekt Bauhaus, eine Kooperation der Universität Stuttgart, der Universität Bremen und der Firma Axivion GmbH ist ein Analyseframework für Softwaresysteme. Es kann auf unterschiedlichen Ebenen eingesetzt werden, wie beispielsweise zur Beobachtung von Softwarearchitektur, zur Ermittlung geklonter Quelltexte oder für die automatische Generierung von Qualitätsmetriken. Ein weiterer Schwerpunkt ist die automatische Identifikation von Programmierfehlern, mit speziellem Fokus auf Synchronisationsfehler nebenläufiger Programme.

Abbildung 12: Aufteilung der Fachbereiche bei Bauhaus[69].

Im Zentrum von Bauhaus steht der IML-Graph, der ein Abbild der Programmstruktur, inklusive des ASTs, aller Typdeklaration und der darauf aufbauenden Daten gibt. Die IML ist eine sprachübergreifende Zwischendarstellung, in die Bauhaus jedes Programm nach der Kompilierung überführt. In dieser Zwischendarstellung können einheitliche Analysen durchgeführt werden, die unabhängig vom verwendeten Frontend funktionieren – neu hinzugefügte Sprachen im Frontend können direkt von den bereits existierenden Analysen profitieren.

Durch den IML kann mittels unterschiedlicher Sichten navigiert werden. Diese Sichten stellen eine Fragestellung dar, die der Anwender im Graph beantworten will, wie

[66] Auch dieser Name ist – wie DECKARD – dem Film Blade Runner (1982) entliehen. Harry Bryant ist der Vorgesetzte von Rick Deckard und ist sozusagen für dessen Funktionieren im größeren Gesamtgefüge zuständig.
[67] Codebeispiele werden sich daher an Ada orientieren – in aller Regel wird dies aus Platzgründen aber kein vollständiger, lauffähiger Programmcode sein.
[68] Die Einführung basiert auf der Selbstbeschreibung der offiziellen Website der Universität Stuttgart zu Bauhaus, siehe [23].
[69] Von [23].

beispielsweise den Verlauf von Datenflussanalysen oder zugehörige Klongruppen eines AST-Knotens.

Bauhaus ist aus mehreren separaten Programmen aufgebaut[70], die für die Durchführung der Analysen nacheinander ausgeführt werden. Die Analyseprogramme erwarten als Eingabegraph immer den Ergebnisgraph des vorigen Schrittes. Die unterschiedlichen Programme annotieren und erweitern den IML-Graphen immer weiter, so dass dieser stetig reicher an Information wird. Dadurch können spätere Programme auf den Ergebnissen voriger Schritte aufbauen.

Auch die Implementierung von Bryant erzeugt eine eigenständige Binary, die als Eingabe einen in einer Datei gespeicherten IML-Graph erwartet. Die einzige Bedingung eines korrekten Durchlaufs ist, dass die PDG-Analyse erfolgreich durchgeführt werden kann. Dies bedeutet, die Voraussetzungen sind identisch mit denen der PDG-Analyse, was die gesamte Analysekette inklusive der SSA-Form einschließt.

3.2 Überblick über Bryant

Einige essenzielle Parameter des Programms sind über einen Konfigurations-Record steuerbar. Dieser ist aktuell statisch, einzelne (oder alle) Parameter davon können jedoch über das Kommandozeileninterface (CLI) verfügbar gemacht werden.

Der Programmablauf hält sich eng an den in Kapitel 2 beschriebenen Ablauf des Verfahrens. Die einzelnen Elemente sind in eigene Module ausgelagert. Diese sind voneinander komplett unabhängig und können einfach aktiviert oder deaktiviert werden, separat weiterentwickelt und getrennt betrieben werden. Die folgende Abbildung gibt einen Überblick über die Programmstruktur.

[70] Die Tools erzeugen zwar unterschiedliche Binaries, verwenden intern aber die gleichen Bibliotheken aus allgemeinen Funktionen und Typdeklarationen.

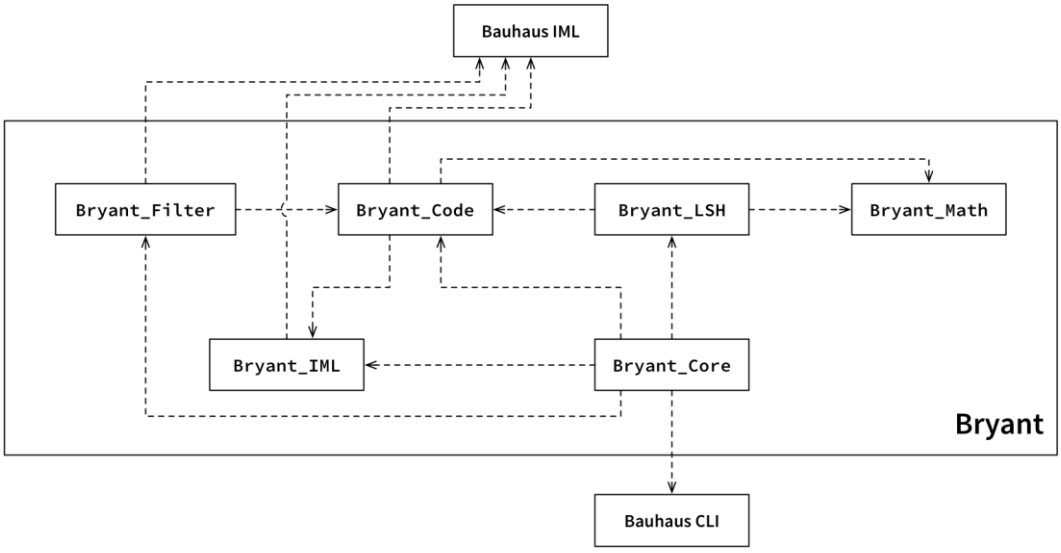

Abbildung 13: Übersicht über die verschiedenen Komponenten von Bryant. Die Pfeile kennzeichnen die Abhängigkeiten. Die Komponenten `Bryant_Debug` und `Bryant_Reporting` sind hier ausgenommen, da sie für die generelle Funktionalität des Tools nicht relevant sind.

Das System besteht aus den folgenden Komponenten:

`(Core)`[71]: Main-Funktion und die Konfigurationsdateien.

`Bryant_Code`: Logik für die Generierung der AST- und PDG-basierten Codefragmente (ein Fragment ist eine Kombination aus IML-Knoten und dem zugehörigen charakteristischen Vektor).

`Bryant_Debug`: Hilfsfunktion um einige essenzielle Datenstrukturen auf dem CLI darzustellen.

`Bryant_Filter`: Ergebnisfilterung, die doppelte Klone oder Klone, die eine Untermenge anderer Klone sind, entfernt.

`Bryant_IML`: Ein Großteil des Adaptercodes zu IML. Dies umfasst den Code für die Indexierung der IML-Knoten und die Funktion um alle Routinen im gegebenen Programm zu finden.

`Bryant_LSH`: Berechnung der LSH-Parameter und das komplette LSH-Hashing und dessen Berechnungen.

`Bryant_Math`: Hilfsfunktionen zu mathematischen Berechnungen, für die Erstellung von Zufallszahlen und die Typ-Definition für die charakteristischen Vektoren.

[71] Diese Dateien liegen im Hauptverzeichnis.

`Bryant_Reporting`: Auswertungsdaten wie Laufzeit, Anzahl der Vektoren und Routinen. Hiermit können nach Durchlauf des Programms Auswertungen erstellt werden.

Im Folgenden wird das Zusammenspiel der einzelnen Komponenten beschrieben. Außerdem werden einige Implementierungsdetails erwähnt, die bei der Funktion des Klonerkenners eine wichtige Rolle spielen.

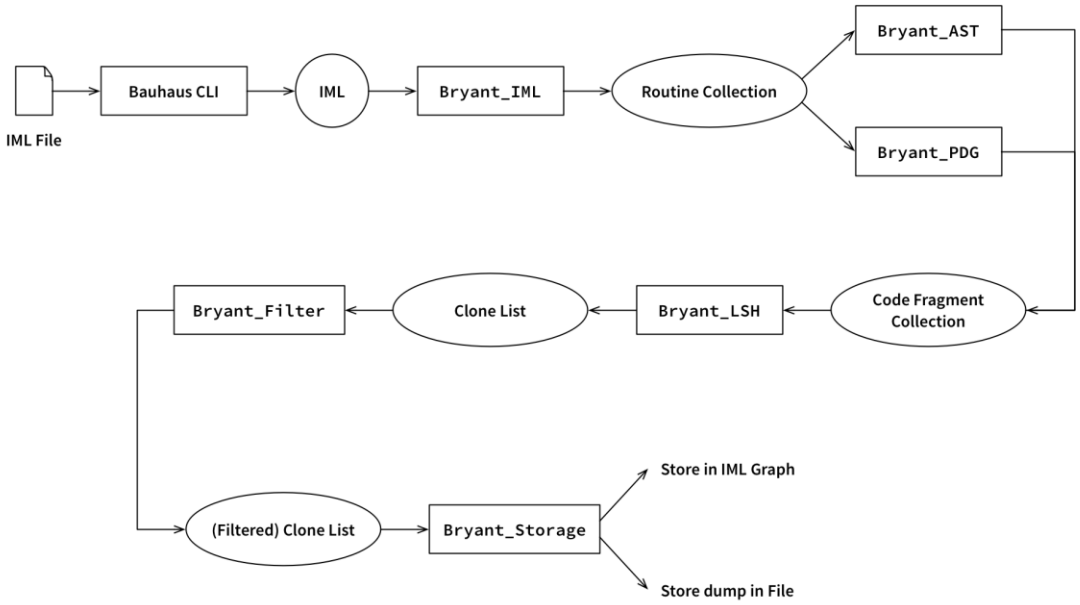

Abbildung 14: Überblick über den Programmfluss und die Zwischenergebnisse in Bryant. Die Kästen kennzeichnen Komponenten, die Kreise und Ellipsen kennzeichnen Zwischenergebnisse (also Daten).

Zunächst wird über die CLI durch Bauhaus der IML bereitgestellt. Dies ist der Einstiegspunkt der eigentlichen Implementierung. Anschließend werden aus dem IML-Graphen alle Routinen extrahiert. Über diese Routinenliste wird iteriert und jede Routine ist Eingabe aller Code-Fragment-Generatoren. Am Ende dieser Phase ist das Zwischenergebnis eine Liste mit allen Code-Fragmenten. Diese wiederum sind erst Eingabe um die LSH-Datenstruktur zu befüllen, anschließend wird mit jedem Fragment aus der Liste nach den benachbarten Vektoren gesucht. Die gefundene Klonliste wird abschließend noch gefiltert.

An diesem Punkt ist die eigentliche Klonsuche abgeschlossen, das Ergebnis sollte allerdings noch persistiert werden. Hierfür gibt es Storage-Adapter, die für die Speicherung im IML oder für anderweitige Speicherziele verantwortlich sind.

3.2.1 Adaption der Referenzimplementierung

Die Originalimplementierung von DECKARD[72] liegt unter einer Open-Source-Lizenz vor und dient teilweise als Vorlage für die Implementierung von Bryant. Die Originalimplementierung ist eine Mischung aus C, C++, Python, Shell-Programmen und einigen anderen Sprachen. Der Kern ist jedoch in den ersten drei Sprachen implementiert, Shell dient hauptsächlich für die Erstellung komfortabler ausführbarer Dateien, die den Kern in korrekter Reihenfolge aufrufen.

Von einer direkten Anbindung der Originalimplementierung (beispielsweise via FFI) wurde abgesehen, da einzig die AST-Vektorgenerierung und LSH adaptierbar gewesen wäre. Die Menge des eingesparten Codes wäre durch die notwendige Umformatierung der Daten nahezu annulliert worden.

Bei enger Betrachtung ist die einzig tatsächlich interessante Bibliothek für eine externe Einbindung LSH. Hier spricht gegen eine Adaption aber, dass die Größe der Bibliothek relativ klein ist (und daher schnell nachprogrammiert ist), und dass die Bibliothek seit der Originalimplementierung von 2005 nicht weiterentwickelt wird. Wenn in der Bibliothek in Zukunft rege Entwicklungsarbeit stattfindet, lohnt sich hier allerdings eine Neubewertung.

Ein rechtlicher Aspekt der gegen eine Adaption der LSH-Implementierung E²LSH[73] spricht ist, dass die Implementierung zwischenzeitlich unter GPL steht. Die Implementierung von E²LSH, die in DECKARD eingebettet ist, steht noch unter der MIT-Lizenz, DECKARD selbst unter der Three-Clause-BSD Lizenz. Aber die aktuellste LSH-Implementierung, von den ursprünglichen Autoren, steht inzwischen unter GPL, eine Lizenz die durch ihr Copyleft nicht kompatibel mit der Lizenz von Bauhaus ist.

3.3 Extraktion der Routine

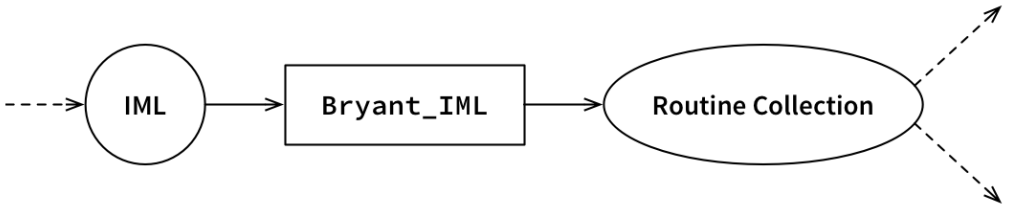

Abbildung 15: Phase in der Implementierung: IML / Routinen.

Der erste Schritt in Bryant ist, alle Routinen im IML zu finden. Die Implementierung ist relativ kurz, da Bauhaus eine Komponente bereitstellt, mit der man mittels des Visitor-

[72] Zu finden unter [24].
[73] Die Originalimplementierung, unter [26] zu finden.

Pattern auf allen Knoten des IML-Graphen Aktionen ausführen kann. Die hier implementierte Aktion ist einfach eine Liste an Routinen mitzugeben, an die bei jedem Routinen-Knoten der aktuelle Knoten angehängt wird.

Besondere Beachtung gilt in dieser Phase totem Code, insbesondere Routinen, die nie verwendet werden. Einige Analysen folgen den Aufrufen aus der main-Prozedur, sodass diese nie aufgerufenen Routinen nicht analysiert werden. Dies wird bei der Suche nach Routinen berücksichtigt. Bei jeder Routine wird überprüft, ob das „Pattern"-Attribute des zugehörigen IML-Knotens leer ist. Falls nicht, wird davon ausgegangen, dass diese Routine analysiert wurde und sie wird zur globalen Routinenliste hinzugefügt.

3.4 Vektorgenerierung

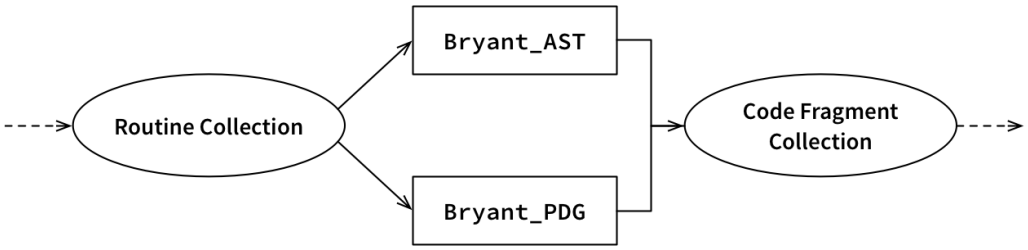

Abbildung 16: Phase in der Implementierung: Vektorgenerierung.

Die Vektorgenerierung aus der theoretischen Betrachtung wird zu einer Codefragment-Generierung in der Implementierung. Ein Codefragment ist eine Liste an IML-Knoten inklusive des zugehörigen charakteristischen Vektors.

Die Codefragmente sind die zentrale Datenstruktur der Implementierung. Ein Codefragment-Generator kann einfach in das System eingehängt werden, solange er das Interface erfüllt, er einen Routinen-Knoten erwartet und an die Vektorliste die ausgewählten Codefragmente anhängt. Durch diese wenig restriktive Bedingung können einfach neue, ausgefeiltere Klongeneratoren in das Verfahren eingebracht werden.

In der Implementierung bisher sind die zwei Generatoren für die Knoten direkt aus dem AST nach DECKARD und für die PDG Slices vorhanden.

3.4.1 Relevante und signifikante charakteristische Vektoren

Bevor die konkreten Algorithmen für das Auswählen der Codefragmente vorgestellt werden, müssen noch die Begriffe *relevanter Knoten* und *signifikanter Knoten* aus der theoretischen Betrachtung für die Implementierung definiert werden.

Relevante Knoten

Die relevanten Knoten definieren die IML-Knotentypen, die einen Einfluss auf den charakteristischen Vektor haben. Die Implementierung ist hierbei losgelöst von der Typstruktur des Bauhaus-IML-Moduls, um BRYANT als komplett eigenständiges Paket zu etablieren. Die wesentliche Aufgabe des `Bryant_IML`-Pakets besteht darin, einem IML-Knoten einen Index im charakteristischen Vektor zuzuweisen. Falls ein solcher Index gefunden wird, ist der Knoten automatisch als relevant definiert. Falls kein Index gefunden wird, ist der Knoten nicht relevant. Dies ist schnell ersichtlich in der zugehörigen Implementierung:

```
function Node_Index_Is_Relevant (Index : Integer) return Boolean is
begin
  return Index > -1;
end Node_Index_Is_Relevant;
```

Codebeispiel 6: Relevanz-Überprüfung eines Knotenindizes[74].

Die erwähnte Unabhängigkeit vom Kern von Bauhaus erkennt man daran, dass die Spezifikation der IML-Knoten nicht berührt wird. Ein gangbarer Weg wäre das Hinzufügen eines neuen Attributes an jeden IML-Knoten gewesen, der den Index zurückgibt. Dies hat aber mehrere Nachteile: zunächst trägt nun jeder Knoten Informationen über BRYANT, auch wenn das Tool selbst möglicherweise gar nicht verwendet wird. Außerdem ist es bei direkter Attributierung nicht trivial die Index-Zuweisung auszutauschen, was gerade bei der Evaluierung unterschiedlicher Relevanzkriterien interessant sein kann.

Daher verwendet die Implementierung eine Hashmap, die einem Tag[75] in Ada einen Index zuordnet. Eine alternative Implementierung wäre ein Test auf den `'Class`-Typ (pro Test also $O(d)$[76] Vergleiche), was den Vorteil hätte, dass die Erkennung die Typhierarchie ausnutzen könnte. Aus Performancegründen aber wurde der Weg der direkten Typvergleiche auf die Ada-Tags mit $O(1)$-Zugriff in der Hashmap implementiert – auch wenn für alle Untertypen ein eigener Eintrag hinzugefügt werden müsste. Da die Typhierarchie der IML-Knoten aber keinem konstanten Wandel unterliegt, ist der Wartungsaufwand für diese Implementierung im vertretbaren Rahmen.

Mit der Zuweisung von IML-Knoten auf Index wird auch ein weiterer Aspekt abgebildet. Einige IML-Knoten sollen explizit vereinheitlicht werden, so dass diese als Duplikate erkannt werden.

Die Conditionals werden als separate Einträge gehandhabt:

[74] Der Fehlerfall, dass ein zu großer Index zurückgegeben wird muss durch das Typsystem abgefangen werden.
[75] Der konkrete Objekttyp.
[76] d ist die Länge des charakteristischen Vektors.

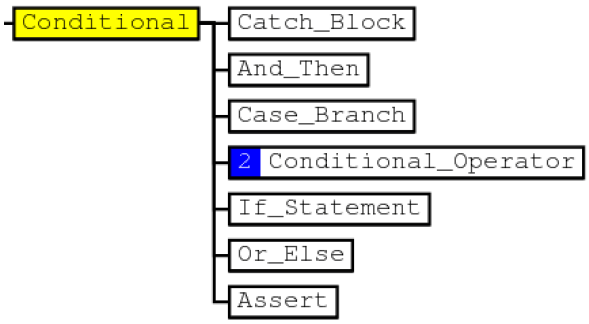

Abbildung 17: Conditionals in der IML-Spezifikation.[77]

Der zugehörige Initialisierungscode weist jedem Knoten einen unterschiedlichen Eintrag zu:

```
Mapping.Insert (Catch_Blocks.Catch_Block_Class'Tag,                    1);
Mapping.Insert (And_Thens.And_Then_Class'Tag,                          2);
Mapping.Insert (Case_Branchs.Case_Branch_Class'Tag,                    3);
Mapping.Insert (Conditional_Operators.Conditional_Operator_Class'Tag, 4);
Mapping.Insert (If_Statements.If_Statement_Class'Tag,                  5);
Mapping.Insert (Or_Elses.Or_Else_Class'Tag,                            6);
```

Codebeispiel 7: Indexierung der unterschiedlichen Typen von Conditionals in Bryant.

Bei den unterschiedlichen Typen von Zuweisungen soll allerdings nicht unterschieden werden:

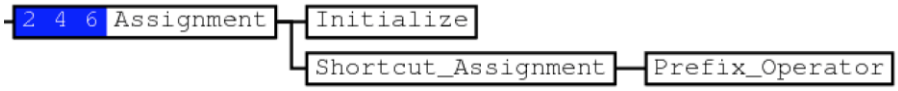

Abbildung 18: Assignments in der IML-Spezifikation.[78]

Sondern es sollen alle Zuweisungen explizit einheitlich behandelt werden. Zwei Codeabschnitte, die sich folglich einzig in der Art der Zuweisung unterscheiden werden als identisch erkannt.

```
Mapping.Insert (Assignments.Assignment_Class'Tag,                    23);
Mapping.Insert (Initializes.Initialize_Class'Tag,                    23);
Mapping.Insert (Shortcut_Assignments.Shortcut_Assignment_Class'Tag, 23);
Mapping.Insert (Prefix_Operators.Prefix_Operator_Class'Tag,          23);
```

Codebeispiel 8: Indexierung der unterschiedlichen Typen von Assignments in Bryant.

Durch diese bewusste Vereinheitlichung kann die Genauigkeit, also im Grunde das semantische Hintergrundwissen des Klonerkenners, kalibriert werden. Mit zu vielen *identischen* Indizes werden Programmabschnitte als identisch angesehen, die nicht

[77] Aus der Visualisierung der Spezifikation der Bauhaus-Dokumentation übernommen.
[78] Aus der Visualisierung der Spezifikation der Bauhaus-Dokumentation übernommen.

identisch sind. Mit zu vielen *unterschiedlichen* Indizes werden nur sehr (auch syntaktisch) ähnliche Programmabschnitte erkannt.

Signifikante Vektoren

Die relevanten Knoten definieren die IML-Knoten, die Einfluss auf den charakteristischen Vektor haben. Aber nicht jeder der relevanten Knoten ist ein geeigneter Kandidat um als AST-Wurzel für einen Klonkandidaten zu fungieren. Literale sind ein relevanter Knoten, jedoch ist ein Literal kein geeigneter Ursprungsknoten für einen Klon.

Daher wurden in Kapitel 2.4 signifikante Vektoren definiert. Dies sind Vektoren die geeignete Kandidaten für Klone sind – weil der Knoten vom richtigen Typ ist oder weil der Knoten ausreichend Kindknoten besitzt. Diese Einschränkung senkt nicht die Zahl der erkannten Klone: bei der AST-Generierung können trotzdem Kindknoten eines nicht-signifikanten Vektors zu einem Kandidatenfragment vereint werden, durch das Vector Merging.

Die getroffene Auswahl signifikanter Knoten basiert auf manuellen Tests, die eine möglichst gute Balance zwischen genug Vektoren und korrektem Ergebnis liefern. Die folgende Liste betrifft die Typhierarchie, es sind also immer der konkrete Typ selbst, als auch alle abgeleiteten Typen enthalten.

- Assignments
- Conditionals
- Loop_Statments
- Routine_Calls
- Unconditional_Branch

Diese Auswahl sollte in zukünftigen Anpassungen der Implementierung weiter überprüft und gegebenenfalls optimiert werden.

Um in die globale Liste aller Codefragmente (mögliche Klonkandidaten) aufgenommen zu werden, muss das Fragment nicht nur relevant und signifikant sein, sondern es muss auch eine ausreichende Größe besitzen. So sollen selbst für signifikante Knoten, die zu einer sehr geringen Menge an Quellcode gehören, keine Vektoren erstellt werden, da sonst die Gefahr von außerordentlich vielen Duplikaten besteht.

Daher wird eine Mindestgröße[79] des Vektors eingeführt, die festlegt, ob für diesen Vektor ein Fragment erstellt wird oder nicht. Diese Mindestgröße gilt für alle Vektoren – unabhängig davon, wie viel konkrete IML-Knoten in diesem Code-Fragment enthalten sind. Die Längenberechnung kann nicht ungewichtet sein. Intuitiv ist eine Initialisierung eines Arrays (5 Literale) weniger geeignet als Klonkandidat, als eine ganze Routine mit mehreren Assignments und If-Conditions (die insgesamt ebenfalls 5 Knoten sein können).

[79] Die Größe ist hier definiert als die Summe aller Einträge.

```
int[] example = {1, 2, 3, 4, 5};
```

Codebeispiel 9: Array-Initialisierung mit 5 Einträgen.

Der Code in Codebeispiel 9 erzeugt für die gesamte Zuweisung einen Vektor der Größe 7 (1x `L_Value`, 1x `Assignment`, 5x `Literal`). Unabhängig von der Länge ist der Vektor dennoch nicht als Wurzel eines Klonkandidaten geeignet. Aus diesem Grund zählt die aktuelle Implementierung momentan die Zahl der signifikanten Knoten. Es wurde eine Mindestzahl von 3 festgelegt – diese Zahl, aber auch die gesamte Logik um einen Knoten als mögliche Klonkandidatenwurzel festzulegen, kann einfach erweitert und getestet werden.

3.4.2 Codefragment-Generierung im AST

Die Implementierung der AST-basierten Codefragment-Generierung hält sich nahe an die Beschreibung des Algorithmus im zugehörigen Artikel[80]. Grob gesagt ist es ein Baumdurchlauf, bei dem in jedem Schritt „on-the-fly“ die Vektoren für den aktuellen Knoten erstellt werden. Hierfür ist nur ein einmaliger Durchlauf aller AST-Knoten des Baumes nötig.

Die AST-Knoten[81] werden in Post-Order-Reihenfolge durchlaufen, da die Vektoren von den Blättern des Baumes aufwärts erstellt werden müssen. Jeder Vektor eines Knoten ist die Summe aller Vektoren der Kinder, inklusive der Erhöhung des Eintrages im Knotenindex, falls der Knoten relevant ist.

Beim Vector Merging in DECKARD wird zunächst für jeden Knoten festgelegt, ob dieser zulässig für solch eine Kombination ist. Da DECKARD auf dem Parsebaum arbeitet, ist diese Unterscheidung wichtig, um unvollständige oder nicht sinnvolle Kombinationen von Knoten zu unterbinden. So sollen keine Knoten die über Block-Grenzen hinausreichen verbunden werden (das letzte Statement eines while-Bodys, mit dem darauffolgenden Statement). Dieses Problem findet sich in dieser Form nicht in einer auf einer AST-basierenden Implementierung. Nichtsdestotrotz können an dieser Stelle zusätzliche Überprüfungen hinzugefügt werden, die ungewollte[82] Kombinationen verhindern.

Bei dem Durchlauf der Kindknoten werden alle Kindknoten in einer Liste zwischengespeichert, aus der dann die zusammengesetzten Vektoren mehrerer IML-Knoten erzeugt werden. Hierbei wird ein *Sliding Window* über die Liste der Vektoren verschoben, ein neues Code-Fragment aus diesen IML-Knoten erstellt und der charakteristische Vektor erzeugt. Wenn dieser Vektor die Signifikanzkriterien erfüllt, wird er zur Liste der Klonkandidaten hinzugefügt. Die Relevanz wird für die

[80] Siehe [6].
[81] „Syntactic Children“ in IML.
[82] Welche Kombinationen im Detail ungewollt sind, muss vorher festgelegt werden.

kombinierten Fragmente nicht geprüft, da es eine Kombination aus mehreren Elementen ist.

Dieses Sliding Window wird nach jedem Durchlauf um den Faktor 1,5 vergrößert (abgerundet) und das Vector Merging beginnt erneut. Dieser Vorgang endet, wenn das Sliding Window länger als die Gesamtlänger der benachbarten Knoten ist. DECKARD implementiert in der Parsebaum-Umsetzung die besondere Bedingung, dass wenn sowohl der Elternknoten, als auch der Kindknoten zu einem Merge kombinierbar sind, der Kindknoten aus dem Sliding Window herausgenommen wird. Auf diese Weise sollen größere Klone favorisiert werden. Es ist noch unklar, wie sich diese Eigenschaft auf die Verwendung des ASTs überträgt. Gabel et al erwähnen diese Beschränkung nicht. In der Implementierung wurde diese Beschränkung so umgesetzt, dass wenn der Elternknoten signifikant ist, alle signifikanten Kindknoten aus dem Sliding Window entfernt werden.

3.4.3 Codefragment-Generierung im PDG

Die Implementierung der Fragmenterzeugung mithilfe des PDG ist in einigen Bereichen nur ein dünner Adapter um die Bauhaus-interne PDG-Komponente. So wird die PDG-Erzeugung und der zugehörige intraprozedurale Backward Slicer verwendet. Dieser Slicer unterstützt bereits die automatische Auftrennung strukturierter Rückgabewerte – so wird das in Codebeispiel 4 beschriebene Problem umgangen.

Ausgehend von der Liste der Backward Slices werden die ISTs erzeugt. Diese ISTs werden anschließend umgewandelt zu Codefragmenten. Dazu werden alle IML-Knoten der Slices zu einer Liste zusammengeführt und der charakteristische Vektor berechnet. Hier ist darauf zu achten, den charakteristischen Vektor für jeden IML-Knoten aus dem Slice *inklusive der AST-Kindknoten* zu berechnen. Nur so können die Vektoren letztendlich korrekt verglichen werden.

In der Implementierung wird von jeder Routine der sogenannte `Link_Out_Use`-Knoten geladen, der Informationen über Rückgabewerte und ausgehende Seiteneffektvariablen bündelt. Ausgehend von diesen Verbindungen in den umliegenden Code werden die Backward Slices erstellt.

Ein Problem dieser Implementierung ist, dass Code, der nicht zu den ausgehenden Verbindungen beiträgt nicht in den Slices auftritt. Dies ist Code, der eigentlich Seiteneffekte besitzt, diese von Bauhaus im Moment aber nicht als solche erkannt werden. Dies betrifft meist Funktionen und Prozeduren der Laufzeitumgebung. So wird Timing-Code (der als Seiteneffekt das Laden der aktuellen Uhrzeit besitzt) und Ausgabecode wie `printf` (der als Seiteneffekt die Ausgabe auf `stdout` besitzt) nicht korrekt markiert. Hierdurch entfallen einige mögliche Threads, da diese in den `Link_Out_Use`-Listen nicht auftauchen.

```
int func (int i, int j) {
  int k = 10;

  long start = get_time_millis();
  long finish;

  while (i < k) {
    i++;
  }

  finish = get_time_millis();
  printf("loop took %dms\n", finish – start);
  j = 2 * k;

  printf("i=%d, j=%d\n", i, j);
  return k;
}
```

Codebeispiel 10: Beispielcode, der sich von Codebeispiel 3 nur durch zusätzlich hinzugefügten Timing-Code unterscheidet.

Obenstehendes Beispiel wird durch die aktuelle PDG-Implementierung korrekt als Klon erkannt. Der Thread um den Timing-Code und um die Variablen `start` und `finish`, zusammen mit dem abschließenden `printf()`-Aufruf wird in der Betrachtung nicht auftauchen. Dies führt in diesem Beispiel weiterhin zu einem korrekten Ergebnis, kann aber beispielsweise doppelten Timing-Code oder alternative Fälle nicht korrekt erkennen.

Ein Spezialfall dieses Problems ist toter Code. Auch dieser wird nicht in den Backward Slices erscheinen. Hier gilt jedoch, dass dieser nicht relevant für diese Untersuchung ist und besser durch eine dafür ausgerichtete Toter-Code-Analyse gefunden und entfernt werden sollte.

Die Implementierung von Gabel et al umgeht dieses Problem, in dem sie Forward Slices für jeden Knoten im AST in aufsteigender Reihenfolge der Quellcode-Zeilennummern erzeugt. Durch dieses Verfahren wird auf jeden Fall jeder AST-Knoten der Routine zu mindestens einem IST hinzugefügt. Möglicherweise könnte diese Erweiterung bei der Bauhaus-PDG-Implementierung ebenfalls unterstützt werden[83].

Eine wichtige Abweichung zur Original-Implementierung von Gabel et al ist, dass Backward Slices anstelle von Forward Slices verwendet werden. Diese erstellen die Datenflüsse nicht vorwärts ausgehend von deklarierten und initialisierten Variablen, sondern rückwärts starten bei Rückgabewerten und Schreibzugriffe auf externe Ressourcen. Dies hat zunächst praktische Gründe (Backward Slices sind in dem zu dieser Arbeit externen Modul implementiert), aber auch kleine praktische Vorteile.

Während sich Timing-Code wie toter Code verhält, im Hinblick darauf, dass beide Varianten in aller Regel keine Datenabhängigkeiten in den aktiven Code haben, agiert

[83] Wobei dies eine Designentscheidung ist, da die Aussagekraft von Codeklonen in Timing-Code, Debug-Code und totem Code diskutiert werden kann.

Debug-Code anders. Wie beispielhaft im Codebeispiel 3 an dem `printf()`-Aufruf zu sehen ist, muss Debug-Code für eine sinnvolle Ausgabe zwangsläufig die existierenden Variablen verwenden. Dies erzeugt eine Datenabhängigkeit, die in einem Forward Slice korrekt erkannt wird. Durch diesen zusätzlichen Eintrag kann ein vorhandener Klon möglicherweise unerkannt bleiben, da der zusätzliche Debug-Aufruf die Vektoren zu weit auseinanderschiebt. Hier sind Backward Slices im Vorteil, da in diesen nur Code auftaucht, der zum „Endergebnis“ der Funktion (sei es eine Zuweisung an eine externe Variable oder ein Rückgabewert) führt. Das schließt reinen Debug-Code aus.

Hier herrscht nun allerdings offensichtlich ein Widerspruch zwischen dem Wunsch auf der einen Seite, seiteneffektbeladenen Laufzeitsumgebungs-Code in den Slices zu finden und der praktischen Eigenschaft auf der anderen Seite, dass Debug-Code in den Slices nicht auftaucht. Diese Entscheidung und Diskussion wird einer zukünftigen Optimierung überlassen, diese Implementierung verbleibt auf dem Stand eines dünnen Adapters um die bestehende PDG-Bibliothek, da die Erstellung von PDGs nicht Kernstück dieser Implementierung ist, sondern nur eine Komponente.

3.5 LSH

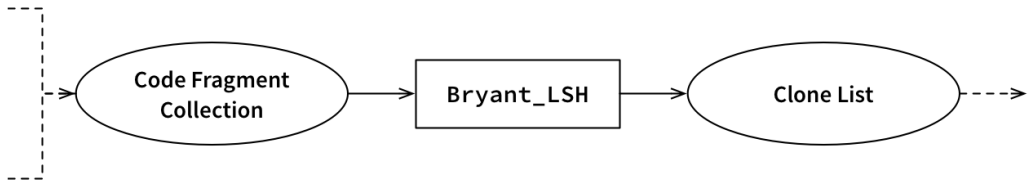

Abbildung 19: Phase in der Implementierung: Locality Sensitive Hashing.

LSH ist – zusammen mit dem Konzept der charakteristischen Vektoren – das Kernstück der Implementierung und des Verfahrens. Diese Kombination ist der Fortschritt, der dieses Verfahren erst praktikabel, performant und skalierbar macht.

3.5.1 Parametergenerierung

Das LSH-Verfahren benötigt eine Menge an Vorverarbeitung. Zunächst werden die LSH-Parameter berechnet, die die Kerneigenschaften des Verfahrens steuern. In dieser Implementierung werden die Parameter L und k statisch nach dem theoretischen Optimum aus Effektivität und geringer Laufzeit berechnet (siehe Kapitel 2.5.2 für die konkreten Formeln), es existieren jedoch alternative Ansätze. Im Ausblick (Kapitel 6) wird eine solche Alternative beschrieben.

Vor der Verwendung von LSH selbst wird über die `Bryant_LSH_Parameters`-Komponente die Konfiguration erstellt, die später als Eingabe in die Hauptfunktion des LSH eingeht. Diese Konfiguration beinhaltet alle für LSH wichtigen Parameter:

- Die Wahrscheinlichkeit, dass zwei kollidierende Hashes zu nahen Vektoren gehören p_1.
- Die Wahrscheinlichkeit, dass zwei kollidierende Hashes nicht zu nahen Vektoren gehören p_2.
- Die maximale Vektor-Entfernung von Codeklonen R.
- Die Breite der Segmente auf der reellen Achse w.
- Die Anzahl der Vektoren n (diese Zahl ist Eingabe für einige der anderen Parameter).
- Die Anzahl, wie oft ein Vektor hintereinander gehasht wird k.
- Die Anzahl, in wie viele Buckets der Vektor maximal gehasht wird L.
- Die Hashfunktionen g_i.
- Die Hashfunktion, die den durch `g()` berechneten Vektoren den Bucketindex zuweist.
- Die Hashmap, in der die Vektoren gespeichert werden.

Wie bereits am Parameter n ersichtlich ist, sind diese Parameter spezifisch für eine Codefragment-Liste. Wenn an diese Liste neue Elemente hinzugefügt oder entfernt werden, sollte die Datenstruktur neu erstellt werden. Die Hashmap selbst wird ebenfalls in den Parametern gespeichert. Dies spiegelt die Tatsache wider, dass die restlichen Parameter zu den Vektoren in der Hashmap gehören.

3.5.2 Befüllung der Hashmap

Nachdem die Parameter erstellt wurden, muss in einem einmaligen Durchlauf jedes Codefragment in die LSH-Struktur eingefügt werden. Dabei wird der Vektor in die L unterschiedlichen Buckets gehasht.

Die Verwendung einer Hashmapstruktur (im Gegensatz zu einem Vektor oder einem Array) ist eine bewusste Entscheidung: der Schlüsselraum der Hashmap ist nicht sehr dicht besiedelt, was bei Verwendung eines Arrays zu einem unnötig großen Speicherbedarf führen würde. Für solch einen Schlüsselraum ist die Verwendung einer Hashmap ideal.

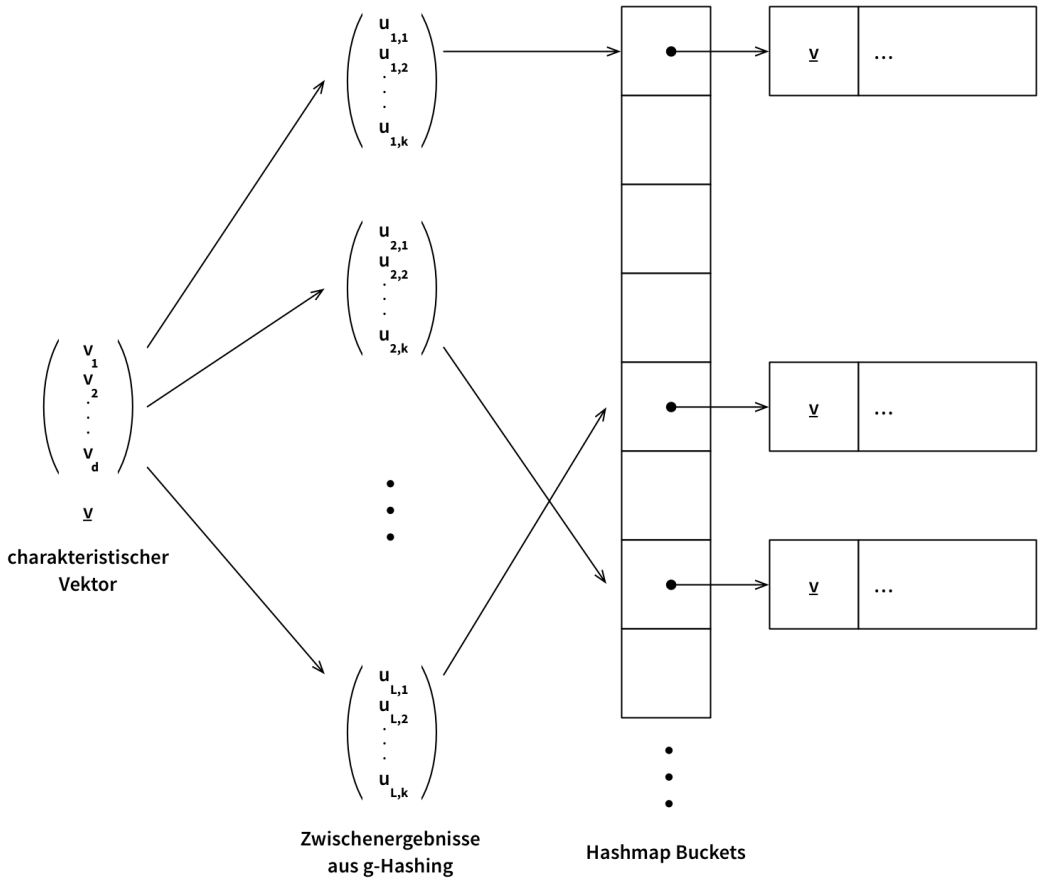

Abbildung 20: Zum besseren Verständnis noch einmal Abbildung 10 aus Kapitel 24, das den Ablauf des Hashings visualisiert.

Bei diesem Hashing wird zunächst ein neuer Float-Vektor mit k Einträgen erstellt. Jeder dieser Einträge ist das Ergebnis des h_i-Hashings ($i = 1 \dots k$)[84]. Diesem neuen Vektor wird mittels der Hashmap-Hashfunktion ein Bucketindex zugewiesen. Diese Hashmap-Hashfunktion besteht abermals aus einem Skalarprodukt mit einem zufälligen Vektor, Modulo einer großen Primzahl (um die Zahl der Kollisionen zu minimieren), Modulo der Hashmap-Größe (die auf die Anzahl der Vektoren gesetzt wird). Dadurch wird sichergestellt, dass auch für kleine Vektormengen keine weit verstreuten Bucketindizes errechnet werden.

Der Parameter L gibt die maximale Zahl der Buckets an, in die ein Vektor eingefügt wird. Für jedes $i = 1 \dots L$ gibt es eine eigene Hashfunktion g_i, das bedeutet es gibt insgesamt $k * l$ unterschiedliche Hashfunktionen h.

[84] Die gesamte Hashingfunktion, die den d-dimensionalen charakteristischen Vektor auf den mehrfach gehashten k-dimensionalen Vektor überführt, heißt g (wie in Kapitel 2.5.2 beschrieben).

In jedem Bucket ist eine Liste an Pointern zu in diesen Bucket gehashten Codefragmenten gespeichert.

3.5.3 Benachbarte Vektoren eines Vektors finden

Nachdem alle Punkte einmalig in die LSH-Datenstruktur eingefügt wurden, können nun Anfragen nach benachbarten Vektoren beantwortet werden.

Hierzu werden zu einem Anfragevektor durch die beiden Hashingfunktionen alle Bucketindizes berechnet. Zwischen dem Anfragevektor und allen Vektoren in diesen Buckets wird die tatsächliche Distanz (euklidische Distanzmetrik) berechnet. Alle Vektoren, zu denen die Distanz geringer als die definierte Maximaldistanz R ist, werden als Ergebnisliste zurückgegeben.

Eine kleine Optimierung ist hier verbaut: die Definition der euklidischen Distanz sieht am Ende der Berechnung vor, die Quadratwurzel aus der restlichen Berechnung zu ziehen[85]. Dies ist in aller Regel eine relativ teure Berechnung. Die Berechnung der Quadratwurzel kann hier ausgelassen werden, wenn das Ergebnis zum Quadrat der Maximaldistanz R^2 verglichen wird. R^2 wird einmalig bei der Berechnung der restlichen LSH-Parameter gespeichert und muss anschließend nicht neu berechnet werden.

3.6 Größensensitive Klonerkennung

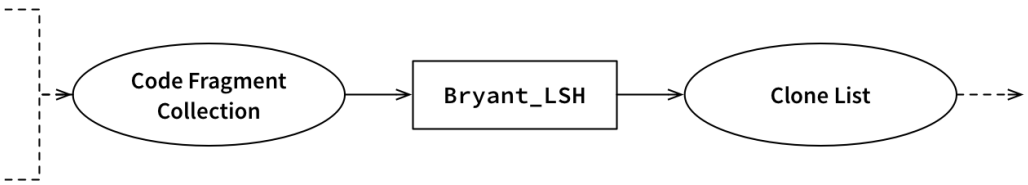

Abbildung 21: Phase in der Implementierung: größensensitive Klonerkennung.

Eine weitere Optimierung, die in DECKARD vorgestellt wird, ist „Size Sensitive Clone Detection“[86]. Bei diesem Verfahren werden nicht alle Vektoren zusammen durch LSH verglichen, sondern es wird eine zusätzliche Gruppierung vorgeschalten.

Dies hat zwei Gründe. Zunächst kann durch ein simples Abschätzen der Länge eines charakteristischen Vektors möglicherweise entschieden werden, ob dieser ein Klon mit einem anderen Vektor sein kann. Wenn man errechnet, dass die Längen zweier Vektoren um mehr als R auseinanderliegen, ist es unmöglich, dass diese zwei Vektoren Klone repräsentieren.

[85] Siehe Kapitel 7.1.
[86] Aus [6], Kapitel 3.4.

Ein weiterer Vorteil der Vorverarbeitung ist, dass unterschiedliche LSH-Parameter verwendet werden können. Bei Codefragmenten die zu größeren Quellcode-Abschnitten gehören, kann die maximale Edit-Distanz im Vergleich zu Fragmenten, die sehr kurze Abschnitte repräsentieren, vergrößert werden. Dies ermöglicht es, die Klonerkennung adaptiv an die Größe der zugrundeliegenden Code-Abschnitte anzupassen.

Beide zusätzlichen Faktoren (Länge des Vektors und Anzahl Zeilen im Quellcode) lassen sich einmalig in der Vektorgenerierung erzeugen und können als zusätzliche Einträge im Code-Fragment gespeichert werden. Die Gruppierungssegmente können frei definiert werden, jedoch sollten sich diese um mindestens R überschneiden, so dass gesichert ist, dass weiterhin alle Klone korrekt erkannt werden.

In Bryant ist diese Optimierung nicht verbaut. Bei der Evaluierung wurde festgestellt, dass die vorliegende Implementierung aus reinen Laufzeiteigenschaften die Optimierung im Moment nicht benötigt. Bei mehreren Tests wurde gemessen, dass die aktuelle Implementierung (für Initialisierung und Querying für alle Vektoren) etwa 500.000 bis 1,5 Millionen Vektoren pro Sekunde schafft. Dies ist für das Laufzeitverhalten ausreichend, sodass die zusätzliche Komplexität explizit nicht in das System aufgenommen wurde.

3.7 Nachbearbeitung der Ergebnisliste

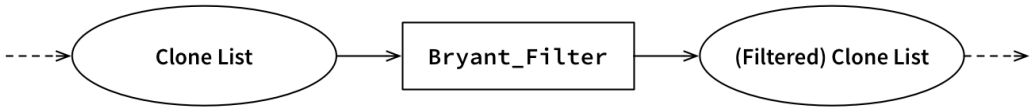

Abbildung 22: Phase in der Implementierung: Ergebnisfilterung.

Nachdem in der Initialisierung von LSH mit allen Codefragmenten die zugehörigen Buckets gefüllt wurden, können anschließend wieder mit einer Iteration über alle Codefragmente die zugehörigen Klone erfragt werden. Diese unbearbeitete Klonliste kann doppelte Ergebnisse enthalten, sowie Ergebnisse mit IML-Knoten, die vollständig in einem anderen Ergebnis enthalten sind. Beispielsweise könnte der Rumpf einer Schleife ein Klon zu einem anderen Fragment sein, die Schleife selbst wurde allerdings ebenfalls als Klon erkannt. In diesem Fall ist immer der größtmögliche Klon interessant, das heißt Duplikate und Teilergebnisse können aus der Liste entfernt werden.

Dies geschieht in der Implementierung in einem Zweipass-Verfahren. Im ersten Durchlauf werden exakte Duplikate aus der Klonliste entfernt. Dies sind Codefragmente, bei denen alle zugehörigen IML-Knoten identisch in einem anderen Fragment enthalten sind.

Im zweiten Durchlauf werden die Typhierarchien betrachtet. Nun werden alle Fragmente entfernt, deren IML-Knoten Kinder der IML-Knoten des anderen Fragments sind. So wird das oben erwähnte Beispiel mit der Schleife entdeckt und herausgefiltert.

Das Ergebnis ist eine Liste an Klonen, die die maximale Anzahl eindeutiger Klone mit maximaler Größe enthält. Dieser Durchgang ist wichtig. Im Regelfall liegt bei naiver Vektorgenerierung der Anteil der redundante (da doppelt vorhanden oder ein AST-Unterbaum eines anderen Unterbaums) bei *70-80%*.

3.8 Speicherung der Ergebnisse

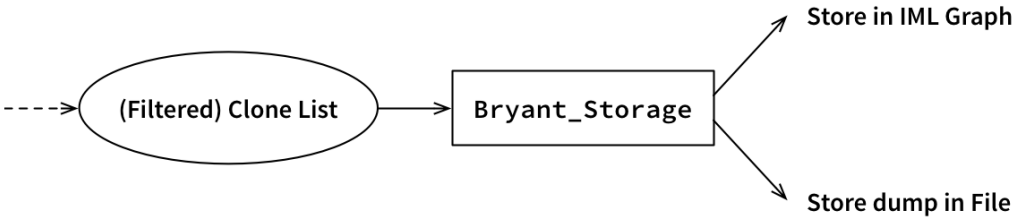

Abbildung 23: Phase in der Implementierung: Speicherung der Ergebnisse.

Die Ergebnisse werden als neue Knoten an den IML-Graphen angehängt und erzeugen damit eine neue Sicht innerhalb des Graphen. Dadurch können in Zukunft die Klongruppen auch in Hilfsprogrammen wie dem IML-Navigator angesehen werden.

Zusätzlich dazu bietet Bryant eine weitere „Speicherart“ an: die Ausgabe auf die Kommandozeile. Hierzu wird von jedem Klonpaar die jeweiligen Dateien inklusive aller im IML verknüpften Zeilennummern angezeigt. Mit Hilfe dieser Ansicht können schnell die Ergebnisse validiert werden.

3.9 Weitere Details der Implementierung

Eine für die Korrektheit von LSH wichtige Eigenschaft wurde in der Theorie zwar beschrieben, blieb im Kapitel über die Implementierung bisher allerdings unerwähnt. Die zugrundeliegende Verteilung der Zufallsvariablen ist essenziell. Wie in Kapitel 2.5.2 beschrieben ist es möglich, eine unter der l_2-Norm stabile Zufallsvariable aus zwei unabhängigen, normalverteilten Zufallsvariablen aus dem Intervall $[0; 1]$ zu erzeugen. Die zugehörige Implementierung wurde direkt von E²LSH[87] adaptiert und sieht wie folgt aus:

[87] Siehe [26].

```
function Generate_Gaussian_Random return Float is
  X1 : Float;
  X2 : constant Float := Generate_Uniform_Random (0.0, 1.0);
begin
  loop
    X1 := Generate_Uniform_Random (0.0, 1.0);
    exit when X1 > 0.0;
  end loop;

  return Math.Sqrt (-2.0 * Math.Log (X1)) * Math.Cos (2.0 * Ada.Numerics.Pi * X2);
end Generate_Gaussian_Random;
```

Codebeispiel 11: Erzeugung einer unter der l_2-Norm stabilen Zufallsvariablen.

Eine kleine Optimierung ist, dass die Anzahl der in einem Codefragment enthaltenen *signifikanten Knoten direkt aus dem charakteristischen Vektor abgelesen* werden kann. Je nach Zuordnung der Indizes der Knotentypen können alle signifikanten Vektoren eigene Einträge erhalten. Die Summe dieser Einträge ist dann die Anzahl der signifikanten Vektoren. Diese Implementierung stellt diese Bedingung an die Indizierungsfunktion allerdings nicht, hier wird die Anzahl der signifikanten Knoten in einem separaten Feld des Codefragments gespeichert.

Eine weitere Entscheidung war es, die *charakteristischen Vektoren als Float-Vektoren* zu implementieren und nicht, wie vermutlich intuitiv angenommen, als Natural-Vektoren. Dies hat den Hintergrund, dass die Einträge der Hashing-Vektoren in den Hashfunktionen $h_{k,L}$ ebenfalls Floats sind (es sind Zufallszahlen im Intervall $[-1; 1]$) und daher die häufige Umwandlung der Typen beim Hashing nicht notwendig ist. Dies führt jedoch dazu, dass für das Addieren der Vektoren die im Vergleich zu Integern langsamere Float-Berechnung verwendet wird. Auf modernen CPUs sollte dieser Unterschied jedoch im Vergleich zu den häufigen Umwandlungen nicht relevant sein.

Die *gewählte Konstante in der L Bucketindex-Hashing-Funktion* sollte zwei wichtige Eigenschaften besitzen. Sie sollte möglichst hoch sein, dass selbst bei vielen Vektoren der Schlüsselraum nicht unnötig verkleinert wird. Außerdem sollte sie eine Primzahl sein, um die Zahl der Kollisionen zu minimieren. Aus diesem Grund wählt E²LSH die Zahl $2^{32} - 5$[88], die in dieser Implementierung übernommen wurde.

Die *Einträge des Hashingvektors im Bucketindex-Hashing* sind Zufallszahlen zwischen 0 und 2^{29}. Dieser Wert wurde ebenfalls von der E²LSH-Implementierung übernommen. Dort wurde dieser Wert empirisch bestätigt.

[88] Ein weiterer wichtiger Grund für die Wahl dieser Zahl in der Originalimplementierung ist, dass sich der Modulo mit dieser Zahl effizient mit Bitshifts berechnen lässt. Für die Details sei auf die Referenzimplementierung unter [26] verwiesen.

4 Evaluierung und Optimierung

Die Tests wurden manuell vorbereitet und durchgeführt. Die Analysen wurden mit *frozen*[89] durchgeführt, eine kleinere Codebasis (727 LLOC), jedoch mit etwa 100 kleineren und größeren Klonen.

Die wichtigsten Kriterien bei der Analyse der Parameter sind:

- Die Anzahl der Vektorvergleiche.
- Die Anzahl der Buckets.
- Die Laufzeit.
- Die Treffer-Rate in LSH (Anzahl der nahen Vektoren geteilt durch die Anzahl aller Vektorvergleiche).
- Die Anzahl der Klonkandidaten (vor der Filterung).
- Der Speicherbedarf.

Diese Eigenschaften und wie diese mit den Parametern zusammenhängen wird in diesem Kapitel erläutert.

4.1 Problem der fehlenden kanonischen Klontestsuite

Die unklare Beschreibung und Modellierung von Codeklonen macht es schwierige eine allgemeingültige Definition zu treffen. Dies ist der Grund, weshalb fast alle verwandten Arbeiten die Codeklone neu definiert haben. Meist hängt diese eng mit dem in diesem Zug neu eingeführten Verfahren zusammen.

Dadurch bedingt existiert auch keine kanonische Klontestsuite, in der die Anzahl und die genaue Position von Klonen definiert ist. Dies macht ein automatisiertes Testen praktisch unmöglich, ebenso wie das Erstellen einer allgemeingültigen Testsuite.

4.2 Qualitative Bewertung der Klone

Eine qualitative Bewertung der Klone kann nur stichprobenhaft und manuell erfolgen. Jeder gefundene Klon muss im Quellcode durch einen Programmierer validiert werden. Dies wurde bei allen Testprogrammen durchgeführt.

Dieser Schritt ist wichtig um Parameter, wie die Relevanz- und Signifikanzkriterien zu kalibrieren. Durch die Feinjustierung kann das Verfahren gegebenenfalls sogar auf einzelne Programme feinjustiert werden, um eine optimale Erkennungsrate zu erreichen. Die hier getroffene Definition ist so gewählt, dass sie für fast alle Programme sehr gut funktionieren.

[89] Siehe [27].

```
int f1 ()
{
   int a, b;

   a = 10;
   b = 5;

   return a + b;
}

int f2 ()
{
   int a, b;

   a = 100;
   b = 50;

   return a + b;
}
```

Codebeispiel 12: Trivialer Codeklon.

Das Problem der qualitativen Tests ist, dass diese schlecht skalieren. Da sie weiterhin manuelle Überprüfung erfordern, kann zwar die Testumgebung so komfortabel wie möglich gemacht werden (man könnte bei der Analyse direkt die betroffenen Quellcode-Abschnitte des Klons in einer Vergleichsansicht anzeigen, um die direkte Bewertung zu ermöglichen). Dies ist aber weit von der Skalierbarkeit einer automatischen Testsuite entfernt.

4.3 Quantitative Bewertung der Klone

Für die quantitative Bewertung wurde eine eigene Reporting-Komponente entwickelt, die viele Metriken aus dem Verfahren extrahiert. Dazu gehören:

- Anzahl der gefundenen Routinen, der besuchten AST-Knoten, der besuchten AST-Sliding Windows und der besuchten `Discriminated_Ref` (eine zentrale Datenstruktur der PDG-Erstellung).
- Anzahl der durch direkte AST-Knoten, durch Sliding Window und durch PDG erstellten Fragmente.
- Anzahl der Fragmente, die mindestens einen Klon besitzen (kann auch ein redundanter Klon sein).
- Anzahl der gefundenen Klone, vor und nach Filterung.
- Anzahl der LSH-Vektorvergleiche.
- Diverse Timings zu Routinen-Suche, Vektorerstellung, LSH-Initialisierung, LSH-Abfrage und Filterung.

Neben der quantitativen Bewertung muss immer zusätzlich eine qualitative Bewertung erfolgen, um abzusichern, dass die Ergebnisse nicht durch beispielsweise viele falsch-

positiven Klone verfälscht sind. Dies kann aufgrund der Menge nur durch Stichproben geschehen, wurde jedoch bei der folgenden Analyse jedes Mal bestätigt.

4.4 LSH-Evaluierung

Als besondere Belastungsprobe wurden einige C++-Programme analysiert, mit absichtlich ungünstig gewählten Parametern, um die Leistungsfähigkeit von LSH zu messen.

Die Analyse erzeugte insgesamt 42.686 Code-Fragmente. Bei einer zu geringen Mindestgröße der charakteristischen Vektoren (in diesem Fall 4 signifikante Knoten) erkannte der Quellcode für fast jedes einzelne Fragmente Klone. Das liegt darin begründet, dass die Programme wenig, aber ausgesprochen lange und tief verzweigte Routinen implementiert. Dies führt dazu, dass fast alle Knoten signifikant sind. Außerdem erzeugen lange und stark verzweigte Routinen sehr viele Backslices.

Bei der Analyse werden nur 181 Routinen besucht, allerdings über 43.000 AST-Knoten besucht. Über die PDG-Backward-Slices wurden 38.686 Code-Fragmente erzeugt. Diese hohe Anzahl der Knoten mit gepaart mit einem stark verzweigten Programmcode führt zu über 580 Millionen Klonkandidaten, die durch LSH gehasht werden müssen.

Doch selbst diese 580 Millionen gehashten Vektoren (über 1,2 Milliarden Vektorvergleiche) führen nur zu einer Laufzeit von etwa 7 Minuten. Dies zeigt, dass LSH außerordentlich effizient ist. Die Initialisierung von LSH (nur das mehrfache Hashen und Anhängen an die Buckets) dauerte trotz der mehr als einer halben Milliarde Vektoren gerade einmal 0,1 Sekunden.

Da das LSH-Verfahren nicht deterministisch ist, schwanken die Zahlen der Vektorvergleiche erheblich. In einigen Tests kamen Schwankungen um über 20% zustande.

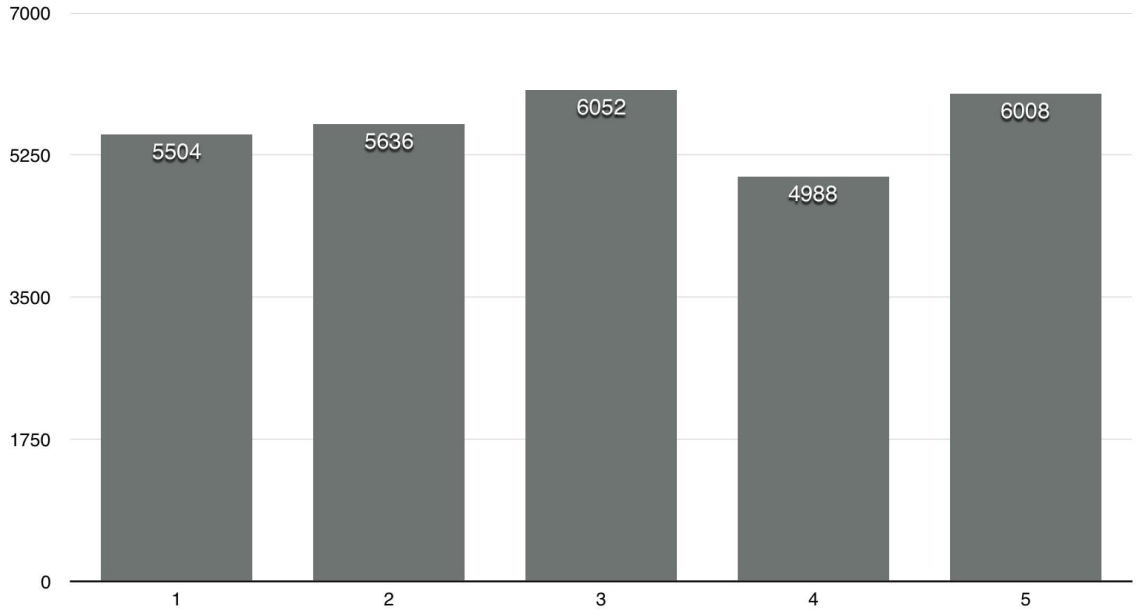

Abbildung 24: Schwankungen der Vektorvergleiche innerhalb einer Messung mit identischen Parametern.

Von der Zahl der Vektorvergleiche abhängig ist die Zahl der Trefferrate der Vektorvergleiche. Dies ist die Zahl der Vektorvergleiche, die zu einem Klon geführt haben. Ein ideales Ergebnis wären keine unnötigen Vektorvergleiche, um eine möglichst optimale Laufzeit zu erhalten.

Eine Optimierung die von den Parametern unabhängig steht, ist es, wenn ein Vektor in L Buckets gehasht wird, dass unter Umständen der Vektor zufällig mehrfach in den gleichen Bucket gehasht wird. Bei Tests mit unterschiedlichen Projekten[90] hat sich herausgestellt, dass die Chance auf solche eine Kollision unter 1% liegt. Weniger als 1% der Fragmente werden in den gleichen Bucket gehasht. Bryant hat diese Optimierung daher nicht implementiert, um die Komplexität der Implementierung für solch einen marginalen Gewinn nicht zu erhöhen.

4.4.1 Parameter: P_1

Der Parameter P_1 ist in der theoretischen Betrachtung sehr wichtig, in der Implementierung ist die einzige Verwendung jedoch die Eingabe zur Berechnung von L und k. Daher erfolgt hier keine gesonderte Betrachtung.

4.4.2 Parameter: R

Der Radius entspricht der maximalen Edit-Distanz der ASTs bzw. AST-Wälder. Er beschreibt die maximale (euklidische) Distanz zu einem anderen Vektor, so dass die zugehörigen Codefragmente als Klone angesehen werden.

[90] *Kleinen* Projekten, bei denen die Wahrscheinlichkeit einer doppelten Bucketzuordnung höher als bei einer großen Zahl an Fragmente ist.

Eine Betrachtung der gefilterten Klonlisten über dem Radius lässt sich ein Rückschluss ziehen:

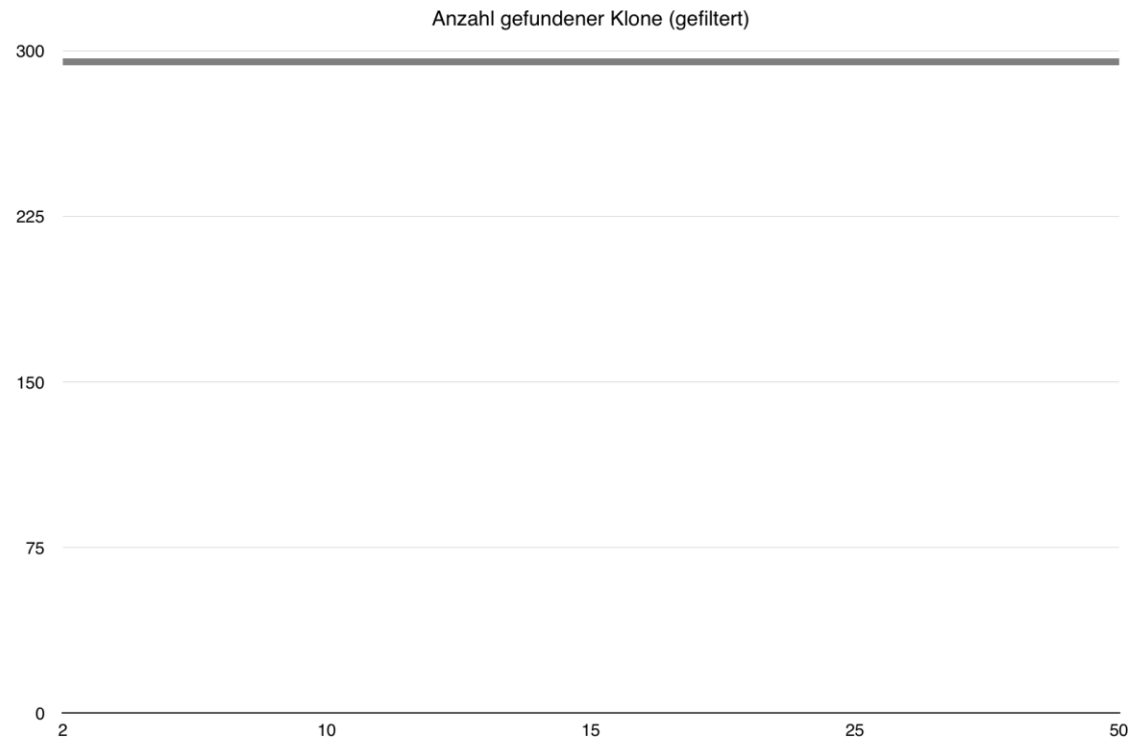

Abbildung 25: Anzahl der (gefilterten) Klone über *R* (die Konstante bei 295 ist die Gerade).

Die Cluster der sehr ähnlichen Vektoren sind relativ eng bestückt, so dass früh die Klone gefunden werden. Die Cluster sind voneinander allerdings relativ weit entfernt. Es scheint also relativ viele sehr ähnlich Muster in der Codebasis zu finden, die Struktur des Codes ist nicht sehr durchmischt.

4.4.3 Parameter: w

Der Parameter w ist für die Segmentzuweisung verantwortlich, in das ein gehashter Vektor zugewiesen wird. Ist w groß, werden viele unterschiedliche Vektoren in den gleichen Bucket gehasht, die Zahl der Vektorvergleiche steigt also. Ist w hingegen sehr klein, besteht die Hashmap aus sehr vielen, leeren Buckets.

Für diese Messung galt: $R = 4.0, L = 2, k = 2$.

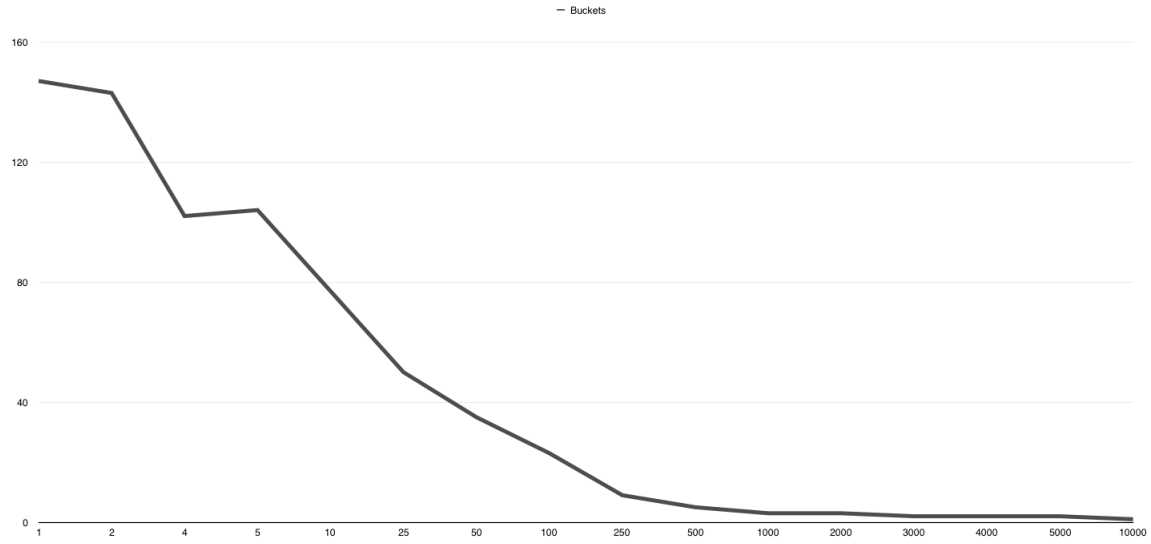

Abbildung 26: Anzahl Buckets über w.

Durch die geringere Anzahl der Buckets steigt die Anzahl der Vektorvergleiche, da mehr entfernte Vektoren in die gleichen Buckets gehasht werden.

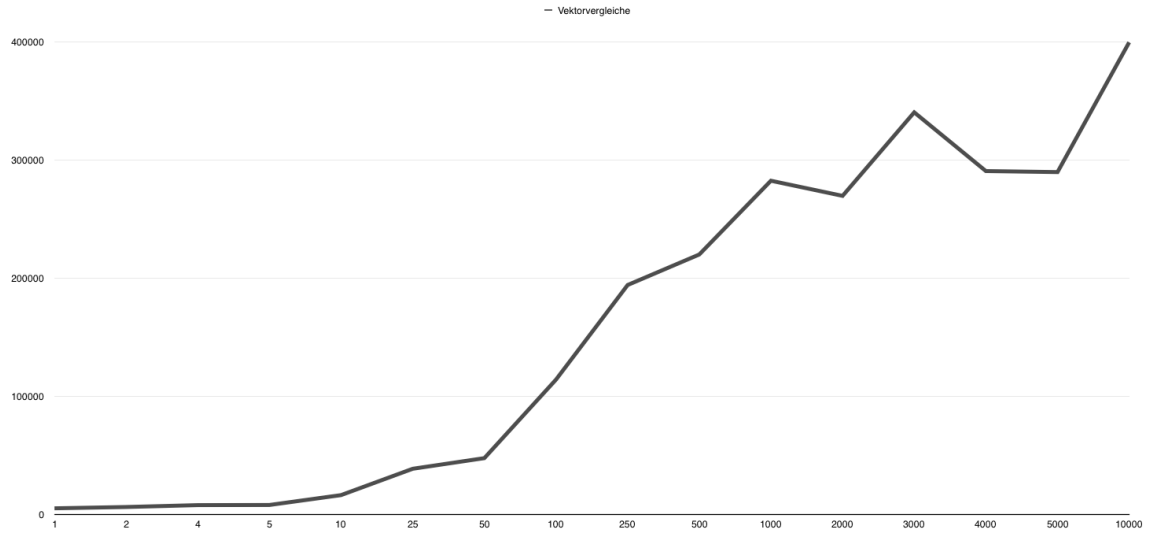

Abbildung 27: Anzahl Vektorvergleiche über w.

Die Laufzeit steigt mit der erhöhten Anzahl der Vektorvergleiche, wobei diese Erhöhung minimal ist. Die gesamte LSH-Berechnung ist selbst im schlechtesten Fall in 0,3 Sekunden abgeschlossen.

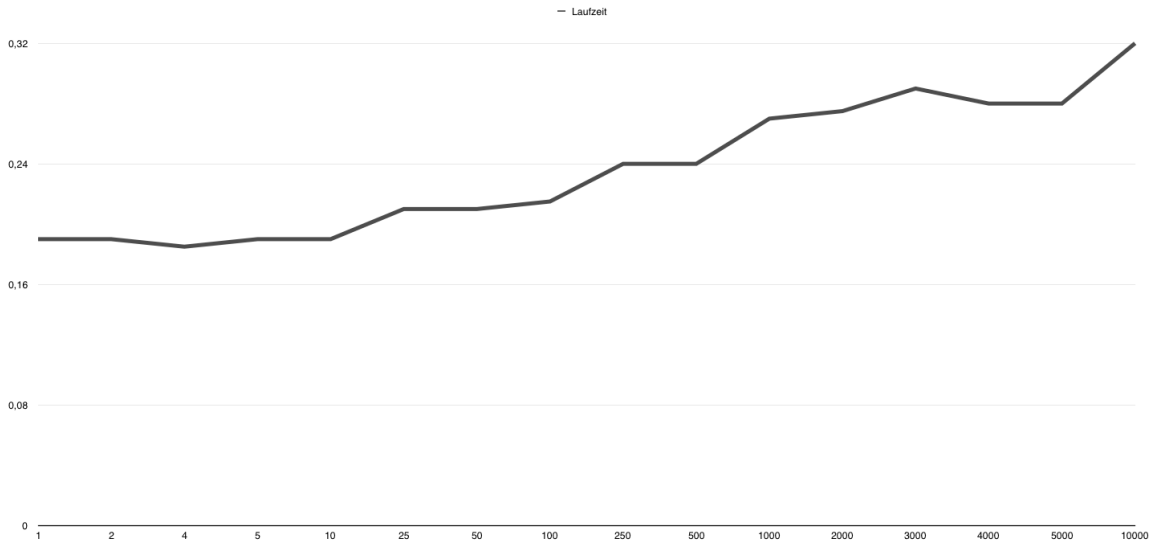

Abbildung 28: Laufzeit über *w*.

Analog zu den erhöhten Vektorvergleichen sinkt die Trefferrate im LSH, da auch weit entfernte Vektoren in den gleichen Bucket gehasht werden:

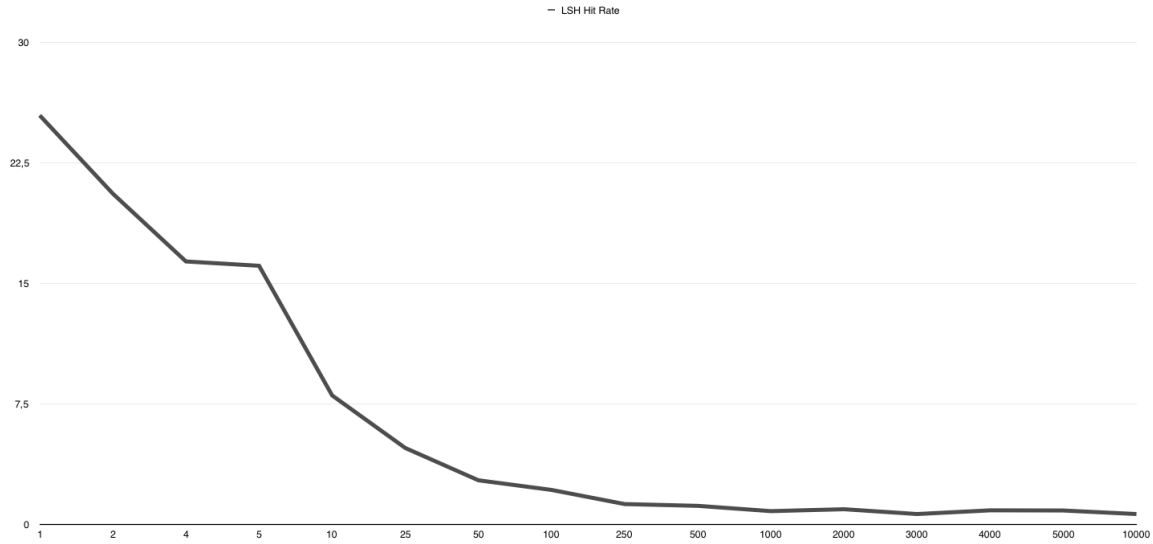

Abbildung 29: LSH-Trefferrate über *w*.

Der Speicherbedarf wird von *w*, außer den üblichen Schwankungen, nicht maßgeblich beeinflusst.

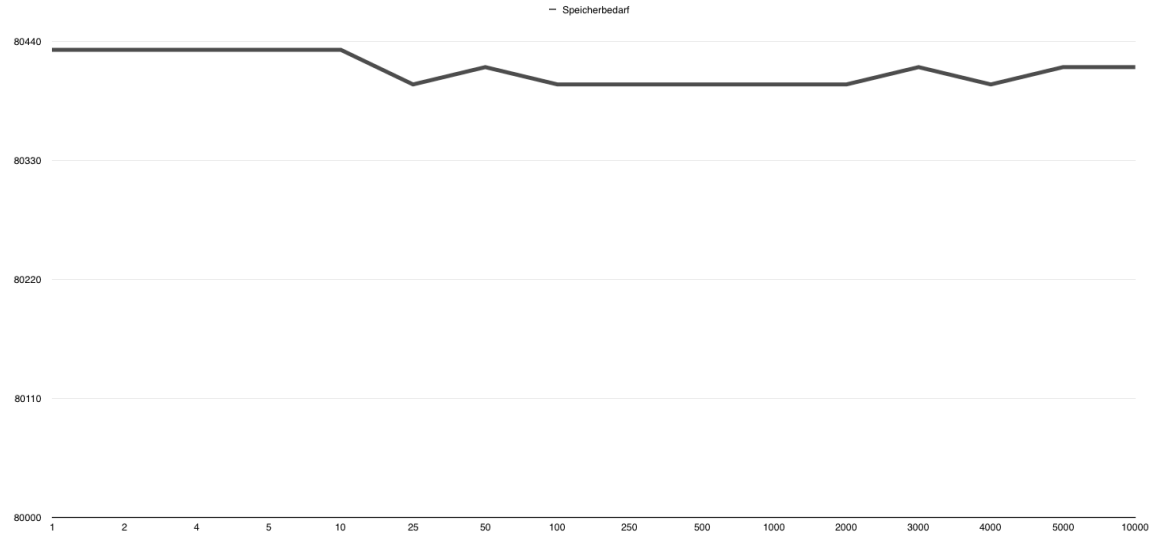

Abbildung 30: Speicherbedarf über w.

Bei der Anzahl der Klonkandidaten erkennt man den Zusammenhang von L und w. Da jeder Vektor in diesem Test fest in 2 Buckets gehasht wird, verdoppelt sich die Zahl der Klonkandidaten, sobald sich die Anzahl der Buckets immer mehr 1 annähert. Ab diesem Punkt enden die 2 Einträge pro Vektor im gleichen Bucket– die Zahl der Klonkandidaten verdoppelt sich.

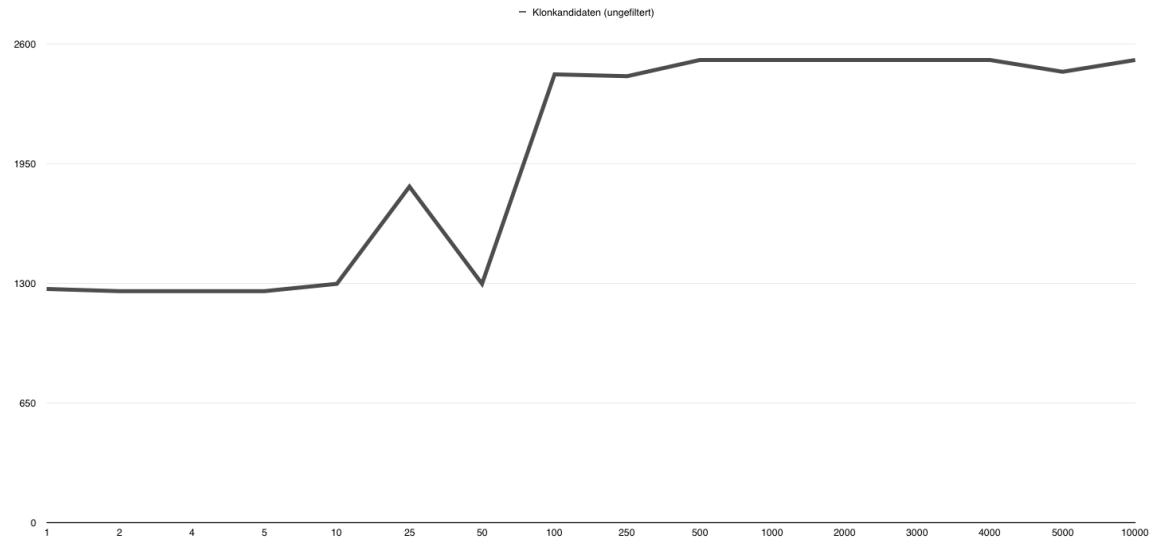

Abbildung 31: Anzahl Klonkandidaten über w.

4.4.4 Parameter: L

Der Parameter L gibt die Anzahl der Buckets an, in die ein Vektor gehasht wird. Der Hintergrund ist, dass es durch die Zufallsverteilung der h-Hashingfunktionen passieren kann, dass der Referenzvektor α zufälligerweise so liegt, dass nahe Vektoren trotzdem in getrennte Buckets gehasht werden.

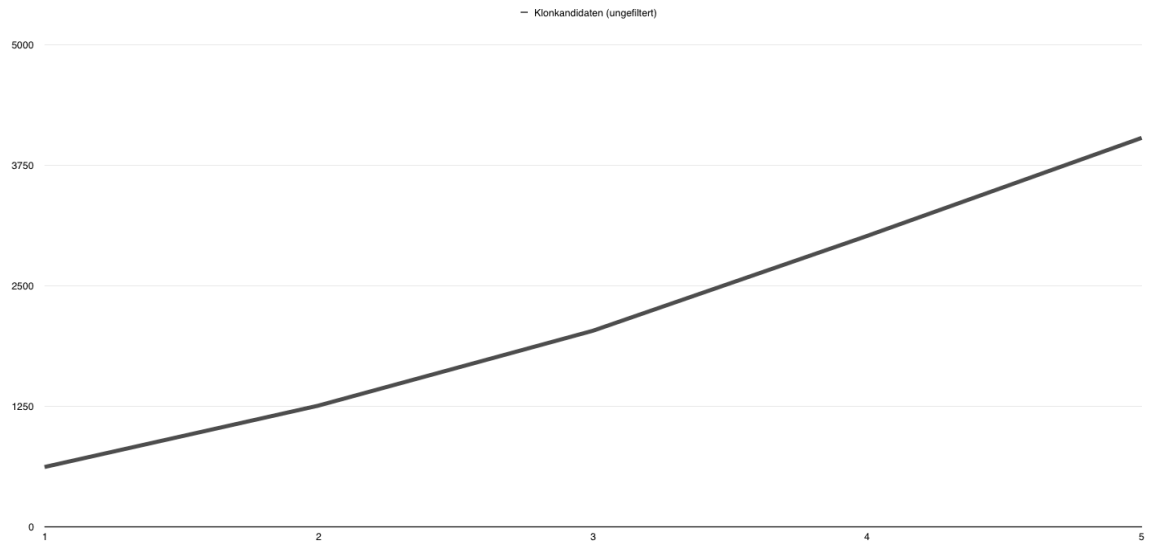

Abbildung 32: Anzahl Klonkandidaten über L.

Die Zahl der erzeugten Klonkandidaten steigt mit steigendem L. Dies ist erwartet, da wenn der Vektor in mehrere Buckets gehasht wird, zwangsläufig die Chance steigt, dass er auch in Buckets gehasht wird, in die seine Klone bereits sind. Die Zahl der steigenden Klonkandidaten sind also keine neuen Klone, sondern nur die erhöhte Anzahl redundanter Klonkandidaten.

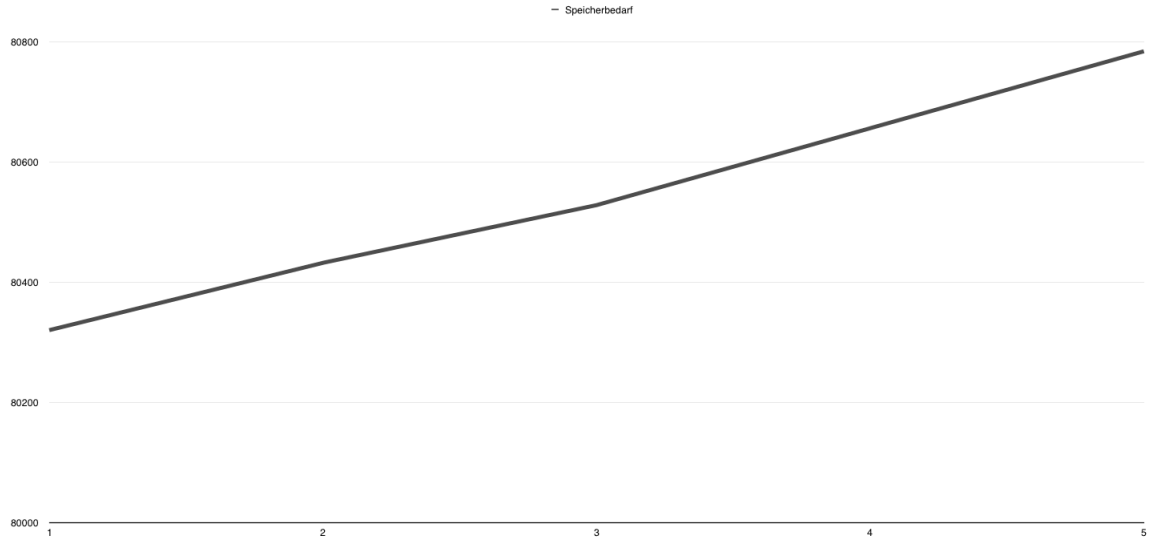

Abbildung 33: Speicherbedarf über L.

Da mit steigendem L mehr Pointer zu den Codefragmenten in den Buckets gehalten werden müssen, steigt der Speicherbedarf entsprechend[91].

[91] Die *absolute* Menge an zusätzlich benötigtem Speicher ist allerdings trotzdem gering.

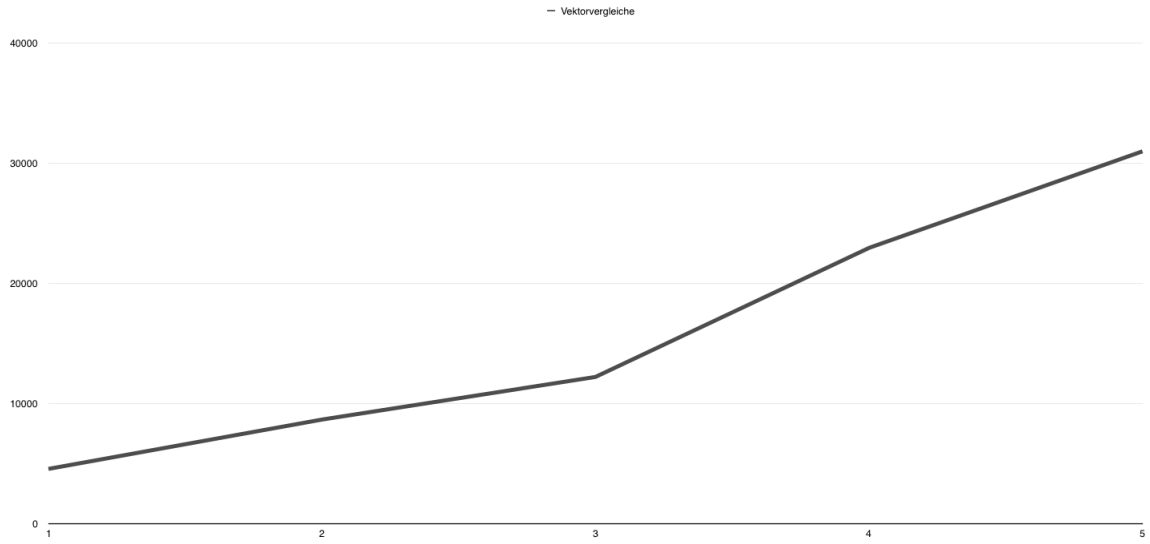

Abbildung 34: Vektorvergleiche über L.

Auch bei L trifft der Zusammenhang zwischen der Anzahl der Vektorvergleiche und der Anzahl der Klonkandidaten zu.

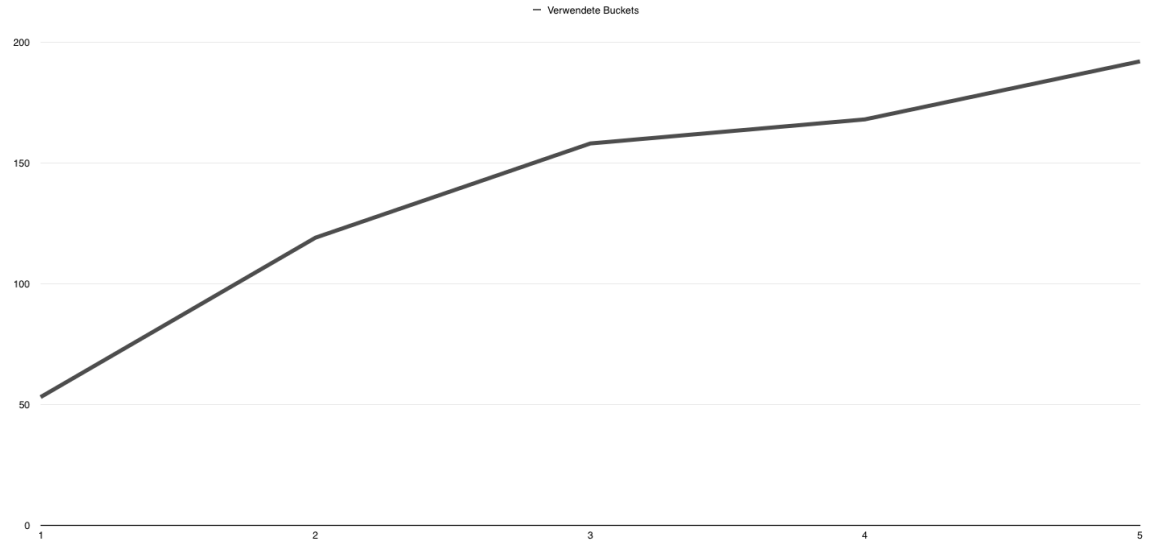

Abbildung 35: Anzahl Buckets über L.

Da insgesamt mehr Vektoren in die Hashmap eingefügt werden, steigt auch die Anzahl der verwendeten Buckets.

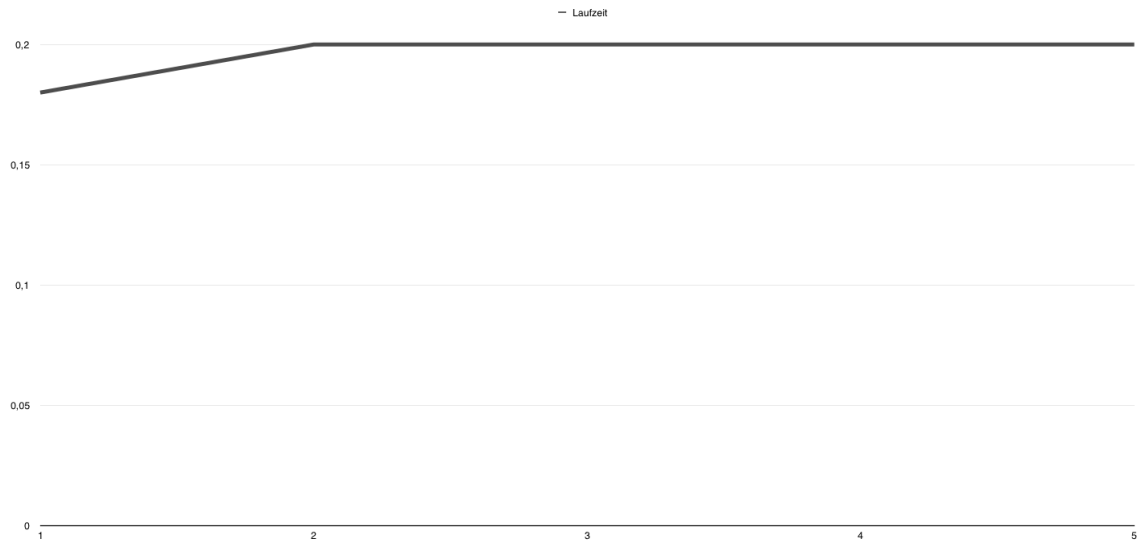

Abbildung 36: Laufzeit über L.

Durch die erhöhte Zahl der Buckets wird die zusätzliche Anzahl der Vektorvergleiche allerdings abgemildert. Diese steigt langsamer als die Zahl der zusätzlichen Vektoren in der Hashmap. Dadurch bleibt die Laufzeit nahezu konstant.

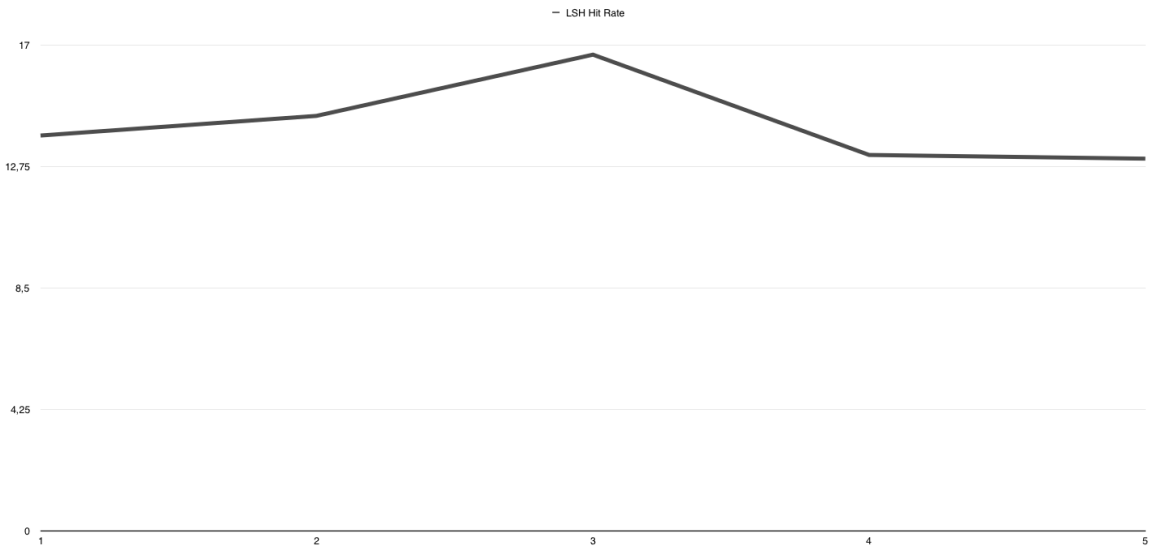

Abbildung 37: LSH-Trefferrate über$L$.

Da die zusätzlichen Vektoren in den Buckets nur Duplikate anderer Vektoren sind, bleibt die Trefferrate nahezu konstant. Es sind insgesamt mehr Vektoren in den Buckets, es werden aber auch mehr Klonkandidaten gefunden.

4.4.5 Parameter: k

Der Parameter k beschreibt die Anzahl der Hashvorgänge, mit der konkateniert der gleiche Vektor gehasht wird. Dies wird implementiert, indem es insgesamt k

Hashfunktionen h_k gibt. Dies verringert die Wahrscheinlichkeit, dass durch ungünstig liegende α-Referenzvektoren zwei weit entfernte Vektoren in die gleiche Bucket gehasht werden.

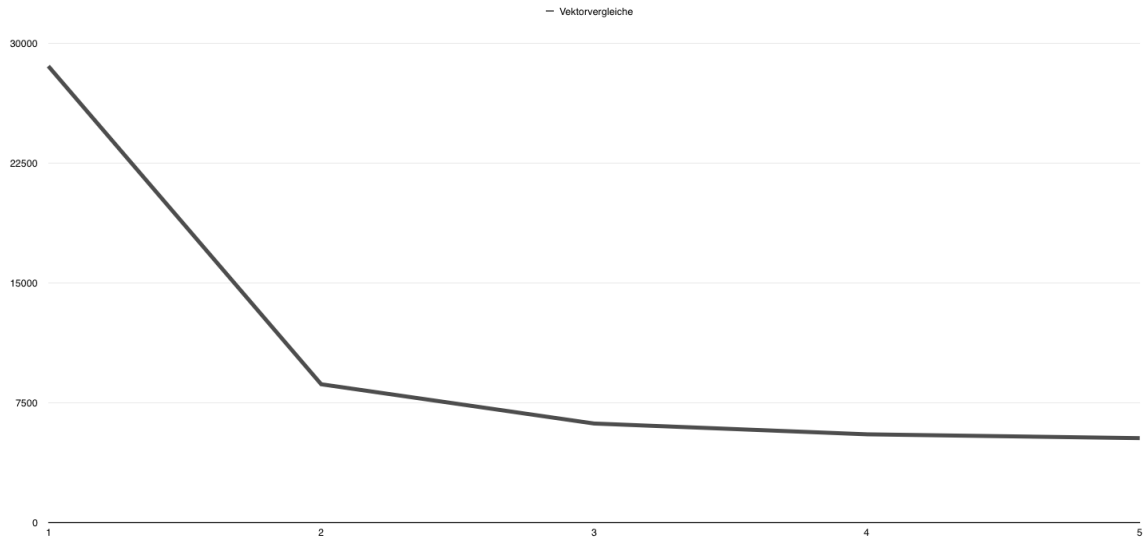

Abbildung 38: Anzahl Vektorvergleiche über k.

Passend zur Theorie ist auch an den Vektorvergleichen ersichtlich, dass deren Anzahl rapide mit steigendem k sinkt. Es ist jedoch relativ schnell ein Plateau erreicht, ab dem der zusätzliche Berechnungsaufwand keinen Mehrwert bringt.

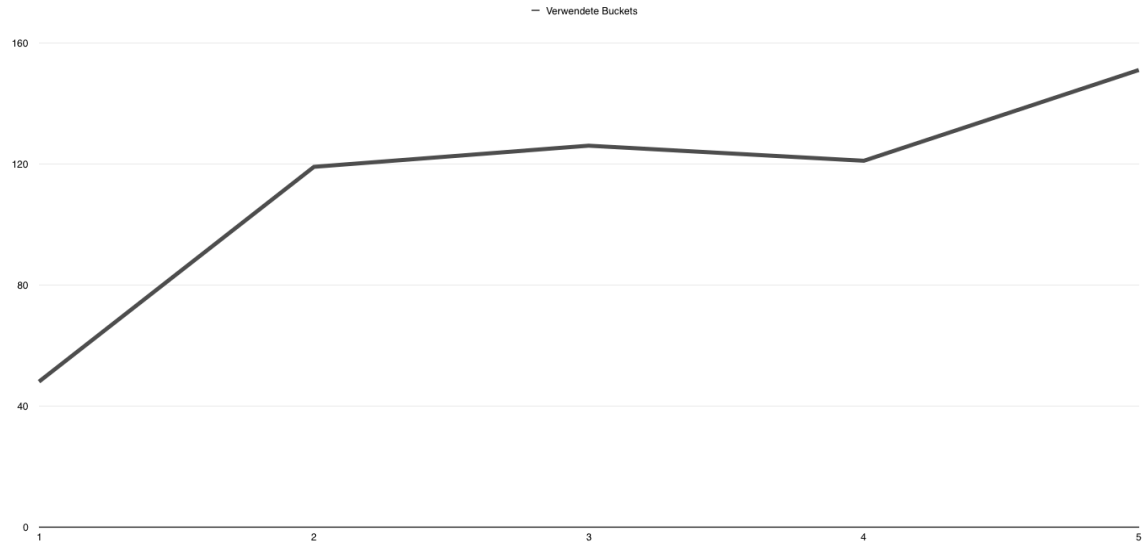

Abbildung 39: Anzahl Buckets über k.

Da die Vektoren durch das zusätzliche Hashing besser voneinander getrennt werden (falls sie unterschiedlich sind), steigt die Anzahl der verwendeten Buckets mit steigendem k.

Die Anzahl ist jedoch begrenzt durch die Codefragmente, die keine Klone in der Codebasis besitzen[92].

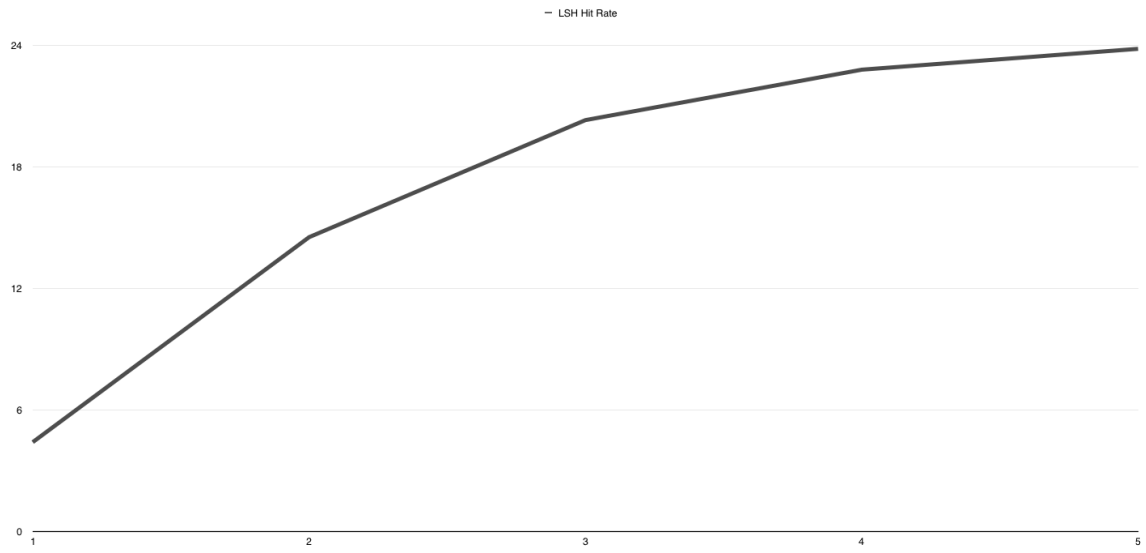

Abbildung 40: LSH-Trefferrate über k.

Die bessere Separierung der Vektoren in mehr, dafür jedoch kleinere Buckets sorgt für eine steigende Trefferrate. Diese hat jedoch, wie auch die Anzahl der Vektorvergleiche ein Plateau, begrenzt durch die Zahl der unabhängigen Vektoren.

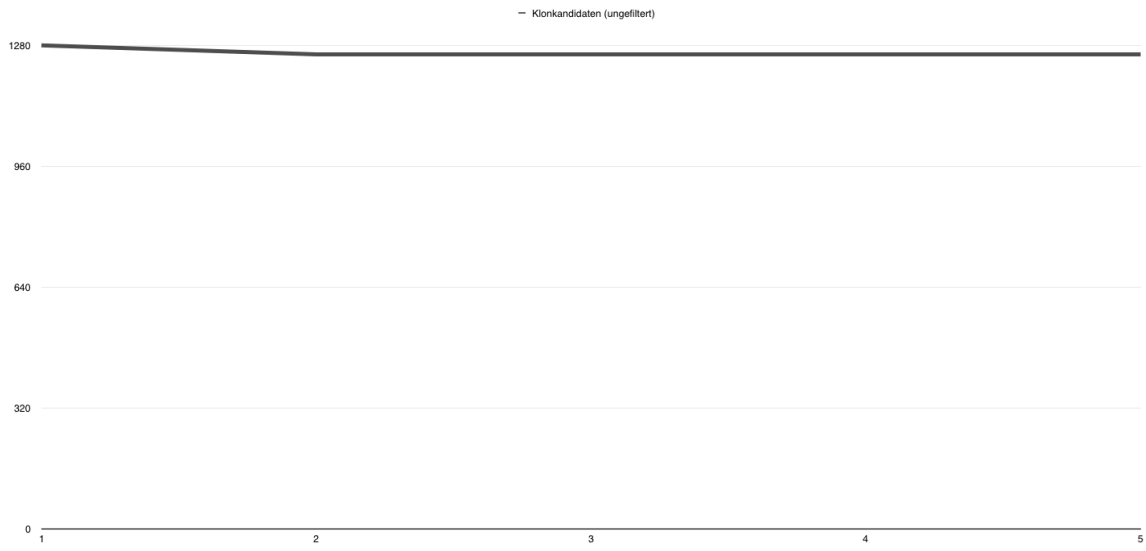

Abbildung 41: Anzahl Klonkandidaten (ungefiltert) über k.

Die gegenläufigen Trends sorgen für keine signifikanten Änderungen bei den Klonkandidaten.

[92] Und durch L.

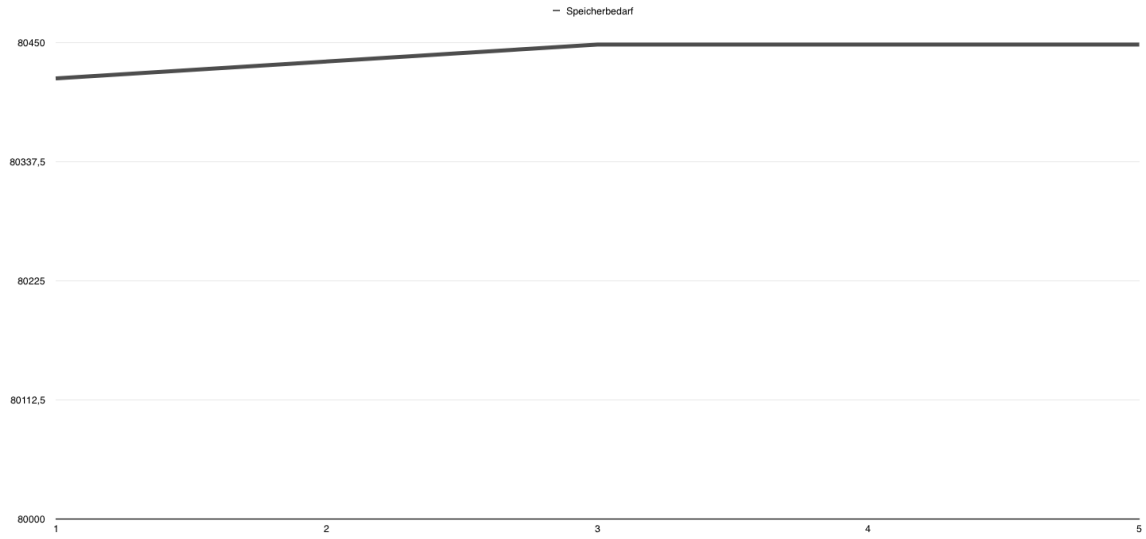

Abbildung 42: Speicherbedarf über k.

Der Speicherbedarf steigt mit zunehmenden k, da die Vektoren der Hashingfunktionen zusätzlichen Speicher benötigen. Dieser Anstieg ist jedoch absolut gesehen vernachlässigbar.

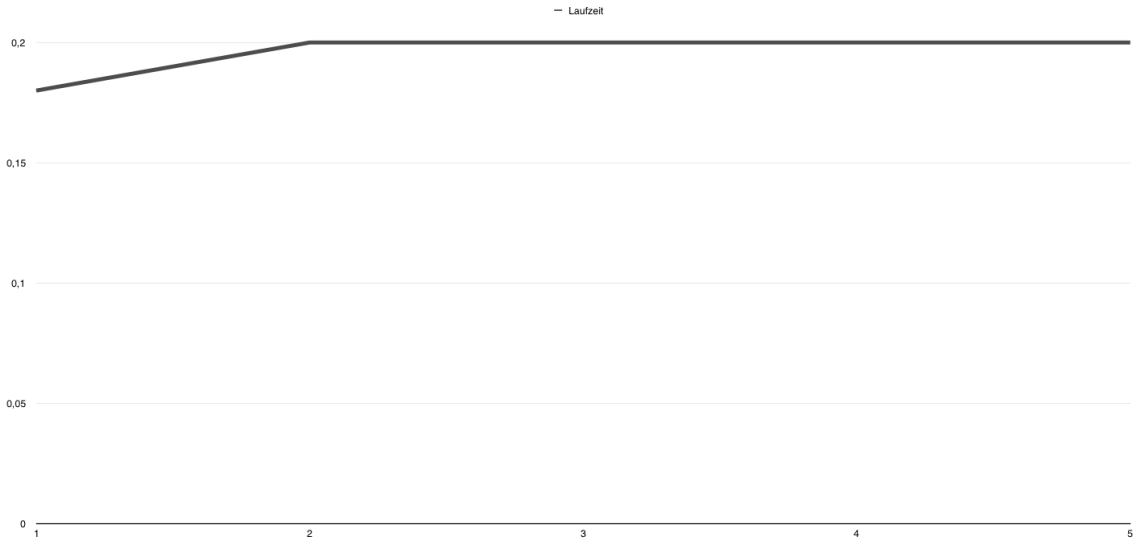

Abbildung 43: Laufzeit über k.

Da die Berechnung der Hashingfunktionen und Hashingwerte in LSH sehr schnell geht, bleibt die Laufzeit auch mit zusätzlichem Rechenaufwand unverändert. Der zusätzlichen Berechnungen des Hashings werden durch weniger Vektorvergleiche eingespart.

4.5 Auswertungen unterschiedlicher Open-Source-Projekte

Es wurden mehrere Open-Source-Programme[93] herausgesucht, die kompatibel mit den Tools von Bauhaus ist. Darüber hinaus wurden einige kleine Testprogramme entwickelt, um bestimmte Erkennungsmuster zu überprüfen. Eine im folgenden verwendete Metrik, um die Größe von Programm zu beschreiben ist die LLOC („Logical Lines of Code“). Diese gibt die Anzahl der Zeilen Programmcode, die Logik beinhalten (zu „Logik“ gehören Statements, nicht jedoch Kommentare, Leerzeilen oder Zeilen, die nur strukturierende Zeichen enthalten[94]).

Unabhängig von einer Klonanalyse wurde festgestellt, dass mit steigender LLOC die Anzahl der Routinen sinkt. Dies kann bei Bryant zu einer schnell wachsenden Menge an Sliding-Window-Fragmenten und PDG-Fragmenten führen. Das ist darauf zurückzuführen, dass die Zahl der AST-Knoten stark mit der Zahl der Zeilen steigen kann.

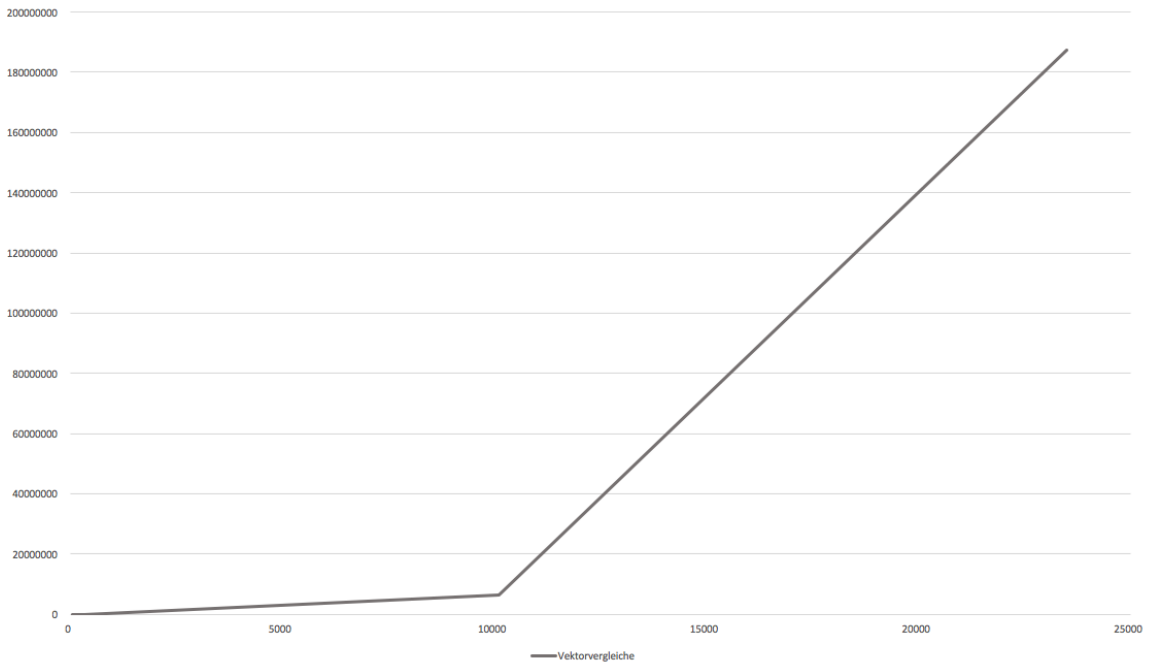

Abbildung 44: Vektorvergleiche über LLOC.

Es ist naheliegend, dass mit steigender LLOC die Zahl der Codefragmente steigt und mit ihr auch die Zahl der Vektorvergleiche.

[93] Unter anderem [27], [28], [29] und [30], sowie einige Test aus einer internen Test-Bibliothek der Universität Stuttgart.
[94] Wie Semikola oder geschweifte Klammern.

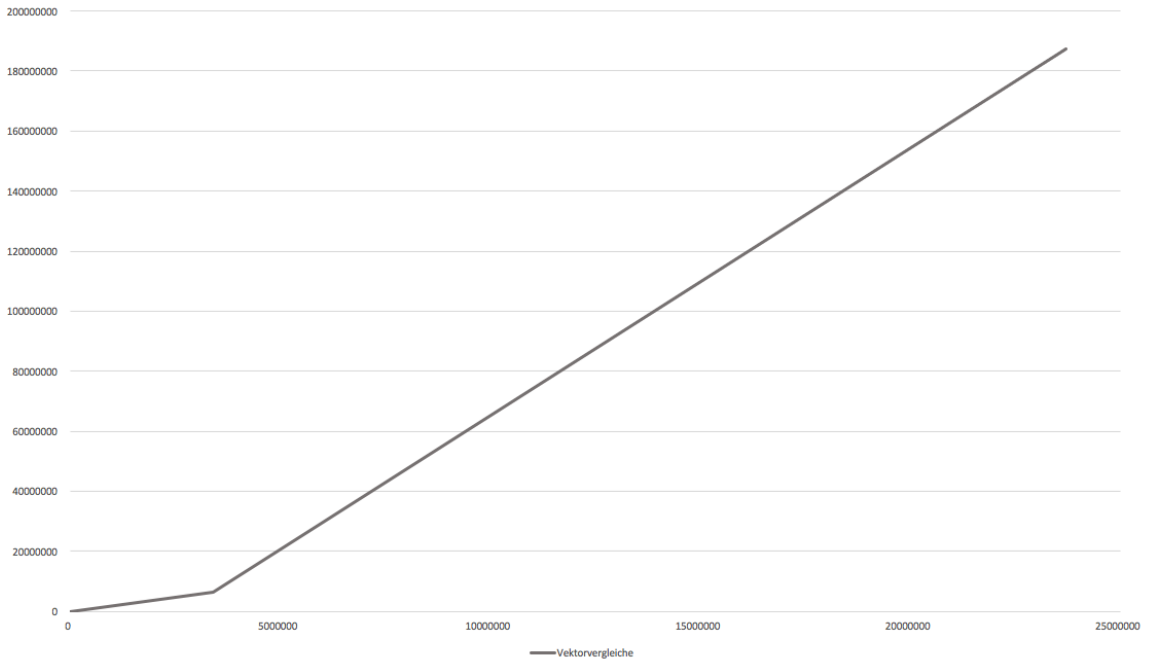

Abbildung 45: Anzahl Vektorvergleiche über die Anzahl der Klonkandidaten.

Ebenfalls lässt sich aus den Daten ablesen, dass die Zahl der Vektorvergleiche linear mit der Zahl der Klonkandidaten steigt. Dies wird durch die automatisch berechneten Parameter gesichert.

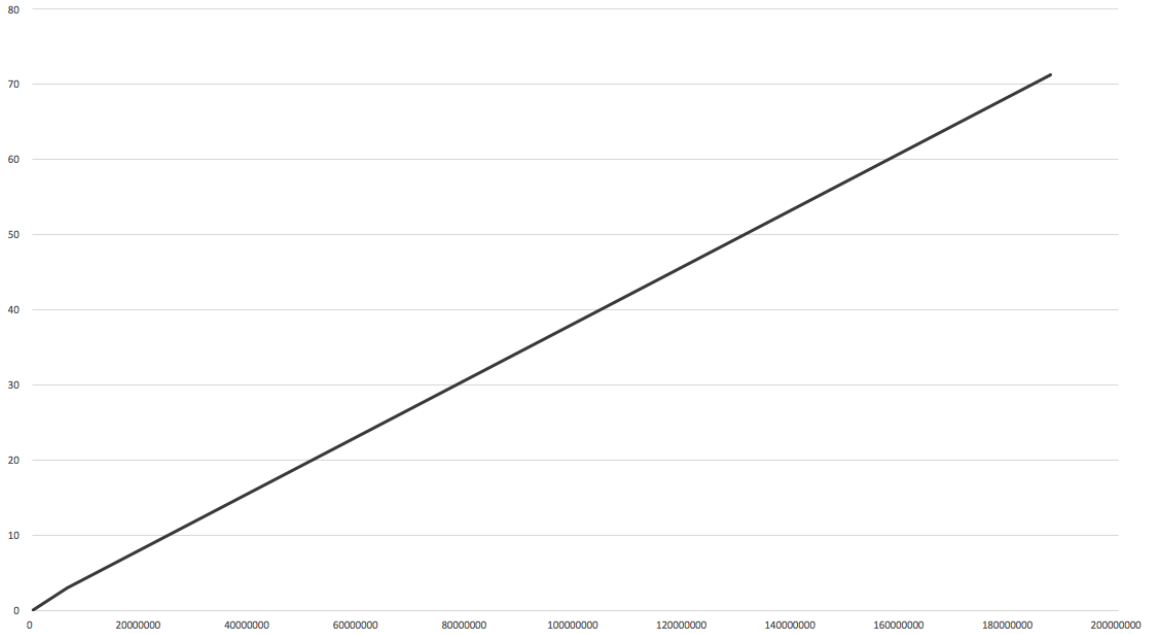

Abbildung 46: Laufzeit LSH über Vektorvergleiche.

Eine triviale Einsicht ist, dass die Laufzeit der LSH-Implementierung linear mit der Zahl der Vektorvergleiche steigt. Dies ist offensichtlich, da die Laufzeit im Grunde einzig von

der Zahl der Vektorvergleiche abhängt. Dies bestätigt allerdings, dass die Implementierung von Bryant hier jedoch zumindest aus Laufzeitsicht valide erscheint.

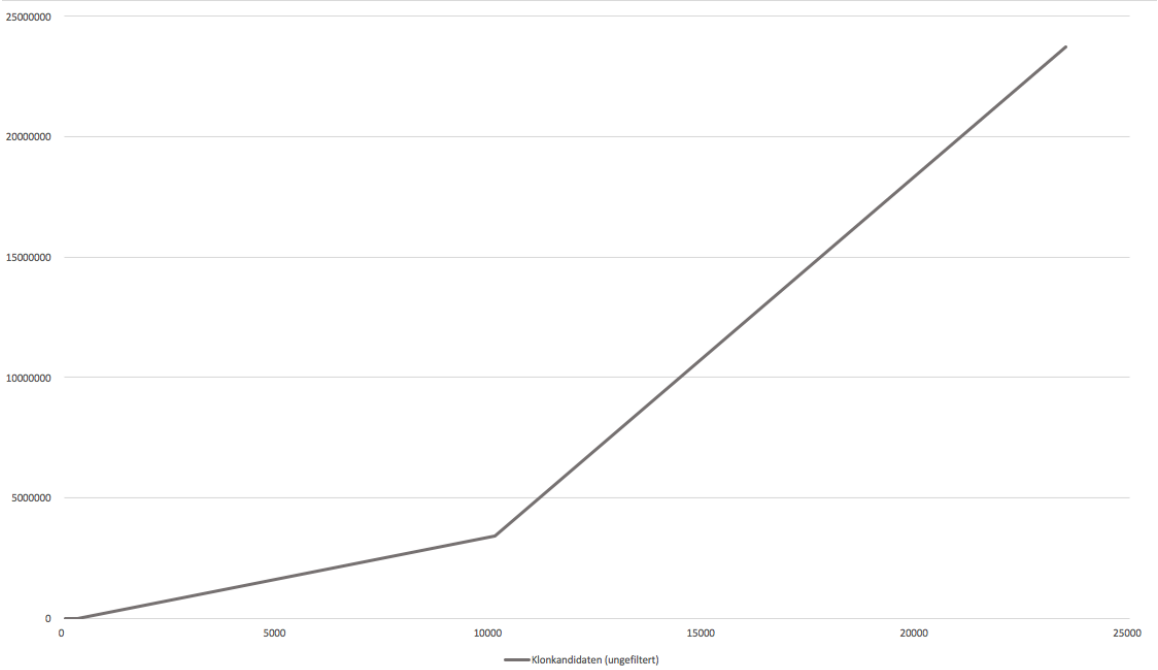

Abbildung 47: Anzahl Klonkandidaten (ungefiltert) über Anzahl Codefragmente.

Die letzte übergreifende Analyse zeigt einen Zusammenhang zwischen der Anzahl der Codefragmente und der Anzahl der erzeugten (ungefilterten) Klonkandidaten. Dies zeigt die Verbindung, dass größere Programme eine höhere Chance besitzen, über die Zeit Codeklone anzuhäufen.

Abschließend zu den Analysen noch ein wichtiger Hinweis bezüglich Laufzeit. Gerade wenn man keine besonders strengen Anforderungen an Codefragment-Generatoren stellen will und lieber mehr als komplizierte Fragmente erzeugen will, ist die korrekte Implementierung des Filters essentiell. In der ersten, rudimentären Implementierung benötigte nur der Filter über 99% der Gesamtlaufzeit. Dies gilt insbesondere für den Fall, dass dieser möglichst wenig Duplikate zulassen und auch Code-Teilbereiche erkennen soll.

5 Zusammenfassung

Bryant bietet eine sehr gute Basis für die Erkennung von Codeklonen. Das Verfahren ist performant, besitzt eine überzeugende Erkennungsrate und minimiert die Zahl der redundanten Klone.

Durch den Aufbau in größtenteils isolierte Teilkomponenten ist das System leicht erweiterbar. Ein weiterer Vorteil ist, dass Teilkomponenten einfach ausgetauscht oder in Isolation optimiert werden können. So kann beispielsweise LSH durch eine komplexere Parameterberechnung (siehe Kapitel 6) erweitert werden, ohne dass die anderen Bereiche ebenfalls Änderungen benötigen.

Die Implementierung basiert aktuell auf der bestehenden PDG-Implementierung in Bauhaus. Hier kann also durch feinere Abstufung, wie die jüngst hinzugefügte Erweiterung der Auftrennung strukturierter Rückgabewerte, eine bessere Identifizierung von ISTs erreicht werden, was insgesamt zu einem besseren Ergebnis führt. Auch diese Änderung ist ohne Anpassungen in Bryant selbst möglich.

Neue Verfahren zur Vektorgenerierung können einfach hinzugefügt und evaluiert werden. Das Tool ist robust gegenüber einer großen Menge an Vektoren, dank des Einsatzes von LSH, das abschließende Filtern und möglicher Erweiterung um größensensitive Klonerkennung.

Durch diese Kombination der verwendeten Verfahren und den internen Aufbau ist das System eher ein Framework für Codeklonerkennung als ein abgeschlossenes Verfahren. Es bietet eine sehr gute Basis um weitere, elaboriertere Methoden der Klonerkennung zu implementieren.

6 Ausblick

Obwohl die vorliegende Implementierung robust ist und bereits im Praxiseinsatz funktioniert, gibt es dennoch einige Optimierungsmöglichkeiten. Diese teilen sich auf zwei Bereiche auf: Performance und noch mehr, „bessere“ Codefragmente.

Im Hinblick auf Performance gibt es insbesondere zwei Aspekte, die von großem Interesse sind. Die Vektorgenerierung für die Routinen ist frei von Seiteneffekten[95], ist also ein idealer Kandidat für Parallelisierung. Hier würde sich eine Implementierung anbieten, die einen Pool von Agenten einsetzt, die die Routinen unter sich aufteilen.

Der zweite interessante Bereich ist LSH, hier beinhaltet E²LSH[96] einige zusätzliche Erweiterungen. Hinter einer der Optimierungen steht, dass bei Tests bemerkt wurde, dass die theoretisch optimalen Werte für L und k nicht diejenigen sind, die auch im praktischen Einsatz hinsichtlich Performance optimal sind. Anstatt die Parameter theoretisch zu berechnen, wird eine Stichprobe der Vektoren gewählt, die mit unterschiedlichen Werten für L und k durchgerechnet werden. Anschließend werden die aus Sicht der Laufzeit optimalen Werte genommen. Dies benötigt einige Vorsicht hinsichtlich Timing-Genauigkeit und paralleler externer Aktivität auf der Testmaschine, kann aber zu besserem Laufzeitverhalten führen.

Eine weitere Zusatzfunktion in E²LSH ist, dass ein alternativer Modus verwendet werden kann, in dem die Hashfunktionen g_i nicht mehr unabhängig voneinander sind, sondern neue, kombinierte Hashfunktionen verwendet werden. Hierdurch kann die Laufzeit abermals reduziert werden, da ein Teil der Hashergebnisse vorberechnet werden kann und die Hashberechnungen dadurch beschleunigt werden[97].

Auf Seite der besseren Codefragmente können neue Verfahren entwickelt werden, die mehr oder andere Codefragmente erzeugen. Die Architektur der Implementierung erlaubt solche Verfahren mit wenig Aufwand hinzuzufügen oder bestehende Verfahren zu ersetzen. Außerdem können die existierenden AST- und PDG-basierten Verfahren verfeinert werden, sodass sie möglicherweise noch bessere Kandidaten erzeugen.

Im Hinblick auf das Gesamtsystem sind weitere Experimente und Tests möglich, um interne Parameter und Definitionen, wie Relevanz und Signifikanz der Vektoren, zu verfeinern.

[95] Die aktuelle Implementierung manipuliert direkt die Vektorliste. Der Umbau, dass die erzeugten Vektoren zurückgegeben werden und daher die Liste nicht mehr direkt bearbeitet wird, ist trivial.
[96] Siehe [26].
[97] Siehe Kapitel 3.4.1 in [25].

7 Appendix

7.1 l_1 und l_2 Norm von Vektoren[98]

Seien $v_1 = \langle x_1, x_2, \dots, x_n \rangle$ und $v_2 = \langle y_1, y_2, \dots, y_n \rangle$ zwei n-dimensionale Vektoren.

Die Hamming-Distanz von v_1 und v_2, $\mathcal{H}(v_1, v_2)$ ist ihre l_1-Norm, das heißt $\mathcal{H}(v_1, v_2) = \|v_1 - v_2\|_1 = \sum_{i=1}^{n} |x_i - y_i|$.

Die euklidische Distanz von v_1 und v_2, $\mathcal{D}(v_1, v_2)$ ist ihre l_2-Norm, das heißt $\mathcal{D}(v_1, v_2) = \|v_1 - v_2\|_2 = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$.

7.2 (Schwache) Zusammenhangskomponenten

Ein ungerichteter Graph $G = (V, E)$ heißt zusammenhängend, wenn es für jede beliebige Knotenkombination u und v einen ungerichteten Weg durch G mit u als Startknoten und v als Endknoten gibt.

[Bild Zusammenhangskomponente mit G, Start- und Endpunkt und zwei Teilgraphen]

Eine Zusammenhangskomponente ist ein maximaler, zusammenhängender Teilgraph.

[98] Aus [6], Definition 3.4.

8 Literaturverzeichnis

[1] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna und L. Bier, „Clone Detection Using Abstract Syntax Trees," *Software Maintenance, 1998. Proceedings., International Conference,* pp. pp. 368-377, November 1998.

[2] M. Datar, P. Indyk, N. Immorlica und V. S. Mirrokni, „Locality-Sensitive Hashing Scheme Based on p-Stable Distributions," *Proceedings of the twentieth annual symposium on Computational geometry,* pp. pp. 253-262, June 2004.

[3] J. Ferrante, K. J. Ottenstein und J. D. Warren, „The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS),* Bd. 9, Nr. 3, pp. pp. 319-349, 1987.

[4] M. Gabel, L. Jiang und Z. Su, „Scalable Detection of Semantic Clones," *Gabel, M., Jiang, L., & Su, Z. (2008, May). Scalable detection of semantic clones. In Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference,* pp. pp. 321-330, May 2008.

[5] A. Gionis, P. Indyk und R. Motwani, „Similarity Search in High Dimensions via Hashing," Nr. Vol. 99, No. 6, pp. pp. 518-529, September 1999.

[6] L. Jiang, G. Misherghi, Z. Su und S. Glondu, „DECKARD: Scalable and accurate tree-based detection of code clones," *Proceedings of the 29th international conference on Software Engineering,* pp. pp. 96-105, May 2007.

[7] T. Kamiya, S. Kusumoto und K. Inoue, „CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *Software Engineering, IEEE Transactions,* Bd. 28, Nr. 7, pp. 654-670, 2002.

[8] Z. Li, S. Lu, S. Myagmar und Y. Zhou, „CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code," *OSDI,* Nr. Vol. 4, No. 19, pp. pp. 289-302, December 2004.

[9] „Oracle America, Inc. v. Google, Inc.," 20 Februar 2016. [Online]. Available: https://en.wikipedia.org/wiki/Oracle_America,_Inc._v._Google,_Inc.. [Zugriff am 18 März 2016].

[10] J. C. Huang und T. Leng, „Generalized Loop-Unrolling: a Method for Program Speed-Up [PDF]," The University of Houston, Department of Computer Science, 18 März 2016. [Online]. Available: http://web.cs.uh.edu/~jhuang/JCH/JC/loop.pdf. [Zugriff am 18 März 2016].

[11] „Search for code duplicates in WebStorm/PhpStorm," 28 September 2011. [Online]. Available: http://blog.jetbrains.com/webide/2011/09/search-for-code-duplicates-in-phpstorm/. [Zugriff am 18 März 2016].

[12] B. S. Baker, „On finding duplication and near-duplication in large software systems," *WCRE,* pp. 85-95, 1995.

[13] B. S. Baker, „Parameterized duplication in strings: Algorithms and an application to software maintenance," *SICOMP,* Bd. 26, Nr. 5, pp. 1343-1362, 1997.

[14] I. D. Baxter, C. Pidgeon und M. Mehlich, „Detecting higher-level similiarity patterns in programs," *ESEC/FSE,* pp. 156-165, 2005.

[15] V. Wahler, D. Seipel, J. W. von Gudenberg und G. Fischer, „Clone detection in source code by frequent itemset techniques," *SCAM,* pp. 128-135, 2004.

[16] K. Kontogiannis, R. de Mori, E. Merlo und J. P. Hudepohl, „Pattern matching for clone and concept detection," *Automated Soft. Eng.,* Bd. 3, Nr. 1/2, pp. 77-108, 1996.

[17] J. Mayrand, C. Leblanc und E. Merlo, „Experiment on the automatic detection of function clones in a software system using metrics," *ICSM,* pp. 244-254, 1996.

[18] R. Komondoor und S. Horwitz, „Using slicing to identify duplication in source code," *SAS,* pp. 40-56, 2001.

[19] R. Yang, P. Kalnis und A. K. H. Tung, „Similarity evaluation on tree-structured data," *SIGMOD,* pp. 754-765, 2005.

[20] J. M. Chambers, C. L. Mallows und B. W. Stuck, „A method for simulating stable random variables," *Journal of the american statistical association,* Bd. 71, Nr. 354, pp. 340-344, 1976.

[21] S. Horwitz, T. Reps und D. Binkley, „Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems (TOPLAS),* Bd. 12, Nr. 1, pp. 26-60, 1990.

[22] M. Weiser, „Program slicing," *Proceedings of the 5th international conference on Software engineering (IEEE Press),* pp. 439-449, 1981.

[23] „Projekt Bauhaus," 6 Dezember 2012. [Online]. Available: http://www.iste.uni-stuttgart.de/ps/projekt-bauhaus.html. [Zugriff am 18 März 2016].

[24] „GitHub - skyhover/Deckard: Code clone detection; clone-related bug detection; sematic clone analysis,“ [Online]. Available: https://github.com/skyhover/Deckard. [Zugriff am 18 März 2016].

[25] A. Andoni und P. Indyk, „E²LSH 0.1 - User Manual,“ 21 Juni 2005. [Online]. Available: http://www.mit.edu/~andoni/LSH/manual.pdf. [Zugriff am 16 März 2016].

[26] „LSH Algorithm and Implementation (E2LSH),“ [Online]. Available: http://www.mit.edu/~andoni/LSH/. [Zugriff am 18 März 2016].

[27] „GitHub - cesanta/frozen: JSON parser and generator for C/C++,“ [Online]. Available: https://github.com/cesanta/frozen. [Zugriff am 18 März 2016].

[28] „bc - GNU Project - Free Software Foundation,“ [Online]. Available: https://www.gnu.org/software/bc/bc.html. [Zugriff am 18 März 2016].

[29] „GitHub - antirez/smaz: Small strings compression library,“ [Online]. Available: https://github.com/antirez/smaz. [Zugriff am 18 März 2016].

[30] „time - GNU Project - Free Software Foundation (FSF),“ [Online]. Available: https://www.gnu.org/software/time/time.html. [Zugriff am 18 März 2016].