Kategorisierung von Programmen anhand ihres Laufzeitverhaltens

Peter P. Ortner

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2017

© Copyright 2017 Peter P. Ortner

Diese Arbeit wird unter den Bedingungen der Creative Commons Lizenz *Attribution-NonCommercial-NoDerivatives 4.0 International* (CC BY-NC-ND 4.0) veröffentlicht – siehe https://creativecommons.org/licenses/by-nc-nd/4.0/.

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 25. September 2017

Peter P. Ortner

Inhaltsverzeichnis

Erklärung iii

Kurzfassung vi

Abstract vii

1 Einleitung 1
- 1.1 Problemstellung . . . 1
- 1.2 Abgrenzung . . . 1
- 1.3 Analyseverfahren . . . 2
  - 1.3.1 Statische Codeanalyse . . . 2
  - 1.3.2 Dynamische Codeanalyse . . . 5
- 1.4 Ansatz . . . 7

2 Versuchsaufbau 8
- 2.1 Setup . . . 8
  - 2.1.1 Laufzeitumgebung . . . 8
  - 2.1.2 Datenaufzeichnung . . . 11
- 2.2 Auswertung . . . 14
  - 2.2.1 Cluster Analyse . . . 14
  - 2.2.2 Visuelle Aufbereitung . . . 16
  - 2.2.3 Analyse anhand von Metriken . . . 16

3 Umsetzung 22
- 3.1 Verwendeter Input . . . 22
  - 3.1.1 Q'Bots . . . 22
  - 3.1.2 Berechnung von π . . . 25
- 3.2 Auswertung . . . 28
  - 3.2.1 Clustering . . . 28
  - 3.2.2 Metriken . . . 31

4 Resümee 37
- 4.1 Rückblick . . . 37
  - 4.1.1 Setup . . . 37
  - 4.1.2 Auswertung . . . 38
- 4.2 Ausblick . . . 38

4.3 Schlusswort 39

A Technische Informationen **40**
A.1 Aktuelle Dateiversionen 40
A.2 Details zur aktuellen Version 40
A.2.1 Allgemeine technische Voraussetzungen 40

B Inhalt der CD-ROM/DVD **41**
B.1 Dokumente 41
B.2 Ausführbare Dateien 41
B.3 Implementierung 42
B.4 Auswertung 42

C Approximationen von π **43**
C.1 Leonhard Euler Version 1 43
C.2 Leonhard Euler Version 2 44
C.3 John Wallis 44
C.4 Gottfried Wilhelm Leibniz 45
C.5 François Viète 45
C.6 Abraham Sharp 46

Quellenverzeichnis **47**
Literatur 47
Online-Quellen 48

Kurzfassung

Bei der Betrachtung von Bildungseinrichtungen im Bereich der Informatik fällt auf, dass jeder, der das Programmieren neu erlernt, vor die gleichen grundlegenden algorithmischen Probleme gestellt wird, wie zum Beispiele das Sortieren von Listen, das Suchen in Fließtexten oder das Implementieren von simplen Datenstrukturen wie Binärbäumen. Geschuldet dem Fakt, dass das Programmieren eine kreative Disziplin ist, ist es kaum verwunderlich, dass bereits diese einfachen Problemstellungen in ihrer jeweiligen programmatischen Umsetzung teils stark variieren.

Ein Übungsleiter, Tutor oder Lehrender ist nun unter Umständen daran interessiert, welche Übungen seiner Studenten Ähnlichkeiten in ihrer jeweiligen Umsetzung aufweisen. Unter dem Begriff der Ähnlichkeit muss zwischen zwei Punkten unterschieden werden, zum einen die semantische Ähnlichkeit, welche auf ein Plagiat hindeutet, und zum anderen die Verhaltensähnlichkeit, welche auf ähnliche Ideen und Umsetzungen hinweist.

Diese Arbeit legt den Schwerpunkt auf die Verhaltensähnlichkeit von Programmen, welche dieselbe Problemstellung lösen. Um diese Ähnlichkeit zu erfassen, ist es leider nicht ausreichend, nur eine Analyse des Quellcodes durchzuführen. Das Hauptproblem bei einer reinen Quellcodeanalyse ist, dass das Verhalten einer Software während der Ausführung nur sehr schwer bis unmöglich vorherzusagen ist. Daher wird im Zuge dieser Arbeit ein genauerer Blick auf das Verhalten von Programmen zu ihrer Ausführungszeit geworfen.

Im Speziellen werden in dieser Arbeit verschiedene Ansätze für dynamische Codeanalyse untersucht und Ideen für die Unterscheidung von Programmen vorgeschlagen. Als Resultat entsteht ein Überblick über die geeignetsten Methoden zur Erkennung von Programmstrukturen.

Abstract

When looking at educational institutions in the field of computer science, it is noticeable that everyone who starts programming is faced with the same basic algorithmic problems as sorting lists, searching in continuous texts or implementing simple data structures such as binary trees. Due to the fact that programming is a creative discipline, it is hardly surprising that even these simple problems vary greatly in their respective programmatic implementation.

A trainer, tutor or teacher may now be interested in the exercises of his or her similarities in their respective implementation. Under the concept of similarity two points have to be distinguished, on the one hand the semantic similarity which indicates a plagiarism and on the other hand the behavioral similarity which points to similar ideas and implementations.

This work focuses on the behavioral similarity of programs that solve the same problem. Unfortunately, it is not sufficient to analyze the source code to understand this similarity. The main problem with a pure source code analysis is that the behavior of a piece of software during execution is very difficult to predict. Therefore, this work looks more closely at the behavior of programs at execution time.

In particular, this thesis examines different approaches to dynamic code analysis and proposes ideas for the differentiation of programs. The result is an overview of the most suitable methods for recognizing program structures.

Kapitel 1

Einleitung

In diesem Abschnitt wird der aktuelle Stand der Technik sowie die Problemstellung die dieser Arbeit zugrunde liegen erläutert. Für den aktuellen Stand der Technik wird auf gängige Methoden zur Softwareanalyse im Bereich der statischen und dynamischen Auswertung eingegangen.

1.1 Problemstellung

Im Bereich der Softwareentwicklung findet man aufgrund der kreativen Natur dieser Disziplin für ein und dasselbe Problem eine Vielzahl an möglichen Lösungen. Diese Ansätze, die sich zum Teil stark unterscheiden, lassen sich nur schwer objektiv beurteilen und bewerten. Je nach verfolgtem Ansatz unterscheiden sich Programme in unterschiedlichen Aspekten, die nicht per se als gut oder schlecht zu kategorisieren sind. So ist ein rechenintensives Programm möglicherweise speicherschonender als ein anderes, welches auf vorberechnete Lookup-Tabellen setzt. Umgekehrt kann ein speicherintensives Programm unter Umständen weniger rechenintensiv sein und eine kürzere Laufzeit aufweisen. Um einen möglichst objektiven Blick auf unterschiedliche Ansätze zu erhalten, wird im Zuge dieser Arbeit versucht, Programme aufgrund ihrer Struktur zu unterteilen und anschließend ähnlich strukturierte Programme anhand weiterer Charakteristika in Relation zu setzen.

1.2 Abgrenzung

Da der Bereich der Softwareanalyse ein weites Spektrum an Betätigungsfeldern beherbergt, wurden auch bereits viele Ansätze bezüglich des Vergleichs von Software anhand einzelner Metriken wie Laufzeitkomplexität, Menge an Zuweisungen und Vergleichen oder Speicherauslastung verfolgt. In dieser Arbeit wird versucht Programme nicht anhand eines spezifischen Kriteriums von gut nach schlecht zu reihen, sondern eine größere Input-Menge an Programmen anhand ihrer strukturellen Ähnlichkeit in Kategorien zu unterteilen und jede Kategorie nach bestimmten Metriken zu reihen. Ziel ist es, ähnliche Lösungsansätze und Herangehensweisen an eine Problemstellung zu erkennen und objektiv zu bewerten.

1.3 Analyseverfahren

Die verschiedenen Analyseverfahren in der Softwareentwicklung werden im Großen und Ganzen in zwei Kategorien unterteilt, zum einen in die statische und zum anderen in die dynamische Codeanalyse. Der große Unterschied zwischen diesen beiden Kategorien ist der Zeitpunkt, wann eines dieser Verfahren Anwendung findet. Die statische Analyse bedient sich der Auswertung des vorliegenden Quellcodes, während die dynamische Variante das fertig erstellte Programm ausführt und dessen Verhalten und Ressourcennutzung analysiert. Im folgenden Abschnitt werden diese beiden Verfahren näher betrachtet.

1.3.1 Statische Codeanalyse

Die statische Analyse definiert eine Reihe von formalen Prüfungen, die ein gegebenes Stück Sourcecode in der Regel vor der Übersetzungszeit durchläuft. Dies reicht von der Überprüfung von Coding-Standards bis hin zur Erkennung von potenziellen Fehlern bzw. Schwachstellen im zu prüfenden Quelltext. Die Durchführung dieses Analyseverfahrens erfolgt in der Regel durch Entwickler in Codereviews, Compiler, Style Checker und speziellen Tools, die für diesen Zweck konzipiert wurden.

Ablauf

Je nach Bedarf wird vor der Analyse ein Set an Kriterien erstellt, welche der zu prüfende Sourcecode erfüllen muss. In der Regel umfassen diese Kriterien sprachspezifische Einschränkungen um eine Übersetzung überhaupt zu ermöglichen, allgemeine oder spezielle Programmierrichtlinien für die Formatierung und falls vorhanden eine Prüfung auf mögliche Fehlerquellen (sogenannte „code smells“). In Abbildung 1.1 ist dieser Ablauf für eine statische Codeanalyse schematisch dargestellt.

Ziel

Das Hauptziel der statischen Codeanalyse liegt darin Sourcecode lesbar und wartbar zu halten, sowie potenzielle Fehler auszuschließen. Aufgrund der einfachen Anwendbarkeit und der vielen möglichen Vorteile ist die statische Codeanalyse das wohl am häufigsten angewandte Analysewerkzeug.

Beispiel

Anhand eines sogenannten „Hello World“-Programms soll der Ablauf einer solchen statischen Codeanalyse kurz erklärt werden. In Listing 1.1 ist eine Implementierung für „Hello World“ zu finden.

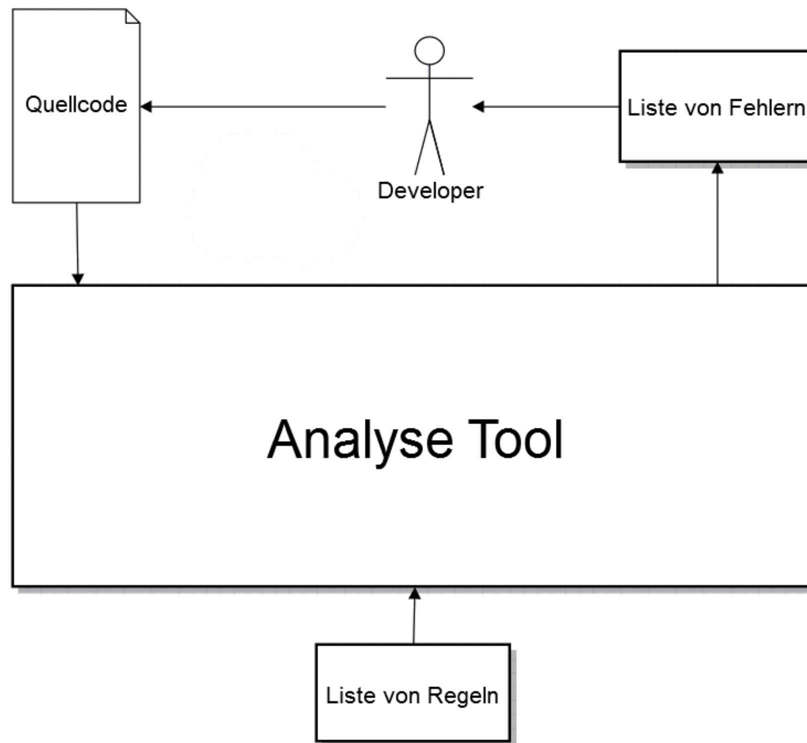

Abbildung 1.1: Schematischer Ablauf einer statischen Analyse.

Listing 1.1: Hello World Example 1.

```
 1 package thesis.analyse;
 2
 3 // Simple Hello Wolrd
 4 public class HelloWorld {
 5
 6   String HELLO_world = "Hello World";
 7
 8   /*
 9   * Main function
10   */
11   public static void main(String[] args) {
12       // Print hello world to the console
13     System.out.println(HELLO_world);
14
15   }
16
17 }
```

Das in Listing 1.1 angeführte Beispiel wirkt auf den ersten Blick nicht zwingend falsch. Um eine statische Codeanalyse durchführen zu können, muss, wie bereits oben erklärt, zuerst eine Liste an Regeln, nach welchen geprüft werden soll, erstellt werden.

Um die Analyse Übersichtlich zu halten wird in diesem Beispiel nur auf folgende Punkte eingegangen:

- Übersetzbarkeit durch den Compiler (formale Richtigkeit).
- Anwendung von Kommentaren im JavaDoc-Stil.
- Allgemeine Formatierungsrichtlinien (Zeileneinzug und Variablenbezeichnung).

Nach genauerer Betrachtung des Quellcodes kann festgestellt werden, dass alle drei definierten Regeln verletzt wurden. Die Variable in Zeile 6 kann so definiert nicht in Zeile 13 verwendet werden, da der Modifier „static" fehlt. Die Kommentare sind nicht JavaDoc-konform und die Tiefe der Einrückung in Zeile 12 entspricht nicht so wie überall sonst zwei Leerzeichen. In Listing 1.2 wurden zum Vergleich alle Mängel aus dem vorherigen Beispiel behoben.

Listing 1.2: Hello World Example 2.

```
1 package thesis.analyse;
2
3 /**
4 * Simple Hello Wolrd
5 *
6 * @author Peter Ortner
7 *
8 */
9 public class HelloWorld {
10
11   private static final String HELLO_WORLD = "Hello World";
12
13   /**
14   * Main function
15   *
16   * @param args
17   *             (not used)
18   */
19   public static void main(String[] args) {
20     // Print hello world to the console
21     System.out.println(HELLO_WORLD);
22   }
23 }
```

Probleme

Die statische Codeanalyse ist weder universell einsetzbar noch für jedes Problem passend und unterliegt einigen Einschränkungen. Die drei schwerwiegendsten Probleme, die dieser Art der Analyse zugrunde liegen, sind folgende:

- Sourcecode muss vorhanden sein. Insbesondere bei der Verwendung von proprietärer Software kann aufgrund des fehlenden Quellcodes keine Aussage über die Qualität der Software getroffen werden.
- Nur eingeschränkte Möglichkeiten bezüglich der Aussage zum Laufzeitverhalten eines Programms. Dies ist eines der größten Defizite bei dieser Art der Analyse. Viele Probleme einer Software werden erst zur Laufzeit ersichtlich. Typische Vertreter von laufzeitspezifischen Problemen sind unter anderem Speicherlecks, Race Conditions oder ungültige Zugriffe auf den Speicher.

- Werkzeuge zur Analyse liefern oft Fehler, welche sich bei genauerer Betrachtung nicht als solche erweisen. Diese „False Positives“ verhindern eine gezielte automatisierte Korrektur aller Fehler und erfordern die nochmalige Kontrolle durch einen Entwickler.

Für die in den ersten beiden Punkten beschriebenen Probleme gibt es als mögliche Lösung ein weiteres Werkzeug - die dynamische Codeanalyse.

1.3.2 Dynamische Codeanalyse

Die dynamische Codeanalyse setzt im Gegensatz zur statischen Codeanalyse während der Laufzeit eines Programms an. Es wird das Verhalten eines Stücks Software zur Laufzeit analysiert, um einen Einblick auf das Verhalten sowie auf die Ressourcennutzung zu gewinnen. Die Anwendungsgebiete der dynamischen Codeanalyse erstrecken sich von der Erkennung von Schadsoftware [16] über Fehlererkennung [12] bis hin zum Vergleich von unterschiedlichen Algorithmen [7].

Da sich diese Arbeit im Bereich des Vergleichs von Software bewegt, wird im folgenden Abschnitt ein besonderer Blick auf den Vergleich von Algorithmen geworfen und es wird nicht näher auf Fehler oder Schadsoftwareerkennung eingegangen. Insbesondere die Klassen der Sortier- und Suchalgorithmen werden hervorgehoben, da diese zu den Grundfähigkeiten eines jeden Softwaresystems gehören und gut verstanden und vergleichbar sind.

Ablauf

Die Herangehensweise an die dynamische Codeanalyse ist stark abhängig vom verfolgten Ziel, verwendeten Tools und der Zielplattform. Um einen Überblick zu erhalten, wird im Folgenden in drei Kategorien unterschieden:

- Code-Injektion zur Übersetzungszeit durch Linking spezieller Bibliotheken oder toolgestützte Erweiterungen des Quellcodes.
- Virtualisierung der Laufzeitumgebung. Einziehen einer Abstraktionsschicht zwischen Programm und Zielplattform.
- Profiling und Debuggen in einer IDE. Diese Methode weist viele Parallelen zur Injektion von Code auf, wird aber als solches oft nicht wahrgenommen, da die IDE die Aufgabe der Manipulation des Quellcodes übernimmt.

Wie bereits angemerkt muss ein Programm, um es einer dynamischen Analyse unterziehen zu können, in einer ausführbaren Form vorliegen. Allerdings erweist sich die Betrachtung eines Programms während der Laufzeit als recht schwierig, da ohne weiteres Zutun kaum Informationen zu gewinnen sind. Um dieses Problem zu beheben, stehen im Grunde nur zwei Möglichkeiten zur Verfügung. Das ist zum einen die Injektion von Überwachungscode zur Übersetzungszeit, was entweder manuell oder automatisiert mit Toolunterstützung geschehen kann. Zum anderen kann die Laufzeitumgebung des Programms manipuliert werden, um zwischen Programm und Hardware alle für den Beobachter interessanten Informationen abgreifen zu können.

Beispiele

Als Beispiel für die dynamische Codeanalyse soll hier der Vergleich von Sortieralgorithmen hinsichtlich ihrer sogenannten Laufzeitkomplexität dienen. Andere Metriken zum Vergleich dieser Klasse von Problemen sind zum Beispiel die Anzahl der vorgenommenen Vergleiche oder Zuweisungen sowie die Speicherkomplexität.

In Tabelle 1.1 ist der Vergleich von 13 Sortieralgorithmen nach der $\mathcal{O}$-Notation [6] zu entnehmen. In diesem Beispiel ist zu erkennen, dass eine einfache Aussage über das Verhalten der Algorithmen nur schwer treffen lässt, da je nach verwendeten Input das Ergebnis aufgrund von Extremfällen stark variieren kann.

Probleme

Weitere Probleme der dynamischen Analyse im Bereich des Vergleichs von Programmen sind, neben der bereits erwähnten Abhängigkeit vom Input:

- Abhängigkeit von der Laufzeitumgebung (Hardware, Betriebssystem, Version der virtuellen Maschine).
- Abhängigkeit vom verwendeten Compiler und der gewählten Optimierungsstufe.

Auch zeigt diese Art der Gegenüberstellung von Programmen keine Anhaltspunkte bezüglich Unterschiede in der zugrunde liegenden Programmstruktur.

Tabelle 1.1: Übersicht Laufzeitkomplexität von verschiedenen Sortieralgorithmen [20].

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	$\Omega(n\log(n))$	$\Theta(n\log(n))$	$\mathcal{O}(n\hat{}2)$	$\mathcal{O}(\log(n))$
Mergesort	$\Omega(n\log(n))$	$\Theta(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n)$
Timsort	$\Omega(n)$	$\Theta(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n)$
Heapsort	$\Omega(n\log(n))$	$\Theta(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(1)$
Bubble Sort	$\Omega(n)$	$\Theta(n\hat{}2)$	$\mathcal{O}(n\hat{}2)$	$\mathcal{O}(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n\hat{}2)$	$\mathcal{O}(n\hat{}2)$	$\mathcal{O}(1)$
Selection Sort	$\Omega(n\hat{}2)$	$\Theta(n\hat{}2)$	$\mathcal{O}(n\hat{}2)$	$\mathcal{O}(1)$
Tree Sort	$\Omega(n\log(n))$	$\Theta(n\log(n))$	$\mathcal{O}(n\hat{}2)$	$\mathcal{O}(n)$
Shell Sort	$\Omega(n\log(n))$	$\Theta(n(\log(n))\hat{}2)$	$\mathcal{O}(n(\log(n))\hat{}2)$	$\mathcal{O}(1)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$\mathcal{O}(n\hat{}2)$	$\mathcal{O}(n)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$\mathcal{O}(nk)$	$\mathcal{O}(n+k)$
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$\mathcal{O}(n+k)$	$\mathcal{O}(k)$
Cubesort	$\Omega(n)$	$\Theta(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n)$

1.4 Ansatz

Wie in diesem Kapitel bisher gezeigt, dienen die dargestellten Analyseverfahren vordergründig der Vermeidung von Fehlern sowie der Verbesserung von Performance und Wartbarkeit eines Softwareproduktes. Da das Hauptaugenmerk dieser Arbeit auf dem Vergleich von Software zur Erkennung von Programmiermustern liegt, sind die gezeigten Methoden nur eingeschränkt nutzbar.

Im Zuge dieser Arbeit wird daher versucht, zwischen Programm und Hardware einzugreifen, alle ausgeführten Schritte der Software aufzuzeichnen und anhand dieser Aufzeichnung Charakteristika bezüglich Struktur des Programms festzumachen.

Kapitel 2

Versuchsaufbau

Dieses Kapitel dient dazu, den Aufbau für die Versuche, die verwendeten Tools und die herangezogenen Metriken für die Auswertung näher zu erläutern.

2.1 Setup

Im Wesentlichen besteht das verwendete Setup aus zwei Teilen. Zum einen die Laufzeitumgebung, welche das sogenannte Q'Wars Setup bereitstellt und zum anderen aus den Tools, welche zur Analyse und Auswertung für die während der Laufzeit der betrachteten Programme generierten Daten implementiert wurden.

2.1.1 Laufzeitumgebung

Das Q'Wars Setup ist ein im Jahr 2016 erstelltes Projekt des Vereins *Quantum Reboot*[1] für das alljährlich stattfindende *Ars Electronica Festival*[2], welches interessierten Personen das Programmieren auf spielerische Art und Weise näher bringen soll. Ziel des Aufbaues war es sogenannte Q'Bots in einem Oberon-Derivat (siehe Abschnitt 2.1.1)zu implementieren. Aufgabe dieser Q'Bots ist das Sammeln von Energiepillen um das eigene Überleben zu sichern und durch Zellteilung eine möglichst große Population zu generieren, um so als dominante Spezies zu überleben. Die Laufzeitumgebung setzt sich aus vier Teilen zusammen:

- **OberonQ:** ist eine an Oberon [17] angelehnte Programmiersprache. Oberon ist eine im Jahre 1991 von Niklaus Wirth und Jürg Gutknecht entwickelte Programmiersprache. Sie ähnelt stark den ebenfalls von Niklaus Wirth entworfenen Sprachen Pascal und Modula-2. Aufgrund der leichten Erlernbarkeit der Sprache fand sie sehr schnell Verwendung in Bildungseinrichtungen. OberonQ wurde gewählt, da Oberon mittlerweile kaum noch in realen Anwendungsfällen eingesetzt wird und daher eine simple Übernahme von bestehenden Code aus dem Internet, ohne Verständnis für diesen zu entwickeln, somit unterbunden wird. Um einen ersten Einblick auf OberonQ zu erhalten, ist in Listing 2.1 ein einfaches Programm angeführt, welches einen sogenannten Q'Bot implementiert.

[1] http://quantumreboot.com/
[2] https://www.aec.at/festival/

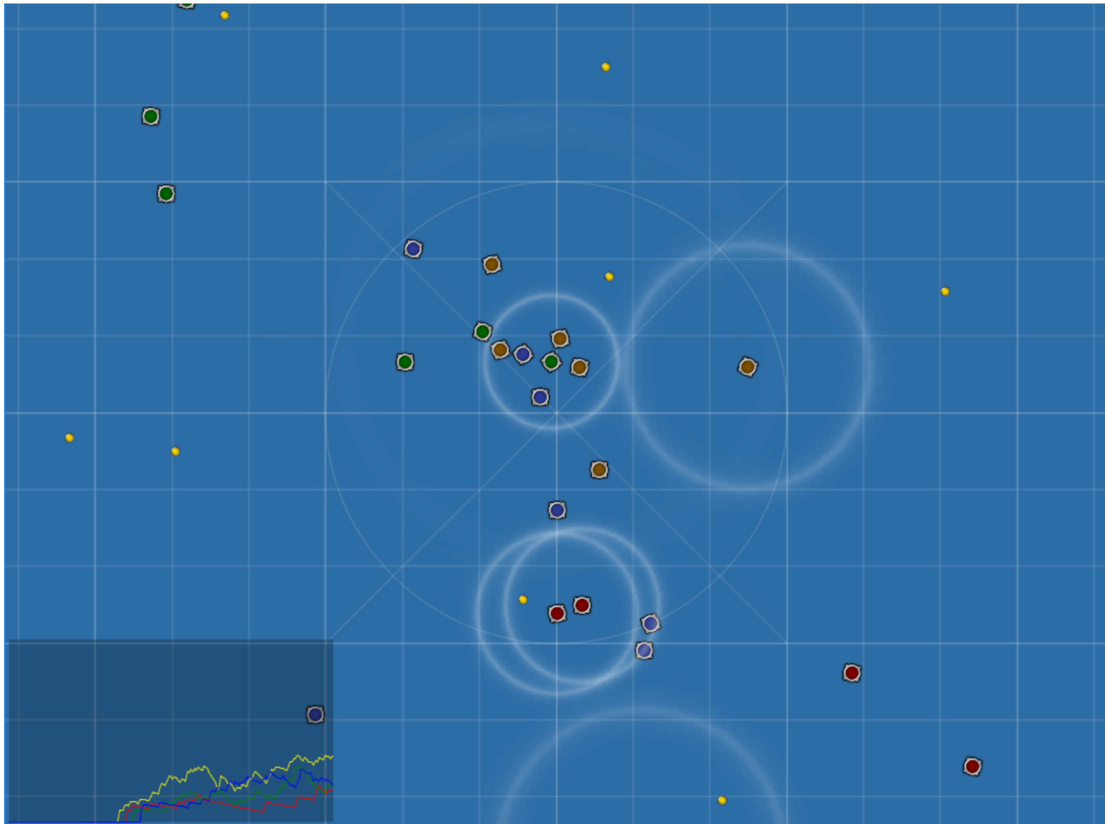

Abbildung 2.1: Q'Wars Simulator.

- **Quasar:** ist der für OberonQ entwickelte Compiler. Er ermöglicht das Übersetzen von OberonQ Programmen in für die virtuelle CPU (siehe Q'Wars Core) ausführbaren Maschinencode.
- **Q'Wars Core:** ist die Implementierung einer virtuellen CPU, welche zur Ausführung der in Maschinencode übersetzten OberonQ Programmen dient. Die CPU verfügt des Weiteren über einen vergleichsweise geringen Umfang an Instruktionen welche in den Tabellen 2.2 und 2.1 aufgelistet sind. Weiter war aufgrund der Zugänglichkeit zum Quellcode eine Adaptierung für die Datenaufzeichnung (siehe Abschnitt 2.1.2) möglich.
- **Q'Wars Simulator:** implementiert die Laufzeitumgebung, mit welcher die entwickelten Programme interagieren. Je nach implementiertem Setup müssen die Anwendungen unterschiedliche Aufgaben lösen. Im Falle des Q'Wars Setups war es notwendig die vom Simulator generierten Energiepillen zu finden und einzusammeln. In Abbildung 2.1 ist der Simulator mit vier sich bekämpfenden Q'Bots zu sehen. Zusammengehörige Q'Bots sind mit der selben Farbe gekennzeichnet. Im linken unteren Bereich ist der Gesamtbestand der einzelnen Populationen abgebildet. Die zu suchende Energie stellen die gelben Punkte dar. Die weißen Kreise symbolisieren das Scannen nach Energie ausgehend von den einzelnen Q'Bots.

Listing 2.1: Nanobot Example.

```
1 MODULE ExampleBot;
2 IMPORT Bot;
3 VAR bearing, dist, bat, agressionLevel: REAL;
4
5 BEGIN
6   (* Main loop *)
7   WHILE TRUE DO
8
9     (* Check battery if battery is more than half full clone*)
10    bat := Bot.GetBattery();
11    IF bat > 5000.0 THEN
12      Bot.Clone();
13    ELSE
14      (* Low energy -> increase speed *)
15      agressionLevel := 10.0 - (bat / 1000.0);
16    END
17
18    (* Scan environment for energy *)
19    Bot.ScanForTreats();
20
21    (* If energy found move to it *)
22    IF Bot.HasNextScan() THEN
23      Bot.GetNextScan(bearing, dist);
24
25      IF dist < agressionLevel THEN
26        dist := agressionLevel;
27      END
28
29      dist := dist / 15.0;
30      Bot.RunEngine(bearing, dist);
31    ELSE
32      (* No energy found go back to old position *)
33      bearing := bearing + 3.14159;
34      Bot.RunEngine(bearing, dist);
35    END
36
37    (* wait for engine to stop before start of the next iteration *)
38    Bot.Sleep(dist * 5.0);
39  END
40
41 END ExampleBot.
```

Der große Vorteil bei der Verwendung des Q'Wars Setups bietet sich in der vergleichsweise geringen Komplexität der CPU, sowie die Möglichkeit Erweiterungen nach eigenen Vorstellungen zu implementieren, da die vollständige Implementierung für diese Arbeit von *Quantum Reboot* zur Verfügung gestellt wurde.

Das gesamte Setup ist unter Verwendung der Programmiersprache Java in der Version 8 implementiert und somit plattformunabhängig.

Systemaufbau

In Abbildung 2.2 ist der Aufbau sowie der Workflow des Q'Wars Setups aufgezeichnet. Es wird damit begonnen, dass ein Entwickler ein Programm in OberonQ implementiert und mittels des bereitgestellten Compilers (Quasar) in eine für die virtuelle CPU

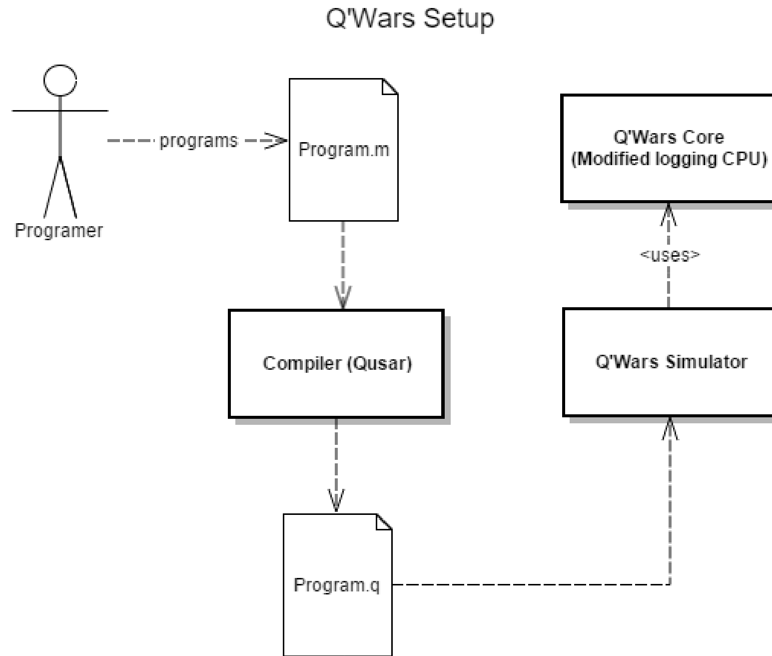

Abbildung 2.2: Schematischer Aufbau des Q'Wars Setups.

verständliche Form bringt. Das nun fertig übersetzte Programm wird dem Simulator übergeben. Dieser wiederum kann anschließend mithilfe der virtuellen CPU das Programm ausführen.

2.1.2 Datenaufzeichnung

Da bei der Entwicklung des Q'Wars Setups keine Möglichkeit zur dynamischen Codeanalyse vorgesehen wurde, musste im Zuge dieser Arbeit ein entsprechender Mechanismus ergänzt werden. Zu diesem Zweck wurde eine zweite Implementierung der virtuellen CPU umgesetzt, welche es ermöglicht jede ausgeführte Instruktion zu erfassen und nach einer zuvor definierten Menge an durchlaufenen Zyklen abzuspeichern.

Um den normalen Betrieb nicht zu beeinflussen, kann das Setup von außen konfiguriert werden und je nach Bedarf die normale CPU durch die Analyse-CPU ausgetauscht werden. Auch kann die Menge an aufgezeichneten Zyklen bis zur Generierung der Auswertung von außen via eines Files konfiguriert werden. Die neue CPU unterscheidet sich von der ursprünglichen nur durch den Aufruf einer Subfunktion, welche die aktuell auszuführende Instruktion sowie den Stackpointer und bei Speicheroperationen die Speicheradresse in einem assoziativen Datenfeld sichert. Nach Ablauf einer zuvor gesetzten Menge an Instruktionen wird die nun fertig befüllte Datenstruktur als File auf die Festplatte des ausführenden Endgerätes serialisiert.

Die konkrete Implementierung ist der beigelegten CD unter dem Ordner Implementierung/Datenaufzeichnung zu entnehmen.

Tabelle 2.1: Übersicht Instruktionen Q'Wars CPU 1.

Instruktion	Beschreibung
Mathematische Operationen	
ADD	Addition zweier ganzen Zahlen
SUB	Subtraktion zweier ganzen Zahlen
MUL	Multiplikation zweier ganzen Zahlen
DIV	Division zweier ganzer Zahlen
MOD	Modulo Division zweier ganzen Zahlen
ADDF	Addition zweier Fließkommazahlen
SUBF	Subtraktion zweier Fließkommazahlen
MULF	Multiplikation zweier Fließkommazahlen
DIVF	Division zweier Fließkommazahlen
NEG	Negieren einer ganzen Zahl
NEGF	Negieren einer Fließkommazahl
ABS	Absolutbetrag einer ganzen Zahl
ABSF	Absolutbetrag einer Fließkommazahl
NOT	Bitweises nicht
XOR	Bitweises XOR
Speicheroperationen	
LDG	Load address global
LDF	Load relative to frame pointer
LDA	Load address (global, pc-relativ)
LDAF	Load address, relative to frame pointer
LOAD	Load from memory
LOADC	Load address
LOADFC	Load address
STG	Store global
STF	Store relativ to frame pointer
STORE	Write to memory
Sprunganweisungen	
RET	Return
BSR	Branch subroutine
BRN	Branch
BRZ	Branch Zero
BNZ	Branch not Zero
SYSCALL	Aufruf Systemfunktion
JUMP	Sprung Operation
JSR	Jump to Subroutine
JSRI	Jump to Subroutine
Konvertierung	
RTI	Fließkommazahl zu Integer
ITR	Integer zu Fließkommazahl

Tabelle 2.2: Übersicht Instruktionen Q'Wars CPU 2.

Instruktion	Beschreibung
Vergleichsoperationen	
EQL	Equals
NEQ	Not Equal
LEQ	Less or Equal
GEQ	Greater or Equal than
LSS	Less than
GTR	Greater than
EQLF	Equal für Fließkomma
NEQF	Not Equal für Fließkomma
LEQF	Less or Equal für Fließkomma
GEQF	Greater or Equal für Fließkomma
LSSF	Less für Fließkomma
GTRF	Greater für Fließkomma
Sonstige	
NOP	Keine Operation
HALT	Beenden der Ausführung
ALLOC	Speicher dynamisch anfordern
FREE	Dynamisch angeforderten Speicher wieder freigeben
UNKNOWN	Keine gültige Instruktion
WRI	Write integer
WRL	Write newline
WRC	Write char
WRF	Write float
WRFL	Write flush
SWAP	Austausch zweier Werte

Aufbau log Dateien

Es wurden mehrere Arten der Datenaufbereitung umgesetzt, um je nach Auswertung nicht unnötige Vorarbeiten hinsichtlich Nachbearbeitung der doch recht umfangreichen Daten treffen zu müssen. Alle aufgezeichneten Daten werden im sogenannten „comma-separated values“ Format [14] abgespeichert, um eine spätere automatisierte Weiterverarbeitung zu vereinfachen. Diese Art von Dateiformat lässt sich programmatisch vergleichsweise einfach einlesen und alle gängigen Tabellenverarbeitungsprogramme können dieses Datenformat ebenfalls verarbeiten. Die so durch die angepasste CPU generierten Daten unterscheiden sich in der Struktur wie folgt:

- Liste von Instruktionen und deren Anzahl.
- Liste von aufeinanderfolgenden Instruktionen, getrennt durch Sprungoperationen.
- Liste von Instruktionen, gereiht nach dem Zeitpunkt, zu der die Instruktion auftrat.

- Liste von Instruktionen, gereiht nach dem Zeitpunkt, zu der die Instruktion auftrat, die aktuelle Position des Stackpointers und für Speicheroperationen die verwendete Speicheradresse.

2.2 Auswertung

Da eine Auswertung per Hand aufgrund der Menge und Struktur der Daten für den Menschen nur schwer möglich ist, wurden für diese Arbeit verschiedene Ansätze verfolgt und Programme entwickelt, um die Auswertung anhand unterschiedlicher Metriken zu vereinfachen.

Im Folgenden werden die Analysemethodiken näher erläutert, welche für diese Arbeit betrachtet wurden.

2.2.1 Cluster Analyse

Der erste Ansatz, die erfassten Daten in Relation zueinander zu setzen, war die Durchführung einer sogenannten Clusteranalyse.

Diese Art der Analyse dient dazu, Ähnlichkeiten in großen Datenbeständen ausfindig zu machen. Gruppen von so als ähnlich definierten Objekten werden als sogenannte Cluster bezeichnet. Die Clusteranalyse zählt zu der Klasse der uninformierten Verfahren, da keine Vorinformationen bezüglich der Daten vorliegen [4].

Für die Durchführung einer Clusteranalyse stehen zahlreiche Algorithmen zur Verfügung, welche sich anhand ihres algorithmischen Aufbaus, der Art des Cluster-Modells sowie der Definition von Ähnlichkeit unterscheiden.

Einer der bekanntesten Algorithmen dieser Klasse ist der sogenannte K-Means-Algorithmus, welcher auch für diese Arbeit herangezogen wurde. Grund für diese Entscheidung war vor allem die Einfachheit des Algorithmus, um somit schneller ein Ergebnis erzielen zu können. Dadurch zeigt sich, ohne erheblichen Mehraufwand, ob eine weitere Betrachtung in Hinblick auf die Analyse via Clustering zielführend ist.

K-Means

Konkret wurde der K-Means-Algorithmus verwendet um erste Ergebnisse hinsichtlich Ähnlichkeiten von Programmen zu erkennen und eine Richtung für weitere genauere Analyseschritte zu finden.

Der K-Means-Algorithmus ist ein sogenannter Prototypbasierter Clusteralgorithmus. Dies bedeutet, dass die einzelnen Cluster durch Prototypen von Punkten repräsentiert werden, um welchen sich die Daten sammeln. Solch ein Prototyp kann zum Beispiel der Mittelpunkt von Punkten im euklidischen Raum sein. Für diesen Algorithmus betrachtet man normalerweise Daten aus dem n-dimensionalen kontinuierlichen Raum [3].

Funktionsweise

Der K-Means-Algorithmus hat die Aufgabe einen Datensatz in k Cluster zu unterteilen. Um dies zu erreichen, werden aus der Datenmenge k Punkte ausgewählt und als Zentren markiert. Im Anschluss werden alle Datenpunkte ihrem am nächsten liegenden Zentrum

zugewiesen. Aus allen Punkten eines so entstanden Clusters wird nun ein neues Zentrum berechnet. Nun werden alle Datenpunkte den neu entstanden Zentren zugewiesen. Dies geschieht solange bis sich die Clusterzentren nicht mehr verändern.

Der Algorithmus lautet somit wie folgt:

1. Auswahl von k Punkten als Anfangszentren.
2. Starte Wiederholung.
3. Ordne Datenpunkte dem nächstliegenden Zentrum zu.
4. Neuberechnung der Zentren.
5. Beende Wiederholung, wenn sich die Zentren nicht mehr ändern.

Die Auffindung der Cluster wird wesentlich durch zwei Faktoren bestimmt. Der erste Faktor ist die Methode zur Wahl der Anfangszentren. Der zweite Faktor ist die Definition des Begriffes der Nähe von zwei Punkten.

Zwei gängige Methoden zur Bestimmung der Anfangszentren sind zum einen die zufällige Wahl von k Punkten aus der gegebenen Datenmenge und zum anderen die Wahl eines beliebigen Punktes und die Berechnung der weiteren Punkte, indem der Datensatz mit der größten Distanz zu den bereits gesetzten Zentren gewählt wird. Mit der ersten Methodik können durch mehrmaliges Ausführen des Algorithmus verschiedene Ergebnisse erzielt werden, was unter Umständen einzelne Ausreißer im Datensatz erkennbar macht. Die zweite Methodik liefert immer die gleiche Lösungsmenge (sofern der erste Datenpunkt immer gleich gewählt wird) und erfasst im Regelfall extreme Ausprägungen im gesamten Datensatz.

Des Weiteren können verschiedene Heuristiken für die Festlegung der Distanz zwischen zwei Punkten definiert werden. In dieser Arbeit wurden zwei Heuristiken betrachtet: zum einen die euklidische Distanz,

$$d(x, y) = \sum_{i=1}^{n} (x_i - y_i)^2 \tag{2.1}$$

welche den Abstand zwischen zweier n-dimensionaler Punkte beschreibt und zum anderen der Kosinus-Ähnlichkeit,

$$\cos(\theta) = \frac{\sum_{i=1}^{n} (x_i - y_i)^2}{\sqrt{\sum_{i=1}^{n} x_i^2} \sqrt{\sum_{i=1}^{n} y_i^2}} \tag{2.2}$$

welche sich durch den Kosinus des durch die beiden Vektoren x und y eingeschlossenen Winkels θ definiert.

Probleme

Die Eigenart der Daten, welche ein Programm als Wertepaare von Instruktionen und deren jeweilige Häufigkeit repräsentiert, hat zur Folge, dass wenn man ein Programm als n-dimensionalen Vektor im euklidischen Raum betrachtet, dieser die Ausprägung $n = 59$ aufweist. Da viele Instruktionen allerdings nur sehr selten oder je nach Programm überhaupt nicht auftreten, kann dies zu schwer interpretierbaren Resultaten führen.

Lösungen

Um der oben beschriebene Problematik entgegen zu wirken, wurden Klassen von Instruktionen erfasst um somit die maximale Ausprägung von $n = 59$ zu reduzieren. In den Tabellen 2.2 und 2.1 sind die Instruktionen in einzelne solche Klassen separiert.

2.2.2 Visuelle Aufbereitung

Aufgrund der Tatsache, dass sich keine klare Abhängigkeit der Daten zueinander in den ersten Versuchen via Clusteranalyse erkennen ließ, mussten weitere Ideen zur Informationsgewinnung gefunden werden.

Ansatz

Da die Analyse der Datenmenge durch algorithmische Aufbereitung nicht aussagekräftig war und die Daten in Rohform für den Menschen ebenfalls keine Schlussfolgerungen zulassen, wurde überlegt, die Daten visuell aufzubereiten. Eine visuelle Darstellung von Datensätzen bietet den Vorteil, dass das menschliche Gehirn im Bereich der Musterkennung einer Maschine deutlich überlegen ist.

Als konkrete Darstellungsmethode, wurde versucht die Datensets zum einen als Wertepaare in Tabellenverarbeitungsprogrammen darzustellen und zum anderen in eigenen Programmen aufzubereiten.

Tabellenverarbeitung

Die Aufbereitung der Daten in gängigen Tabellenverarbeitungsprogrammen wie zum Beispiel Microsoft Excel erwies sich ähnlich wenig aussagekräftig wie die Verarbeitung via K-Means, da nur Ausreißer aus der Gesamtdatenmenge erkennbar gemacht werden konnten.

Lochkarten

Für eine bessere Einsicht in die Struktur einer analysierten Software wurde für diese Arbeit ein experimentelles Programm entwickelt, welches die einzelnen Instruktionen als zeitliche Abfolge aufbereitet und darstellt.

Ziel ist die Erkennung von Mustern innerhalb der Programmstruktur. In Abbildung 2.3 ist eine solche Auswertung zu betrachten. Zu erkennen sind sich wiederholende Sequenzen in einem Programm, wie sie in wiederkehrenden Aufrufen von Funktionen oder Schleifendurchläufen auftreten können. Auch zu erkennen ist die geringe Menge an Instruktionen, welche visualisiert wird, da nur diejenigen Instruktionen dargestellt werden, die auch mindestens einmal durchlaufen wurden.

2.2.3 Analyse anhand von Metriken

Dieser Abschnitt dient zur näheren Erklärung der schlussendlich verwendeten Metriken, welche zur Unterscheidung und Kategorisierung der betrachteten Programme verwendet wurden. Des Weiteren wird auf weitere mögliche Metriken eingegangen, welche im Zuge dieser Arbeit keine Verwendung gefunden haben.

Übersicht

Zu Beginn wird eine einfache Übersicht über alle Metriken mit einer kurzen Erklärung gegeben. Diejenigen Metriken, welche in der Arbeit Verwendung gefunden haben, werden im Anschluss nochmals näher erläutert.

- **Instruction Mix:** beschreibt die relative Verteilung aller Instruktionen eines Programms, welche während der Laufzeit ausgeführt werden. Diese Metrik eignet sich recht gut als Basis für die Clusteranalyse und lässt eine grobe Vorsortierung zu, um Ausreißer in den Datensätzen festzumachen.
- **Branch Direction:** beschreibt die Richtung von Sprunganweisungen. Diese können je nach der Richtung, in welche von der aktuellen Position aus gesprungen wird, in zwei Arten unterschieden werden. Zum einen in „forward“ und zum anderen in „backward“ Branches. Diese Metrik berechnet den prozentualen Anteil von „forward“ Branches aus der Gesamtanzahl an ausgeführten Sprungoperationen.
- **Taken Branches:** gibt die prozentuale Anzahl an Sprungoperationen aus der Gesamtanzahl an durchlaufenen Instruktionen an.
- **Dynamic Basic Block Size:** beschreibt die relative Größe einer Programmsektion mit einem Eintritts- und einem Austrittspunkt. Berechnet wird für diese Metrik die durchschnittliche Größe eines solchen Blocks in einem Programm. Diese Metrik gibt einen guten Einblick darauf wie fein granular ein Programm in Subfunktionen zerlegt wurde.
- **Dependency Distance:** gibt die durchschnittliche Anzahl an Instruktionen zwischen einer Leseoperation und Schreiboperation auf ein- und demselben Speicherbereich an. Diese Metrik dient neben der Auswertung des Stackpointers dazu die Häufigkeit von Variablennutzung zu erkennen.
- **Instruction Temporal Locality:** gibt die durchschnittliche Distanz (Anzahl an Instruktionen) zwischen zwei gleichen Instruktionen an. Diese Metrik gibt ebenfalls Einblick auf die Struktur des analysierten Programms.
- **Instruction Pairs:** sind aufeinanderfolgende Paare von Instruktionen. Dient zur Erkennung von häufig verwendeten Ablauffolgen innerhalb einer Software.
- **Average Stack Usage:** beschreibt den vom Programm während der Laufzeit benötigten Speicherbedarf. Diese Metrik liefert den durchschnittlichen Speicherbedarf eines Programms und lässt Rückschlüsse auf die Ressourcennutzung zu.
- **Laufzeit:** ist die Dauer die ein Programm vom Start bis zum Erreichen der angestrebten Lösung benötigt.
- **Programmgröße:** ist die Größe eines Programms nach der Übersetzung durch den Compiler.

Diese Metriken lassen sich wiederum recht einfach in drei Kategorien einteilen:

- **Struktur:** bestehend aus Instruction Mix, Branch Direction, Taken Branches, Dynamic Basic Block Size, Instruction Temporal Locality und Instruction Pairs
- **Speicherverhalten:** bestehend aus Dependency Distance und Average Stack Usage
- **Konstanten:** bestehend aus Laufzeit, Programmgröße

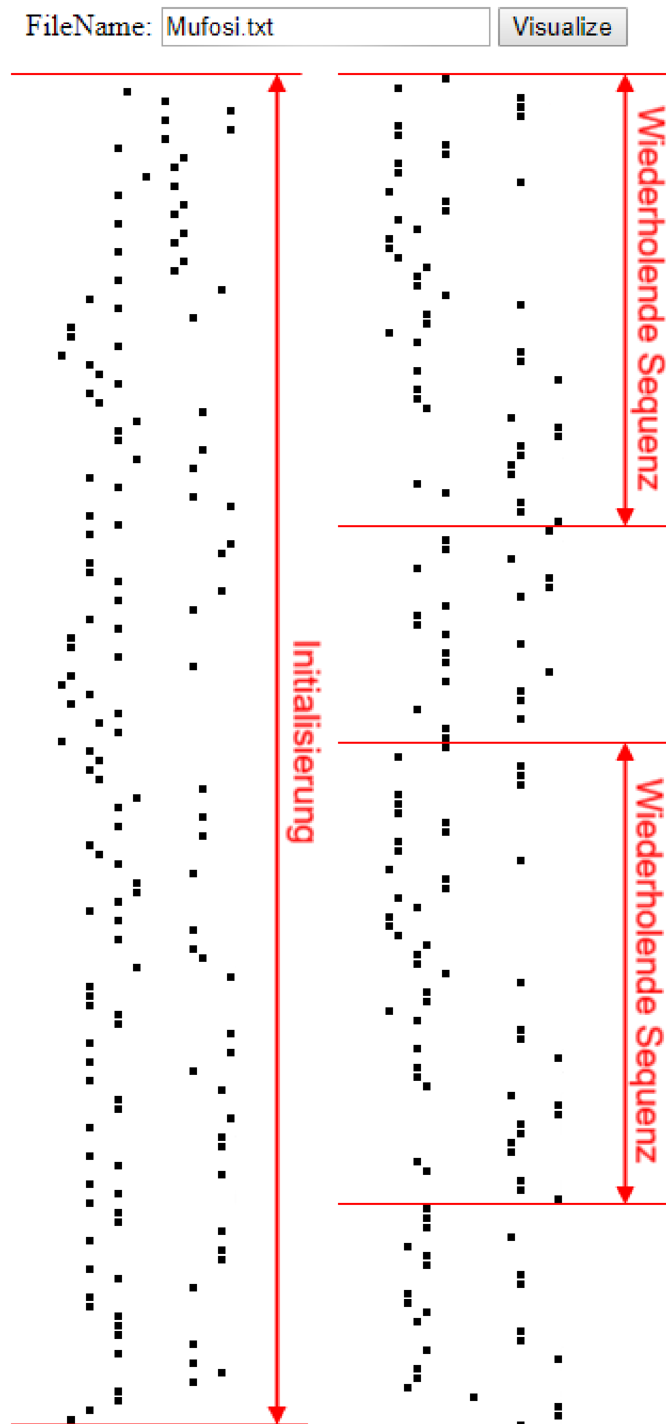

Abbildung 2.3: Beispiel für die visuelle Aufbereitung als Lochkartenmuster.

Wie unschwer zu erkennen ist, sind in der Kategorie „Struktur“ vergleichsweise viele Metriken zu verzeichnen. Als Resultat dieser Beobachtung wird hier im späteren Verlauf (siehe Kapitel 3) eine genauere Unterscheidung getroffen. Bezüglich der „Laufzeit“ ist zu erwähnen, dass hier eine Zeitmessung, wie sie für diese Metrik üblich wäre, nicht zur Anwendung kommt. Grund hierfür ist, dass eine solche Messung stark von der zu Grunde liegenden Hardware abhängig ist. Ein besseres Maß für die Laufzeit würde die Anzahl der durchlaufenen Instruktionen bieten, wobei jede Instruktion je nach ihrer zugrunde liegenden Implementierung gewichtet wird und somit eine konstante Metrik gewährleistet werden kann. Allerdings wird im Versuchsaufbau je Programm nur ein Ausschnitt von 1000 Instruktionen betrachtet und somit ist die Laufzeit für die hier angewandten Betrachtungen irrelevant.

Instruktion Mix

Der sogenannte Instruktion Mix gibt die relative Anzahl der verwendeter Instruktionen innerhalb eines Programms wieder [11]. Hier wird eine allgemeine Aussage über die Struktur des Programms gemacht. Je nach Zusammenfassung von verschiedenen Instruktionen in einzelne Kategorien (z.B. alle arithmetischen Operationen) kann eine grobe Unterscheidung und mögliche Vorsortierung mit Hilfe einer Clusteranalyse getroffen werden.

Dynamic Basic Block Size

Stellt eine Kennzahl zur Strukturgröße des betrachteten Programms dar [11]. Ein „gut“ implementiertes Programm zeichnet sich durch seine geringe Größe von einzelnen Sektionen aus. Ein bekanntes Paradigma in der Programmierung sagt aus, dass pro Funktion nur eine Aufgabe abgearbeitet werden sollte [9]. Eine kleine Zahl in dieser Metrik lässt darauf schließen, dass dieses Paradigma eingehalten wurde und der Code eine vergleichsweise gute Grundstruktur aufweist.

Dependency Distance

Diese Metrik ist definiert durch die absolute Anzahl an Instruktionen zwischen einer Leseoperation und einer Schreiboperation auf ein und dasselbe Register [5, 10]. Da das Q’Wars Setup eine registerlose CPU implementiert, wird für diese Metrik die Distanz zwischen der Lese und Schreibzugriffe auf ein und dieselbe Speicheradresse ermittelt. Durch diese Änderung kann die durchschnittliche Lebenszeit einer Variable im Speicher ermittelt werden. Somit sind auch Informationen über die Variablennutzung des analysierten Programms verfügbar.

Instruction Temporal Locality

Hier wird die durchschnittliche Anzahl an Instruktionen zwischen zwei gleichen Instruktionen gemessen. Ein im Vergleich hoher Wert dieser Metrik kann darauf schließen, dass es kaum wiederkehrende Sequenzen im Programmablauf bzw. wiederkehrende Sequenzen sehr weit voneinander entfernt sind und gibt somit ebenfalls einen Einblick in die Struktur des Programms.

Eval Tool

K:\IM3_IGT\IGT_Master\Auswertung\ExampleBots\Export

Browse

..
AllInst
Clust1
Clust2
Count
SPPC
Trace

Name	Average Stack Usage	Dependency Distance	Instruction Temporal Locality	Dynamic Basic Block Size	Program Size	Cluster
TMNT.q.csv	62.56	52.74	52.39	16.18	2884.0	0.0
CircleOfFriss.m.csv	62.56	38.05	37.72	19.52	2400.0	0.0
Shredder.q.csv	143.0	65.3	44.56		788.0	1.0
Hybris-1.2.0.q.csv	43.52	91.0	92.45	16.18	3124.0	2.0
GloryBot.q.csv	62.56	43.4	48.87	18.66	2308.0	2.0
Meebo.q.csv	58.88	105.81	105.94	21.05	3568.0	2.0
ExampleBot.m.csv	43.52	111.91	59.83	22.08	2864.0	2.0
Hbnm.q.csv	50.05	108.94	77.55	22.21	3620.0	2.0
CloneWars.q.csv	52.68		82.89	29.74	3104.0	2.0
fz_Test_Bot.q.csv	77.0	32.6	24.65	11.23	1900.0	3.0
fz_Test_Bot_v3.q.csv	77.0	57.15	42.11	11.87	2084.0	3.0
SoloPlayer.m.csv	77.0	42.94	49.77	12.38	2340.0	3.0
DoomBot.m.csv	62.56	139.5	79.62	13.22	2556.0	3.0
YOUGONNALOSEAGAINSTTHISBOT.q.csv	62.56	63.76	81.36	13.25	2676.0	3.0
KingOfHill.m.csv	62.56	65.11	101.64	13.46	2592.0	3.0
Mufosi.m.csv	62.56	87.8	62.96	13.71	2164.0	3.0
ZergRush.m.csv	62.56	44.72	41.76	15.82	2752.0	3.0
Herbert.q.csv	62.56	51.92	45.79	16.06	2348.0	3.0
ZergRushCrazy.m.csv	62.56	43.32	55.34	16.26	2812.0	3.0
ZergRushMetal.m.csv	62.56	50.85	69.59	16.6	2972.0	3.0
dr_NoBo.q.csv	62.56	68.11	60.8	17.32	2736.0	3.0
Wasti.q.csv	47.66	92.7	111.02	17.46	2172.0	3.0
LoKr_LoBot.m.csv	62.56	57.41	68.51	17.89	2280.0	3.0

Clusters
4
Algorithm
k-means
Similarity
euclidean di
Analyse

Abbildung 2.4: Tool zur aufbereiten der Metriken.

Average Stack Usage

Durch die Beobachtung des Stackpointers kann der durchschnittliche Bedarf an statischem Speicher eines Programms ermittelt werden. Mithilfe dieser Metrik können Rückschlüsse auf die Anzahl an verwendeten Variablen gezogen werden.

Programmgröße

Die Programmgröße kann, je nach verwendetem Compiler und Optimierungsstufe, stark variieren. Da das verwendete Setup nur einen Compiler ohne Optimierung zur Verfügung stellt, können durch diese Metrik Programme erkannt werden, die eine unübliche Größe aufweisen.

Aufbereitung

Zur Berechnung und Darstellung der Metriken wurde für diese Arbeit ein Programm entwickelt, das die zuvor ermittelten Metriken aufbereitet. In Abbildung 2.4 ist das Tool zu sehen, welches eine größere Menge von Programmen übersichtlich darstellt. Für jedes analysierte Programm sind die Metriken Average Stack Usage, Dependency Distance, Instruction Temporal Locality und Dynamic Basic Block Size abgebildet. Weiter wird zusätzlich eine Clusteranalyse durchgeführt und die jeweilige Programmgröße angezeigt. In Abbildung 2.5 ist zum Abschluss noch das gesamte Analyse-Setup schematisch dargestellt.

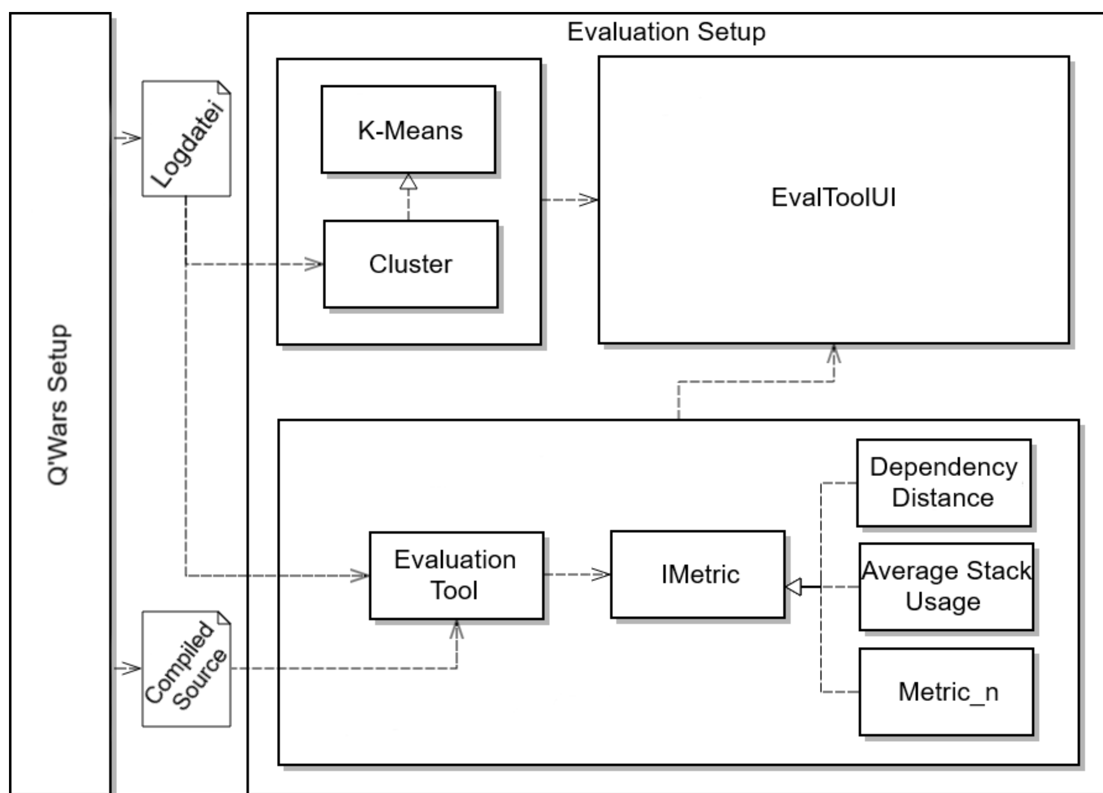

Abbildung 2.5: Übersicht über den Ablauf der Analyse.

Kapitel 3

Umsetzung

Dieses Kapitel präsentiert die Auswertungen und Analysen zu den in den vorhergegangen Kapiteln beschriebenen Problemstellungen.

3.1 Verwendeter Input

Im Zuge dieser Arbeit wurden bestehende OberonQ Programme analysiert, welche für die Q'Bot-Challenge beim Ars Electronica Festival 2016 erstellt wurden. Diese Auswertungen waren allerdings aufgrund der zum Teil fehlenden Quelldateien und der Art der Aufgabenstellung nicht sonderlich aussagekräftig. Um ein besseres Ergebnis zu erzielen und aussagekräftige Schlussfolgerungen treffen zu können, wurde im späteren Verlauf stattdessen unterschiedliche Näherungsverfahren für die Zahl π implementiert und analysiert.

3.1.1 Q'Bots

Wie bereits in Kapitel 2 angemerkt, wurde das Q'Wars Setup als Umgebung für diese Arbeit gewählt. Im Zuge der Entwicklung dieses Setups sowie dem Einsatz beim Ars Electronica Festival 2016 wurden bereits einige Programme in OberonQ abgefasst und standen von Beginn an für diese Arbeit zur Verfügung. Leider waren diese zumeist nur als übersetztes Programm ohne Sourcecode vorhanden, was auch mitunter zur Wahl des Themas dieser Arbeit geführt hat. Zur Auswertung wurden 23 Programme von acht Entwicklern herangezogen und mittels unterschiedlicher Methoden analysiert. Hauptaugenmerk lag auf der Clusteranalyse via dem K-Means-Algorithmus. Aufgrund der zumeist fehlenden Quelldateien und der sehr variablen Lösbarkeit für die gestellte Aufgabe eine primitive künstliche Intelligenz zu Implementieren, war die Aussagekraft dieser Analysen nicht sehr ergiebig.

Probleme

Wie bereits erwähnt, kam es bei der Betrachtung und Auswertung der Q'Bots zu einigen Problemen. Diese beliefen sich vordergründig auf die hohe Variation bei den Lösungsansätzen. Um dies besser zu veranschaulichen, sind in den Listings 3.1 und 3.2 zwei sich stark unterscheidende Q'Bots dargestellt. Die erste Implementierung mit dem Namen

„Shredder“, zeigt eine recht übliche Herangehensweise an die Problemstellung. Es wird die Umgebung, auf sich in der nähe befindliche Energiepillen überprüft und bei positiver Detektierung, wird die erste gefundene Pille angesteuert. Weitere Features sind, dass zurückfahren auf die alte Position, falls keine Pille gefunden wurde, die Zellteilung bei Überschreitung von 50% Energie und das erhöhen der Bewegungsgeschwindigkeit bei sinkendem Energievorrat. Die zweite Implementierung mit dem Namen „CircleOfFriss“, setzt hingegen auf die Beobachtung, dass der Simulator die Energiepillen im Zentrum der Simulation in einem kreisförmigen Bereich generiert. Daher wird nicht auf die Detektierung von Energiepillen gesetzt, sondern schlicht ein kreisförmiger Weg abgefahren in der Hoffnung zufällig Energiepillen zu finden. Des Weiteren kann dieser Q’Bot bei einer Zellteilung, seinen Klone durch Mutation anders parametrisieren.

Listing 3.1: Q’Bot „Shredder“.

```
1 MODULE Shredder;
2 IMPORT Bot;
3 VAR bearing, dist, bat, agressionLevel: REAL;
4
5 BEGIN
6   (* Main loop *)
7   WHILE TRUE DO
8
9     (* Check battery if battery is more than half full clone*)
10    bat := Bot.GetBattery();
11    IF bat > 5000.0 THEN
12      Bot.Clone();
13    ELSE
14      (* Low energy −> increase speed *)
15      agressionLevel := 10.0 - (bat / 1000.0);
16    END
17
18    (* Scan environment for energy *)
19    Bot.ScanForTreats();
20
21    (* If energy found move to it *)
22    IF Bot.HasNextScan() THEN
23      Bot.GetNextScan(bearing, dist);
24
25    IF dist < agressionLevel THEN
26      dist := agressionLevel;
27    END
28
29    dist := dist / 15.0;
30    Bot.RunEngine(bearing, dist);
31    ELSE
32      (* No energy found go back to old position *)
33      bearing := bearing + 3.14159;
34      Bot.RunEngine(bearing, dist);
35    END
36
37    (* wait for engine to stop before start of the next iteration *)
38    Bot.Sleep(dist * 5.0);
39  END
40 END Shredder.
```

Listing 3.2: Q'Bot „CircleOfFriss“.

```
1 MODULE CircleOfFriss;
2 IMPORT Bot, Math, Random;
3 VAR bearing, offset, speed, rad, correction: REAL;
4
5   (* calculation of current energy level *)
6   PROCEDURE BatLevel(): REAL;
7   VAR max, cur: REAL;
8   BEGIN
9     max := Bot.GetMaxBattery();
10    cur := Bot.GetBattery();
11    RETURN cur / max;
12  END BatLevel;
13
14  (* generating a mutation value i.e. +0.5 or -0.5 *)
15  PROCEDURE Mutate(): REAL;
16  BEGIN
17    IF Random.NextReal() < 0.5 THEN
18      RETURN -0.01;
19    ELSE
20      RETURN 0.01;
21    END
22  END Mutate;
23
24 BEGIN
25  bearing := 0;
26  offset := 0;
27  speed := 0;
28  rad := 10.0;
29  correction := 5.0;
30
31  (* main loop *)
32  WHILE TRUE DO
33    (* movement calculation *)
34    bearing := bearing + ((3.14159 / 180.0) * rad);
35    Bot.RunEngine(bearing+offset, 0.1+speed);
36
37    (* clone when 70% energy is reached *)
38    IF BatLevel() > (0.7) THEN
39      IF Bot.Clone() THEN
40        offset := Mutate();
41        speed := Mutate();
42      END
43    END
44
45    rad := rad + correction;
46    IF rad >= (90.0) THEN
47      correction := -5.0;
48    END
49    IF rad <= 0.0 THEN
50      correction := 5.0;
51    END
52  END
53 END CircleOfFriss.
```

3.1.2 Berechnung von π

Um die in der vorherigen Sektion Q'Bots beschriebenen Probleme zu umgehen und aussagekräftige Werte zu generieren wurden Programme geschrieben, welche bessere Eigenschaften bezüglich ihrer Vergleichbarkeit aufweisen. Zu diesem Zweck bieten sich mathematische Problemstellungen der klasse Approximation recht gut an, da sie beliebig lange ausgeführt werden können und es für ein und dieselbe Lösung mehrere Ansätze zur Berechnung gibt. Insbesondere die Kreiszahl π hat über die Jahrhunderte viele verschiedene Mathematiker beschäftigt und daher wurden hierfür viele Berechnungsmodelle entwickelt.

Näherungsverfahren

Für die Auswertungen in dieser Arbeit wurden sechs Näherungsverfahren für die Kreiszahl π in verschiedenen Varianten implementiert und miteinander verglichen. Es folgen alle für die Auswertung verwendeten Näherungsverfahren mit einer kurzen Beschreibung der zugrunde liegenden mathematischen Definition, die konkreten Implementierungen in OberonQ sind dem Anhang C zu entnehmen.

- Die Approximation nach Leonhard Euler [1] definiert sich durch

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}. \tag{3.1}$$

- Eine alternative Approximation zur Berechnung nach Euler, welche nur eine geringe Abwandlung der Ersten darstellt, lautet

$$\sum_{n=1}^{\infty} \frac{1}{n^4} = \frac{\pi^4}{90} \tag{3.2}$$

  und weist eine vergleichsweise höhere Konvergenzgeschwindigkeit auf.

- Die Approximation nach John Wallis aus dem Jahr 1655 [15], ist durch die vergleichsweise einfachen Berechnung

$$\prod_{n=1}^{\infty} (1 - \frac{1}{4n^2}) = \frac{2}{\pi} \tag{3.3}$$

  definiert.

- Die Approximation nach Abraham Sharp [13], welche um 1700 entstand, definiert sich durch

$$\sum_{n=0}^{\infty} \frac{2(-1)^n 3^{\frac{1}{2-n}}}{2n+1} = \pi. \tag{3.4}$$

  .

- Die Approximation nach Gottfried Wilhelm Leibniz [13] aus dem Jahr 1682 ist durch

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{\pi}{4} \tag{3.5}$$

  definiert. Ein Manko dieser recht einfachen Art der Berechnung, ist die vergleichsweise langsame Konvergenzgeschwindigkeit.

- Die Approximation nach François Viète[1] aus dem Jahr 1593 ist durch den Ausdruck

$$\lim_{n \to \infty} \prod_{i=1}^{n} \cos\left(\frac{x}{2^{i+1}}\right) = \frac{2}{\pi} \tag{3.6}$$

  definiert.

Anpassungen

Alle aufgelisteten Näherungsverfahren für π ähneln sich stark in ihrer jeweiligen Art der zugrunde liegenden Berechnung. Die Struktur, welche den Programmen zu Grunde liegt, weist ebenfalls große Ähnlichkeiten auf. Daher wurde, um eine aussagekräftigere Auswertung zu erreichen, jede Berechnung noch anhand einiger Punkte angepasst und jeweils als eigenständiges Programm implementiert. Diese Anpassungen sind:

- Berechnung von Zwischenergebnissen (z.B. Nenner und Zähler) und deren Speicherung in Variablen.
- Verzicht auf die Kapselung der eigentlichen Berechnungslogik für die Zahl π in eine eigene Unterfunktion.
- Die gesamte Implementierung in der Hauptfunktion gekapselt ohne jegliche Unterteilung in Subfunktionen.

Um die Anpassungen besser zu verstehen, sind in den Listings 3.3, 3.4 und 3.5 die Veränderungen am Beispiel des Programms zur Approximation nach Euler in der ersten Version (siehe Listing C.1) angegeben.

Listing 3.3: π nach Euler (Berechnung für π nicht in eigener Methode).

```
1  MODULE Euler1V3;
2  VAR t_1, e: REAL;
3  n: INTEGER;
4    PROCEDURE sqrt(a: REAL): REAL;
5    VAR s: INTEGER;
6    BEGIN
7      s := 8;
8      SYSCALL(16, a, s);
9      RETURN a;
10   END sqrt;
11
12 BEGIN
13   n := 1;
14   (* replace of the first euler1 function call *)
15   t_1 := 1 / sqrt(n);
16
17   WHILE TRUE DO
18     n := n + 1;
19     (* replace of the euler1 function call in the main loop *)
20     t_1 := t_1 + (1 / sqrt(n));
21     e := sqrt(t_1 * 6);
22   END
23 END Euler1V3.
```

[1] Viète, F. und van Schooten, F., Francisci Vietae Opera mathematica in vnum volumen congesta ac recognita, Opera atque studio Francisci a Schooten (1646). Zitiert nach [8]

Listing 3.4: π nach Euler (Zwischenergebnisse in Variablen).

```
1 MODULE Euler1V2;
2 VAR t_1, e: REAL;
3 n: INTEGER;
4   PROCEDURE sqrt(a: REAL): REAL;
5   VAR s: INTEGER;
6   BEGIN
7     s := 8;
8     SYSCALL(16, a, s);
9     RETURN a;
10  END sqrt;
11
12  PROCEDURE euler1(glied : REAL) : REAL;
13  (* create variable root *)
14  VAR root : REAL;
15  BEGIN
16    (* save the sqrt result in the root variable *)
17    root := sqrt(glied);
18    (* calculate the approximation step *)
19    RETURN 1 / root;
20  END euler1;
21 BEGIN
22  n := 1;
23  t_1 := euler1(1);
24
25  WHILE TRUE DO
26    n := n + 1;
27    t_1 := t_1 + euler1(n);
28    e := sqrt(t_1 * 6);
29  END
30 END Euler1V2.
```

Listing 3.5: π nach Euler (keine Verwendung von Subfunktionen).

```
1 MODULE Euler1V4;
2 VAR t_1, e, a, s: REAL;
3 n: INTEGER;
4 BEGIN
5   n := 1;
6   s := 8; (* 's' must be a variable for the system call *)
7   a := n;
8   (* replace the wrapped syscall for the sqrt *)
9   SYSCALL(16, a, s);
10  t_1 := 1 / a;
11
12  WHILE TRUE DO
13    n := n + 1;
14    a := n;
15    (* replace the wrapped syscall for the sqrt *)
16    SYSCALL(16, a, s);
17    t_1 := t_1 + 1 / a;
18    a := t_1 * 6;
19    (* replace the wrapped syscall for the sqrt *)
20    SYSCALL(16, a, s);
21    e := a;
22  END
23 END Euler1V4.
```

Tabelle 3.1: Übersicht Approximationen für π.

	Keine Anpassung	Zusatz Variablen	Keine Kapselung	Keine Subfunktionen
Euler1V1	X			
Euler1V2		X		
Euler1V3		X	X	
Euler1V4		X	X	X
Euler2V1	X			
Euler2V2		X		
Euler2V3		X	X	
Euler2V4		X	X	X
LeibnizV1	X			
LeibnizV2		X		
LeibnizV3		X	X	
LeibnizV4		X	X	X
SharpV1	X			
SharpV2		X		
SharpV3		X	X	
SharpV4		X	X	X
VietaV1	X			
VietaV2		X		
WallisV1	X			
WallisV2		X		
WallisV3				X

Aufgrund der rekursiven Implementierung von Vieta (siehe Listing C.5) und der Einfachheit der Berechnung nach Wallis (siehe Listing C.3) konnten für diese beiden Algorithmen nicht alle Änderungen eingebaut werden. Eine Übersicht über die Anpassungen der einzelnen Implementierungen kann daher der Tabelle 3.1 entnommen werden.

3.2 Auswertung

Die Auswertung der generierten Daten gliedert sich in zwei Sektionen zum einen die Betrachtung mittels K-Means mit unterschiedlichen Werten für K und zum anderen eine Betrachtung der Instruktionen anhand von den bereits in Kapitel 2 beschriebenen Metriken.

3.2.1 Clustering

Auch wenn die Clusteranalyse keine direkten Aussagen über das Verhalten eines Programms liefert und die Ergebnisse stark von der Inputmenge, der Wahl von K sowie der verwendeten Ähnlichkeitsfunktion abhängig sind, können die Ergebnisse dennoch als Ausgangsbasis für nähere Untersuchungen genutzt werden. Aufgrund der Abhängigkeit von K und der Ähnlichkeitsfunktion wird für die Auswertung der Wert für K beginnend

Tabelle 3.2: K-Means angewandt an Implementierungen von π. $k = 2$, Ähnlichkeit: euklidische Distanz.

Name	Cluster		Name	Cluster
Euler1V1	0		Euler2V1	0
Euler1V2	0		Euler2V2	0
Euler1V3	0		Euler2V3	0
Euler1V4	1		Euler2V4	1
LeibnizV1	0		SharpV1	0
LeibnizV2	0		SharpV2	0
LeibnizV3	0		SharpV3	0
LeibnizV4	1		SharpV4	1
WallisV1	1		VietaV1	0
WallisV2	0		VietaV2	0
WallisV3	1			

Tabelle 3.3: K-Means angewandt an Implementierungen von π. $k = 2$, Ähnlichkeit: Kosinus.

Name	Cluster		Name	Cluster
Euler1V1	0		Euler2V1	0
Euler1V2	0		Euler2V2	0
Euler1V3	0		Euler2V3	0
Euler1V4	1		Euler2V4	1
LeibnizV1	0		SharpV1	0
LeibnizV2	0		SharpV2	0
LeibnizV3	0		SharpV3	0
LeibnizV4	1		SharpV4	1
WallisV1	1		VietaV1	0
WallisV2	0		VietaV2	0
WallisV3	1			

mit 2 immer um 2 erhöht bis zu einem Maximalwert von 6 und jeweils einmal für die Kosinus-Ähnlichkeit und einmal mittels dem euklidischen Abstand durchgeführt.

Ergebnisse K-Means

1. **$k = 2$, Ähnlichkeit: Euklid** (siehe Tabelle 3.2): Durch die Unterscheidung in nur zwei Cluster können im Normalfall sowie bei einer zu hohen Anzahl keine wirklichen Aussagen getroffen werden. Jedoch ist hier bereits zu beobachten, dass die jeweiligen Implementierungen welche keine Subfunktionen aufweisen, sich stark ähneln und somit einen eigenen Cluster bilden.
2. **$k = 2$, Ähnlichkeit: Kosinus** (siehe Tabelle 3.3): Durch die Änderung der Abstandsmetrik können hier mittels des Kosinus-Abstand, keine anderen Ergebnisse als via der euklidischen Distanz erkannt werden.

Tabelle 3.4: K-Means angewandt an Implementierungen von π. $k = 4$, Ähnlichkeit: euklidische Distanz.

Name	Cluster		Name	Cluster
Euler1V1	0		Euler2V1	0
Euler1V2	0		Euler2V2	0
Euler1V3	0		Euler2V3	0
Euler1V4	2		Euler2V4	2
LeibnizV1	0		SharpV1	0
LeibnizV2	0		SharpV2	0
LeibnizV3	0		SharpV3	0
LeibnizV4	2		SharpV4	2
WallisV1	1		VietaV1	3
WallisV2	1		VietaV2	3
WallisV3	1			

Tabelle 3.5: K-Means angewandt an Implementierungen von π. $k = 4$, Ähnlichkeit: Kosinus.

Name	Cluster		Name	Cluster
Euler1V1	0		Euler2V1	0
Euler1V2	0		Euler2V2	0
Euler1V3	0		Euler2V3	0
Euler1V4	2		Euler2V4	2
LeibnizV1	0		SharpV1	0
LeibnizV2	0		SharpV2	0
LeibnizV3	0		SharpV3	0
LeibnizV4	2		SharpV4	2
WallisV1	1		VietaV1	3
WallisV2	1		VietaV2	3
WallisV3	1			

3. **$k = 4$, Ähnlichkeit: Euklid** (siehe Tabelle 3.4): Auch bei der Unterteilung in 4 Cluster ist zu beobachten, dass die Implementierungen ohne Unterfunktionen bis auf WallisV3 eine eigene Gruppe bilden. Auch zu erkennen ist, dass alle Implementierungen von Wallis nun eine eigene Gruppe ergeben was vermutlich daran liegt, dass keine Systemfunktionen genutzt werden. Auch die beiden Implementierungen von Vieta, welche beide rekursiver Natur sind bilden einen eignen Cluster.
4. **$k = 4$, Ähnlichkeit: Kosinus** (siehe Tabelle 3.5): Auch hier kann wie schon in Punkt 2 kein Unterschied aufgrund der abgeänderten Distanzfunktion festgestellt werden.
5. **$k = 6$, Ähnlichkeit: Euklid** (siehe Tabelle 3.6): Auch hier kann man erkennen, dass die Implementierung ohne Subfunktion scheinbar eine starke Auswirkung auf die Clusterfindung hat, da immer noch drei Programme, die diese Eigenschaft

Tabelle 3.6: K-Means angewandt an Implementierungen von π. $k = 6$, Ähnlichkeit: Euklidischedistanz.

Name	Cluster		Name	Cluster
Euler1V1	0		Euler2V1	0
Euler1V2	0		Euler2V2	0
Euler1V3	0		Euler2V3	0
Euler1V4	4		Euler2V4	2
LeibnizV1	0		SharpV1	0
LeibnizV2	0		SharpV2	0
LeibnizV3	0		SharpV3	0
LeibnizV4	4		SharpV4	4
WallisV1	1		VietaV1	5
WallisV2	1		VietaV2	3
WallisV3	1			

aufweisen, gruppiert werden. Auch die Implementierungen für Wallis bilden noch eine gemeinsame Gruppe.

6. ***k* = 6, Ähnlichkeit: Kosinus** (siehe Tabelle 3.7): Hier ist nun zum ersten mal ein anderes Ergebnis aufgrund der Änderung auf die Kosinus-Ähnlichkeit festzustellen. Im Vergleich zu der Auswertung mittels euklidischer Distanz können hier wie bereits in Punkt 3 wieder die beiden Implementierungen für Leibniz einem Cluster zugeordnet werden. Allerdings bilden nun die Implementierungen für Wallis keine gemeinsame Gruppe. Die restlichen Cluster weisen keine Veränderung auf.

Durch die Clusteranalyse zeichnet sich bereits zum Teil ein Muster für Ähnlichkeiten von Implementierungen ab. Allerdings gibt es immer noch einen großen Bereich von Programmen, die sich im selben Cluster befinden, aber erhebliche Unterschiede in der jeweiligen Umsetzung aufweisen. Um hier noch mehr Klarheit zu erlangen, erfolgt im nächsten Schritt die Analyse via unterschiedlicher Metriken.

3.2.2 Metriken

Die Analyse anhand von Metriken dient dazu, einen genaueren Einblick in die Struktur eines Programms zu gewinnen und soll somit einen Vergleich von unterschiedlichen Lösungen ermöglichen.

Verwendete Metriken

Wie bereits in Kapitel 2 erwähnt, wurden nicht alle Metriken zur Analyse herangezogen. Dies hat vor allem den Hintergrund die Auswertung übersichtlich zu halten. Daher wurden nur die erfolgversprechendsten Metriken ausgewählt und umgesetzt.

Diese Metriken sind:

1. **Average Stack Usage** (siehe Tabelle 3.8): Die Average Stack Usage gibt wie der Name bereits sagt die durchschnittliche Menge an Daten, die am Stack während des betrachteten Zeitraums abgelegt wurden an. Ein hoher Wert in dieser Metrik

Tabelle 3.7: K-Means angewandt an Implementierungen von π. $k = 6$, Ähnlichkeit: Kosinus.

Name	Cluster		Name	Cluster
Euler1V1	0		Euler2V1	0
Euler1V2	0		Euler2V2	0
Euler1V3	0		Euler2V3	0
Euler1V4	5		Euler2V4	2
LeibnizV1	0		SharpV1	0
LeibnizV2	0		SharpV2	0
LeibnizV3	0		SharpV3	0
LeibnizV4	5		SharpV4	5
WallisV1	4		VietaV1	3
WallisV2	4		VietaV2	3
WallisV3	1			

Tabelle 3.8: Analyse nach der Metrik Average Stack Usage.

Name	Wert		Name	Wert
Euler1V1	66.73		Euler2V1	83.41
Euler1V2	62.56		Euler2V2	83.41
Euler1V3	91.00		Euler2V3	83.41
Euler1V4	166.83		Euler2V4	166.83
LeibnizV1	58.88		SharpV1	52.68
LeibnizV2	52.68		SharpV2	50.05
LeibnizV3	77.00		SharpV3	66.73
LeibnizV4	143.00		SharpV4	143.00
WallisV1	100.10		VietaV1	22.24
WallisV2	91.00		VietaV2	18.53
WallisV3	166.83			

lässt zum Beispiel auf die Verwendung einer globalen Lookup-Tabelle schließen. Im betrachteten Fall (siehe Tabelle 3.8) kann man erkennen, dass die jeweiligen Implementierungen ohne Subfunktionen einen höheren Wert als die anderen Programme aufweisen. Dies ergibt sich daraus, dass die gesamte Berechnung und somit auch alle benötigten Variablen in einer Methode gesammelt sind und somit der Stack wie z.B. bei dem Verlassen einer Methode nie aufgeräumt wird. Neben diesen hohen Werten sind auch noch die beiden Implementierungen nach Vieta zu betrachten, welche beide einen vergleichsweise geringen Wert aufweisen. Dies lässt sich am ehesten durch die rekursive Natur dieser Algorithmen und die daraus folgenden Eigenheiten einer solchen Implementierung erklären.

2. **Dependency Distance** (siehe Tabelle 3.9): Die Dependency Distance gibt die durchschnittliche Anzahl an Operationen zwischen einer Lese- und Schreiboperation auf ein und demselben Speicherbereich an. Diese Metrik kann auch als

Tabelle 3.9: Analyse nach der Metrik Dependency Distance.

Name	Wert	Name	Wert
Euler1V1	50.31	Euler2V1	52.61
Euler1V2	47.42	Euler2V2	49.33
Euler1V3	46.21	Euler2V3	36.81
Euler1V4	20.03	Euler2V4	26.59
LeibnizV1	45.84	SharpV1	69.90
LeibnizV2	28.15	SharpV2	50.94
LeibnizV3	40.60	SharpV3	68.39
LeibnizV4	30.17	SharpV4	26.13
WallisV1	33.56	VietaV1	146.55
WallisV2	23.14	VietaV2	38.17
WallisV3	30.16		

Abstandsmaß innerhalb eines Programms verstanden werden. In der Auswertung nach Tabelle 3.9 ist zu erkennen, dass die Abstände eine Abhängigkeit aufgrund der Menge an verwendeten Subfunktionen zu haben scheinen. Dies lässt sich durch den fehlenden Overhead durch einen Funktionsaufruf erklären. Auch wieder auffällig ist die Implementierung VietaV1, die einen sehr hohen Wert in dieser Metrik aufweist. Eine Erklärung hierfür ist wieder die rekursive Implementierung. Dass VietaV2 diese Eigenschaft nicht aufweist kommt daher, dass für diese Implementierung innerhalb der rekursiven Methode nun Berechnungen durchgeführt werden, welche auf sich im Speicher befindliche Variablen zurück greifen.

3. **Instruction Temporal Locality** (siehe Tabelle 3.10): Gibt die durchschnittliche Anzahl an Instruktionen zwischen zwei gleichen Instruktionen an. Dieser Wert sollte insbesondere unter die Lupe genommen werden, wenn er eine extreme Ausprägung aufweist. Da in Tabelle 3.10 nur geringe Schwankungen bei dieser Metrik auszumachen sind, lässt sich hier nichts Relevantes feststellen. Auffällig ist nur, dass wieder eine Abhängigkeit von der Anzahl der implementierten Subfunktionen zu erkennen ist.
4. **Dynamic Basic Block Size** (siehe Tabelle 3.11): Diese Metrik spiegelt die durchschnittliche Größe von Programmsektionen wieder. Wie zu erwarten ist in Tabelle 3.11 kein Wert für die Programme zu finden, welche keine Subfunktionen und somit keine abgeschlossenen Sektionen aufweisen. Auch in dieser Metrik ist vor allem auf Ausreißer zu achten, so weist z.B. VietaV2 im Vergleich zu Euler2V3 einen ca. viermal so hohen Wert auf.
5. **Programmgröße** (siehe Tabelle 3.12): Die Programmgröße dient lediglich als zusätzliche Information. Allerdings kann auch hier, wenn ein Programm einen vergleichsweise hohen oder niedrigen Wert aufweist, eine nähere Untersuchung sinnvoll sein.

Die Betrachtung der Metriken lässt erkennen, dass eine genaue Unterscheidung und Eingrenzung scheinbar nicht möglich ist. Allerdings können gröbere Abweichungen von der Norm festgemacht werden. Insbesondere fällt die rekursive Implementierung von

Tabelle 3.10: Analyse nach der Metrik Instruction Temporal Locality.

Name	Wert		Name	Wert
Euler1V1	27.09		Euler2V1	27.31
Euler1V2	29.14		Euler2V2	28.78
Euler1V3	26.62		Euler2V3	27.63
Euler1V4	19.67		Euler2V4	20.42
LeibnizV1	31.33		SharpV1	40.54
LeibnizV2	34.08		SharpV2	42.68
LeibnizV3	31.39		SharpV3	39.07
LeibnizV4	25.98		SharpV4	33.71
WallisV1	21.14		VietaV1	46.34
WallisV2	21.18		VietaV2	42.92
WallisV3	14.72			

Tabelle 3.11: Analyse nach der Metrik Dynamic Basic Block Size.

Name	Wert		Name	Wert
Euler1V1	16.07		Euler2V1	12.69
Euler1V2	16.40		Euler2V2	14.09
Euler1V3	12.00		Euler2V3	11.65
Euler1V4	—		Euler2V4	—
LeibnizV1	23.00		SharpV1	23.43
LeibnizV2	25.25		SharpV2	25.53
LeibnizV3	12.40		SharpV3	12.40
LeibnizV4	—		SharpV4	—
WallisV1	17.33		VietaV1	49.12
WallisV2	23.91		VietaV2	59.16
WallisV3	—			

Tabelle 3.12: Analyse nach der Metrik Programmgröße.

Name	Wert		Name	Wert
Euler1V1	440.00		Euler2V1	472.00
Euler1V2	476.00		Euler2V2	512.00
Euler1V3	420.00		Euler2V3	436.00
Euler1V4	424.00		Euler2V4	400.00
LeibnizV1	468.00		SharpV1	692.00
LeibnizV2	524.00		SharpV2	748.00
LeibnizV3	500.00		SharpV3	796.00
LeibnizV4	516.00		SharpV4	908.00
WallisV1	340.00		VietaV1	536.00
WallisV2	400.00		VietaV2	540.00
WallisV3	356.00			

Vieta auf. Des Weiteren hat die Average Stack Usage bei den Implementierungen von WallisV1 und WallisV2 einen vergleichsweise hohen Wert.

Ergebnis

Führt man nun die Clusteranalyse und die Auswertungen der Metriken zusammen, können für die unterschiedlichen Implementierungen zur Approximation der Zahl π einige Aussagen getroffen werden. Anhand der Vorsortierung durch den K-Means Algorithmus, welche auf die Erkennung von Ausreißern abzielt, bilden die Implementierungen für Vieta und Wallis je eine eigene Gruppe sowie auch die Implementierungen, welche keine Unterfunktionen verwenden. Die restlichen 12 Algorithmen bilden den letzten und größten Cluster, welcher die breite Masse an Implementierungen darstellt. Diese breite Masse kann nun anhand der Metriken noch feiner sortiert werden, um noch weitere unübliche Eigenschaften festzumachen.

Da die Auswertungen je nach Fokus der analysierenden Person variieren kann, sind die Metriken wertfrei zu betrachten und je nach Anwendungsfall unterschiedlich zu gewichten. Für das hier betrachtete Beispiel der Approximation von π könnte bei einem Fokus auf eine möglichst kleine Strukturgröße gesagt werden, dass die Implementierung nach Euler2V3 mit einer Dynamic Basic Block Size von 11,56 die beste Wahl darstellt. Da aber innerhalb von nur einer Instruktion unterschied vier weitere Implementierungen liegen, ist ein zweiter Blick auf andere Metriken möglicherweise wünschenswert. So weist die Umsetzung nach SharpV3 lediglich eine Average Stack Usage von 66,73 auf und ist somit möglicherweise den anderen Programmen überlegen. Mit diesem Beispiel soll gezeigt werden, dass keine einfache Aussage über die Qualität eines Programms getroffen werden kann.

In Abbildung 3.1 sind zur Übersicht alle Metriken zusammengefasst. Diese Zusammenfassung wurde mithilfe des Analyseprogrammes, welches in Kapitel 2 beschrieben wurde erstellt. Die farblichen Abstufungen von Grün nach Rot dienen zur Erkennung von vergleichsweise hohen und niedrigen werten innerhalb einer Metrik.

Name	Average Stack Usage	Dependency Distance	Instruction Temporal Locality	Dynamic Basic Block Size	Program Size	Cluster
Euler1V1.q.csv	66.73	50.31	27.9	16.07	440.0	0.0
Euler1V2.q.csv	62.56	47.42	29.14	16.4	476.0	0.0
Euler1V3.q.csv	91.0	46.21	26.62	12.0	420.0	0.0
Euler1V4.q.csv	166.83	20.03	19.67		424.0	2.0
Euler2V1.q.csv	83.41	52.61	27.31	12.69	472.0	0.0
Euler2V2.q.csv	83.41	49.33	28.78	14.09	512.0	0.0
Euler2V3.q.csv	83.41	36.81	27.63	11.65	436.0	0.0
Euler2V4.q.csv	166.83	26.59	20.42		400.0	2.0
LeibnizV1.q.csv	58.88	45.84	31.33	23.0	468.0	0.0
LeibnizV2.q.csv	52.68	28.15	34.08	25.25	524.0	0.0
LeibnizV3.q.csv	77.0	40.6	31.39	12.4	500.0	0.0
LeibnizV4.q.csv	143.0	30.17	25.98		516.0	2.0
SharpV1.q.csv	52.68	69.9	40.54	23.43	692.0	0.0
SharpV2.q.csv	50.05	50.94	42.68	25.53	748.0	0.0
SharpV3.q.csv	66.73	68.39	39.07	12.68	796.0	0.0
SharpV4.q.csv	143.0	26.13	33.71		908.0	2.0
VietaV1.q.csv	22.24	146.55	46.34	49.12	536.0	3.0
VietaV2.q.csv	18.53	38.17	42.92	59.61	540.0	3.0
WallisV1.q.csv	100.1	33.56	21.14	17.33	340.0	1.0
WallisV2.q.csv	91.0	23.14	21.18	23.91	400.0	1.0
WallisV3.q.csv	166.83	30.16	14.72		356.0	1.0

Abbildung 3.1: Auswertung der Approximationsalgorithmen für π.

Kapitel 4

Resümee

Als letztes Kapitel soll noch einmal die Arbeit rekapituliert werden und ein Ausblick für zukünftige Arbeiten in diesem Bereich gegeben werden.

4.1 Rückblick

Der Bereich der Programmanalyse umfasst ein weites Spektrum an Betätigungsfeldern. Jeder Programmierer sollte sich zumindest in einem gewissen Ausmaß mit dieser Thematik beschäftigt haben. Insbesondere die Erkenntnis, dass jede Lösung noch verbessert werden kann, sollte man immer im Hinterkopf behalten.

4.1.1 Setup

Das verwendete Setup, welches einem eher akademischen Hintergrund entsprungen ist, kann in seiner Umsetzung nicht eins zu eins auf ein reales Szenario übertragen werden. Da die angewandten Methoden recht allgemein gehalten wurden, sollte eine Anwendung sowie Erweiterung auf ein reales Setup dennoch gut möglich sein. Der große Vorteil und der Grund warum das Q'Wars Setup gewählt wurde, ist die Einfachheit mit welcher hier ein Einblick in die Materie der dynamischen Codeanalyse gewährt wird. Da diese Arbeit auch dazu dienen soll, ohne großen Aufwand eine Analyse auch von Personen durchführen zu lassen, welche sich nicht eingängig mit der Materie beschäftigt haben, bietet ein solch simples Setup natürlich einen Vorteil.

Für den interessierten Leser ist dennoch eine Anwendung auf praxisnaher Ebene wünschenswert. Um eine solche Umsetzung zu ermöglichen, gibt es verschiedenste Tools zur dynamischen Codeanalyse. Bei diesen Tools ist zu beachten, dass die erhobene Datenmenge schnell unüberschaubar werden kann und somit vor der Betrachtung der Daten Einschränkungen getroffen werden müssen. Hierfür würde sich beispielsweise die Hauptkomponenten Analyse [2] anbieten welche versucht, die markantesten Werte in einem Datenset zu fixieren. Diese Technik wurde unter anderem in der Arbeit „Measuring Program Similarity“ [11] angewandt.

Tools, welche sich für eine derartige Analyse eignen, wie sie hier in dieser Arbeit durchgeführt wurde, sind unter anderem:

- **PIN [19]:** ist ein von Intel bereitgestelltes Softwareprodukt zur Aufzeichnung von Instruktionen auf IA-32 und x86-64 Architekturen und bietet durch die bereitgestellte API dem Endnutzer viele Freiheiten für Auswertungen.
- **DynamoRIO [18]:** bietet wie PIN die Möglichkeit zur Erstellung eigener Auswertungen von ausführbaren Programmen. Im Unterschied zu PIN ermöglicht dieses Programm die Auswertung auf IA-32, AMD64, ARM und AArch64 Architekturen.

4.1.2 Auswertung

Wie in Kapitel 3 bereits ausführlich behandelt wurden Implementierungen für die Approximation der Zahl π herangezogen. Zu erkennen war, dass vor allem drei Klassen von Lösungen eine hohe Ähnlichkeit aufzeigten. Zum einen die rekursiven Ansätze zum anderen die Implementierungen nach Wallis, welche keine Systemfunktionen verwenden und allgemein eine einfachere Grundstruktur aufweisen. Auch erkennbar waren die konstruierten Fälle, welche keine Subfunktionen für ihre jeweilige Implementierungen verwendeten. Da der Verzicht auf Unterfunktionen jedoch kaum in der Praxis zu finden ist, ist diese Eigenschaft mehr als Beweis für die Erkennung von extremen Ausprägungen innerhalb eines Datensets zu verstehen.

4.2 Ausblick

Aufgrund der hier gelegten Basis bieten sich nun viele Möglichkeiten, die in dieser Arbeit gewonnenen Erkenntnisse weiter zu verfolgen.

Wie bereits erwähnt ist der Umstieg auf ein anderes Setup, welches für die Auswertung von „realen“ Anwendungen konzipiert wurde, einen Versuch wert. Durch die Änderung des Setups bietet sich auch eine breitere Möglichkeit im Bereich der Analyse nach Metriken. So kann unter anderem beispielsweise die Cache-Miss-Rate ermittelt werden, welche in modernen Rechnerarchitekturen eine gute Aussage über die Performance eines Programms liefert.

Auch bei der Beibehaltung des Q’Wars Setups ergeben sich noch weitere Möglichkeiten. So können die in Kapitel 2 angesprochenen, jedoch nicht umgesetzten Metriken implementiert werden. Insbesondere die Branch Direction könnte hier noch interessante Ergebnisse liefern. Bei den Implementierungen von π, könnte eine Abbruchbedingung nach der erfolgreichen Berechnung von n Stellen eingebaut werden, um einen Wert für die Performance der jeweiligen Approximation zu ermitteln. Auch könnte die Implementierung weiter verändert werden, um spezielles Verhalten zielgerichteter analysieren zu können.

Eine weitere Idee wäre das Gegenüberstellen der π Auswertung mit der Auswertung der Q’Bots um ähnliche strukturelle Muster, zwischen den beiden doch sehr unterschiedlichen algorithmischen Aufgabenstellungen, zu erkennen.

4.3 Schlusswort

Wie diese Arbeit zeigt, ist es möglich, durch die Betrachtung von Programmen während der Laufzeit Gemeinsamkeiten in einzelnen Lösungen zu ermitteln. Eine genaue Festlegung in wieweit sich die Programme ähneln ist jedoch immer noch für jeden Anwendungsfall einzeln zu evaluieren und eine feste Abgrenzung und Definition von Kategorien ist pauschal nicht möglich. Ebenso hat die Auswertung gezeigt, dass eine Erkennung und Unterscheidung von Programmqualität durchaus möglich ist.

Anhang A

Technische Informationen

A.1 Aktuelle Dateiversionen

Datum	Datei
2017/10/12	Ortner_Peter_2017.pdf

A.2 Details zur aktuellen Version

Die Aktuelle Version 1.2.2 dieser Arbeit ist die vollständige Abfassung mit zusätzlich eingearbeiteten Formatierungswünschen.

A.2.1 Allgemeine technische Voraussetzungen

Für die korrekte Ausführung aller beigelegten Programme sind folgende Voraussetzungen mindestens zu erfüllen:

- Java Runtime Environment in der Version 8 oder neuer,
- Windows 10 64 Bit.

Anhang B

Inhalt der CD-ROM/DVD

Format: CD-ROM, Single Layer, ISO9660-Format

B.1 Dokumente

Pfad: /Dokumentation/

Ortner_Peter_2017.pdf Diese Arbeit im PDF Format.

SystemDoku/ . . . . . . Enthält eine allgemeine Dokumentation über das verwendete Setup.

B.2 Ausführbare Dateien

Pfad: /Programme/MetrikenAnalyse

User Dokumentation/ . Enthält die Anleitung zur Verwendung der Metriken Analyse.

EvalBotCsvs.jar . . . . . Ausführbares Java-Programm zur Analyse von aufgezeichneten Instruktionen via Metriken.

Pfad: /Programme/QWarsSimulator

qwars-simulator.cfg . . Konfigurationsdatei für den Q'Wars Simulator.

simulator-1.2.3.jar . . . Ausführbarer Q'Wars Simulator in der Version 1.2.3.

Pfad: /Programme/Visualisierung

htdocs/ . . . . Enthält die in d3.js[1] geschriebene Implementierung für die Lochkartenvisualisierung.

ServerLocalhost.exe . . Eigenständig ausführbarer HTTP Server[2] für Windows.

[1] https://d3js.org/

[2] http://miniweb.sourceforge.net/

B.3 Implementierung

Pfad: /Implementierung/

- Clustering/ . . . . Enthält die Implementierung des K-Mean-Algorithmus
- Datenaufzeichnung/ . . Enthält die modifizierte LoggingCPU für das Q'Wars Setup, sowie die Serialisierung der CSV-Dateien.
- Metriken Analyse/ . . . Enthält die Implementierung des Tools zur Metriken Analyse.
- Visualisierung/ . . . . . Enthält eine unter Linux lauffähige HTTP Server Implementierung für die in d3.js geschriebene Lochkartenvisualisierung.

B.4 Auswertung

Pfad: /Auswertung/

- BeschreibungInstruktionAufteilung/ Enthält mögliche Aufteilungen der Instruktionen für die Clusteranalyse.
- ExampleBots/ . . . . . Enthält die unterschiedlichen Implementierungen für die Q'Bots sowie die generierten Log-Dateien für diese.
- PICalc/ . . . . Enthält die unterschiedlichen Implementierungen für die Zahl PI sowie die generierten Log-Dateien für diese.

Anhang C

Approximationen von π

Die hier angeführten Implementierungen von Approximationsverfahren zur Berechnung der Zahl π, sind für OberonQ adaptierte Versionen nach den Ausarbeitungen von Werner Scholz [21].

C.1 Leonhard Euler Version 1

Listing C.1: π nach Euler Version 1 (vgl. Formel 3.1).

```
1 MODULE Euler1V1;
2 VAR t_1, e: REAL;
3 n: INTEGER;
4
5   (* wrapper for syscall to calculate sqrt *)
6   PROCEDURE sqrt(a: REAL): REAL;
7   VAR s: INTEGER;
8   BEGIN
9     s := 8;
10    SYSCALL(16, a, s);
11    RETURN a;
12  END sqrt;
13
14  (* main calculation for the euler approximation *)
15  PROCEDURE euler1(glied : REAL) : REAL;
16  BEGIN
17    RETURN 1 / sqrt(glied);
18  END euler1;
19
20 BEGIN
21  n := 1;
22  t_1 := euler1(1);
23
24  (* main loop *)
25  WHILE TRUE DO
26    n := n + 1;
27    t_1 := t_1 + euler1(n);
28    e := sqrt(t_1 * 6);
29  END
30 END Euler1V1.
```

C.2 Leonhard Euler Version 2

Listing C.2: π nach Euler Version 2 (vgl. Formel 3.2).

```
1 MODULE Euler2V1;
2 VAR t_1, e: REAL;
3 n: INTEGER;
4
5   PROCEDURE sqrt(a: REAL): REAL;
6   VAR s: INTEGER;
7   BEGIN
8     s := 8;
9     SYSCALL(16, a, s);
10    RETURN a;
11  END sqrt;
12
13  PROCEDURE euler2(glied : REAL) : REAL;
14  BEGIN
15    RETURN 1 / glied / glied / glied / glied;
16  END euler2;
17
18 BEGIN
19   n := 1;
20   t_1 := euler2(1);
21
22   WHILE TRUE DO
23     n := n + 1;
24     t_1 := t_1 + euler2(n);
25     e := sqrt(sqrt(t_1 * 90));
26   END
27 END Euler2V1.
```

C.3 John Wallis

Listing C.3: π nach Wallis (vgl. Formel 3.3).

```
1 MODULE WallisV1;
2 VAR t_1, e: REAL;
3 n: INTEGER;
4
5   PROCEDURE wallis(glied : INTEGER) : REAL;
6   BEGIN
7     RETURN (glied + glied % 2) / (glied + (glied + 1) % 2);
8   END wallis;
9
10 BEGIN
11   n := 1;
12   t_1 := wallis(1);
13
14   WHILE TRUE DO
15     n := n + 1;
16     t_1 := t_1 * wallis(n);
17     e := t_1 * 2;
18   END
19 END WallisV1.
```

C.4 Gottfried Wilhelm Leibniz

Listing C.4: π nach Leibniz (vgl. Formel 3.5).

```
1 MODULE LeibnizV1;
2 VAR t_1, e: REAL; n: INTEGER;
3   PROCEDURE potenz*(a, b: REAL): REAL;
4   VAR s: INTEGER;
5   BEGIN
6     s := 9;
7     SYSCALL(16, a, b, s);
8     RETURN a;
9   END potenz;
10  PROCEDURE leibniz(glied : INTEGER) : REAL;
11  BEGIN
12    RETURN potenz(-1, (glied + 1) % 2) / (2 * glied - 1);
13  END leibniz;
14 BEGIN
15  n := 1;
16  t_1 := leibniz(1);
17  WHILE TRUE DO
18    n := n + 1;
19    t_1 := t_1 + leibniz(n);
20    e := t_1 * 4;
21  END
22 END LeibnizV1.
```

C.5 François Viète

Listing C.5: π nach Viète (vgl. Formel 3.6).

```
1 MODULE VietaV1;
2 VAR t_1, e: REAL; n: INTEGER;
3   PROCEDURE sqrt(a: REAL): REAL;
4   VAR s: INTEGER;
5   BEGIN
6     s := 8;
7     SYSCALL(16, a, s);
8     RETURN a;
9   END sqrt;
10
11  PROCEDURE vieta(glied : INTEGER) : REAL;
12  BEGIN
13    IF glied = 1 THEN RETURN sqrt(0.5);
14    ELSE RETURN sqrt(0.5 + 0.5 * vieta(glied-1));
15    END
16  END vieta;
17 BEGIN
18  n := 1;
19  t_1 := vieta(1);
20  WHILE TRUE DO
21    n := n + 1;
22    t_1 := t_1 * vieta(n);
23    e := 2 / t_1;
24  END
25 END VietaV1.
```

C.6 Abraham Sharp

Listing C.6: π nach Sharp (vgl. Formel 3.4).

```
1 MODULE SharpV1;
2 VAR t_1, e: REAL;
3 n: INTEGER;
4
5   (* wrapper for the syscall of the power function *)
6   PROCEDURE potenz*(a, b: REAL): REAL;
7   VAR s: INTEGER;
8   BEGIN
9     s := 9;
10    SYSCALL(16, a, b, s);
11    RETURN a;
12  END potenz;
13
14  PROCEDURE sqrt(a: REAL): REAL;
15  VAR s: INTEGER;
16  BEGIN
17    s := 8;
18    SYSCALL(16, a, s);
19    RETURN a;
20  END sqrt;
21
22  PROCEDURE sharp(glied : INTEGER) : REAL;
23  BEGIN
24    RETURN potenz(-1, (glied + 1) % 2) / ((2 * glied - 1) * potenz(3, glied-1));
25  END sharp;
26
27 BEGIN
28  n := 1;
29  t_1 := sharp(1);
30
31  WHILE TRUE DO
32    n := n + 1;
33    t_1 := t_1 + sharp(n);
34    e := 2 * sqrt(3) * t_1;
35  END
36
37 END SharpV1.
```

Quellenverzeichnis

Literatur

[1] Dario Castellanos. „The Ubiquitous π“. *Mathematics Magazine* 61.2 (1988), S. 67–98 (siehe S. 25).

[2] G. H. Dunteman. *Principal Components Analysis*. Sage Publications, 1989 (siehe S. 37).

[3] Anil K. Jain und Richard C. Dubes. *Algorithms for Clustering Data*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988 (siehe S. 14).

[4] L. Kaufman und Peter J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Hoboken, New Jersey: John Wiley, 1990 (siehe S. 14).

[5] Ho Young Kim und Tag Gon Kim. „Performance simulation modeling for fast evaluation of pipelined scalar processor by evaluation reuse“. In: *Proceedings. 42nd Design Automation Conference, 2005.* Juni 2005, S. 341–344 (siehe S. 19).

[6] Donald E. Knuth. „Big Omicron and Big Omega and Big Theta“. *SIGACT News* 8.2 (Apr. 1976), S. 18–24 (siehe S. 6).

[7] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998 (siehe S. 5).

[8] Aaron Levin. „A New Class of Infinite Products Generalizing Viète’s Product Formula for π“. *The Ramanujan Journal* 10.3 (Dez. 2005) (siehe S. 26).

[9] Robert C. Martin. *Clean Code – A Handbook of Agile Software Craftsmanship*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2009 (siehe S. 19).

[10] Derek B. Noonburg und John Paul Shen. „A Framework for Statistical Modeling of Superscalar Processor Performance“. In: *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*. HPCA ’97. Washington, DC, USA: IEEE Computer Society, 1997 (siehe S. 19).

[11] A. Phansalkar u. a. „Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites“. In: *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*. ISPASS ’05. Washington, DC, USA: IEEE Computer Society, 2005, S. 10–20 (siehe S. 19, 37).

[12] Feng Qin, Shan Lu und Yuanyuan Zhou. „SafeMem: exploiting ECC-memory for detecting memory leaks and memory corruption during production runs“. In: *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. Feb. 2005, S. 291–302 (siehe S. 5).

[13] K. H. Schmidt. *Pi Geschichte und Algorithmen Einer Zahl*. Books on Demand, 2001 (siehe S. 25).

[14] Y. Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. Network Working Group, RFC 4180. RFC Editor, Okt. 2005. URL: http://www.rfc-editor.org/rfc/rfc4180.txt (siehe S. 13).

[15] Johan Wästlund. „An Elementary Proof of the Wallis Product Formula for pi“. *The American Mathematical Monthly* 114.10 (2007), S. 914–917 (siehe S. 25).

[16] C. Willems, T. Holz und F. Freiling. „Toward Automated Dynamic Malware Analysis Using CWSandbox“. *IEEE Security Privacy* 5.2 (März 2007), S. 32–39 (siehe S. 5).

[17] Niklaus Wirth und Jürg Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1992 (siehe S. 8).

Online-Quellen

[18] DynamoRIO. *DynamoRIO Dynamic Instrumentation Tool Platform*. 2009. URL: http://www.dynamorio.org/ (besucht am 14. 09. 2017) (siehe S. 38).

[19] Intel. *Pin - A Dynamic Binary Instrumentation Tool*. 2016. URL: www.intel.com/software/pintool (besucht am 14. 09. 2017) (siehe S. 38).

[20] Eric Rowell. *Big-O Algorithm Complexity Cheat Sheet*. 2013. URL: http://www.bigocheatsheet.com/ (besucht am 06. 09. 2017) (siehe S. 6).

[21] Werner Scholz. *Die Geschichte der Approximationen der Zahl Pi*. 2001. URL: http://www.cwscholz.net/projects/fba/ (besucht am 11. 09. 2017) (siehe S. 43).