Fachhochschule Nordwestschweiz
Hochschule für Technik

MASTER OF SCIENCE
IN ENGINEERING

FHNW Brugg/Windisch

Projektbericht P7

AutoFeedback – Performance

Student	Joel Emmenegger joel.emmenegger@fhnw.ch
Advisor	Prof. Dr. Christoph Stamm christoph.stamm@fhnw.ch

Emmenegger Joel
5.2.2021

SUMMARY

Die Arbeit beschäftigt sich mit der automatischen Evaluation studentischer Java-Programmierlösungen bezüglich Performance. Dafür soll ein Evaluationstool erstellt werden, welches die asymptotische Zeitkomplexität von verschiedenen studentischen Java-Programmierlösungen messen kann.

Um das Evaluationstool testen zu können, wurden in einem ersten Schritt Beispielprobleme für studentische Java-Programmierlösungen erstellt. Dafür wurden verschiedene Lösungsansätze für die sechs Probleme implementiert.

Das Perfomance-Messtool analysiert die asymptotische Zeitkomplexität eines Algorithmus. Damit das Messtool den Algorithmus richtig verwenden kann, wird ein Performancetest benötigt, welcher das Vorbereiten und das Aufrufen des Algorithmus übernimmt. Damit kann die Ausführungszeit eines Algorithmus mit unterschiedlich grossen Inputdaten gemessen werden. Aus den Resultaten wird mit Hilfe einer linearen Regression eine Zeitkomplexität für den Algorithmus ermittelt.

Die Tests des Performance-Messtools mit den Beispielproblemen ergaben bei fünf von sechs Problemen konstante und korrekte Resultate. Beim sechsten Problem kam es teilweise zu inkonsistenten und faschen Resultaten. Das Resultat lag in jedem Fall maximal bei der nächstkleineren bzw. grösseren Zeitkomplexität.

Um das erstellte Tool benutzbar zu machen, wurde dieses in das abgeschlossene Lehrfonds-Projekt «Automatisches Feedback zu Programmieraufgaben» des Instituts für Mobile und Verteilte Systeme integriert.

INHALTSVERZEICHNIS

1. EINLEITUNG

Für Programmieranfänger ist es wichtig Programmieraufgaben selbständig lösen zu können und für ihre Lösungen ein sinnvolles und individuelles Feedback zu erhalten. Im Studiengang Informatik an der Hochschule für Technik fallen ins erste Studienjahr Module, in welchen die Grundlagen der Programmierung mit Java vermittelt werden. Das Betreuungsintensive aber besonders Wertvolle an diesen Modulen ist, das Begutachten der Lösungen von Programmieraufgaben, welche die Studierenden im Rahmen des begleiteten Selbststudiums erarbeitet haben. Die Beurteilung beinhaltet Hinweise zu falschen, verbesserungswürdigen und guten Teilen in ihren Programmen. Diese Beurteilung sollte den Studierenden möglichst zeitnah zur Verfügung gestellt werden. Um bei wachsender Anzahl Studierender die personellen Ressourcen für die enge Betreuung nicht zu überbeanspruchen, wird ein automatisches Feedbacksystem benötigt.

Im abgeschlossenen Lehrfonds-Projekt «Automatisches Feedback zu Programmieraufgaben» des Instituts für Mobile und Verteilte Systeme sind bereits erste Teile eines automatischen Feedbacksystems umgesetzt worden. Daraus resultierte eine Webapplikation, welche die Korrektheit von Studentenprogrammen serverseitig anhand von vordefinierten Unit Tests überprüft.

1.1. AUTOFEEDBACK PROJEKT

1.1.1. AUFBAU DER WEBAPPLIKATION

Es gibt zwei verschiedene Benutzertypen: «Dozenten» und «Studenten». Mit einem Dozentenaccount können neue Aufgaben «Tasks» erstellt werden. Ein Task beinhaltet einen Titel, eine Beschreibung, eine Zip-Datei mit den Unit Tests und eine Liste von Klassennamen, welche beim Testen ausgeführt werden sollen. Um das automatische Testen von Studentenlösungen zu ermöglichen definiert der Dozent Interfaces, welchen die Studentenlösungen entsprechen müssen.

Um das Benutzen der Applikation für die Studenten zu vereinfachen, werden die Tasks in «Modulen» zusammengefasst. Dabei ist ein Modul eine sortierte Sammlung von Tasks, die beispielsweise während eines Semesters von den Studenten gelöst werden sollen. Ein Task kann dabei in verschiedenen Modulen enthalten sein. Die Zugriffsrechte auf Tasks sind auf der Ebene der Module geregelt, damit Studenten nicht zu jedem Task separat eingeladen werden müssen.

Wenn ein Student seine Lösung zu einer Aufgabe testen möchte, wählt er den dazugehörigen Task aus und lädt seine Java-Klassen als Zip-Datei hoch. Anschliessend wird der Code kompiliert und evaluiert. Nach der Evaluation werden die Resultate dem Studenten mit einem Feedback im Task angezeigt.

1.1.2. EVALUATION DER PROGRAMME

Die Evaluation einer Studentenlösung auf dem Server läuft wie folgt ab: Sobald eine noch nicht evaluierte Studentenlösung auf dem Server verfügbar ist, wird diese von einem Hintergrundprozess bearbeitet. Dabei werden in einem ersten Schritt sowohl die Java-Klassen der Unit Tests als auch der Lösungen in einem Docker-Container kompiliert. Anschliessend wird eine Java-Applikation ausgeführt, welche mittels Reflexion die Tests des Dozenten und die Studentenlösung lädt. Die Tests werden ausgeführt und die Resultate inklusive Feedbacks in eine Datenbank geschrieben. Das Feedback kann anschliessend von den Studenten in der Webapplikation eingesehen werden.

1.2. Ziele

Die Hauptaufgabe der Arbeit besteht in der automatischen Evaluation studentischer Java-Programmierlösungen bezüglich Performance und die Integration in die AutoFeedback-Applikation. Dies umfasst die Beantwortung folgender Fragen:

1. Welche Tools zur Evaluation der Performance von Java-Programmcode sind bereits verfügbar? Was leisten sie? Wo sind ihre Grenzen?
2. Wie kann die Performance von Java-Programmen aussagekräftig gemessen werden?
3. Wie kann die asymptotische Zeitkomplexität anhand von Messungen automatisch ermittelt werden?
4. Wie können anhand einer Musterlösung für eine Programmieraufgabe automatisch Performance-Tests erstellt werden?

1.3. Aufbau des Berichts

Im folgenden Bericht werden im Kapitel 2 bestehende Tools zur Evaluation der Performance von Java-Programmcodes analysiert. Anschliessend werden im Kapitel 3 verschiedene Beispielprobleme definiert und beschrieben. Diese dienen zur Bewertung der entwickelten Performanceevaluation und werden teilweise auch für die Analyse der bestehenden Tools eingesetzt. Im Kapitel 4 wird die entwickelte Performanceevaluation erläutert und anhand der Beispielprobleme getestet. Im Kapitel 5 werden alle entwickelten und geänderten Projekte beschrieben. Dies beinhaltet einerseits, wie die Performanceevaluation in das abgeschlossene Lehrfonds-Projekt «Automatisches Feedback zu Programmieraufgaben» integriert wird und andererseits wird gezeigt wie Dozenten und Studenten die Applikation verwenden können, sowie welche Anforderungen an ihre Projekte gestellt werden. Der Bericht wird mit der Diskussion im Kapitel 6 abgeschlossen.

2. Analyse bestehender Tools

Die Abschätzung der Performance von Java Programmen wird bereits von verschiedenen vorhandenen Tools gemacht. In diesem Kapitel wird für verschiedene Tools geprüft, ob eine Verwendung für dieses Projekt vorteilhaft sein könnte oder ob gar schon ganze Teile dieses Projekts übernommen werden.

2.1. Kriterienliste

Nachfolgend sind die Kriterien zur Evaluation von bestehenden Tools zur Messung der Performance von Java Programmen beschrieben. Das Verhalten und die Resultate der bestehenden Tools werden anhand des Beispielproblems *Quicksort* (sehe Kapitel 3.2) evaluiert.

Was	Mögliche Punkte
Lizenz	Ist es möglich dieses Tool gratis zu testen und zu verwenden?
Aktualität	- Für welche Java-Version ist es entwickelt? - Wann wurde es das letzte Mal verbessert (Bugfixes und neue Funktionen)?
Liegt Code zu diesem Tool vor	- Ist eine Version zum Testen vorhanden? - Kann der Code eingesehen werden (GitHub Repository)?
Dokumentation	- Ist eine Dokumentation vorhanden? - Wie einfach kann das Tool mit Dokumentation verwendet werden?
Performance	Zeit bis ein Resultat vorliegt.
Reproduzierbarkeit der Resultate	Wie konstant sind die Resultate?
Resultate Qualität	- Wird die Ausführungszeit gemessen? - Wird eine asymptotische Zeitkomplexität abgeschätzt oder berechnet?
Automatisierung	- Können Tests nachträglich und einfach in ein bestehendes Programm integriert werden? - Können Tests anhand einer Musterlösung (automatisch oder manuell) erstellt werden für den Einsatz mit Studentenlösungen?
Integration	Wie gross ist der Aufwand für eine Integration in die bestehende AutoFeedback-Applikation?
Erweiterbarkeit	Kann das Tool auf die Bedürfnisse des AutoFeedback Projektes angepasst werden?

Tabelle 1: Kriterienliste

2.2. JMH (Java Microbenchmark Harness)

JMH ist ein Tool, um Java Projekte auszuführen und zum Analysieren von Nano- / Mikro- / Milli- und Makro-Benchmarks auf der JVM [1].

Was	Mögliche Punkte
Lizenz	GNU General Public License (GPL)
Aktualität	Die neuste JMH Version wurde im Oktober 2020 veröffentlicht und funktioniert mit Java OpenJDK 14.
Liegt Code zu diesem Tool vor	Es sind Testbeispiele vorhanden und der Code kann im GitHub Repositorien eingesehen werden.
Dokumentation	Es ist eine Javadoc Dokumentation vorhanden [2]. Das Benutzen mit Dokumentation ist jedoch nicht immer logisch und durch viele Möglichkeiten kompliziert.
Performance	Die Dauer einer Performancemessung kann eingestellt werden. Im Allgemeinen dauern Messungen jedoch relativ lange. Das Tool versucht sämtliche Einflüsse der JVM auf die Performance möglichst gut zu reduzieren. Wenn eine hohe Genauigkeit gewünscht ist, dauert die Analyse eines Algorithmus mehrheitlich über eine Minute. Zum Beispiel dauerte die Evaluation eines Quicksort Algorithmus, welcher 10`000 Elemente sortieren sollte, über drei Minuten. Dabei wurde eine durchschnittliche Ausführungszeit von 55.3 ms mit +/- 3 ms Abweichung gemessen.
Reproduzierbarkeit der Resultate	Die Resultate sind bei mehrfachen Ausführungen sehr ähnlich. Das Tool hat eine sehr gute Präzision.
Resultate Qualität	Die durchschnittliche Zeit pro Operation wird gemessen, es werden jedoch keine Abschätzungen zur asymptotischen Zeitkomplexität vorgenommen.
Automatisierung	Es wird standardmässig Maven als Building Tool verwendet. Die Resultate können aus einer Resultatklasse ausgelesen werden und die Zeitkomplexität kann anschliessend anhand der Messungen berechnet werden. Tests können mit Hilfe einer Musterlösung einfach manuell erstellt werden. Die automatische Generierung ist schwieriger vorzunehmen.
Integration	Die bestehende AutoFeedback Applikation verwendet Gradle und nicht Maven als Buildtool. Eine Integration sollte jedoch mit etwas Mehraufwand möglich sein.
Erweiterbarkeit	Nicht erweiterbar, weil die Lizenz es nicht erlaubt.

TABELLE 2: KRITERIENBEWERTUNG JMH

Fazit:

Das Tool ist sehr gut geeignet, um genaue Zeitmessungen vorzunehmen, diese dauern jedoch lange. Falls die Resultate mit weniger zeitaufwendigen Messungen nicht schlecht sind, lohnt sich der Aufwand für die Verwendung dieses Tools nicht. Zumal genauere Messungen nicht unbedingt zu einer genaueren Abschätzung der Zeitkomplexität führen.

2.3. CALIPER

Caliper ist ein Tool, um die Performance von Java Code zu messen, mit primärem Fokus auf Microbenchmarks. Der Release der neusten Version war 2015. Diese funktionierte mit der aktuellen Java Version nicht mehr einwandfrei. Das Tool ist nicht in der Lage die asymptotische Zeitkomplexität von Programmen zu evaluieren, sondern nur um Microbenchmark-Zeitmessungen vorzunehmen [3].

Was	Mögliche Punkte
Lizenz	Das Ändern und Benutzen der Software ist erlaubt unter der Apache License 2.0 Lizenz
Aktualität	Die neuste Version 1.0-beta-2 wurde im März 2015 veröffentlicht. Diese Version funktioniert mit der aktuellen Java Version nicht mehr einwandfrei und ist damit ungeeignet für diese Projekt.
Liegt Code zu diesem Tool vor	Das GitHub Repository von Caliper ist zugänglich.
Dokumentation	Wurde nicht analysiert.
Performance	Wurde nicht analysiert.
Reproduzierbarkeit der Resultate	Wurde nicht analysiert.
Resultate Qualität	Das Tool ist nicht in der Lage die asymptotische Zeitkomplexität von Programmen zu evaluieren, sondern nur um Microbenchmark-Zeitmessungen vorzunehmen.
Automatisierung	Wurde nicht analysiert.
Integration	Wurde nicht analysiert.
Erweiterbarkeit	Das Ändern ist unter Einhaltung der Lizenzen möglich.

TABELLE 3: KRITERIENBEWERTUNG CALIPER

Fazit:
Das Tool wird als untauglich erachtet und deshalb nicht weiter analysiert.

2.4. CONTIPERF

Mit ContiPerf können in Java Performancetests erstellt werden, welche Schwächen im Code schnell und einfach aufdecken. Die Performancetests werden von Entwicklern erstellt und sind eine Erweiterung zu den JUnit 4 Test von Java [4].

Was	Mögliche Punkte
Lizenz	Das Ändern und Benutzen der Software ist erlaubt unter einer der folgenden Lizenzen: Apache License 2.0, Lesser GNU Public License (LGPL) 3.0, Eclipse Public License 1.0 oder BSD License
Aktualität	Die neuste Version 2.4.3 von ContiPerf ist vom Juni 2019 und hat eine Mindestanforderung von Java 5 und JUnit 4.7.
Liegt Code zu diesem Tool vor	Das GitHub Repository von ContiPerf ist zugänglich.

Was	Mögliche Punkte
Dokumentation	Wurde nicht analysiert.
Performance	Wurde nicht analysiert.
Reproduzierbarkeit der Resultate	Wurde nicht analysiert.
Resultate Qualität	Die Resultate werden als HTML-Dateien für eine online Anzeige zusammengestellt. Diese Form der Resultate ist für den Zweck dieser Arbeit ungeeignet.
Automatisierung	Wurde nicht analysiert.
Integration	Wurde nicht analysiert.
Erweiterbarkeit	Das Ändern ist unter Einhaltung der Lizenzen möglich.

TABELLE 4: KRITERIENBEWERTUNG CONTIPERF

Fazit:
Die Resultate die ContiPerf als HTML-Dateien liefert, sind für den Zweck dieser Arbeit ungeeignet. Die Funktionalität bietet ausserdem keine Analyse der asymptotischen Zeitkomplexität. Das bestehende AutoFeedback Projekt setzt bei Unit Test auf die neueren JUnit 5 Tests und nicht wie ContiPerf auf JUnit 4. Einige Punkte wurden nicht analysiert, weil von einer Verwendung dieses Tools abgesehen wurde.

2.5. JUNITPERF

JUnitPerf ist ein API Performancetest Framework, welches das JUnit4 Framework erweitert. Die Ersteller des Frameworks raten dem Benutzer für das Benchmarken von Code-Blöcken mit wenig Latenz JMH (siehe Kapitel 2.2) zu verwenden [5].

Was	Mögliche Punkte
Lizenz	Das Ändern und Benutzen der Software ist erlaubt unter der Apache License 2.0 Lizenz
Aktualität	Die neuste Version 1.16.1 wurde im September 2020 veröffentlicht.
Liegt Code zu diesem Tool vor	Das GitHub Repository von JUnitPerf ist zugänglich.
Dokumentation	Eine Dokumentation ist im GitHub Repository vorhanden und empfiehlt für das Benchmarken von Code-Blöcken JMH. Deshalb wurde dieses Framework nicht weiter evaluiert.
Performance	Wurde nicht analysiert.
Reproduzierbarkeit der Resultate	Wurde nicht analysiert.
Resultate Qualität	Wurde nicht analysiert.
Automatisierung	Wurde nicht analysiert.
Integration	Wurde nicht analysiert.

Was	Mögliche Punkte
Erweiterbarkeit	Das Ändern ist unter Einhaltung der Lizenzen möglich.

TABELLE 5: KRITERIENBEWERTUNG JUNITPERF

Fazit:
Für AutoFeedback wäre das Benchmarken von Code-Blöcken sinnvoll, dafür wird jedoch von den Herstellern dieses Tools JMH empfohlen. Deshalb wurde auch dieses Framework nicht weiter evaluiert.

2.6. PERFORMANCE AWARENESS IN SOFTWARE DEVELOPMENT

Performance Awareness in Software Development ist ein Projekt der tschechischen Charles Universität. In diesem Projekt wird versucht die Software Performance bereits während der Implementierungsphase in Projekten zu messen und die Entwickler damit zu unterstützen. Dafür wurden Performance Unit Tests entwickelt, welche die Performance von einem Programm nach Änderungen im Code validieren können. Das Projekt ist jedoch nicht darauf ausgelegt, eine asymptotische Zeitkomplexität zu bestimmen, sondern dem Entwickler Zeitmessungen zur Verfügung zu stellen. Es wurden verschiedene Projekte dafür entwickelt, unter anderem das Stochastice Performance Logic (SPL) und Microbenchmarking Agent for Java. Für diese zwei wurde geprüft, ob sie für diese Arbeit verwendet werden können. Die meisten Papers und Tools wurden 2012 und 2013 entwickelt und anschliessend noch einige Jahre gewartet und weiterentwickelt. Das Software Projekt ist dokumentiert und auch in verschiedenen wissenschaftlichen Arbeiten beschrieben. Die Dokumentation zu den einzelnen Projekten ist unvollständig und es ist kein aktuelles Projekt vorhanden [6].

Stochastice Performance Logic (SPL) wurde entwickelt, um Performance Abschätzungen zu machen. Dabei sollte die Performance von verschiedenen Versionen eines Programms verglichen werden. Die neuste Version von SPL ist drei Jahre alt und verlangt mindestens Java 8. Eine Dokumentation zu dem Programm ist nur rudimentär vorhanden und die aktuellste Version funktioniert auf meinem Rechner nicht. Das Tool verwendet im Hintergrund JMH für das Erstellen der Benchmarks. Weil dieses Tool keine asymptotische Zeitkomplexität abschätzen kann, veraltet und schlecht dokumentiert ist, wird von einer Nutzung abgesehen [7].

Was	Mögliche Punkte
Lizenz	Das Ändern und Benutzen der Software ist erlaubt unter der Apache License 2.0 Lizenz
Aktualität	Die neuste Version 1.0.4 wurde im Mai 2018 veröffentlicht.
Liegt Code zu diesem Tool vor	Das GitHub Repository von Stochastice Performance Logic ist zugänglich.
Dokumentation	Eine rudimentäre Dokumentation ist auf GitHub vorhanden. Die verlinkten Webseiten für weitere Informationen sind jedoch nicht mehr vorhanden.
Performance	Wurde nicht analysiert.
Reproduzierbarkeit der Resultate	Wurde nicht analysiert.
Resultate Qualität	Das Tool ist nicht in der Lage die asymptotische Zeitkomplexität von Programmen zu evaluieren.

Automatisierung	Wurde nicht analysiert.
Integration	Wurde nicht analysiert.
Erweiterbarkeit	Das Ändern ist unter Einhaltung der Lizenzen möglich.

TABELLE 6: KRITERIENBEWERTUNG SPL

Mit Hilfe des Projekts Java Microbenchmarking Agent for Java können JVM Events wie GC oder JIT aufgezeichnet werden und Performance Zähler und genaue Zeitmessungen aus dem JNI herausgelesen werden. Die aktuellste Version dieses Tools ist ebenfalls drei Jahre alt [8].

Was	Mögliche Punkte
Lizenz	Das Ändern und Benutzen der Software ist erlaubt unter der Apache License 2.0 Lizenz
Aktualität	Die neuste Version 1.0.0 wurde im November 2018 veröffentlicht.
Liegt Code zu diesem Tool vor	Das GitHub Repository von Stochastice Java Microbenchmarking Agent for Java ist zugänglich.
Dokumentation	Eine rudimentäre Dokumentation ist auf GitHub vorhanden.
Performance	Wurde nicht analysiert.
Reproduzierbarkeit der Resultate	Wurde nicht analysiert.
Resultate Qualität	Das Tool ist nicht in der Lage die asymptotische Zeitkomplexität von Programmen zu evaluieren.
Automatisierung	Wurde nicht analysiert.
Integration	Wurde nicht analysiert.
Erweiterbarkeit	Das Ändern ist unter Einhaltung der Lizenzen möglich.

TABELLE 7: KRITERIENBEWERTUNG JAVA MICROBENCHMARKING AGENT FOR JAVA

Fazit:
Weil dieses beiden Tools keine asymptotische Zeitkomplexität abschätzen können und veraltet sind, wurden auch diese Tools nicht weiterverfolgt bzw. wurde von einer Nutzung abgesehen.

2.7. FAZIT DER TOOLANALYSE

Keines der geprüften Tools kann eine Abschätzung zur Zeitkomplexität abgeben. Die meisten Tools messen die Ausführungszeiten von einzelnen Methoden oder ganzen Programmen. Nur bei JMH war auch eine aktuelle Version vorhanden. Alle Projekte sind kompliziert und mit oder teilweise ohne Dokumentation schwer zu verstehen und zu erweitern. Der zeitliche Aufwand für die Verwendung eines dieser Tools ist sehr gross, da viel Zeit für das Verstehen des Tools, sowie in die Integration in AutoFeedback investiert werden müsste. Zudem kann kein Tool Abschätzungen zur Zeitkomplexität machen. Aus diesen Gründen wurde die Verwendung eines dieser Tools verworfen.

Option: Falls die Genauigkeit der Zeitmessung nicht ausreichend ist, könnte JMH für das Messen der Zeit eingesetzt werden.

3. BEISPIELPROBLEME

Für die Überprüfung der Performancemessungen und für die Evaluation der bestehenden Tools werden repräsentative Beispielprobleme benötigt. Die Probleme sollen Aufgaben aus den Programmier-Grundlagen-Modulen der ersten Studienjahre simulieren. Die Probleme sind unterschiedlich in ihrer Komplexität und sollen dabei einen möglichst vielseitigen Test des Performance-Messtools zulassen. Die folgenden Probleme wurden ausgewählt: das Sieb des Eratosthenes, der Quicksort-Algorithmus, Binärzahlen in Dezimalzahlen umwandeln, eine *IntArrayList*, der Dijkstra-Algorithmus und das Rucksackproblem.

Für jedes Problem wurde ein Interface definiert, welches die Struktur und verschiedene Implementierungen vorgibt. Die Implementierungen sollen verschiedene Studenten- oder Dozentenlösungen repräsentieren sowie verschiedene Lösungsansätze aufzeigen.

Jedes der sechs Beispielprobleme wird in einem der folgenden Kapitel dargestellt. Dabei wird zuerst das zu lösende Problem erklärt. Anschliessend werden das Interface und die verschiedenen Implementierungen beschrieben. Der Code zu den Beispielproblemen ist im Projekt Example Problems[1] zu finden. Jedes Problem hat ein eigenes Package im Package *ch.fhnw.autofeedback.exampleproblems*. Darin befinden sich sowohl das Interface als auch die Implementierungen. Die Packages sind nach dem jeweiligen Problem benannt. Das Interface und die implementierten Varianten haben denselben Namen wie in der Beschreibung.

3.1. SIEB DES ERATOSTHENES

Das Sieb des Eratosthenes ist ein alter Algorithmus, um Primzahlen bis zu einem gegebenen Höchstwert zu finden. Dafür werden in einem ersten Schritt alle Zahlen von zwei bis zum Höchstwert in ein Array geschrieben und als potenzielle Primzahlen markiert. Anschliessend wird die kleinste unmarkierte Zahl im Array als Primzahl markiert. Alle Vielfachen dieser Primzahl werden als Nichtprimzahl markiert. Diese beiden Schritte werden solange wiederholt, bis alle Zahlen im gegebenen Wertebereich entweder als Primzahlen oder als Nichtprimzahl markiert wurden [9].

IMPLEMENTIERUNG

Das *SieveOfEratosthenes* Interface definiert die Methode *int[] sieveOfEratosthenes(int n)*, diese erwartet eine Maximalzahl n. Als Resultat wird ein *int* Array mit allen Primzahlen von 2 bis n zurückgegeben.

Es wird mit zwei unterschiedlichen Varianten des *SieveOfEratosthenes* getestet:

1. *SieveOfEratosthenesSimple*: Das Sieb wird wie oben beschrieben implementiert.
2. *SieveOfEratosthenesCache*: Das Sieb erhält in dieser Variante einen Cache. Dabei werden die gefundenen Primzahlen in einem Array gespeichert. Bei weiteren Aufrufen des Algorithmus können nun bereits gefundene Primzahlen aus dem Cache gelesen werden und müssen nicht nochmals gesucht werden.

[1] Projekt Example Problems: https://gitlab.fhnw.ch/autofeedback/performance/exampleproblems

3.2. QUICKSORT

Der Quicksort Algorithmus ist ein effizienter Sortieralgorithmus. Als Beispielproblem soll eine Liste von Zahlen der Grösse nach sortiert werden [10].

IMPLEMENTIERUNG

Das *Quicksort* Interface definiert eine Methode *void sort(int arr[], int low, int high)*, diese erwartet ein *int* Array, welches sortiert werden soll. Die Werte *low* und *high* geben den Bereich im Array an, welcher sortiert werden soll. Dabei sollen alle Werte inklusive jenem an Position *low* bis zum Wert an Position *high* sortiert werden. Der Wert an Position *high* wird nicht mehr sortiert.

Ein Quicksort Algorithmus besteht aus drei Schritten:

1. Ein Pivot Element wird aus der zu sortierenden List ausgewählt.
2. Partitionierung: Die Liste wird aufgeteilt in einen ersten Teil, indem sich alle Elemente befinden, welche kleiner sind als das Pivot Element und einem zweiten Teil, indem sich alle Elemente befinden, welche grösser sind als das Pivot Element. Werte, die gleich dem Pivot Element sind, können sich in beiden Teilen befinden.
3. Rekursion: Die beiden oberen Schritte werden auf dem ersten und dem zweiten Teil der Liste separat angewendet. Dies wird solange wiederholt, bis sich nur noch ein Element in einer Liste befindet.

Es wird mit vier verschiedenen Varianten von *Quicksort* getestet:

1. *QuicksortWrapper*: Die Input Parameter werden an die Funktion *java.utils.Arrays.sort()* aus der Standartbibliothek übergeben. Diese übernimmt das gesamte Sortieren.
2. *QuicksortLomuto*: Für die Partitionierungen wird das Schema von Nico Lomuto verwendet. Das Array wird dabei von Indices i und j durchlaufen. Dabei sollen alle Zahlen von Start bis $(i - 1)$ kleiner dem Pivot Element und die Zahlen von i bis j grösser dem Pivot Element sein. Als Pivot Element wird jeweils das letzte Element des Arrays verwendet.
3. *QuicksortLomutoMid*: Die Partitionierung wird nach demselben Schema wie beim *QuickSort-Lomuto* Algorithmus vorgenommen. Nur die Auswahl des Pivot Elements ändert sich. Es werden das erste Element, das mittlere Element und das letzte Element miteinander verglichen und der mittlere Wert wird als Pivot Element verwendet.
4. *QuicksortHoare*: Für die Partitionierung wird das Schema von Tony Hoare verwendet. Dabei laufen zwei Indices von beiden Seiten gegeneinander. Sobald je ein Element gefunden wird, welches grösser gleich bzw. kleiner gleich dem Pivot Element ist, werden diese getauscht. Als Pivot Element wird das mittlere Element aus dem Array verwendet.

3.3. BINÄRZAHLEN IN DEZIMALZAHLEN UMWANDELN

Die Darstellung einer beliebig grossen positiven Zahl soll von der binären Darstellung in die dezimale Darstellung umgerechnet werden.

IMPLEMENTIERUNG

Das *BinaryToDecimal* Interface definiert eine Methode *String binaryToDecimal(String binary)*. Diese erwartet eine Binärzahl als Input und gibt die umgerechnete Dezimalzahl zurück. Damit die Zahlengrösse nicht limitiert ist, wird sowohl für den Input als auch für den Rückgabetyp ein *String* verwendet.

Es wird mit zwei verschiedenen Varianten von *BinaryToDecimal* getestet:

1. *BinaryToDecimalChar*: Die Binärzahl wird vom LSB her umgerechnet. Dabei werden zwei Werte hochgezählt: einerseits der Akkumulator, dieser Wert entspricht 2^x, wobei x die aktuelle Position in der Binärzahl ist, angefangen beim LSB mit 0. Andererseits wird das Resultat mit dem Akkumulator addiert, wenn sich in der Binärzahl an der Position x eine 1 befindet. Die beiden Zahlen werden in einem *char* Array gespeichert, in welchem jedes Element einer Dezimalzahl entspricht. Für die Rückgabe wird das *char* Array wieder in einen *String* umgerechnet, wobei alle führenden Nullen entfernt werden.
2. *BinaryToDecimalInt*: Diese Variante funktioniert prinzipiell genau gleich wie die *BinaryToDecimalChar* Variante. Sie unterscheiden sich nur in der Speicherung des Resultats und der Akkumulatorzahl. Diese werden in *int* Arrays gespeichert. Dabei entspricht jedoch ein Element nicht nur einer Zahl, sondern neun Stellen. Damit kann der Speicherbedarf reduziert werden und das Addieren wird effizienter.

3.4. IntArrayList

In der Java Standardbibliothek gibt es eine generische *ArrayList<E>*, welche mit einem beliebigen Objektdatentyp verwendet werden kann. Es existieren jedoch keine *ArrayLists* für die primitiven Datentypen. Die *IntArrayList* ist eine proprietäre Implementierung einer *ArrayList* für den primitiven Datentypen *int* mit den Funktionen *add, remove, get*, *set* und *size.*

Implementierung

Das *IntArrayList* Interface definiert die fünf Methoden: *void add(int e), int remove(int index), int get(int index), int set(int index, int e)* und *int size()*. Mit der *add* Methode wird das Element *e* hinten an die bestehende Liste angehängt. Die Methode *remove* entfernt das Element an Position *index* aus der Liste, anschliessend werden alle dahinterliegenden Elemente eine Position nach vorne geschoben und das gelöschte Element zurückgegeben. *Get* gibt das Element an Position *index* zurück. *Set* ersetzt das Element an Position *index* mit dem Element *e* und gibt das ersetzte Element zurück. *Size* gibt die aktuelle Grösse der Liste zurück.

Es wird mit drei verschiedenen Varianten von Binärzahlen in *IntArrayList* getestet:

1. *IntArrayListWrapper*: Die Wrapper-Liste verwendet intern eine *java.util.ArrayList<Integer>*. Die fünf Methoden rufen in dieser Version die gleichnamigen Methoden der *ArrayList* auf. Damit wurde die gesamte Funktionalität der *IntArrayList* ausgelagert.
2. *IntArrayListX1_5*: Die Variante 1.5 verwendet eine *int* Array als internen Speicher, welche standardmässig mit Grösse 10 initialisiert wird. Mit der *add* Methode wird ein Element an die erste nicht besetzt Position im Array gespeichert und der interne Zähler der Arraygrösse um 1 erhöht. Die Grösse des internen Speichers wird nach oben angepasst, sobald alle Positionen des Arrays gefüllt sind und ein weiteres Element eingefügt wird. Dabei wird das neue Array mit der 1.5-fachen Grösse des bestehenden Arrays initialisiert und alle Werte werden ins neue Array kopiert. Die restlichen Methoden verändern die Werte des internen Speichers und geben entsprechende Elemente zurück wie im Interface spezifiziert.
3. *IntArrayList2*: Die *IntArrayList2* ist identisch zur *IntArrayList1_5* mit einem Unterschied, das Array wird um das Zweifache vergrössert, sobald alle Positionen des Arrays gefüllt sind und ein weiteres Element eingefügt wird. Dies reduziert die Anzahl der zeitaufwendigen Vergrösserungen, erhöht jedoch auch den Speicherbedarf der Liste.

3.5. DIJKSTRA-ALGORITHMUS

Der Algorithmus von Dijkstra findet den kürzesten Pfad von einem gegebenen Startknoten zu allen anderen Knoten in einem gewichteten Graphen [11].

IMPLEMENTIERUNG

Das *Dijkstra* Interface definiert eine Methode *int[] dijkstra(Graph graph, int sourceVertex)*, diese erwartet einen Graphen und einen Startknoten und gibt die minimale Distanz zwischen dem Startknoten und jedem anderen Knoten als Array zurück. Es wird dabei erwartet, dass die Knoten im Graph von 0 bis $(n-1)$ nummeriert sind, wobei n die Anzahl der Konten ist.

Der Graph besteht aus Knoten mit gewichteten Kanten, dabei entspricht das Kantengewicht der Distanz zwischen den beiden Knoten. Das Kantengewicht muss positiv sein, damit der Algorithmus funktioniert. Existiert eine Kante zwischen zwei Knoten so sind diese verbunden. Existiert keine Kante so gibt es keine Verbindung zwischen den Knoten. Zunächst wird die Distanz zwischen Startknoten und jedem anderen Knoten in der Distanzliste auf Unendlich gesetzt und die Distanz zu sich selbst auf 0. Der Startknoten wird anschliessend in die Liste der zu bearbeitenden Knoten gesetzt. Solange die Liste, der zu bearbeitenden Knoten nicht leer ist, werden die folgenden Schritte ausgeführt: Es wird der Knoten u mit der momentan kürzesten Distanz zum Startknoten aus der Liste entfernt. Anschliessend wird für jede Kante, die vom Knoten u zu einem anderen Knoten v geht überprüft ob die aktuelle Distanz zu v oder die Distanz zu u plus das Kantengewicht kleiner ist. Falls der Weg über u der Kürzere ist, wird die Distanzliste aktualisiert und v in die Liste der zu bearbeitenden Knoten eingefügt.

Zwei verschiedenen Varianten von Dijkstra Algorithmen werden getestet:

1. *DijkstraPriorityQueue*: Die Variante *priority queue* verwendet den Dijkstra Algorithmus mit einer priorisierten Warteschlange als Datenstruktur für die Liste der zu bearbeitenden Knoten. Aus einer priorisierten Warteschlange kann effizient das Element mit der höchsten Priorität entfernt werden. Die Elemente werden für den Dijkstra Algorithmus anhand der Distanz zum Startknoten priorisiert. Dabei ist eine tiefe Distanz höher gewichtet.
2. *DijkstraSet*: Die Variante *set* verwendet ein Set als Datenstruktur für die Liste der zu bearbeitenden Knoten. Dabei muss das Set durchsucht werden, um das Element mit der kleinsten Distanz zum Startknoten zu finden.

3.6. RUCKSACKPROBLEM

Das Rucksackproblem ist ein Optimierungsproblem und gehört zu den 21 klassischen NP-vollständigen Problemen. Es wird aus einer Menge von Objekten, bestehend aus Gewicht und Nutzwert, die Teilmenge mit dem grössten Nutzwert bestimmt, bei welcher das Gesamtgewicht kleiner als das vorgegebene Maximalgewicht ist [12].

IMPLEMENTIERUNG

Das *Knapsack* Interface definiert eine Methode *int knapSack(int w, int wt[], int val[], int n)*, diese erwartet die Kapazität des Rucksackes w, ein Array mit den gewichteten Objekten, ein Array mit den Nutzwerten der Objekte sowie die Anzahl der Objekte die vorhanden sind. Als Resultat wird der Nutzwert aller Objekte, die sich im Rucksack befinden erwartet.

Es wird mit drei verschiedenen Varianten von Knapsack Implementierungen getestet:

1. *KnapsackBranchAndBound*: In der Branch and Bound (B&B) [13] Implementierung vom Rucksackproblem wird mit Branch-and-Bound ein Entscheidungsbaum aufgebaut und durchsucht. Dafür werden zur Vorbereitung alle Objekte nach Nutzwert pro Gewicht sortiert. Somit befindet sich das wertvollste Objekt am Anfang der Liste. Anschliessend wird der Entscheidungsbaum aufgebaut.
   In jedem Knoten werden so lange Objekte in den Rucksack gepackt, bis keines mehr Platz hat. Dabei wird immer das Wertvollste, sich noch nicht im Rucksack befindende Objekt reingelegt, solange bis das wertvollste, sich noch nicht im Rucksack befindende Objekt o nicht mehr komplett in den Rucksack passt. Der totale Nutzwert, der sich aktuell im Rucksack befindet, gibt eine untere Schranke für das Problem. Die wertvollste untere Schranke wird global vermerkt und dient später dazu, ganze Teilbäume aus dem Entscheidungsbaum zu schneiden. Für die obere Schranke wird das wertvollste sich noch nicht im Rucksack befindenden Objekt o, anteilsmässig in den Rucksack gepackt. Somit ist der Rucksack nun komplett gefüllt. Die obere Schranke definiert den maximalen Nutzwert, der im darunterliegenden Teilbaum noch möglich ist.
   Der Startknoten hat noch keine Auflagen und berechnet beide Schranken. Falls beide Schranken gleich sind, wurde der optimale Nutzwert für diesen Teilbaum gefunden. Falls die obere Schranke kleiner ist als die globale untere Schranke, kann in diesem Teilbaum nicht der optimale Nutzwert liegen.
   Falls keines dieser Kriterien erfüllt ist, wird in zwei Teilbäumen weitergesucht. Für den linken Teilbaum wird das Objekt o aus der Liste aller Objekte entfernt. Für den rechten Teilbaum wird das Objekt o bereits in den Rucksack gelegt. Bei der Evaluation wird immer erst der rechte Teilbaum evaluiert. Der Entscheidungsbaum wird mit Tiefensuche durchsucht, dabei hat der rechte Teilbaum immer Priorität.
   Ist der gesamte Baum durchsucht, wird die globale untere Schranke als optimaler Nutzwert zurückgegeben.
2. *KnapsackDynamicProgramming*: In der Dynamic Programming Variante werden schon berechnete Teilresultate in einer Tabelle K zwischengespeichert. Die zwischengespeicherten Werte können später aus der Tabelle K gelesen werden und müssen nicht nochmals berechnet werden. In der Tabelle K entsprechen die Spalten den werten von 0 bis w und die Reihen beinhalten die Gewichte. Die Tabelle K wird nun mit zwei ineinander verschachtelten Schleifen ausgefüllt. In der äusseren Schleife wird i von 0 bis Anzahl Objekte n gezählt und in der inneren Schleife von 0 bis zum maximalen Gewicht w. Innerhalb der beiden Schleifen wird die Tabelle anhand der Formel 1 ausgefüllt. Der maximale Nutzwert steht nach den Schleifen in der Tabelle an Position $K[n, w]$ [14].

$$K[i,w] = \begin{cases} 0 & i = 0 \bigvee w = 0 \\ \max\begin{cases} val[i-1] + K[i-1, w - wt[i-1]] \\ K[i-1,w] \end{cases} & wt[i-1] \leq w \\ K[i-1,w] & otherwise \end{cases} \quad (1)$$

3. *KnapsackRecursive*: Die rekursive Implementierung des Rucksackproblems evaluiert die maximale Rucksackkapazität mit der rekursiven Formel 2. Die Formel $M(n, w)$ findet die optimale Lösung für n Objekte mit einem maximalen Gewicht w. Die variable v_n entspricht dabei dem Nutzwert des n-ten Objekts und w_n dem Gewicht des n-ten Objekts. Die Rekursion wird mit der Anzahl Objekte als n und des Maximalgewichtes als w gestartet.

$$M(n,w) = \begin{cases} 0 & n \leq 0 \\ M(n-1,w) & w_n > w \\ \max\begin{cases} M(n-1,w) \\ v_n + M(n-1, w-w_n) \end{cases} & w_n \leq w \end{cases} \quad (2)$$

4. PERFORMANCE-MESSTOOL

In diesem Kapitel wird in mehreren Schritten erklärt, wie die asymptotische Zeitkomplexität für einen Algorithmus abgeschätzt wird. Dies bildet die Basis für die Umsetzung des Performance-Messtools. Der Aufbau des Performance-Messtools ist in Abbildung 1 zu sehen. Die einzelnen Schritte werden in den folgenden Kapiteln genauer beschrieben.

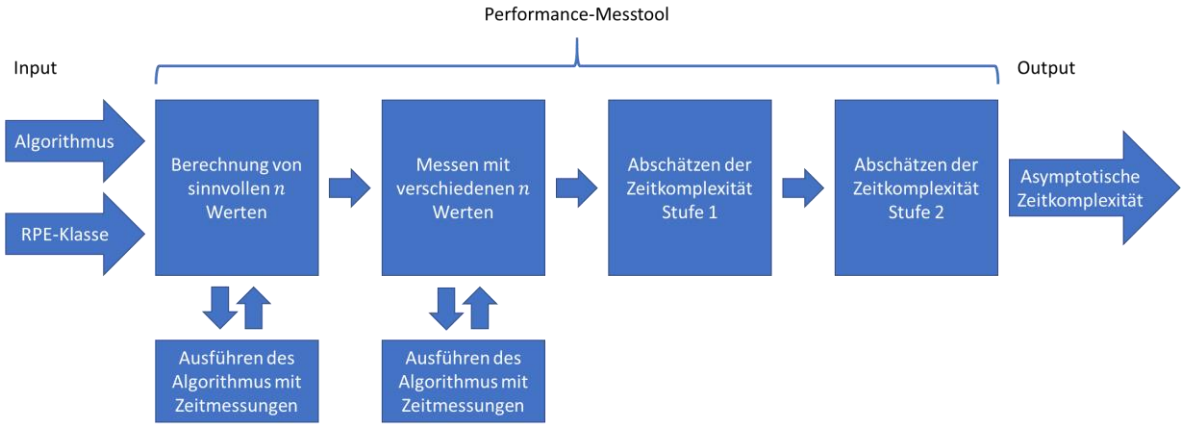


ABBILDUNG 1: AUFBAU PERFORMANCE-MESSTOOL

Der implementierte Code zum Performance-Messtool befindet sich im Projekt AutoFeedback Library[2].

4.1. VORAUSSETZUNGEN

Für das Testen eines Algorithmus werden zwei verschiedene Klassen benötigt. Erstens der Algorithmus selbst und zweitens eine RPE-Klasse, die dem Performance-Messtool mitteilt, wie der Algorithmus zu verwenden ist. Die RPE-Klasse benötigt ein Interface, welches die Struktur des Algorithmus definiert. Der Algorithmus selbst und das Interface dazu können ohne Einschränkungen definiert werden.

Die RPE-Klasse erweitert die abstrakte Klasse *RunPerformanceEvaluation*, welche aus den beiden abstrakten Methoden *run()* und *setup(int n)* besteht. Die *setup* Methode wird vor jedem *run* ausgeführt und soll die Daten für die Ausführung bereitstellen. Der Parameter n steht für die Grösse, anhand welcher die asymptotische Komplexität berechnet werden soll. Die Methode *run* wird zur Ausführung des Algorithmus verwendet. Die errechnete Zeitkomplexität wird vom Algorithmus nach der Analyse in die RPE-Klasse gespeichert. Um dem Studenten nicht nur die ermittelte Zeitkomplexität mitteilen zu können, erwartet der Konstruktor auch die für die Studenten anzustrebende asymptotische Zeitkomplexität und den Namen des Tests. Die Implementierung der *RunPerformanceEvaluation* Klasse befindet sich im Projekt AutoFeedback Library im package *ch.fhnw.autofeedback.performanceevaluation*.

4.2. ZEITMESSUNGEN

Damit die asymptotische Zeitkomplexität eines Algorithmus berechnet werden kann, muss dieser mit verschiedenen Inputgrössen ausgeführt werden. Die Inputgrösse wird mithilfe des n in der *setup* Methode gesteuert. Um aussagekräftige Resultate zu erhalten, kann dieser Wert nicht zufällig gewählt werden.

[2] Projekt AutoFeedback Library: https://gitlab.fhnw.ch/autofeedback/autofeedback-library

4.2.1. Berechnung von sinnvollen n Werten

Für eine Analyse der Zeitkomplexität werden Ausführungen mit möglichst weit gestreuten Inputgrössen n benötigt. Sehr kleine Inputgrössen n sind nicht aussagekräftig. Dies liegt sowohl an den Algorithmen selbst, weil nicht mehr der Term mit der höchsten Ordnung dominiert, als auch an den Zeitmessungen, die eher die Dauer des Methodenaufrufs messen. Deswegen sollte ein n gewählt werden, bei welchem die Ausführung bereits einige Zeit dauert. Bei zu grossen n gibt es nur das Problem, dass der Algorithmus länger dauert.

Um diese Probleme zu minimieren, können minimale und maximale Ausführungszeiten für einen durchlauf des Algorithmus definiert werden. Anhand dieser Zeiten werden anschliessend minimale und maximale Werte für n berechnet. Für die Berechnung werden die Formeln 3 bis 6 verwendet. Die Variable $time$ entspricht dabei der Ausführungszeit des Algorithmus mit der Input-Grösse n. Die Variable $target$ ist die definierte maximale Ausführungszeit für den Algorithmus. Die Berechnung wird solange fortgesetzt, bis die Ausführungszeit die definierte maximale Ausführungszeit überschreitet. Dieser letzte n Wert ist nun der maximale n Wert. Der minimale n Wert entspricht dem ersten n Wert bei welchem die $time$ die minimale Ausführungsdauer überschreitet. Gestartet wird die Berechnung mit den n Werten eins und zwei (siehe Formel 3).

$$n_1 = 1, \quad n_2 = 2 \tag{3}$$

$$longer = \frac{time_{i-1}}{time_{i-2}} \tag{4}$$

$$increase = \begin{cases} n_{i-1} * \left(\frac{n_{i-1}}{n_{i-2}} - 1\right) & time_{i-1} * longer < target \\ \frac{n_{i-1} - n_{i-2}}{longer} & otherwise \end{cases} \tag{5}$$

$$n_i = \max\begin{cases} n_{i-1} + increase \\ n_{i-1} + 1 \end{cases} \tag{6}$$

Die Ausführungszeiten für ein n wird mit Messungen ermittelt. Für diese Zeitmessungen kann sowohl die Anzahl an Aufwärmiterationen als auch die Anzahl der gezeiteten Iterationen konfiguriert werden. Die Messungen werden im folgenden Kapitel erklärt. Die Implementierung der Berechnung von sinnvollen n Werten ist im Projekt AutoFeedback Library, im Package *ch.fhnw.autofeedback.performanceevaluation* in der Klasse *PerformanceEvaluation* zu finden. Die Berechnung der verschiedenen n Werte wird im ersten Teil der *runEvaluation()* Methode vorgenommen.

4.2.2. Messen mit verschiedenen n Werten

Der zu bewertende Algorithmus kann nun ausgeführt und die Zeit gemessen werden. Dafür wird er eine konfigurierbare Anzahl, mit logarithmisch verteilten n Werten zwischen dem minimalen und maximalen n Wert, ausgeführt. Die Schwierigkeit dabei ist, dass eine einzige Ausführung aus mehreren Gründen nicht Aussage kräftig sein kann. Der Input für den Algorithmus sollte für gleiche n Werte gleich schwierig sein. Dabei ist zu beachten, dass z.B. beim Sortieren nicht jedes Array mit derselben Länge gleich schwierig zu sortieren ist. Deshalb sollten verschiedene Arrays mit derselben Länge gemessen werden, um einen Durchschnittswert zu berechnen. Ein weiteres Problem sind die inkonsistenten Ausführungszeiten von Java für den exakt gleichen Code. Die Ursachen der Messungenauigkeiten werden im Kapitel 4.2.2.1 besprochen. Um die Auswirkungen dieser Probleme zu reduzieren, wird der zu bewertende Algorithmus erst einige Zeit ohne zu Messen als Aufwärmephase ausgeführt. Anschliessend wird gemessen. Der

Durchschnittswert für eine Ausführung wird aus allen Ausführungen berechnet und zusammen mit dem n Wert gespeichert. Diese Ausführungszeiten können konfiguriert werden. Implementiert sind die Zeitmessungen im Projekt AutoFeedback Library, im Package *ch.fhnw.autofeedback.performanceevaluation* in der Klasse *PerformanceEvaluation*. Die Ausführungszeit kann mit den Methoden *testXsec()* und *testXTimes()* gemessen werden. Während der Evaluation werden diese Methoden aus der *runEvaluation()* Methode heraus aufgerufen.

4.2.2.1. Performance Messungen in Java

Eine automatische Performanceevaluation hängt unter anderem von der Genauigkeit und der Konstanz der Zeitmessungen ab. Die Messungen sollten wiederholt durchgeführt werden, um eine höhere Genauigkeit zu erreichen. Der Zeit- und Ressourcenaufwand pro Test erhöht sich somit um die Anzahl der Wiederholungen. Ebenfalls ist zu beachten, dass die Schwankungen in den Zeitmessungen nicht grösser sein sollten, als die Ausführungszeit des Programms und somit sichergestellt ist, dass nicht nur Schwankungen in der Systemperformance gemessen werden.

Java wird zu Bytecode kompiliert, welcher vor der Java-Virtual-Machine (JVM) ausgeführt wird. Der Code wird in einem Interpreter Modus ausgeführt. Die JVM verfolgt häufig aufgerufene Methoden und kompiliert diese mit dem Java-Just-In-Time-Compiler (JIT) zu Maschinencode. Diese Optimierung benötigt Zeit, welche mit den nachfolgenden schnelleren Ausführungen kompensiert werden soll [15]. Dieser Effekt könnte bei wiederholtem Ausführen oder bei Inputs, welche aufwendig zu berechnen sind, zum Tragen kommen und die Resultate verfälschen. Bei der Kompilierung durch den JIT Compiler können auch Performance Optimierungen durchgeführt werden. Dabei können nicht performante Studentenlösungen intern zu performantem Code kompiliert werden.

Einen Teil der Verarbeitungszeit verbringt eine Java Applikation mit Garbage Collection. Es wird Speicherplatz freigegeben und der Heap aufgeräumt. Somit wird das Speichermanagement von der JVM übernommen und nicht vom Entwickler selbst. Der Garbage Collector kann nun eine Reihe von Zeitmessungen verfälschen, indem er nicht bei jeder Messung in Aktion tritt [16].

4.3. Abschätzung der Zeitkomplexität

Die Abschätzung der Zeitkomplexität wird in zwei Stufen vorgenommen. In der ersten Stufe wird eine grobe Abschätzung der Laufzeit berechnet. In einer zweiten Stufe wird die grobe Abschätzung präzisiert und in eine Zeitkomplexität umgerechnet.

Stufe 1

In der ersten Stufe wird versucht die polynomielle Abhängigkeit der Input Grösse n zur Ausführungsdauer $time$ zu berechnen ($time = an^b + \cdots$). Das b aus dieser Funktion gibt eine grobe Abschätzung der asymptotischen Komplexität. Für die Berechnung der Werte von a und b wird eine lineare Regression verwendet. Die lineare Regression versucht, beobachtete abhängige Variablen mit einer unabhängigen Variabel mit der Formel $y = a + bx$ zu erklären. Die lineare Regression wird im Kapitel 4.3.1 behandelt.

Die beiden Formeln können mit einer Logarithmus-Transformation aneinander angeglichen werden mit den Formeln 7 bis 10 [17].

$$time = an^b \tag{7}$$

$$\ln time = \ln(an^b) \tag{8}$$

$$\ln time = \ln a + b \ln n \tag{9}$$

$$y = k + bx, \quad y = \ln time, \quad \mathrm{x} = \ln n, \quad \mathrm{k} = \ln a \tag{10}$$

Die Formel der linearen Regression entspricht der Formel $y = k + bx$, damit können die gesuchten Werte für a und b mit Hilfe der linearen Regression berechnet werden.

Stufe 2

In der zweiten Stufe wird anhand der Variablen b, n und $time$ die Zeitkomplexität ermittelt, welche die Abhängigkeit von n zu $time$ am besten darstellt. Mögliche Zeitkomplexitäten, welche geschätzt werden können, sind $O(1)$, $O(\log n)$, $O(\sqrt{n})$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, $O(n^4)$, $O(2^n)$ und $O(n!)$. Für die Auswahl der Zeitkomplexität wird, anhand der zuvor errechneten Variabel b, die Anzahl der möglichen Zeitkomplexitäten reduziert. Anschliessend wird für diese eine lineare Regression berechnet und anhand des r2 Scores die am besten passende ausgewählt. Die Auswahlkriterien und die linearen Funktionen für die Zeitkomplexität sind in Tabelle 8 zu sehen und der r2 score wird im Kapitel 4.3.2 erklärt.

Zeitkomplexität	Bereich von b	Lineare Funktion
$O(1)$	$b < 0.3$	$time = a + b0 \Rightarrow y = a$
$O(\log n)$	$b > 0 \wedge b < 1$	$time = a + b(\log_2 n)$
$O(\sqrt{n})$	$b > 0 \wedge b < 1$	$time = a + b\sqrt{n}$
$O(n)$	$b > 0.4 \wedge b < 1.4$	$time = a + bn$
$O(n \log n)$	$b > 1 \wedge b < 2$	$time = a + b(n * \log_2 n)$
$O(n^2)$	$b > 1.4 \wedge b < 2$	$time = a + b(n^2)$
$O(n^3)$	$b > 2 \wedge b < 4$	$time = a + b(n^3)$
$O(n^4)$	$b > 3 \wedge b < 5$	$time = a + b(n^4)$
$O(2^n)$	$b > 4$	$time = a + b(2^n)$
$O(n!)$	$b > 5$	$time = a + b(n!)$

TABELLE 8: ZEITKOMPLEXITÄTEN

Die Evaluation der Zeitkomplexität ist im Projekt AutoFeedback Library im Package *ch.fhnw.autofeedback.performanceevaluation* in der Klasse *TimeComplexity* zu finden.

4.3.1. LINEARE REGRESSION

Eine lineare Regression [18] versucht beobachtete abhängige Variablen mit einer unabhängigen Variablen zu erklären. Dies bedeutet, dass eine Gerade durch die Datenpunkte gelegt wird, welche möglichst nahe an allen Datenpunkten liegt. Dies ist in Abbildung 2 sichtbar. Bei der einfachen linearen Regression werden nur zwei metrische Grössen berücksichtigt, die Einflussgrösse X und die Zielgrösse Y. Die Gleichung der linearen Regression ist damit durch die Formel 11 gegeben. Die Variablen x und y entsprechen dabei der Einfluss- und Zielgrösse. Die Variablen β_0 und β_1 entsprechen den zu berechnenden

Variablen. ε ist die Error Variable, welche das zufällige Rauschen in den Daten abbildet und n ist die Anzahl an Variablen im Vektor X und Y.

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i, \quad i = 1, \dots, n \tag{11}$$

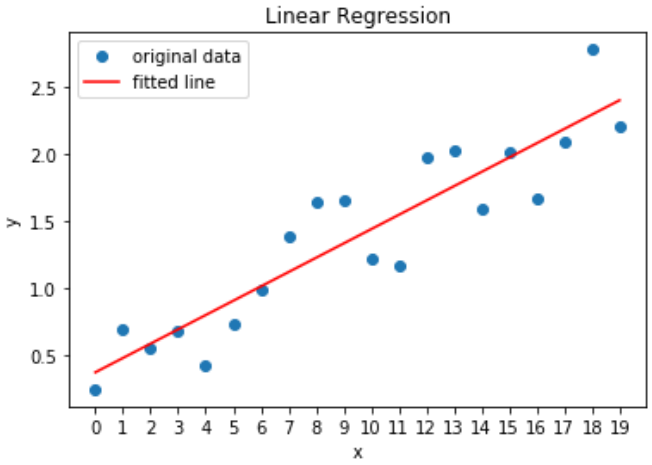


ABBILDUNG 2: BEISPIEL EINER LINEAREN REGRESSION

Es gibt verschiedene Möglichkeiten den Abstand zwischen den Datenpunkten und der Regressionslinie zu messen. In der verwendeten Variante wird die Differenz zwischen Vorhersage und effektivem Wert quadriert. Die Parameter β_0 und β_1 der linearen Formel können anhand der Einflussgrösse x und der Zielgrösse y berechnet werden. Die Berechnung erfolgt mit den Formeln 12, 13 und 14.

$$\beta_1 = \frac{SS_{xy}}{SS_{xx}}, \quad \beta_0 = \bar{y} - \beta_1 \bar{x} \tag{12}$$

$$\bar{x} = \frac{\sum x}{n}, \quad \bar{y} = \frac{\sum y}{n} \tag{13}$$

$$SS_{xy} = \frac{1}{n}\sum (x - \bar{x})(y - \bar{y}), \quad SS_{xx} = \frac{1}{n}\sum (x - \bar{x})^2 \tag{14}$$

4.3.2. R2 SCORE

Das Bestimmtheitsmass oder r2 Score [19] ist eine Performance Metrik für Regressionsmodelle. Der r2 Score beurteilt die Anpassungsgüte einer Regression. Es beruht dabei auf der Quadratsummenzerlegung. Es wird die totale Quadratsumme, in die vom Regressionsmodell erklärte Quadratsumme und die Residuenquadratsumme zerlegt. Die Berechnung erfolgt mit Formel 15 wobei y den Messwerten und $\hat{y}$ den vorhergesagten Werten entspricht.

$$R^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2}, \quad \bar{y} = \frac{\sum y}{n} \tag{15}$$

4.4. TESTS

Das Performance-Messtool wird anhand der Beispielprobleme getestet. Für eine Analyse wird für jedes Problem eine RPE-Klasse benötigt und es können einige Konfigurationsparameter eingestellt werden. Die Konfigurationsparameter wurden mithilfe des Quicksort Beispielproblems evaluiert und sind in Tabelle 9 zu sehen. Anschliessend wird für jedes Problem die RPE-Klasse vorgestellt und die Resultate der Tests dargestellt.

Konfigurationsparameter	Wert
Minimale Ausführungszeit für n	20'000 Nanosekunden
Maximale Ausführungszeit für n	0.025 Sekunden
Anzahl der Aufwärmiterationen für das Finden des minimalen und maximalen N-Werts	2 Iterationen
Anzahl der Iterationen für das Finden der minimalen und maximalen N-Werts	6 Iterationen
Aufwärmzeit für das Messen der Performance	1 Sekunde
Zeit für das Messen der Performance	2 Sekunden
Anzahl verschiedene N die, logarithmisch verteilt zwischen dem minimalen und maximalen N, getestet werden	6 Ausführungen

TABELLE 9: KONFIGURATIONSPARAMETER

Die Implementierung der RPE-Klassen und der Tests ist im Projekt AutoFeedback Library zu finden. Die Tests und alle RPE-Klassen befinden sich im Test Ordner im package *ch.fhnw.autofeedback.performance*. Die RPE-Klassen beginnen mit dem Kürzel RPE, gefolgt vom Namen des Beispielproblems. Die Tests beginnen mit dem Kürzel Test, gefolgt vom Namen des Beispielproblems. Weil in den Beispielproblemen nicht alle Zeitkomplexitäten einmal vorhanden sind, wurde das BusyWaiting Problem zum Testen aller Zeitkomplexitäten hinzugefügt.

4.4.1. BUSYWAITING

Die RPE-Klasse des *BusyWaiting* Problems speichert in der *setup* Methode den Wert für n. Die *run* Methode berechnet anhand von n wie lange sie Warten soll und wartet anschliessend genau solange, bis ein Resultat zurückgegeben wird. An die *RPEBusyWaiting* Klasse kann im Konstruktor eine Methode übergeben werden, welche die Laufzeit für ein n berechnen soll. Für die Zeitkomplexitäten $O(n^2)$ wird eine Methode übergeben welche n quadriert und für die Zeitkomplexitäten $O(\log n)$ wird eine Methode übergeben welche den Logarithmus von n nimmt usw.. So kann ein Test für jede Zeitkomplexität erstellt werden, bei welchem die Ausführungszeiten perfekt auf die Zeitkomplexität passen. Die Tests für alle Zeitkomplexitäten waren erfolgreich.

4.4.2. SIEB DES ERATOSTHENES

Das Sieb des Eratosthenes hat eine sehr einfache RPE-Klasse. Das n welches die *setup* Methode bekommt muss nur in der *run* Methode an den Algorithmus weitergegeben werden. Zusätzlich wird in der *setup* Methode die Instanz der Sieb-Klasse erneuert, um Caching Vorteile zu minimieren. Die Zeitkomplexität wurde bei beiden Varianten meistens korrekt evaluiert (sehe Tabelle 10).

Variante	Zu erwartende Zeitkomplexität	Analysierte Zeitkomplexität
SieveOfEratosthenesSimple	Die implementierte Variante von Sieb des Eratosthenes hat eine asymptotische Zeitkomplexität von $O(n \log\log n)$ [9].	Es wurde eine der nächstmöglichen asymptotische Zeitkomplexität von $O(n)$ oder $O(n \log n)$ evaluiert.

Variante	Zu erwartende Zeitkomplexität	Analysierte Zeitkomplexität
SieveOfEratosthenesCache	Der Algorithmus hat grundsätzlich eine asymptotische Zeitkomplexität von $O(n \log \log n)$ [9]. Durch das Cachen von Lösungen wird der Algorithmus zum Teil gar nicht ausgeführt.	Es wurde eine der nächstmöglichen asymptotische Zeitkomplexität von $O(n)$ oder $O(n \log n)$ evaluiert. Die Effekte des Caches kommen nicht zum Tragen, dank der neu Instanziierung der Klasse vor jedem Test.

TABELLE 10: RESULTATE SIEB DES ERATOSTHENES

4.4.3. QUICKSORT

Die RPE-Klasse des Quicksort-Algorithmus erstellt in der *setup* Methode ein zufälliges Array der Länge n welches anschliessend in der *run* Methode sortiert wird. Bei den Tests wurde für alle Quicksort Algorithmen die erwartete asymptotische Zeitkomplexität bestimmt (sehe Tabelle 11).

Variante	Zu erwartende Zeitkomplexität	Analysierte Zeitkomplexität
QuicksortWrapper	Ein Quicksort Algorithmus hat eine durchschnittliche asymptotische Zeitkomplexität von $O(n \log n)$. Diese kann im schlechtesten Fall zu $O(n^2)$ werden [10].	Durch das mehrfache Ausführen des Sortierens und der zufälligen Arrays, die sortiert werden, wird die Zeitkomplexität $O(n \log n)$ korrekt evaluiert.
QuicksortLomuto	$O(n \log n)$ für die Erklärung siehe Zeile *QuicksortWrapper*.	$O(n \log n)$
QuicksortLomutoMid	$O(n \log n)$ für die Erklärung siehe Zeile *QuicksortWrapper*.	$O(n \log n)$
QuicksortHoare	$O(n \log n)$ für die Erklärung siehe Zeile *QuicksortWrapper*.	$O(n \log n)$

TABELLE 11: RESULTATE QUICKSORT

4.4.4. BINÄRZAHLEN IN DEZIMALZAHLEN UMWANDELN

Die RPE-Klasse für das Binärzahlen in Dezimalzahlen Umwandeln, erzeugt in der *setup* Methode einen String mit Länge n der zufällig mit Nullen und Einsen befüllt wird. Dieser String wird anschliessend in der *run* Methode dem Algorithmus zum Umwandeln übergeben. Die Resultate des Tests sind korrekt und können in der Tabelle 12 angesehen werden.

Variante	Zu erwartende Zeitkomplexität	Analysierte Zeitkomplexität
BinaryToDecimalChar	Die Implementierte Variante hat eine asymptotische Zeitkomplexität von $O(n^2)$.	$O(n^2)$
BinaryToDecimalInt	Die Implementierte Variante hat eine asymptotische Zeitkomplexität von $O(n^2)$.	$O(n^2)$

TABELLE 12: RESULTATE BINÄRZAHLEN IN DEZIMALZAHLEN UMWANDELN

4.4.5. IntArrayList

Die *IntArrayList* besteht aus den Methoden *add*, *get*, *remove*, *set* und *size*. Für jede dieser Methoden wird eine separate RPE-Klasse erstellt. Somit kann die Performance von allen fünf Methoden getestet werden. Die Aufführungsdauer der Methoden, welche die IntArrayList anbietet, hängt nicht vom Input der Methoden ab. Die Ausführungsdauer hängt von der Länge der Liste ab oder ist konstant.

Die RPE-Klasse der *add* Methode sorgt in der *setup* Methode dafür, dass falls die Liste zu gross wird (über eine Million Elemente) eine neue Liste erstellt wird. In die Liste wird anschliessend in der *run* Methode ein Element eingefügt. Das Neuerstellen der Liste verhindert Speicherprobleme, die bei wiederholtem Einfügen von Elementen entstehen könnten. Die Resultate der *add* Methode sind in Tabelle 13 zu sehen.

Variante	Zu erwartende Zeitkomplexität	Analysierte Zeitkomplexität
IntArrayListWrapper	Das Einfügen eines Elementes in eine bestehende Liste hat im Normalfall eine asymptotische Zeitkomplexität von $O(1)$. Sobald die Liste jedoch keine freie Kapazität mehr hat, wird eine neue grössere Liste erstellt und alle bestehenden Elemente werden in die neue Liste kopiert. Diese Operation hat eine asymptotische Zeitkomplexität von $O(n)$, welches in einer amortisierten asymptotischen Zeitkomplexität von $O(1)$ für ein Element resultiert.	$O(1)$
IntArrayListX1_5	$O(1)$ für die Erklärung siehe Zeile *IntArrayListWrapper*.	$O(1)$
IntArrayList2	$O(1)$ für die Erklärung siehe Zeile *IntArrayListWrapper*.	$O(1)$

TABELLE 13: RESULTATE DER METHODE ADD DER INTARRAYLIST

Die RPE-Klasse der *get* Methode bereitet in der *setup* Methode eine *IntArrayList* der Länge n vor. Die *IntArrayList* wird bei mehrfachem Aufrufen der *setup* Methode falls nötig erweitert. Falls sie bereits zu lange ist wird eine neue Liste erstellt und wieder aufgefüllt mit n Elementen. Da der Arbeitsspeicher auf einem Computer begrenzt ist, kann jedoch die *IntArrayList* nicht beliebig gross werden. Um

Speicherprobleme zu verhindern wurde die IntArrayList auf 50 Millionen Elemente limitiert. Das bedeutet, dass alle Tests mit n grösser als 50 Millionen ein verfälschtes Resultat zurückliefern und somit auch die Performanceevaluation verfälschen. In der *run* Methode wird zufällig eine Position in der ArrayList ausgewählt und das Element an dieser Position mit *get* geholt. Die Resultate der *get* Methode sind in Tabelle 14 zu sehen.

Variante	Zu erwartende Zeitkomplexität	Analysierte Zeitkomplexität
IntArrayListWrapper	Eine einzelne *get* Operation hat eine asymptotische Zeitkomplexität von $O(1)$.	$O(1)$
IntArrayListX1_5	$O(1)$ für die Erklärung siehe Zeile *IntArrayListWrapper*.	$O(1)$
IntArrayList2	$O(1)$ für die Erklärung siehe Zeile *IntArrayListWrapper*.	$O(1)$

TABELLE 14: RESULTATE DER METHODE GET DER INTARRAYLIST

Die RPE-Klasse der *remove* Methode bereitet in der *setup* Methode eine *IntArrayList* der Länge von n vor. Die *IntArrayList* wird bei mehrfachem Aufrufen der *setup* Methode falls nötig erweitert. Falls sie bereits zu lange ist wird eine neue Liste erstellt und wieder aufgefüllt mit n Elementen. Da der Arbeitsspeicher auf einem Computer begrenzt ist, kann jedoch die *IntArrayList* nicht beliebig gross werden. Um Speicherprobleme zu verhindern wurde die IntArrayList auf 50 Millionen Elemente limitiert. Das bedeutet, dass alle Tests mit n grösser als 50 Millionen ein verfälschtes Resultat zurückliefern und somit auch die Performanceevaluation verfälschen. In der *run* Methode wird das erste Elemente aus der *IntArrayList* entfernt. Die Resultate der *remove* Methode sind in der Tabelle 15 zu sehen. Die Zeitkomplexitäten wurden korrekt ermittelt.

Variante	Zu erwartende Zeitkomplexität	Analysierte Zeitkomplexität
IntArrayListWrapper	Ein aus der *IntArrayList* gelöschtes Element hinterlässt eine Lücke im Array. Die dahinterliegenden Elemente müssen um eine Position nach vorne vorschoben werden. Weil in diesem Test das erste Element gelöscht wird, müssen alle anderen $n-1$ Elemente einen Platz nach vorne geschoben werden. Dies entspricht einer asymptotischen Zeitkomplexität von $O(n)$ für ein Element.	Das Resultat ist inkonsistent und schwankt zwischen $O(n)$ und $O(n \log n)$. Dies liegt an Schwankungen in den Messungen.

Variante	Zu erwartende Zeitkomplexität	Analysierte Zeitkomplexität
IntArrayListX1_5	$O(n)$ für die Erklärung siehe Zeile *IntArrayListWrapper*.	Das Resultat ist inkonsistent und schwankt zwischen $O(\log n)$, $O(\sqrt{n})$ und $O(n)$. Dies hängt mit der Limitierung der Listenlänge und der damit verbundenen Ausführungsdauer zusammen.
IntArrayList2	$O(n)$ für die Erklärung siehe Zeile *IntArrayListWrapper*.	Das Resultat ist inkonsistent und schwankt zwischen $O(\log n)$, $O(\sqrt{n})$ und $O(n)$. Dies hängt mit der Limitierung der Listenlänge und der damit verbundenen Ausführungsdauer zusammen.

TABELLE 15: RESULTATE DER METHODE REMOVE DER INTARRAYLIST

Die RPE-Klasse der *set* Methode bereitet in der *setup* Methode eine *IntArrayList* der Länge von n vor. Die *IntArrayList* wird bei mehrfachem Aufrufen der *setup* Methode falls nötig erweitert. Falls sie bereits zu lange ist wird eine neue Liste erstellt und wieder aufgefüllt mit n Elementen. Da der Arbeitsspeicher auf einem Computer begrenzt ist, kann jedoch die *IntArrayList* nicht beliebig gross werden. Um Speicherprobleme zu verhindern wurde die IntArrayList auf 50 Millionen Elemente limitiert. Das bedeutet, dass alle Tests mit n grösser als 50 Millionen ein verfälschtes Resultat zurückliefern und somit auch die Performanceevaluation verfälschen. In der *run* Methode wird ein zufälliger Wert der *IntArrayList* mit einem neuen Werten überschrieben. Die Resultate der *set* Methode sind in der Tabelle 16 zu sehen und die Zeitkomplexitäten wurden korrekt ermittelt.

Variante	Zu erwartende Zeitkomplexität	Analysierte Zeitkomplexität
IntArrayListWrapper	Eine einzelne *set* Operation hat eine asymptotische Zeitkomplexität von $O(1)$.	$O(1)$
IntArrayListX1_5	$O(1)$ für die Erklärung siehe Zeile *IntArrayListWrapper*.	$O(1)$
IntArrayList2	$O(1)$ für die Erklärung siehe Zeile *IntArrayListWrapper*.	$O(1)$

TABELLE 16: RESULTATE DER METHODE SET DER INTARRAYLIST

Die RPE-Klasse der *size* Methode bereitet in der *setup* Methode eine *IntArrayList* der Länge von n vor. Die *IntArrayList* wird bei mehrfachem Aufrufen der *setup* Methode falls nötig erweitert. Falls sie bereits zu lange ist wird eine neue Liste erstellt und wieder aufgefüllt mit n Elementen. Da der Arbeitsspeicher auf einem Computer begrenzt ist, kann jedoch die *IntArrayList* nicht beliebig gross werden. Um Speicherprobleme zu verhindern wurde die IntArrayList auf 50 Millionen Elemente limitiert. Das bedeutet,

dass alle Tests mit n grösser als 50 Millionen ein verfälschtes Resultat zurückliefern und somit auch die Performanceevaluation verfälschen. In der *run* Methode wird die Länge der *IntArrayList* abgefragt. Die Resultate der *size* Methode sind in der Tabelle 17 zu sehen.

Variante	Zu erwartende Zeitkomplexität	Analysierte Zeitkomplexität
IntArrayListWrapper	Eine einzelne *size* Operation hat eine asymptotische Zeitkomplexität von $O(1)$.	$O(1)$
IntArrayListX1_5	$O(1)$ für die Erklärung siehe Zeile *IntArrayListWrapper*.	$O(1)$
IntArrayList2	$O(1)$ für die Erklärung siehe Zeile *IntArrayListWrapper*.	$O(1)$

TABELLE 17: RESULTATE DER METHODE SIZE DER INTARRAYLIST

4.4.6. DIJKSTRA

Die RPE-Klasse des Dijkstra Algorithmus bereitet in der *setup* Methode jeweils einen Graphen für den Algorithmus vor. Der Graph besteht dabei aus n Knoten und $2n$ Kanten. In der *run* Methode wird Dijkstra Algorithmus für den erstellten Graphen ausgeführt. Die Resultate der Zeitkomplexitätsanalyse sind in der Tabelle 18 zu sehen. Sie sind wie erwartet ausgefallen.

Variante	Zu erwartende Zeitkomplexität	Analysierte Zeitkomplexität
DijkstraPriorityQueue	Die asymptotische Zeitkomplexität des Dijkstra Algorithmus hängt von der Datenstruktur der Kanten und der Liste der zu bearbeitenden Knoten ab. Wenn für die Kanten eine Adjazenzliste und für die noch zu bearbeitenden Knoten eine Priority Queue verwendet wird, erreicht der Dijkstra Algorithmus eine asymptotische Zeitkomplexität von $O(E \log V)$. Dabei entspricht E der Anzahl Kanten und V der Anzahl Knoten [11]. Dies bedeutet das mit $E = 2n$ und $V = n$ eine asymptotische Zeitkomplexität von $O(2n \log n) \Leftrightarrow O(n \log n)$ erwartet wird.	$O(\mathrm{n} \log n)$

DijkstraSet	Die asymptotische Zeitkomplexität des Dijkstra Algorithmus hängt von der Datenstruktur der Kanten und der Liste der zu bearbeitenden Knoten ab. Wenn für die Kanten eine Adjazenzliste und für die noch zu bearbeitenden Knoten ein Set verwendet wird, erreicht der Dijkstra Algorithmus eine asymptotische Zeitkomplexität von $O(E + V^2)$. Dabei entspricht E der Anzahl Kanten und V der Anzahl Knoten [11]. Dies bedeutet das mit $E = 2n$ und $V = n$ eine asymptotische Zeitkomplexität von $O(2n + n^2) \iff O(n^2)$ erwartet wird.	$O(n^2)$

TABELLE 18: RESULTATE DES DIJKSTRA-ALGORITHMUS

4.4.7. RUCKSACKPROBLEM

Die RPE-Klasse des Rucksackproblems definiert in der *setup* Methode n Items welche später in den Rucksack gelegt werden können. Dafür werden zwei zufällige Arrays mit den Werten und dem Gewicht der Items erstellt. Die Werte in beiden Arrays sind zwischen 0 und $4n$. Als maximal Kapazität W für den Rucksack wird ca. ein Achtel aller Items festgelegt ($W = 2n * \frac{n}{8}$). In der *run* Methode wird anschliessend ein Algorithmus ausgeführt, um das Rucksackproblem zu lösen. Die Resultate des Rucksackproblems sind in Tabelle 19 zu sehen.

Variante	Zu erwartende Zeitkomplexität	Analysierte Zeitkomplexität
KnapsackBranchAndBound	Für den Branch and Bound Algorithmus werden in einem ersten Schritt alle Items nach dem Wert pro Gewicht sortiert. Dies hat eine asymptotische Zeitkomplexität von $O(n \log n)$. Anschliessend wird mit Branch and Bound ein Baum durchsucht, was im schlechtesten Fall eine asymptotische Zeitkomplexität von $O(2^n)$ hat [20]. Die effektive asymptotische Zeitkomplexität wird allerdings dank Schnittregel, die den Baum verkleinern, deutlich geringer ausfallen. Im besten Fall muss nur ein Element aus dem Baum bearbeitet werden, dies führt mit dem Sortieren zu einer minimalen asymptotischen Zeitkomplexität von $O(n \log n)$.	Mit den zufälligen Objekten wurde eine asymptotische Zeitkomplexität von $O(n^2)$ evaluiert. Dies Variante ist wie erwartet schneller als der Ansatz mit dynamischer Programmierung.

Variante	Zu erwartende Zeitkomplexität	Analysierte Zeitkomplexität
KnapsackDynamicProgramming	Die dynamisch programmierte Variante hat eine asymptotische Zeitkomplexität von: $O(n * W) = O\left(n * 2n * \frac{n}{8}\right) = O\left(\frac{n^3}{4}\right) \Leftrightarrow O(n^3)$	$O(n^3)$
KnapsackRecursive	In der rekursiven Variante wird das Gewicht und der Wert für alle möglichen Kombinationen rekursive evaluiert was zu eine asymptotischen Zeitkomplexität von $O(2^n)$ führt.	$O(2^n)$

TABELLE 19: RESULTATE DES RUCKSACKPROBLEMS

4.5. RESULTATE DER TESTS

Test	Resultat
BusyWaiting	Erwartete Resultate
Sieb des Eratosthenes	Erwartete Resultate
Quicksort	Erwartete Resultate
Binärzahlen in Dezimalzahlen umwandeln	Erwartete Resultate
IntArrayList	Aufgrund von Mess- oder Evaluationsfehlern bei der *remove* Methode konnte die erwartete Zeitkomplexität nicht evaluiert werden.
Dijkstra	Erwartete Resultate
Rucksackproblem	Erwartete Resultate

TABELLE 20: RESULTATE DES TESTS

Bei fast allen Tests konnten konstante Resultate erzielt werden. Die Resultate waren bis auf wenige Fälle auch korrekt. Anhand des Busy Waiting Tests ist zu sehen, dass die Performanceevaluation imstande ist, jede mögliche Zeitkomplexität zu ermitteln. Die Auswahl der verschiedenen Zeitkomplexitäten, die evaluiert werden können, wurde eingeschränkt. Damit kann trotz Schwankungen in den Ausführungszeiten von Java-Code die Zeitkomplexität konsistent ermittelt werden. Es wurden jedoch auch Schwachstellen aufgedeckt, diese werden in den folgenden Abschnitten erklärt.

Die Ausführungsdauer der Methoden der *IntArrayList* hängt nicht vom Input der jeweiligen Methode ab. Die Methoden *get*, *set* und *size* haben eine konstante Ausführungsdauer. Trotzdem wurde die Länge der Liste von n abhängig gemacht. Falls nun eine Studentenlösung von der Länge der Liste abhängen würde, könnte der Fehler gefunden werden. Ein Student könnte zum Beispiel eine LinkedList einreichen, bei welcher die *get* Methode von der Länge der Liste abhängt.

Die Ausführungsdauer der *remove* und der *add* Methode hängt von der Länge der Liste ab. Weil nicht unendlich viel Speicherplatz zur Verfügung steht, muss die Länge der Liste begrenzt werden. Falls die

Grenze überschritten wird, würde dies zu verfälschten Resultaten führen. Deshalb sind die Resultate der *remove* Methode inkonsistent. Die *add* Methode ist von diesem Problem nicht betroffen. Die *get*, *set* und *size* Methoden haben eine konstante Laufzeit und sind daher ebenfalls nicht betroffen.

Die asymptotische Zeitkomplexität des Dijkstra Algorithmus hängt sowohl von der Anzahl an Kanten als auch von der Anzahl an Knoten ab. Eine von zwei variablen abhängigen Zeitkomplexitäten kann jedoch nicht ermittelt werden. Dieses Problem kann mit einem konstanten Verhältnis der beiden Variablen zu einander umgangen werden. Damit nicht nur mit einem Verhältnis getestet werden kann, könnten mehrere Performance-Tests mit unterschiedlichen Verhältnissen geschrieben werden.

Die Schwachstellen des Performance-Messtools können mit einfachen Anpassungen reduziert werden. Es kann für verschiedene Beispielprobleme, welche Aufgaben aus den Programmier-Grundlagenmodulen der ersten Studienjahre simulieren, die asymptotische Zeitkomplexität ermittelt werden. Damit erfüllt das Performance-Messtool seine grundsätzliche Anforderung.

5. Integration in AutoFeedback

Um die Performanceevaluation für Studenten nutzbar zu machen, soll diese ins Lehrfonds-Projekt «Automatisches Feedback zu Programmieraufgaben» integriert werden. Die Funktionsweise des bestehenden AutoFeedback Projekts wird im Kapitel 1.1 beschrieben. Der Code des AutoFeedback Projekts wurde in verschiedene Projekte unterteilt. In diesem Kapitel wird auf die Änderungen an den bestehenden Projekten, auf die neu dazugekommenen Projekte und auf deren Zusammenspiel eingegangen. Wie die Projekte zusammenhängen ist in Abbildung 3 dargestellt. Jedes der grossen viereckigen Kästchen stellt dabei ein Projekt dar. Die kleinen viereckigen Kästchen zeigen, das ein Projekt von einem anderen Projekt abhängig ist.

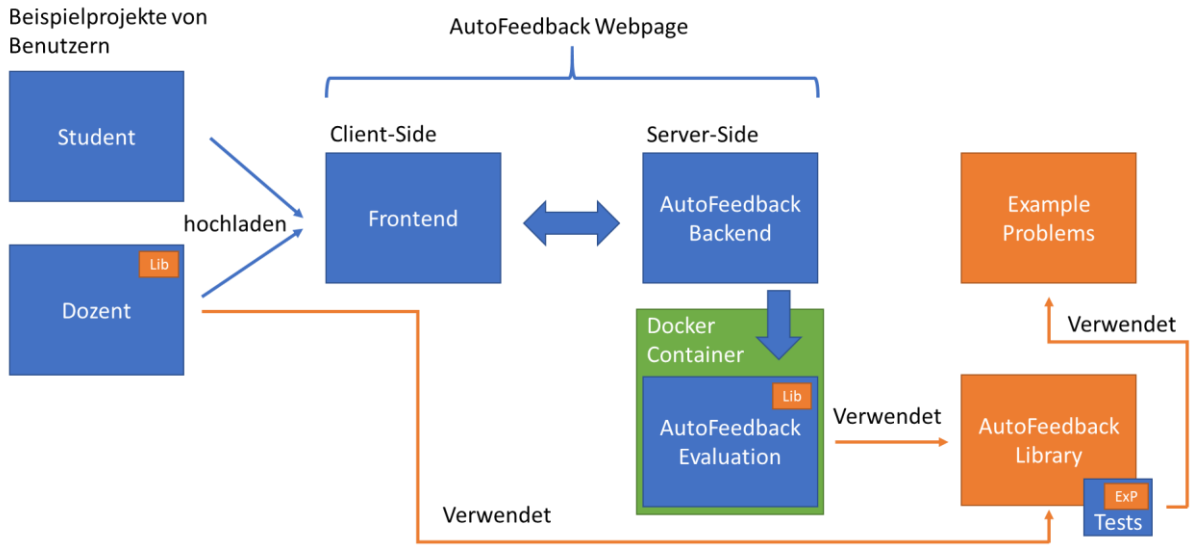


Abbildung 3: Übersicht über die AutoFeedback Projekte

5.1. Example Problems

Das Projekt Example Problems ist ein neues Projekt und beinhaltet alle Beispielprobleme, welche im Rahmen dieser Arbeit entwickelt wurden. Dazu gehören die Interfaces, die verschiedenen Implementierungen und die Testfälle. Auf die einzelnen Beispielprobleme und deren Implementierung wurde im Kapitel 3 eingegangen. Das Projekt Example Problems ist auf dem FHNW GitLab unter https://git-lab.fhnw.ch/autofeedback/performance/exampleproblems zu finden.

5.2. AutoFeedback Library

Das Projekt AutoFeedback Library beinhaltet alle Klassen, welche von verschiedenen anderen Projekten benötigt werden. Das Projekt wurde im Rahmen dieser Arbeit neu erstellt. Es besteht aus Klassen für die Performanceevaluation und aus bereits vorhandenen Klassen, die für das Testen der Korrektheit von Studentenlösungen mit Unit Tests nötig sind.

Die Funktionsweise der Performanceevaluation wurde im Kapitel 4 beschrieben und befindet sich in der Library. Somit kann sowohl vom AutoFeedback Evaluation Projekt als auch vom Dozenten Projekt darauf zugegriffen werden.

Die Performanceevaluation wird, wie im Kapitel 4.4 beschrieben und anhand der Beispielprobleme getestet. Die dafür benötigten RPE-Klassen und die Tests befinden sich im Test Ordner der Library. Für die Ausführung der Tests wird das Projekt Example Problems mit den Implementierungen der

Beispielprobleme benötigt. Das Projekt AutoFeedback Library ist auf dem FHNW GitLab unter https://gitlab.fhnw.ch/autofeedback/autofeedback-library zu finden.

5.3. Frontend

Im Projekt Frontend befindet sich die AutoFeedback React Webseite. Auf der Webseite können von Dozenten neue Tasks mit Testfällen erstellt werden. Anschliessend können die Studenten dort ihre Lösungen bewerten lassen. Dafür werden die Unit Tests und die Performance Tests, welche vom Dozenten im Task eingerichtet wurden, auf die Studentenlösung angewendet. Das Frontend Projekt ist auf dem FHNW GitLab unter https://gitlab.fhnw.ch/autofeedback/frontend zu finden.

Die Webseite wurde im Lehrfonds-Projekt «Automatisches Feedback zu Programmieraufgaben» erstellt. Die Webseite wurde damals nicht auf einen Stand gebracht, der veröffentlicht werden könnte. Es fehlen dafür wichtige Funktionen und Dokumentationen: das Editieren und Löschen von jeglichen Elementen; die Möglichkeit Accounts zu erstellen und erstellte Accounts wieder zu löschen. Das Routing zwischen den verschiedenen Ansichten funktioniert nicht einwandfrei. Es sind einige optische Verbesserungen im Design notwendig, sowie Zusatzfunktionen, welche das Benutzen der Webseite vereinfachen. Eine generelle Benutzeranleitung fehlt, sowie Beispiele für das Erstellen der Testfälle und das Einreichen von Studentenlösungen. Die Verbesserung der Webseite ist nicht Teil dieser Arbeit. Die Performanceevaluation soll jedoch ins Projekt integriert und die Resultate sollen auf der Webseite angezeigt werden. Die Anzeige der Test Resultate wurde, wie in Abbildung 4 zu sehen, um die Performance Resultate erweitert.

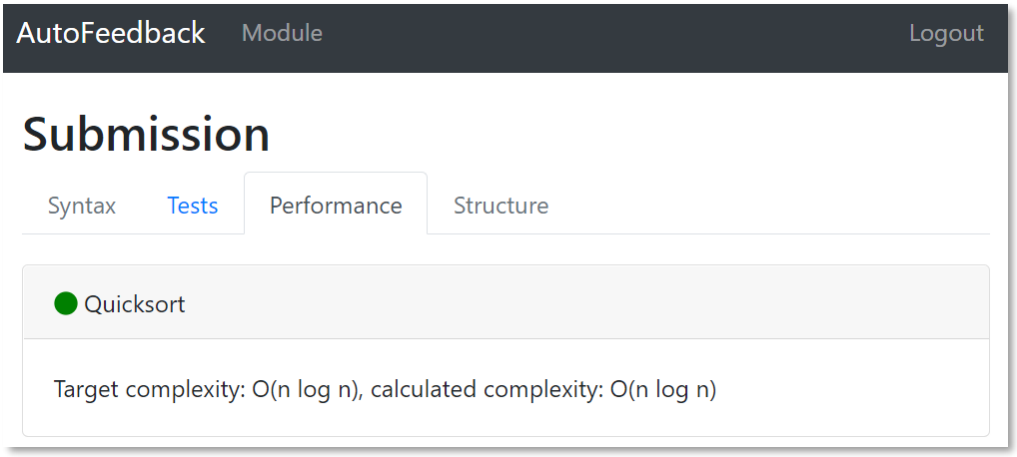


Abbildung 4: View Performance Resultate

Die Submissionsliste wurde um den aktuellen Status der Evaluation und um eine Ampel, ob alle Tests korrekt waren erweitert. Die Liste wird neu jede Sekunde neu geladen, um das Resultat der noch laufenden Tests schnellstmöglich anzeigen zu können. Die neue Submissionsliste ist in Abbildung 5 zu sehen.

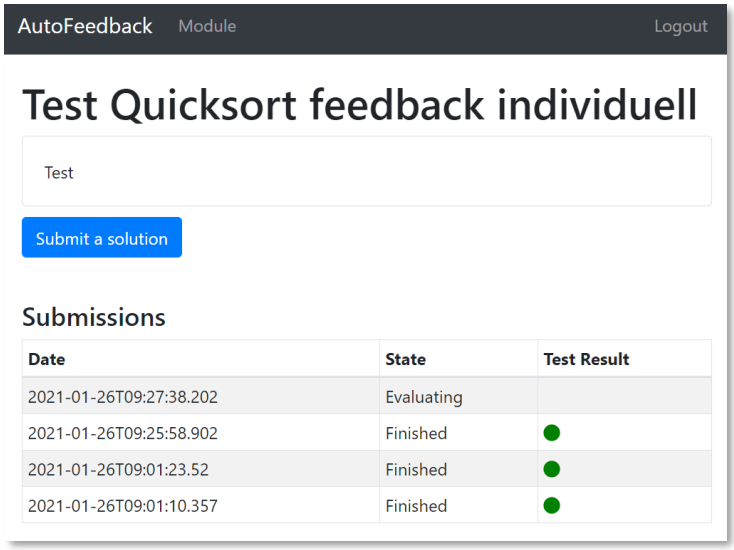


ABBILDUNG 5: VIEW SUBMISSIONSLISTE

Fehler im Routing wurden behoben und die Webseite um einen Logout Button erweitert.

5.3.1. ERSTELLEN EINES NEUEN TESTS

In einem Modul können neue Tests erstellt werden. Jeder Test hat einen Namen und eine Beschreibung. Dabei können unter «Modules» verschiedene Module ausgewählt werden, in welchen dieser Test angezeigt und von den Studenten angewendet werden kann. Anschliessend wird eine Zip Datei mit den Testfällen des Dozenten hochgeladen. Unter «Test Classes» werden alle Klassen der zu verwendenden Tests, inklusive der maximalen Ausführungszeit für den Test angegeben. Das Formular für die Erstellung eines Tests ist in Abbildung 6 ersichtlich.

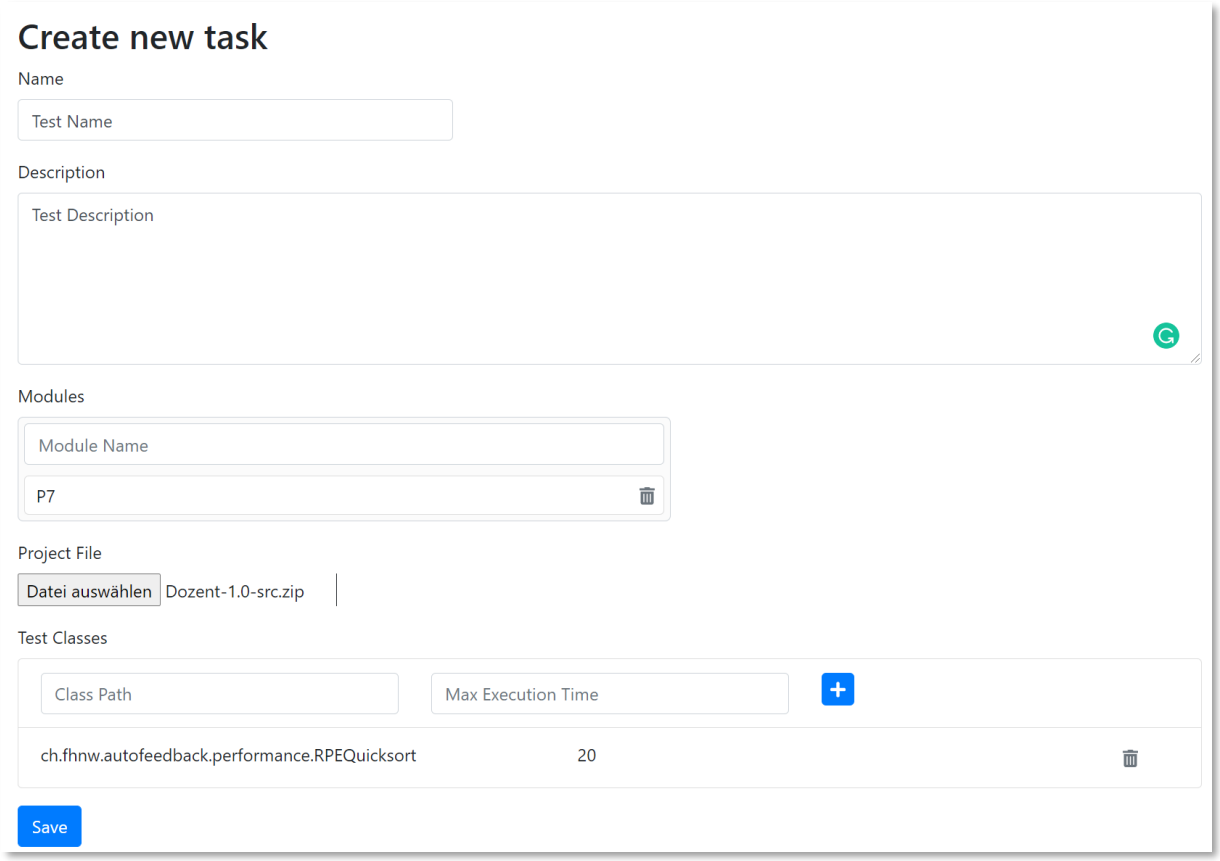


ABBILDUNG 6: TEST ERSTELLEN

5.3.2. HOCHLADEN EINER STUDENTEN LÖSUNG

Ein Student kann seine Lösung als Zip Datei für einen bestimmten Test hochladen. Die Abgabe erfolgt über den «Submit a Solution» Button eines Tests, wie er in Abbildung 5 zu sehen ist. Anschliessend kann er seine Zip-Datei auswählen, mit «Submit» hochladen und testen lassen. Das Formular zum Hochladen einer Lösung ist in Abbildung 7 zu sehen.

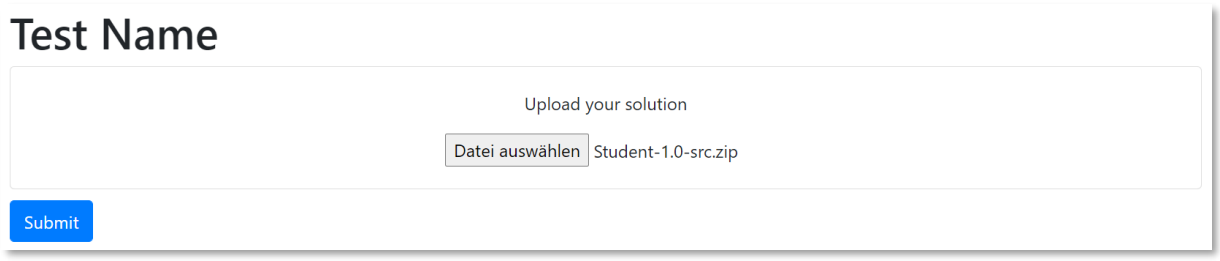


ABBILDUNG 7: HOCHLADEN EINER STUDENTENLÖSUNG

5.4. AUTOFEEDBACK BACKEND

Das AutoFeedback backend ist das Spring Boot Java backend zur React Webseite und wurde im Rahmen des Lehrfonds-Projekts «Automatisches Feedback zu Programmieraufgaben» entwickelt. Das AutoFeedback backend Projekt ist auf dem FHNW GitLab unter https://gitlab.fhnw.ch/autofeedback/autofeedback-backend zu finden.

Die Studentenlösung wird nicht direkt im Backend getestet. Die Studentenlösung und die Testfälle der Dozenten werden in einem Docker Container geladen. Dort werden sie kompiliert und vom

Performance-Messtool getestet. Dadurch können Seiteneffekte des Kompilierens und das Ausführen des Studenten-/ Dozentencodes das Backend nicht beeinträchtigen. Weil die Evaluation der Lösungen nicht im Backend stattfindet, muss dieses für die Integration der Performanceevaluation kaum angepasst werden. Damit die Testresultate auf der Webseite in die Kategorien «Tests» und «Performance» aufgeteilt werden können, wird ein zusätzliches Attribut Kategorie bei den Testresultaten benötigt. Dieses Attribut wird neu vom Backend ebenfalls ans Frontend gesendet.

5.5. AutoFeedback Evaluation

Die Studentenlösungen und die Testfälle der Dozenten werden wie bereits erwähnt, in einem Docker Container kompiliert und anschliessend vom der AutoFeedback Evaluation ausgeführt. Für jeden Test wird erst die Testklasse des Dozenten mit reflection geladen. Anschliessend wird überprüft, ob es sich um eine Performance oder einen Unit Test handelt. Der Test wird ausgeführt und die Resultate werden in die Datenbank gespeichert.

Die AutoFeedback Evaluation wurde im Rahmen des Lehrfonds-Projekts «Automatisches Feedback zu Programmieraufgaben» entwickelt und jetzt um das in Kapitel 4 beschriebene Performance-Messtool erweitert. Das AutoFeedback Evaluation Projekt ist auf dem FHNW GitLab unter https://gitlab.fhnw.ch/autofeedback/autofeedback-backend-evaluation zu finden.

5.6. Student

Im Projekt Student befindet sich ein Beispielprojekt eines Studenten. Dieses Projekt soll im Zusammenspiel mit dem Projekt Dozent (sehe Kapitel 5.7) zum Test der AutoFeedback Applikation dienen. Damit ein Studentenprojekt von der AutoFeedback Applikation bewertet werden kann, muss dieses auf die Webseite hochgeladen werden. Dafür wird der «src» Ordner mit allen Klassen und die «build.gradle» Datei zusammen gezipt und hochgeladen. Die hochgeladene ZIP Datei wird vom AutoFeedback Projekt in einem Docker Container entpackt und anhand der Gradle Datei kompiliert.

Dieses Projekt wurde um eine Lösung für das Quicksort Problem erweitert. Dafür wurde sowohl das *Quicksort* Interface als auch die Implementierung von *QuicksortHoare* unverändert aus dem Kapitel 3.2 in dieses Projekt übernommen. Das Projekt Student ist auf dem FHNW GitLab unter https://gitlab.fhnw.ch/autofeedback/reflection-student zu finden.

5.7. Dozent

Im Projekt Dozent befindet sich ein Beispielprojekt eines Dozenten. Darin werden Tests für die Studentenlösung definiert, welche in der AutoFeedback Applikation getestet werden sollen. Damit ein Dozentenprojekt von der AutoFeedback Applikation zum Bewerten von Studentenlösungen verwendet werden kann, muss auf der Webseite ein neuer Test erstellt und das Projekt hochgeladen werden. Dieser Vorgang wird in Kapitel 5.3.1 erklärt. Von einem Dozentenprojekt wird der «src» Ordner mit allen Klassen und die *build.gradle* Datei zusammen gezipt und hochgeladen. Die hochgeladene ZIP Datei wird vom AutoFeedback Projekt in einem Docker Container entpackt und anhand der Gradle Datei kompiliert. Damit ein Performance Test entwickelt werden kann, werden verschiedene Klassen wie die Klasse *RunPerformanceEvaluation,* die für einen Test erweitert werden soll, benötigt. Alle benötigten Klassen befinden sich im Projekt AutoFeedback Library. Dieses Projekt kann in der Gradle Datei als Abhängigkeit angegeben werden. Dabei ist zu beachten, dass alle Abhängigkeiten, die in der Gradle Datei definiert werden, auch im Docker Container geladen werden können. Das Projekt Dozent ist auf dem FHNW GitLab unter https://gitlab.fhnw.ch/autofeedback/reflection-teacher zu finden.

5.7.1. Implementierung eines Performance Tests

Ein Performance Test erweitert die Klasse *RunPerformanceEvaluation* wie in Kapitel 4.1 erklärt wird. Eine solche RPE-Klasse wird erstellt, um eine Studentenlösung zu testen. Eine RPE-Klasse benötigt jedoch einige Anpassungen, damit eine Studentenlösung aus einem anderen Projekt getestet werden kann. Im Konstruktor der RPE-Klasse wird die Studentenlösung mit Reflection in das Projekt geladen und ein Objekt erstellt. Der Konstruktor kennt das Studentenprojekt jedoch nicht. Deshalb bekommt der Konstruktor den Speicherort des Studentenprojekts als String übergeben. Anschliessend kann mit der Methode *getProxyObjectOfStudentClass* aus der Klasse *Util* des AutoFeedback Library Projekts ein Objekt der Studentenlösung erstellt werden. Die Methode *getProxyObjectOfStudentClass* benötigt dafür den Speicherort, den Klassennamen inklusive Package und ein Interface der Klasse, die geladen werden soll.

Im Dozentenprojekt wird die *RPEQuicksort* Klasse aus Kapitel 4.4.3 übernommen um das Studentenprojekt aus Kapitel 5.6 zu testen. Der Konstruktor wird dabei wie oben beschrieben angepasst. In der *RPEQuicksort* Klasse wird die Methode *getProxyObjectOfStudentClass* mit «*ch.fhnw.autofeedback.exampleproblems.sort.QuicksortHoare*» als Klassenname, das *Quicksort* Interface als Interface und dem Speicherort der Studentenlösung aufgerufen. Als Resultat wird eine Instanz der Studentenklasse, in diesem Fall der *QuicksortHoare* Klasse, zurückgegeben. Die *setup* und *run* Methode können ohne Änderung übernommen werden.

Ein Dozent hat die Möglichkeit die *String feedback()* Methode der abstrakten Klasse *RunPerformanceEvaluation* in seiner RPE-Klasse zu überschreiben. Der Output dieser Methode wird gespeichert und dem Studenten, nach der Evaluation, auf der Webseite angezeigt. Standardmässig gibt diese Methode folgenden Text zurück “Target complexity: {TARGETCOMP}, calculated complexity: {CALCCOMP}”. Die Werte «{TARGETCOMP}» und «{CALCCOMP}» werden dabei durch die gesuchte und die evaluierte Zeitkomplexität vor dem Speichern ersetzt. Durch das Überschreiben dieser Methode hat ein Dozent die Möglichkeit den Studenten ein angepasstes Feedback oder Tipps für ihre Implementierungen weiterzugeben.

5.7.2. Asymptotische Zeitkomplexität einer Musterlösung Testen

Ein Dozent gibt bei einem Test die geforderte asymptotische Zeitkomplexität an. Falls er seinen Test überprüfen möchte oder die asymptotische Zeitkomplexität seiner Musterlösung berechnen möchte, kann er dies auf zwei Arten tun. Option eins: er lädt, wie die Studenten, seine Lösung auf die Webseite und lässt sie dort bewerten. Option zwei: Er erledigt dies direkt im Dozentenprojekten. Dafür kann er eine Main in einer beliebigen Klasse erstellen. Darin wird ein Objekt seiner RPE-Klasse erstellt. Der Konstruktor der RPE-Klasse verlangt als Parameter den Speicherort der Studentenlösung. Dieser entspricht in diesem Fall demjenigen der Musterlösung. Anschliessend kann, auf dem Objekt der RPE-Klasse die Methode *calculateComplexity* ausgeführt werden. Diese berechnet danach die asymptotische Zeitkomplexität und gibt das Resultat auf der Konsole aus.

Im Projekt Dozent befindet sich die Main Methode direkt in der *RPEQuicksort* Klasse und lädt die Musterlösung *QuicksortHoare* aus dem Projekt. Als Speicherort der Musterlösung kann dabei "/" angegeben werden.

6. DISKUSSION

Verschiedene bestehende Tools zur Performanceanalyse wurden auf ihre Tauglichkeit für dieses Projekt geprüft. Die Analyse hat ergeben, dass keines dieser Tools die asymptotische Zeitkomplexität einer Methode messen kann. Die meisten Tools sind dazu gedacht die Performance von verschiedenen Implementierungen zu vergleichen oder Zeitmessungen vorzunehmen. Ausser JMH waren alle Tools bereits ein paar Jahre alt. Zudem sind die Tools kompliziert, nicht sehr verbreitet und nicht genügend gut dokumentiert. Deshalb wurde von der Verwendung dieser Tools abgesehen.

Damit ein Performance-Messtool getestet werden kann, wurden verschiedene Varianten von sechs Beispielproblemen implementiert. Die Beispielprobleme waren sehr nützlich bei der Implementierung und Verbesserung des Performance-Messtools. Es konnten Fehler festgestellt, verbessert und behoben, sowie die Möglichkeiten und Grenzen des fertigen Performance-Messtools aufgezeigt werden.

Mit dem erstellten Performance-Messtool kann die Performance von verschiedenen Algorithmen mithilfe eines Performance Tests gemessen und die asymptotische Zeitkomplexität ermittelt werden. Damit konsistente Resultate ermittelt werden können, wurde die Anzahl an Zeitkomplexitäten auf zehn der meist Gebrauchten eingeschränkt. Das Performance-Messtool wurde mit den sechs Beispielproblemen getestet. Bei fünf konnte die erwartete Zeitkomplexität ermittelt werden. Beim sechsten Beispielproblem hängt die Ausführungszeit des Algorithmus nicht vom Input ab, dies führt zu Fehlern in der Analyse. Die asymptotische Zeitkomplexität von Algorithmen, bei welchen die Komplexität von mehr als einer Variablen abhängt, wie z.B. dem Dijkstra Algorithmus, kann korrekt ermittelt werden, wenn ein konstantes Verhältnis zwischen den Variablen besteht. Sowohl die Anzahl an Zeitkomplexitäten, die analysiert werden können, als auch eine Analyse der Zeitkomplexität, die von mehreren Variablen abhängig ist, könnte in Zukunft realisiert werden. Damit weiter konsistente und korrekte Resultate mit dieser Erweiterung möglich wären, müssten die Schwankungen in den Zeitmessungen reduziert werden. Dabei könnte eventuell ein Tool wie JMH helfen. Dies würde jedoch die Laufzeit einer Analyse massiv verlangsamen und damit die Benutzerfreundlichkeit senken.

Das erstellte Performance-Messtool konnte in das Lehrfonds-Projekts «Automatisches Feedback zu Programmieraufgaben» integriert werden. Damit können Performance Tests von Dozenten entwickelt und den Studenten zum Testen zur Verfügung gestellt werden. Zudem haben Dozenten die Möglichkeit ihre Musterlösungen zu analysieren und ihre Tests darauf anzupassen. Die Webseite sollte in Zukunft auf einen Stand gebracht werden, der veröffentlicht werden kann. Dazu sind noch einige Usability Probleme und Fehler zu beheben.

ANHANG

LITERATUR

[1] OpenJDK, «Java Microbenchmark Harness (JMH),» GitHub repository, [Online]. Available: https://github.com/openjdk/jmh/tree/1.26. [Zugriff am 25 Januar 2021].

[2] «JMH Core 1.27 API,» Oracle, [Online]. Available: https://javadoc.io/doc/org.openjdk.jmh/jmh-core/latest/index.html. [Zugriff am 25 Januar 2021].

[3] Google, «Caliper,» Google, [Online]. Available: https://github.com/google/caliper. [Zugriff am 25 Januar 2021].

[4] javatlacati, «ContiPerf,» GitHub repository, [Online]. Available: https://github.com/javatlacati/contiperf. [Zugriff am 25 Januar 2021].

[5] noconnor, «JUnitPerf,» GitHub repository,, [Online]. Available: https://github.com/noconnor/JUnitPerf. [Zugriff am 25 Januar 2021].

[6] D. o. D. a. D. Systems, «Performance Awareness in Software Development,» Faculty of Mathematics and Physics, Charles Univercity, [Online]. Available: https://d3s.mff.cuni.cz/research/performance-awareness/. [Zugriff am 25 Januar 2021].

[7] SemaiCZE, «Stochastic Performance Logic evaluation engine,» GitHub repository, [Online]. Available: https://github.com/SemaiCZE/spl-evaluation-java. [Zugriff am 25 Januar 2021].

[8] D. o. D. a. D. Systems, «Microbenchmarking Agent for Java,» GitHub repository, [Online]. Available: https://github.com/D-iii-S/java-ubench-agent. [Zugriff am 25 Januar 2021].

[9] Wikipedia contributors, «Sieve of Eratosthenes,» Wikipedia, The Free Encyclopedia., 12 Januar 2021. [Online]. Available: https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes. [Zugriff am 25 Januar 2021].

[10] Wikipedia contributors, «Quicksort,» Wikipedia, The Free Encyclopedia., 23 Januar 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Quicksort&oldid=1002204629. [Zugriff am 25 Januar 2021].

[11] Wikipedia contributors, «Dijkstra's algorithm,» Wikipedia, The Free Encyclopedia., 24 Januar 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Dijkstra%27s_algorithm&oldid=1002424875. [Zugriff am 25 Januar 2021].

[12] Wikipedia contributors, «Knapsack problem,» Wikipedia, The Free Encyclopedia., 24 Januar 2021. [Online]. Available:

https://en.wikipedia.org/w/index.php?title=Knapsack_problem&oldid=1002537677. [Zugriff am 25 Januar 2021].

[13] GeeksforGeeks, «0/1 Knapsack using Branch and Bound,» GeeksforGeeks, 20 November 2018. [Online]. Available: https://www.geeksforgeeks.org/0-1-knapsack-using-branch-and-bound/. [Zugriff am 25 Janauar 2021].

[14] GeeksforGeeks, «0-1 Knapsack Problem | DP-10,» GeeksforGeeks, 3 November 2020. [Online]. Available: https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/. [Zugriff am 25 Januar 2021].

[15] V. Horký, P. Libic, A. Steinhauser und P. Tuma, «DOs and DON'Ts of Conducting Performance Measurements in Java,» in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, Austin, TX, USA, January 31 - February 4, 2015*, 2015.

[16] P. Pufek, H. Grgic und B. Mihaljevic, «Analysis of Garbage Collection Algorithms and Memory Management in Java,» in *42nd International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2019, Opatija, Croatia, May 20-24, 2019*, 2019.

[17] S. Goldsmith, A. Aiken und D. S. Wilkerson, «Measuring empirical computational complexity,» in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, 2007.

[18] Wikipedia contributors, «Linear regression,» Wikipedia, The Free Encyclopedia., 13 Januar 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Linear_regression&oldid=1000082798. [Zugriff am 25 Januar 2021].

[19] Wikipedia contributors, «Coefficient of determination,» Wikipedia, The Free Encyclopedia., 10 Januar 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Coefficient_of_determination&oldid=999556832. [Zugriff am 25 Januar 2021].

[20] D. R. Morrison, S. H. Jacobson, J. J. Sauppe und E. C. Sewell, «Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning,» *Discrete Optimization,* Bd. 19, pp. 79-102, 2016.

[21] P. Stefan, V. Horky, L. Bulej und P. Tuma, «Unit Testing Performance in Java Projects: Are We There Yet?,» in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, New York, NY, USA, 2017.

[22] F. Hewson, J. Dietrich und S. Marsland, «Performance Regression Testing on the Java Virtual Machine Using Statistical Test Oracles,» in *24th Australasian Software Engineering Conference, ASWEC 2015, Adelaide, SA, Australia, September 28 - October 1, 2015*, 2015.

[23] A. Georges, D. Buytaert und L. Eeckhout, «Statistically rigorous java performance evaluation,» in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, 2007.

[24] H. Li, M. Wu und H. Chen, «Analysis and Optimizations of Java Full Garbage Collection,» in *Proceedings of the 9th Asia-Pacific Workshop on Systems, APSys 2018, Jeju Island, Republic of Korea, August 27-28, 2018*, 2018.

Tabellen Verzeichnis

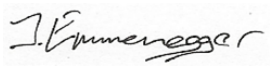
Abbildung Verzeichnis

EHRLICHKEITSERKLÄRUNG

Hiermit erkläre ich, den vorliegenden Projektbericht P7 AutoFeedback – Performance für selbständig, ohne Hilfe Dritter und nur unter Benutzung der angegebenen Quellen verfasst zu haben.

Küttigen, 05.02.2021

Ort, Datum



Joel Emmenegger