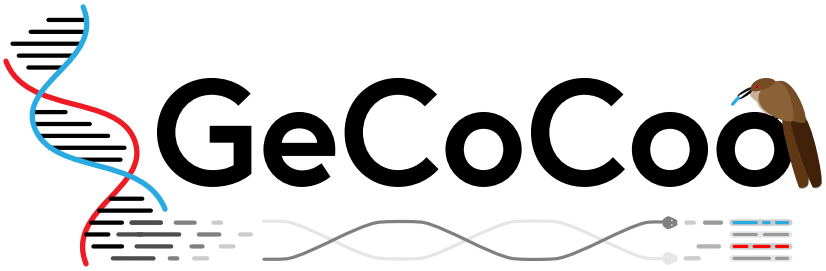
Abschlussbericht der Projektgruppe 633

GeCoCoo



Teilnehmer:
Dennis Martin Bednarek
Alwin Berger
Thomas Alessandro Buse
Sven Deichsel
Niklas Domagalla
Leon Motzet
Erik Rieping
Fabian Sieber
Jakob Vogt
Felix Wiegand
Jannik Wolff

Betreuer:
Prof. Dr. Sven Rahmann
Jens Zentgraf
Elias Kuthe

Datum:
24. April 2021

Technische Universität Dortmund
Fakultät für Informatik
Ls11: Algorithm Engineering
`http://ls11-www.cs.tu-dortmund.de`

Inhaltsverzeichnis

Kapitel 1 — Einleitung

Der vorliegende Bericht fasst das Vorgehen, Entwicklungen und die Ergebnisse der Projektgruppe (PG) „Codegenerierung für schnelle Cuckoo-Hashing-Varianten in Genomanalysen“ (GeCoCoo) zusammen.

1.1 Motivation

Als Genomanalyse wird ein Verfahren zur Analyse des Erbguts bezeichnet. Sie wird in der biologischen Forschung, vor allem aber in der Medizin im Bereich der Humangenetik verwendet. Zu den Anwendungen gehören die Feststellung von Dispositionen gegenüber erblichen Krankheiten, pränatale Diagnostik und Identifizierung von Personen und Verwandtschaftsverhältnissen.

Bei der Genomanalyse können große Mengen von Daten anfallen, die verarbeitet werden müssen. So weist das menschliche Genom über 3 Milliarde Basenpaare auf. Die Verarbeitung geschieht oft in mehreren Schritten mit einer Vorverarbeitung vor der eigentlichen Anwendung. Dies stellt auch für die moderne Datenverarbeitung eine Herausforderung dar, insbesondere, wenn viele Datensätze miteinander verglichen werden. Aufgrund der Menge an Daten ist nicht nur die Geschwindigkeit der Verarbeitung eine Herausforderung, sondern auch der Speicherbedarf während der Verarbeitung. Ein Teil der Vorverarbeitung von Gensequenzdaten ist die Qualitätskontrolle. Diese Kontrolle ist notwendig, da bei der Sequenzierung von Genen Fehler nie gänzlich vermeidbar sind. Dabei werden verschiedene Analysen durchgeführt, um sicherzustellen, dass die Daten und später daraus hergeleitete Erkenntnisse zuverlässig sind.

1.2 Zielsetzung

Die Herausforderungen bei der Verarbeitung von Genomdaten sind, aufgrund der hohen Menge an Daten, sowohl die Geschwindigkeit der Verarbeitung als auch die effiziente Speicherung der Daten. Im Rahmen dieser Projektgruppe soll also eine speichereffiziente Hashtabelle entwickelt werden, die nach den Wünschen des Benutzers spezifiziert und generiert werden kann und schnellen Zugriff auf die Daten ermöglicht. Die genetischen Daten, die hier gespeichert werden sollen, sind kurze Stücke des Erbguts, sogenannte k-mere (erläutert in Kapitel 2.1.1). Diese k-mere sollen also mitsamt ihren assoziierten Informationen in der Hashtabelle gespeichert und schnell auf sie zugegriffen werden können. Aufgrund dieser spezifischen Form von genetischen Daten können dazu hoch angepasste Algorithmen verwendet werden. Anschließend soll ein Tool erstellt werden, das diese Hashtabelle nutzt, um genetische Daten einer Qualitätskontrolle zu unterziehen. Dieses Tool soll gegebene Daten nach verschiedenen Merkmalen und Statistiken beurteilen, um damit Aufschluss über die Brauchbarkeit und Verlässlichkeit der Daten zu geben.

1.3 Aufbau des Berichts

Im Rahmen dieses Berichts wird zuerst in Kapitel 2 auf die thematischen Grundlagen eingegangen. Zu diesem Zwecke werden die biologischen Grundbegriffe erläutert und die verschiedenen verwendeten Werkzeuge vorgestellt. Ebenso werden das Hashing und Methoden der Arbeitsorganisation erläutert.

Danach werden die drei Miniprojekte in Kapitel 3 vorgestellt, die der ersten Erarbeitung der Themen für die Projektgruppe und zur Implementierung einiger Funktionen dienen, die im weiteren Verlauf benötigt werden. Die Miniprojekte beschäftigen sich mit dem Einlesen und dem Verarbeiten von genetischen Daten sowie der Beschleunigung dieser Verarbeitung.

Zur effizienten Verarbeitung der Genomdaten wurde auf die Just-in-time-Kompilierung und die Verwendung von *Numba* gesetzt, das im Abschnitt 2.3.3 näher erläutert wird. Die Just-in-time-Kompilierung ermöglicht die Generierung von effizienterem Code, da Parameter, die zur Laufzeit übergeben werden, als Konstanten im kompilierten Code verwendet werden können. In einem Miniprojekt wird die Verwendung von JIT-annotierten Klassen und Funktionen verglichen. Beim Vergleich stellte sich die Funktionen wie in Abschnitt 3.1.5 als signifikant schneller heraus. Somit wurde in den folgenden Projekten auf JIT-annotierte Klassen verzichtet. Diese Erkenntnis fand insbesondere bei der Implementierung der Hashtabelle Verwendung. Die Verwendung von *FASTQ*-Dateien ist bei der Verarbeitung von Genomdaten elementar. In einem weiteren Miniprojekt wird das parallelisierte Einlesen und Verarbeiten thematisiert. Im dritten Miniprojekt wird die Erstellung eines k-mer Processors vorgestellt, der die Grundlage für die folgenden Projekte Low Complexity Filter und *YAFQC* darstellt. Bei der Umsetzung des k-mer Processors werden zunächst unterschiedliche Encodierungsmethoden für k-mere aus einer DNA-Sequenz verglichen, wobei sich die Übersetzung mittels des 2-Basen-Arrays auf dem Eingabetyp *NumPy*-Array als effizienteste Lösung hinsichtlich der Laufzeit herausstellt. Ebenso wird die Implementierung des k-mer Processors besprochen.

Im Weiteren wird in Kapitel 4 der Low Complexity Filter vorgestellt. Genetische Daten weisen häufig Regionen geringer Komplexität auf, die Analysen auf diesen Daten verzerren können. Um Verzerrungen der Ergebnisse zu reduzieren, dient der Low Complexity Filter dazu, Regionen mit geringer Komplexität aus einem gegebenen Datensatz zu filtern. Der umgesetzte Low Complexity Filter konnte als Grundlage für das Tool *YAFQC* genutzt werden.

Die erarbeiteten Konzepte für die Hashtabelle werden im darauf folgenden Kapitel 5 vorgestellt. Hierbei wird insbesondere auf die konfigurierbare Generierung der Hashtabelle, der Umgang mit dieser über eine Schnittstelle, sowie die Verwendung von Value Modulen zur Schlüssel-Wert-Paar Berechnung und Hashfunktionen zum Ermitteln der Position eines Schlüssels in der Hashtabelle eingegangen.

Aufbauend auf den vorherigen Konzepten werden Details zur Implementierung der Hashtabelle aus dem zweiten Semester im Kapitel 6 erläutert. Zusätzlich werden die Speicherabstraktion sowie die zugrundeliegenden Hashtabellenoperationen besprochen. Außerdem werden Benchmarks für verschiedene Konfigurationen der Hashtabelle vorgestellt.

Im Kapitel 7 wird ein Tool zur Qualitätssicherung von *FASTQ*-Dateien vorgestellt. Dazu werden existierende Tools auf ihre Funktionalität analysiert. Darauf folgend wird auf die Implementierung des Tools, die umgesetzten Qualitätsmerkmale sowie die Kontaminationsprüfung unter Nutzung der implementierten Hashtabelle, eingegangen. Zudem werden ein Benchmark und ein Vergleich zu anderen existierenden Tools vorgestellt.

Abschließend wird in Kapitel 8 ein Fazit über die Ergebnisse der Projektgruppe gezogen.

Kapitel 2 — Grundlagen

In diesem Kapitel werden zuerst biologische Grundlagen erklärt, die zum Verständnis der behandelten Daten und Abläufe notwendig sind. Daraufhin gibt es einen Einblick in die Funktionsweise des Hashings, wobei insbesondere auf die Variante Cuckoo Hashing eingegangen wird. Zuletzt werden Werkzeuge und Arbeitsweisen vorgestellt, welche, insbesondere für die Umsetzung des Projektes und die kollaborative Arbeit, von den Mitgliedern der Projektgruppe verwendet werden.

2.1 Datenverwaltung in der Bioinformatik

Die Projektgruppe befasst sich im Folgenden mit Themen der Bioinformatik. Daher soll dieses Unterkapitel einen Einblick in die biologischen Hintergründe, die verwendeten Formate und die Quellen der zu verarbeitenden Daten geben.

2.1.1 Biologische Grundlagen

Als Genom werden die Erbinformationen eines lebenden Organismus bezeichnet [24, 6]. Die Informationen werden dabei in der Desoxyribonukleinsäure (DNS) oder auf Englisch *deoxyribonucleic acid* - im Folgenden nur noch als DNA bezeichnet - gespeichert. Beim Menschen wird Ribonukleinsäure (RNS) bzw. auf Englisch *ribonucleic acid* - im Folgenden nur noch als RNA bezeichnet - aus der DNA transkribiert, also synthetisiert. RNA dient unter anderem der Proteinsynthese und kann so in Form der *messenger RNA* als Zwischenspeicher der Information verstanden werden. Damit sind DNA und RNA ein wesentlicher Bestandteil bei der Funktion von zellulären Prozessen. Bei manchen Organismen kann dieses Zusammenspiel in umgekehrter Form vorliegen, sodass die RNA der Träger der Erbinformation ist und die DNA zur Proteinsynthese genutzt wird.

Aufbau

Sowohl DNA als auch RNA sind Makromoleküle, die sich aus Nukleotiden zusammensetzen. Man spricht dabei von Polynukleotiden. Wobei DNA doppelsträngig und RNA zumeist einzelsträngig vorliegt. Jedes Nukleotid besteht dabei aus einem Zucker, einem Phosphorsäurerest und einer Base. Im Falle der DNA kommt als Zucker Desoxyribose, bei der RNA die Ribose vor. Es gibt fünf verschiedene Basen: *Cytosin* (C), *Guanin* (G), *Adenin* (A), *Thymin* (T) und *Uracil* (U), wobei Uracil nur in RNA und Thymin nur in DNA vorkommt. Jede Base weist Verbindungen in Form von schwachen Wasserstoffbrückenbindungen zu seinem jeweiligen Komplement auf. Zwischen A-T liegen zwei Wasserstoffbrücken und G-C drei Wasserstoffbrücken vor. Aufgrund dieser Basenpaarungen und des doppelsträngigen Vorliegens der DNA-Moleküle bilden diese zwei gegenläufige Nukleotidstränge, die sich spiralförmig umeinander winden, aus. Diese Doppelhelix kann als Leiter betrachtet werden, bei der die Wasserstoffbrücken als Sprossen angesehen werden können. Die Erbinformation liegt so in doppelter Form vor. Bei der DNA-Replikation können anhand des komplementären Nukleotidstrangs fehlerhafte Basenpaarungen erkannt werden.

Codierung

Die Basen codieren die Erbinformationen, wobei DNA-Sequenzen Zeichenketten sind, die aus dem Alphabet $\{A, C, G, T\}$ gebildet werden. Ein k-mer hingegen bezeichnet eine zusam-

menhängende Teilsequenz von Basen der Länge k. Demnach ist „ATG“ ein Beispiel für ein 3-mer und „ATGC“ ein Beispiel für ein 4-mer. Die Menge aller k-mere einer DNA-Sequenz besteht also aus allen zusammenhängenden Teilsequenzen der Länge k. Für die DNA-Sequenz oder Zeichenkette „GATCTAG“ repräsentiert die Menge $\{GAT, ATC, TCT, CTA, TAG\}$ die Menge der 3-mere (also k-mere für $k = 3$). Aufgrund der vier unterschiedlichen Basen der DNA ergeben sich so 4^k unterschiedliche k-mere. Die Anzahl der k-mere in einer DNA-Sequenz der Länge n ist somit $n - k + 1$.

Reverse Complement

Ein reverse Complement (deutsch: *umgekehrtes Komplement*) ist eine DNA-Sequenz, die bei umgekehrter Leserichtung und Komplementbildung aus einer Quellsequenz entsteht. Im Kontext von DNA-Sequenzen bezeichnet das Komplement die Ersetzung jeder Base durch ihr zugehöriges Inverses. So wird jedes A durch ein T, jedes C durch ein G, jedes G durch ein C und jedes T durch ein A ersetzt. Die DNA-Sequenz „GATTACA“ besitzt so das reverse Complement „CTAATGT“. Diese Sequenzen sind für die Verarbeitung von k-meren von Bedeutung, da dieses reverse Complement in der Natur praktisch immer auf dem gegenüberliegenden DNA-Strang vorkommen. Bei der Verarbeitung von DNA-Sequenzen sind diese gegebenenfalls explizit zu ermitteln.

2.1.2 Dateiformate

Für die Speicherung von Sequenzdaten existieren in der Bioinformatik unterschiedliche Dateiformate. Im Folgenden sollen die in dieser Projektgruppe verwendeten Formate vorgestellt und erklärt werden.

FASTQ-Format

Das *FASTQ*-Format[1] ist ein textbasiertes Dateiformat zur Speicherung von DNA Sequenzen. Die Anordnung der Sequenzen entspricht der in Abbildung 2.1 dargestellten Syntax.

```
@ML-P2-14:9:000H003HG:1:11102:17290:1073 1:N:0:TCCTGAGC+GCGATCTA
TTTGGTAACAGCATGAATTATTCTAGCCACTAAAACTCTATGAACATCTTGTGAAGGTTTCAGATAGAGCCTGAAGTACACAGAGAACAATTCTTAAAAAA
+
AAAAAEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEAEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE<AEEEEEEE
```

Abbildung 2.1: Beispiel eines Eintrages einer *FASTQ*-Datei

Jeder Eintrag, im Folgenden auch *read* genannt, besteht aus vier Zeilen. Die erste Zeile beginnt mit dem Zeichen `@` und enthält Informationen über den jeweiligen Namen der identifizierten Sequenz, sowie Informationen über den Sequenzierer und der Position auf der Durchflusszelle. In der zweiten Zeile ist die Basenabfolge der identifizierten Sequenz abgespeichert. Die dritte Zeile beginnt mit dem Zeichen `+`, welches als Trennzeichen dient, und kann erneut Zusatzinformationen aus der ersten Zeile enthalten. Oft enthält die dritte Zeile allerdings lediglich das Trennzeichen. Informationen über die Qualität der jeweiligen identifizierten Basen sind in der letzten Zeile gespeichert.

FASTA-Format

Bei dem sogenannten *FASTA*-Format handelt es sich ebenfalls um ein rein textbasiertes Format, in dem Nukleotidsequenzen bzw. Proteinsequenzen gespeichert werden. Jede Datei beginnt mit einer Kopfzeile (engl. Header), welche stets mit einem `>` beginnt. Diese enthält, jeweils getrennt durch ein `|`, den Namen sowie eine Beschreibung der jeweiligen Sequenz.

[1] `https://emea.support.illumina.com/bulletins/2016/04/fastq-files-explained.html`

Für die Kopfzeilen einer *FASTA*-Datei existieren verschiedene Standardisierungen, welche sich abhängig von der Sequenzdatenbank unterscheiden. Danach folgen die Sequenzdaten, in denen die Basen als jeweils ein Buchstabe in 5′- nach 3′-Richtung angeordnet sind[2]. Eine Datei mit mehreren Sequenzen erfordert für jede einzelne Sequenz eine eigene Kopfzeile als Bezeichner. Zwei *FASTA*-Dateien mit jeweils einer Sequenz lassen sich also einfach durch Konkatenation zu einer einzelnen zusammenführen. Zusätzlich können sich noch einzelne Zeilen mit Kommentaren in der *FASTA*-Datei befinden, welche laut der Spezifikation des Formates mit einem Semikolon beginnen müssen.

2.1.3 Datenbanken

Es gibt verschiedene Datenbanken, über die genetische Daten für die Forschung zur Verfügung gestellt werden. Diese sind frei zugänglich und jeder kann diese zur eigenen Verwendung herunterladen oder Daten hochladen, die durch eigene Forschung produziert wurden. Die verfügbaren Daten umfassen für verschiedene Spezies sowohl ganze Genome, als auch die Sequenzen der Proteine, die durch Transkription und Translation der Gene gebildet werden. Im Folgenden werden die zwei wichtigsten DNA-Datenbanken, die als Quelle für Testdaten für unsere Projektgruppe dienen, erläutert.

Sequence Read Archive

Das *Sequence Read Archive*[3] (kurz SRA) ist eine öffentliche Datenbank für DNA-Sequenz-Daten und wird verwaltet durch das *National Center for Biotechnology Information* (kurz NCBI). Der Großteil der Daten im SRA besteht aus kurzen *reads*, die von Hochdurchsatz-Sequenzierern wie dem *Ilumina Genome Analyzer* produziert werden. Diese haben eine Länge von 100 bis 200 Basenpaaren. Das SRA umfasst aktuell eine Datenmenge von etwa 39260 Tera-Basen. Zum Vergleich: Das menschliche Genom hat eine Größe von $3,27$ Giga-Basen. Das SRA selbst enthält nur Daten über DNA-Sequenzen, das NCBI unterhält neben dem SRA jedoch auch weitere Datenbanken über RNA- und Proteindaten.

European Nucleotide Archive

Das *European Nucleotide Archive*[4] (kurz ENA) ist ebenfalls eine öffentliche Datenbank für Nucleotidsequenzen. Es wird verwaltet durch das *European Bioinformatics Institue*.

Datenmodell

Das SRA und das ENA verwenden ein bestimmtes hierarchisches Metadatenmodell zur Strukturierung ihrer Daten. Eine Veranschaulichung des Modells kann in Abbildung 2.2 betrachtet werden.

Das Modell folgt gedanklich dem Vorgehen bei der Durchführung einer wissenschaftlichen Studie - dementsprechend ist die oberste Struktur die *study*. Eine Studie hat, wie auch alle untergeordneten Strukturen, einen Identifikationscode, die *accession number*. Bei Studien beginnt die *accession number* grundsätzlich mit `SRP`, beispielsweise `SRP289452`. Zur Durchführung einer Studie werden im Bereich der Medizin/Biologie Proben benötigt - die *samples*. Diese enthalten Daten über den Ort oder das Individuum, von denen die Probe stammt. *Accession numbers* von Proben beginnen mit `SRS`.

Als Nächstes gibt es die *experiments*, die im Rahmen einer *study* mit den *samples* ausgeführt werden. Diese enthalten Informationen zum Sequenziervorgang, wie etwa das

[2] `https://zhanglab.ccmb.med.umich.edu/FASTA/`
[3] `https://www.ncbi.nlm.nih.gov/sra`
[4] `https://www.ebi.ac.uk/ena`

Gerät und die Sequenzierstrategie. Experimente sind erkennbar an dem Präfix `SRX` in der *accession number*. Die *experiments* verbinden die *samples* mit der *study*.

Ein Experiment produziert Ausgabedaten. Diese werden *runs* genannt. Die *accession number* eines *runs* beginnt mit `SRR`. Ein *run* enthält eine oder zwei Ausgabedateien, je nachdem, ob *single-end* oder *paired-end* Sequenzierung durchgeführt wurde. Diese Dateien enthalten die eigentlichen DNA-Sequenzdaten und sind üblicherweise komprimierte *FASTQ*-Dateien. Eine solche *FASTQ*-Datei enthält die DNA-Sequenz-Daten in Form von vielen einzelnen *reads*.

Ein *read* ist ein zusammenhängendes Stück DNA aus der Probe. Neben den DNA-Basen A, C, G und T können auch andere Buchstaben vorkommen, wie etwa N für eine unbestimmte Base.

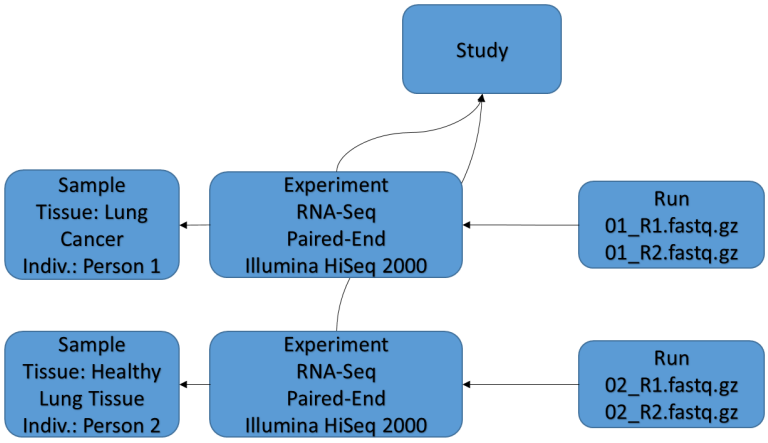


Abbildung 2.2: Metadatenmodell der genetischen Datenbanken.

2.2 Hashing

Im Rahmen der Projektgruppe wird Cuckoo-Hashing verwendet um eine Hashtabelle zu erzeugen und zu füllen. Im Folgenden werden die Grundlagen von Hashing sowie das Cuckoo-Hashing kurz erklärt.

2.2.1 Grundlagen

Hashing wird benutzt um eine Eingabemenge (die *Keys*) auf eine Menge von Hashwerten abzubilden. Eine Funktion, die das erreicht, nennt man Hashfunktion. Es gibt also eine beliebige Menge U, die als das Universum bezeichnet wird, und die Menge $K \subseteq U$ der *Keys*. Die Menge der Hashwerte V ist ein von der Anwendung abhängiger Zahlenraum, wird aber typischerweise als ein Anfangsstück der natürlichen Zahlen gewählt, also $V = \{0, \ldots, P-1\}$. Die Hashfunktion ist dann definiert als $h : K \to V$ Durch eine Hashfunktion bekommt man außerdem eine Hashtabelle, in der eine Position dem Hashwert entspricht, auf den der dort abgelegte *Key* abgebildet wird. Hashfunktionen kommen vor allem in drei Anwendungsgebieten zum Einsatz: Kryptographie, Berechnung von Prüfsummen und bei sogenannten *dictionaries*. Da die jeweiligen Anwendungsgebiete unterschiedliche Ergebnisse erzielen wollen, sind auch jeweils unterschiedliche Eigenschaften für die Hashfunktionen interessant, auf die später noch näher eingegangen wird. In der Projektgruppe kommen die Hashfunktionen vor allem im Rahmen von *dictionaries* zum Einsatz.

Hashfunktionen

Wie bereits erwähnt können Hashfunktionen verschiedene Eigenschaften haben, die je nach Anwendungsbereich mehr oder weniger attraktiv sind. Im Bereich der Kryptographie beispielsweise sollen sich aus den berechneten Hashwerten keine Rückschlüsse auf die ursprünglichen *Keys* ziehen lassen. Bei den in der Projektgruppe genutzten Hashfunktionen interessieren wir uns aber vor allem für die Effizienz hinsichtlich Laufzeit und Speichernutzung. Um die Laufzeit möglichst gering zu halten soll also die Funktion möglichst schnell berechenbar sein, aber auch wenig Kollisionen erzeugen. Außerdem sollen möglichst viele Speicherplätze der Hashtabelle auch tatsächlich belegt werden.

Bei Hashfunktionen in Datenbanken und insbesondere im Kontext der Projektgruppe ist auch die Invertierbarkeit eine wichtige Eigenschaft. Eine Funktion h ist invertierbar, wenn es eine Funktion h^{-1} gibt mit $h^{-1}(h(x)) = x$. So kann aus dem Hashwert wieder der *Key* berechnet werden, was vor allem im Kontext von Datenbanken interessant ist, da dann der *Key* nicht gespeichert werden muss. Um eine Funktion zu invertieren, muss diese allerdings injektiv sein, was bei Hashfunktionen oft nicht der Fall ist. Um dieses Problem zu umgehen kann zu den *Keys* jeweils ein Fingerabdruck gespeichert werden. Dabei soll aus dem Hashwert und dem Fingerabdruck der *Key* wieder eindeutig berechnet werden können. So muss möglicherweise nur ein kleiner Bruchteil anstelle des ganzen *Keys* gespeichert werden.

Die meisten Hashfunktionen, die für Datenbanken genutzt werden, bestehen aus Bitweisen Operationen, wie zyklischen Rotationen oder bitweisem XOR, oder Multiplikationen mit Konstanten. Die XOR-Operation und die bitweise Rotation sind schnell durchzuführen, da pro Bit nur wenige Operationen erforderlich sind. Das XOR ist direkt auf Hardware-Ebene implementiert und erfordert nur wenige Taktzyklen. Bei der bitweisen zyklischen Rotation muss jedes Bit nur einmal um einen bestimmten Wert nach links und nach rechts verschoben werden, dann erfolgt eine bitweises Oder auf die beiden entstehenden Bitstrings (auch das ist auf der Hardware-Ebene implementiert). Die Multiplikation ist eine eher lange Operation und wird daher nur mit Konstanten durchgeführt. Dadurch, dass eine der Zahlen konstant ist, kann der Compiler bereits einige der erforderlichen Additionen kürzen und so die Laufzeit verkürzen. Außerdem muss der Hashwert immer mit einer Modulo-Operation an die Größe der Tabelle angepasst werden. Es gibt verschiedene Typen von Hashfunktionen, von denen im Folgenden einige näher erklärt werden.

Einfache Hashfunktionen funktionieren wie die in 2.1 gezeigte Funktion. Sie bestehen vor allem aus Rotationen um bestimmte Werte, bitweise XOR-Operationen, sowie Multiplikationen mit Konstanten. Oft wird dabei der *Key* in gleich große *Chunks* aufgeteilt, auf denen dann die Operationen durchgeführt werden. Die verarbeiteten *Chunks* werden dann wiederum mit Operationen (wie dem bitweisen XOR) zu einem Hashwert zusammengefügt. Ein Beispiel für eine simple Hashfunktion ist in Listing 2.1 zu sehen. Die Hashfunktion bekommt hier eine Zahl als *Key* gegeben und gibt wiederum eine Zahl als Hashwert zurück. Die neue Zahl wird berechnet, indem der *Key* zunächst um 32 Bits nach links rotiert wird, wobei `rol(x, y)` eine zyklische Linksrotation von x um y Bits implementiert. Danach wird das Ergebnis noch mit einem Seed, der von der äußeren Funktion vorgegeben wird, multipliziert. Da diese Funktionen nur die oben beschriebenen Operationen verwenden, lassen sie sich vergleichsweise schnell durchführen und auch die Invertierbarkeit ist gegeben, abgesehen von den Modulo-Operationen.

Beim Tabulation Hashing wird für das Berechnen des Hashwertes eine zuvor (z.B. zufällig) generierte Tabelle verwendet. Dafür wird der *Key* zunächst wieder in gleich große

```
def make_invertible_hash(seed = 238172381943):
    def hash_function(key: int):
        h = (rol(key, 32))
        h = (seed * h)
        return h
```

Listing 2.1: Beispiel für eine Hashfunktion

Chunks aufgeteilt und für jeden dieser *Chunks* wird ein Wert aus der generierten Tabelle gesucht. Diese Werte werden dann mittels XOR zu dem Hashwert kombiniert. Das genaue Vorgehen ist wie folgt: sei p die Anzahl der Bits des *Keys*, q die gewünschte Anzahl der Bits des Hashwertes und $r \leq p$ die Größe der *Chunks*, in die der *Key* aufgeteilt wird. Zuerst wird die Anzahl $t = \lceil \frac{p}{r} \rceil$ der *Chunks* berechnet. Damit wird dann eine Tabelle T mit den Maßen $t \times 2^r$ erzeugt und mit zufälligen Werten gefüllt. Für den i-ten *Chunk* x_i sei nun $h_i = T[i][x_i]$. Der Hashwert zu dem *Key* x ergibt sich dann als

$$h(x) = h_1 \oplus h_2 \oplus \cdots \oplus h_t$$

Auch hier werden nur XOR-Operationen verwendet. Allerdings muss zuvor die Tabelle generiert werden und da viele Suchoperationen auf der Tabelle durchgeführt werden, sollte eine Datenstruktur verwendet werden, die in dieser Hinsicht möglichst effizient ist. Außerdem sollte die Tabelle klein genug sein für den Arbeitsspeicher sein.

Der ntHash [17] ist eine Hashingmethode, die auf eine Sequenz von k-meren ausgelegt ist. Die Berechnung des Hashwertes eines k-mers ergibt sich dabei aus einer zuvor generierten Tabelle (wie beim Tabulation Hashing) und aus dem Hashwert des vorherigen k-mers. Für eine k-mer-Sequenz r wird der Hashwert des ersten k-mers $k\text{-mer}_0$ berechnet als

$$H(k\text{-mer}_0) = rol^{k-1}h(r[0]) \oplus rol^{k-2}h(r[1]) \oplus \cdots \oplus h(r[k-1])$$

Dabei ist rol eine zyklische Rotation nach links und h ist eine Tabelle, mit der A,C,G und T (also die Zeichen innerhalb eines k-mers) auf zufällige 64-Bit-Integer abgebildet werden. Alle weiteren Hashwerte werden berechnet als

$$H(k\text{-mer}_i) = rol^1 H(k\text{-mer}_{i-1}) \oplus rol^k h(r[i-1]) \oplus h(r[i+k-1])$$

Der ntHash ist bezüglich k-meren schneller als die meisten anderen darauf ausgelegten Hashfunktionen. Allerdings ist die Funktion nur schwer invertierbar, da sie auf den Hashwerten der vorherigen k-mere basiert.

Kollisionen

In der Praxis sind Hashfunktionen oft nicht injektiv, das heißt, dass verschiedene *Keys* auf denselben Hashwert abgebildet werden. Für zwei *Keys* x und y mit $x \neq y$ gilt also $h(x) = h(y)$. Einen solchen Fall nennt man Kollision. Da gerade im Bereich der Datenbanken aber meistens alle *Keys* gespeichert werden sollen, müssen Kollisionen aufgelöst werden. Für die Kollisionsbehandlung gibt es zwei unterschiedliche Herangehensweisen, die oft auch gemischt zum Einsatz kommen.

Zum einen gibt es das geschlossene Hashing mit offener Adressierung. Hierbei wird versucht eine neue Speicherposition für den *Key* zu finden. Drei Methoden dafür werden im Folgenden kurz erläutert: lineares Sondieren, quadratisches Sondieren und doppeltes Hashing. Beim linearen Sondieren wird von dem berechneten Hashwert aus die nächste

um ein bestimmtes Intervall versetzte Position überprüft, bis ein freier Speicherplatz gefunden wird. Der verwendete Hashwert im i-ten Versuch einen *Key* x einzufügen wird also berechnet als $h_i(x) = h(x) + i \cdot a$, wobei a die Größe des Intervalls ist.

Das quadratische Sondieren benutzt zwei Konstanten c_1 und c_2 um die nächste Speicherposition zu berechnen. Dabei fließt c_1 weiterhin linear in die Berechnung ein, während c_2 mit einem quadratischen Faktor einfließt. Der Hashwert im i-ten Versuch ist also $h_i(x) = h(x) + c_1 \cdot i + c_2 \cdot i^2$.

Beim doppelten Hashing nimmt man sich für die Berechnung des Intervalls eine zweite Hashfunktion zu Hilfe, die die Größe des Sondierungsintervalls bestimmt. Für zwei gewählte Hashfunktionen h und h' ergibt sich der Hashwert im i-ten Versuch als $h_i(x) = h(x) + i \cdot h'(x)$. Weitere Methoden umfassen Hopscotch-Hashing, Robin Hood Hashing sowie Cuckoo-Hashing, welches später noch näher erklärt wird.

Die zweite Herangehensweise ist das offene Hashing mit geschlossener Adressierung, auch bekannt als Separate Chaining. Hierfür ist die Hashtabelle so aufgebaut, dass für jeden Hashwert alle eingefügten *Keys* mit diesem Hashwert separat in einer Liste gespeichert werden. Um einen *Key* zu finden muss also erst der Hashwert berechnet und dann die entsprechende Liste nach dem *Key* durchsucht werden. Meistens wird als Liste eine Datenstruktur verwendet, die vor allem bei kleiner Größe effizient ist, da die Liste nur bei Kollisionen vergrößert wird und die Anzahl der Kollisionen für eine effiziente Hashtabelle klein gehalten werden soll.

2.2.2 Cuckoo Hashing

Viele Verfahren zur Kollisionsauflösung in Hashtabellen resultieren in schwer vorhersehbaren Lookup-Zeiten bei hohen Füllraten oder erfordern dynamische Datenstrukturen, wie Listen, welche aufgrund der Nutzung von nicht zusammenhängendem Speicher zu häufigen Cache Misses führen, für die Speicherung der einzelnen Buckets. Cuckoo Hashing vermeidet beide Probleme und garantiert konstante Lookup-Zeiten und funktioniert effizient mit gleichförmigen einfachen Array-Strukturen. Dies sind die Gründe weshalb wir uns für die Durchführung unseres Projektes primär auf das Cuckoo-Hashing fokussieren.

Bei der Grundform des Cuckoo-Hashings werden zwei Hashfunktionen, hier f_1 und f_2 genannt, verwendet, um Schlüssel auf jeweils eine separate Hashtabelle pro Hashfunktion zu verteilen. Soll ein Schlüssel x eingefügt werden, so kann er also entweder an der Stelle $f_1(x)$ in die erste Tabelle eingefügt werden oder an der Stelle $f_2(x)$ in die zweite Tabelle. Da die Stellen, also Buckets in der Grundform aber höchstens jeweils einen Schlüssel aufnehmen können, also explizit auf verkettete Listen verzichtet wird, kann es dazu kommen, dass beide Buckets, an denen ein Schlüssel gespeichert werden könnte, bereits voll sind. Um in dieser Situation trotzdem den Schlüssel einfügen zu können muss einer der beiden Schlüssel der die potentiellen Buckets des aktuellen Schlüssels belegt, aus diesem gelöscht werden und vom einzufügenden Schlüssel verdrängt werden. Für den verdrängten Schlüssel können dann wieder beide Buckets, in die er eingefügt werden könnte, ermittelt werden und darauf überprüft werden, ob einer von ihnen leer ist, was zum Einfügen dieses Schlüssels führt, sonst zu einer erneuten Verdrängung usw.

Da die beschriebene Einfügeoperation insbesondere bei hohen Auslastungen der Hashtabellen manchmal überhaupt nicht abbricht, weil die Verdrängungen in einer zyklischen Form ablaufen, worauf im Abschnitt zum Cuckoo-Graphen genauer eingegangen wird, ist die Laufzeit einer Einfügeoperation in eine Cuckoo-Hash-Tabelle im Worst-Case in $O(n)$, da ein solcher Kreis sich im schlimmsten Fall über alle vorhandenen Schlüssel erstrecken kann. Im Gegensatz dazu ist die Laufzeit einer Suchoperation selbst im Worst-Case in $O(1)$, da lediglich zwei Buckets überprüft werden müssen um einen Schlüssel zu finden oder festzustellen, dass er nicht vorliegt. Neben der asymptotischen Laufzeit hat insbe-

sondere in der Praxis die Anzahl der Cache-Misses bei Zugriffen auf Datenstrukturen eine große Bedeutung, da in der Grundform des Cuckoo-Hashings allerdings exakt ein Schlüssel pro Bucket gespeichert wird und somit nicht auf die Größe einer Cache-Line geachtet wird, ist die Grundform des Cuckoo-Hashings gegenüber Verfahren die, die gesamte Größe für Buckets ausnutzen im Nachteil. Ein weiterer Nachteil der Grundform des Cuckoo-Hashings ist, dass üblicherweise nur geringe Auslastungen der Tabellen erzielt werden können, also der Quotient aus der Anzahl der mit Schlüsseln belegten Bucket-Speicherplätze und den angeforderten Bucket-Speicherplätzen gering ist, da es oft schon bei Auslastungen von ungefähr 50% [31] vorkommt, dass Schlüssel nicht mehr eingefügt werden können, weil sich die zuvor erwähnten Kreise bilden.

Der Grundform des Cuckoo-Hashings steht eine Reihe von Erweiterungen gegenüber. Im Folgenden wird auf das (h, b) Cuckoo-Hashing eingegangen, welches es erlaubt anstelle von zwei Hashfunktionen h, verschiedene Hashfunktionen zu benutzen, sowie anstelle von Buckets die nur einen Schlüssel speichern können, solche zu verwenden die b Schlüssel speichern können. Auch die Nutzung exklusiver Tabellen ist beim (h, b) Cuckoo-Hashing nicht festgelegt und es kann eine gemeinsame Tabelle für alle Hashfunktionen verwendet werden. Die asymptotische Laufzeit des (h, b) Cuckoo-Hashings verhält sich genau so wie die der Grundform, da sich auch hier beliebig große Kreise formen können und damit die Worst-Case Laufzeit fürs Einfügen in $O(n)$ liegt und die Worst-Case Laufzeit fürs Suchen in $O(h \cdot b)$ ist. Dies liegt daran, dass höchstens in jedem Bucket gesucht werden muss, auf den über eine Hashfunktion gezeigt wird und jeder Bucket b Schlüssel einhält. Da h und b für eine Hashtabelle konstant gewählt werden, ist die Laufzeit damit in $O(1)$.

Während die asymptotischen Laufzeit-Verhalten der beiden Variationen des Cuckoo-Hashings gleich sind, ist in der Praxis für Datenstrukturen besonders die durchschnittliche Anzahl der Cache-Misses pro Suchoperation von Bedeutung. Hierbei ist es beim (h, b) Cuckoo-Hashing möglich das b so zu wählen, dass der Speicher für einen Bucket möglichst genau in eine Cache-Line passt, was darin resultiert, dass die Wahrscheinlichkeit sinkt, dass ein erneuter Cache-Miss notwendig ist, um den Schlüssel zu finden, da er wenn bei einem Mapping auf das erste Bucket dort auch gespeichert werden kann wenn schon $b - 1$ Schlüssel dort gespeichert werden. Damit hat das (h, b) Cuckoo-Hashing in dieser Hinsicht einen Vorteil gegenüber der Grundform. Die Möglichkeit, mehr als zwei Hashfunktionen beim (h, b) Cuckoo-Hashing zu verwenden, hat hingegen sowohl Vorteile als auch Nachteile gegenüber der Grundform, welche beim Einsatz entsprechend abzuwägen sind. Einerseits führt der Einsatz von mehr Hashfunktionen dazu, dass alle Schlüssel auf mehr Buckets verteilt werden können und somit die Wahrscheinlichkeit sinkt, dass es nicht möglich ist einen Schlüssel einzufügen. Dadurch ist es möglich deutlich höhere Füllraten als bei der Grundform zu erreichen. Bei $h = 3$ und $b = 1$ sind bereits Füllraten von über 0.9 üblich und bei $h = 3$ und $b = 3$ Füllraten von über 0.99 [31]. Andererseits kann es in diesem Fall eben auch dazu kommen, dass Schlüssel in einen Bucket eingefügt werden, der ihm über eine Hashfunktion mit einem Index > 2 zugeordnet wird, was dazu führt, dass mehr Cache-Misses bei der Suche stattfinden, da mehr Speicherstellen betrachtet werden müssen.

Der Cuckoo-Graph

Das Graphen-Modell des Cuckoo-Hashings stellt die Belegung einer Hashtabelle mit einer Menge von Schlüsseln für eine gegebene Menge an Hashfunktionen dar. Sowohl die Buckets als auch die Schlüssel werden in dem Modell als Knoten dargestellt. Die Zuweisung der Buckets zu den Schlüsseln wird über die gerichteten und gewichteten Kanten des Graphen dargestellt. Die Gewichte sind ganze Zahlen aus den Intervallen $[-h, -1]$ und $[1, h]$ und werden im folgenden auch Kosten genannt. Jeder Schlüssel-Knoten hat so viele anliegende Kanten wie Hashfunktionen für die Hashtabelle verwendet werden, also in der Grundform

2 sonst h, und der Betrag des Gewichts ist der Index der verwendeten Hashfunktion. Der andere Knoten ist stets ein Bucket, und zwar der den die entsprechende Hashfunktion für den Schlüssel berechnet, der Cuckoo-Graph ist also bipartit, wobei die Schlüssel-Knoten untereinander nicht benachbart sind und die Bucket-Knoten ebenso.

Die Richtung der Kanten sowie das Vorzeichen der Gewichte wird durch die Verwendung der Hashfunktionen festgelegt. Für alle $h - 1$ Kanten, deren Hashfunktionen nicht verwendet werden, also die einen Schlüssel mit Buckets verbinden in denen er nicht gespeichert wird, gehen die Kanten von den entsprechenden Buckets aus und zeigen auf den Schlüssel. Die Kantengewichte sind dann positiv. Für die eine Kante die einen Schlüssel mit dem Bucket verbindet, in dem er gespeichert wird, ist die Richtung umgekehrt, also geht sie vom Schlüssel aus und zeigt auf den Bucket. Außerdem ist das Kantengewicht für diese Kante negativ.

Die Kosten der verwendeten Kante sind insofern von Bedeutung, als dass sie bei einer Suche nach dem Schlüssel der Anzahl an zu Besuchenden Buckets entspricht. Da jeder Bucket ungefähr eine Cache-Line in Anspruch nehmen sollte, resultieren Kosten der Höhe u bei der Suche nach einem Schlüssel auch in u Cache-Misses. Cuckoo-Graphen, die besonders niedrige Kosten-Betrags-Summen für die verwendeten Kanten haben, sind also besonders „gut“.

Der Begriff eines Kreises im Cuckoo-Graphen, der bereits zuvor verwendet wurde, soll hier erklärt werden. Wenn in einer Hashtabelle mit $(2, 1)$ Cuckoo-Hashing drei Schlüssel x_1, x_2 und x_3 alle über ihre Hashfunktionen auf die selben drei Buckets abgebildet werden, so liegt ein Kreis vor, da ein weiterer Schlüssel, dessen beide Buckets zwei der drei vollen Buckets sind, nun nicht mehr in die Tabelle eingefügt werden könnte. Parallel dazu existieren im Cuckoo-Graphen beim (h, b) Cuckoo-Hashing Mengen an Schlüsseln X, der Größe $|X| = a \cdot b$ für welche alle ihre Nachbarn im Cuckoo-Graphen Buckets aus der Menge Y sind mit der Größe $|Y| = a$. Diese Mengen sind zwar in der Graphentheorie keine Kreise, haben aber beim (h, b) Cuckoo-Hashing den selben Effekt wie ein Kreis beim $(2, 1)$ Cuckoo-Hashing, wehalb sie im folgenden als **(h,b)-Kreise** bezeichnet werden. Falls auch nur ein weiterer Schlüssel nur in Buckets aus der Menge Y gespeichert werden kann, so ist es unmöglich ihn und alle Schlüssel der Menge X gleichzeitig in der Hashtabelle zu speichern.

Ein Beispiel für einen solchen Graphen ist in Abbildung 2.3 zu sehen.

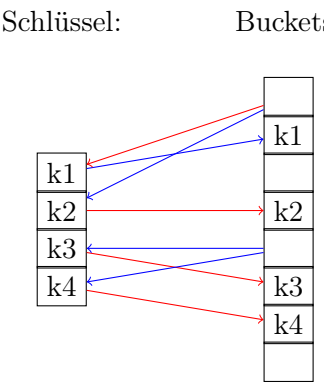


Abbildung 2.3: Ein Cuckoo-Graph. Rote Kanten besitzen Kosten 1 und Blaue sind die Kanten mit Kosten 2.

Einfügestrategien für das (h,b) Cuckoo-Hashing

Bis jetzt wurden das Cuckoo-Hashing, dessen Erweiterung das (h, b) Cuckoo-Hashing und der Cuckoo-Graph vorgestellt. Während für die Grundform des Cuckoo-Hashings bereits

eine Einfügestrategie vorgestellt wurde, sind für das (h, b) Cuckoo-Hashing einige verschiedene Einfügestrategien bekannt.

Da die Implementierung dieser Strategien natürlich in Form von Algorithmen geschieht, ist insbesondere deren Laufzeit und Speicherverbrauch von Interesse. Ein weiteres Merkmal welches von Interesse ist, ist die Güte der erzeugten Belegung der Hashtabelle. Da die durch Einfügestrategien erzeugten Hashtabellenbelegungen in der Praxis verwendet werden, um gespeicherte Daten auszulesen, und die Laufzeit durch die Anzahl der Cache-Misses dominiert wird, sind Belegungen besonders „gut“, wenn sie in der Praxis beim Suchen in möglichst wenigen Cache-Misses resultieren. Weil die Anzahl an Cache-Misses pro Such-Operation beim (h, b) Cuckoo-Hashing meist gleich der Anzahl der zu besuchenden Buckets ist, und die Buckets eines Schlüssels in einer festen Reihenfolge im Bezug auf die Hashfunktionen besucht werden, kann die Anzahl der zu besuchenden Buckets minimiert werden, wenn alle Schlüssel mit möglichst geringen Hashfunktions-Indizes auf die Buckets verteilt werden. Im Folgenden bezeichnet das Wort Kosten die Summe der Indizes der Hashfunktionen, die für die Belegung verwendet wurden und eine Belegung ist „gut“ wenn ihre Kosten gering sind, da diese Kosten der Anzahl der erwarteten Cache-Misses gleich kommen.

Im Folgenden wird auf die beiden Einfügestrategien Random Walk und die kostenoptimale Einfügestrategie eingegangen.

Der Random Walk

Im Folgenden bezeichnet der Begriff „voller Bucket“ einen Bucket dessen b Plätze alle einen Schlüssel speichern, und der Begriff „offener Bucket“ einen Bucket der 0 bis $b-1$ Schlüssel speichert, und somit noch mindestens einen Schlüssel aufnehmen kann. Beim Random Walk werden die Schlüssel in die Tabelle eingefügt, indem für sie in der gegebenen Reihenfolge die ihnen über ihre Hashfunktion zugewiesenen Buckets ermittelt werden, bis so ein offener Bucket ermittelt wird. In diesen wird der Schlüssel dann eingefügt. Kommt es zu einer Situation, in der alle Buckets, in die ein Schlüssel x eingefügt werden kann, voll sind, so wird ein zufälliger, bereits in der Tabelle liegender Schlüssel y, aus einem der Buckets an einer der Stellen $f_1(x) \dots f_h(x)$, in die der Schlüssel x eingefügt werden kann gelöscht und x an die Stelle von y geschrieben. Der gelöschte Schlüssel y wird dann nach dem gleichen Verfahren wie x eingefügt. Auf diese Art und Weise können beim Einfügevorgang, insbesondere bei hohen Füllraten lange Verdrängungsketten entstehen.

Zusätzlich können die im Abschnitt zum Cuckoo-Graphen beschriebenen (h, b)-Kreise entstehen, was dazu führt, dass ein Schlüssel nicht in die Tabelle eingefügt werden kann. Dies bedeutet, dass die Tabelle mit der aktuellen Konfiguration die gegebene Schlüsselmenge überhaupt nicht speichern kann. Da lange Verdrängungsketten allerdings suggerieren, dass die Tabelle nah an ihrer über ihre Parameter wahrscheinlichen Füllrate ist und Einfügeoperationen zunehmend langsamer werden, kann der Benutzer ein Maximum an Verdrängungen pro Einfügeoperation festlegen. Wird dieses erreicht, so kann die Tabelle mit anderen Parametern erneut aufgebaut werden, in der Hoffnung, dass sich dann keine langen Ketten bilden. Diese Einfügestrategie ist zwar leicht zu implementieren, bietet allerdings keine Garantie für die Summe der Kosten, weshalb im Folgenden die kostenoptimale Einfügestrategie diskutiert wird.

Da die Random Walk Einfügestrategie praktisch dem Standard-Einfügeverfahren bei der Grundform entspricht und lediglich auf das (h, b) Cuckoo-Hashing angepasst ist, ist sie das übliche Einfügeverfahren beim (h, b) Cuckoo-Hashing und kann praktisch immer eingesetzt werden, wenn Daten einfach gespeichert werden müssen und Cache-Misses nicht besonders teuer sind. Außerdem bietet sie sich auch an wenn pro Einfügeverfahren nur einzelne oder wenige Schlüssel eingefügt werden müssen, da sie im Vergleich zum kostenoptimalen Verfahren auch dann recht schnell ist. Das Verfahren ist also sinnvoll, falls

laufend Änderungen an den Schlüsseln der Tabelle vorgenommen werden müssen.

Die kostenoptimale Einfügestrategie

Im Gegensatz zum Random Walk stellt die kostenoptimale Einfügestrategie[33] sogar sicher, dass die minimalen möglichen Kosten für die gegebene Tabellenkonfiguration und Schlüsselmenge erreicht wird. Um dies zu erreichen, arbeitet sie in Durchläufen und erhält dabei die Invariante aufrecht, dass vor jedem Durchlauf alle Schlüssel, die bereits in der Tabelle sind, die minimalen Kosten für ihre Menge haben.

Die ersten beiden Durchläufe sind dabei Initialisierungsdurchläufe, bei denen im ersten alle Schlüssel eingefügt werden, die mit der ersten Hashfunktion, also Kosten 1 pro Schlüssel, eingefügt werden können. Beim zweiten Durchlauf wird genauso für die verbleibenden Schlüssel vorgegangen, nur dass nun die zweite Hashfunktion verwendet wird. Nach beiden Durchläufen wurde noch keine Verdrängung durchgeführt und die Invariante ist aufrechterhalten.

Sollten noch weitere Schlüssel verbleiben, so muss ein komplexeres Verfahren eingesetzt werden, um das Einfügen dieser ohne das Erzeugen überflüssiger Kosten zu gewährleisten. Die folgenden Durchläufe sind dann Standarddurchläufe, welche aus zwei Phasen bestehen, der Suchphase und der Bewegungsphase.

In der Suchphase werden für alle Schlüssel ohne Bucket die kürzesten Pfade zu offenen Buckets ermittelt. Dazu wird eine Abwandlung des Bellmann-Ford-Algorithmus eingesetzt. Es wird der Cuckoo-Graph eingesetzt und alle offenen Buckets werden mit Kosten 0 initialisiert und alle anderen Knoten mit unendlichen Kosten. Es wird eine Menge A verwendet, um zu speichern, welche Knoten im Augenblick aktiv sind. Diese Menge wird initial mit den offenen Buckets initialisiert.

Da der Cuckoo-Graph bipartit ist, enthält die Menge A zu jedem Zeitpunkt entweder nur Schlüssel-Knoten oder nur Bucket-Knoten. Es wird dann jeweils für jeden Knoten in A jeder Nachbar betrachtet und die aktuellen Kosten der Nachbarn c_y mit den aktuellen Kosten c_x des Knoten $x \in A$ plus den Kosten der Kante z zwischen den beiden Knoten verglichen. Falls $c_y > c_x + c_z$ so wird c_y auf $c_x + c_z$ gesetzt und für den Knoten y wird x als Vorgänger gespeichert. Außerdem wird y zur Menge A' der aktiven Knoten für den nächsten Durchlauf hinzugefügt. Nachdem alle Nachbarn aus A betrachtet und gegebenenfalls aktualisiert wurden, wird A auf A' gesetzt. Ist A dann leer, so wurden alle verbessernden Pfade im Graphen gefunden und die Suchphase endet und es wird in die Bewegungsphase übergegangen.

Am Ende einer Suchphase kann es dazu kommen, dass für einen Schlüssel ohne Bucket kein Pfad zu einem offenen Bucket gefunden werden konnte. Dies kann nur eintreten, falls alle Buckets, in denen dieser Schlüssel gespeichert werden kann, Teil eines (h, b)-Kreises sind. In diesem Fall kann die Tabelle die gegebene Menge an Schlüsseln nicht aufnehmen.

Da in diesem Zustand nun alle kürzesten Pfade für jeden Schlüssel ohne Bucket bekannt sind, beginnt nun die Bewegungsphase, in welcher die Schlüssel entlang dieser Pfade bewegt werden. Da für einen Pfad allerdings nicht sichergestellt werden kann, dass er für mehrere Schlüssel, die an ihm verschoben werden, einen offenen Bucket am Ende hat bzw. dieser den kürzesten Weg hat, wenn er von mehreren Schlüsseln benutzt wird, wird jeder Pfad von höchstens einem Schlüssel benutzt. Es wird also wie folgt vorgegangen: Es wird über die Schlüssel ohne Bucket iteriert und sie folgen der Reihe nach den Pfaden, ersetzen also ihren Vorgänger, welcher seinen Vorgänger ersetzt usw. bis ein offener Bucket erreicht wird. Wenn ein solcher Ersetzungsvorgang durchgeführt wird, so wird jeder Knoten auf dem Weg als besucht markiert. Wenn ein Schlüssel anschließend bei seinem Bewegungsvorgang über einen Knoten verschoben werden muss, der bereits berührt wurde, so wird er erst im nächsten Standarddurchlauf eingefügt.

Auf diese Art und Weise werden über die Standarddurchläufe die verbliebenen Knoten eingefügt, wobei immer mindestens ein Knoten pro Durchlauf eingefügt wird, falls alle Schlüssel eingefügt werden können, und durch die kürzesten Wege sichergestellt wird, dass die Invariante aufrecht erhalten wird.

Die Laufzeit des Algorithmus ist im Worst-Case in $O(n^3)$, da diese durch die Suchphasen dominiert wird, welche selber n Mal durchgeführt werden können. Sowie n Iterationen enthalten können, welche wiederum selber jeweils eine Laufzeit in $O(n)$ haben. Praktisch ist jedoch meist eine Laufzeit zu beobachten, die mit einem Algorithmus mit Laufzeit in $O(n \cdot \log^2\ n)$ zu vergleichen ist.

Die kostenoptimale Einfügestrategie bietet also ein relativ performantes Verfahren an, um Batches von Schlüsseln so in eine Cuckoo-Hash-Tabelle einzufügen, dass bei der Suche die Anzahl der Cache-Misses minimiert wird. Da das Verfahren aber beim Einfügen weniger oder einzelner Schlüssel pro Batch in der Suchphase große Teile der Tabelle berücksichtigen muss, um wenige Schlüssel einzufügen, ist es nicht sinnvoll, es für Hashtabellen einzusetzen, in die ständig neue Schlüssel eingefügt werden sollen. Es ist also insbesondere da sinnvoll, diese Strategie einzusetzen, wo alle Schlüssel von vornherein bekannt sind und lediglich die Nutzlast der Schlüssel aktualisiert werden muss. Dementsprechend ist der Einsatz auch bei bereits bestehenden Hashtabellen, insbesondere Cuckoo-Hashtabellen, denkbar, deren Schlüssel-Wert-Paare zuvor mit sub-optimalen Kosten eingefügt wurden, um eine Tabelle mit optimalen Kosten zu erzeugen.

2.3 Werkzeuge

In diesem Unterkapitel werden die von der Projektgruppe verwendeten Werkzeuge vorgestellt. Dies umfasst sowohl die Werkzeuge für die Umsetzung des Projektes und das gemeinsame Arbeiten an dem Projekt, als auch die Vorgehensweise bei der Implementierung und die Dokumentation des Projektes. Darüber hinaus wird die Notwendigkeit und Bedeutung der einzelnen Werkzeuge für die Nutzung in der Projektgruppe diskutiert.

2.3.1 Python

Im Rahmen der Projektgruppe wird hauptsächlich mit der Programmiersprache Python gearbeitet. Python ist eine interpretierte, dynamisch typisierte Programmiersprache, die sich dadurch auszeichnet, dass der Programmcode gut lesbar ist und sehr knapp gehalten werden kann. Interpretierte Programmiersprachen unterscheiden sich von kompilierten Programmiersprachen insofern, dass der Quellcode erst bei der Ausführung und Zeile für Zeile übersetzt wird. Dies bietet den Vorteil, dass der Entwicklungsaufwand reduziert wird, da keine Wartezeiten durch das Kompilieren entstehen und eventuelle Fehlermeldungen zeilenspezifisch ausgegeben werden können. Durch die höhere Entwicklungsgeschwindigkeit eignet sich Python besonders gut für das *rapid prototyping* und für nicht zeitkritische Anwendungen. Oft sind nur wenige Codezeilen für die Laufzeit des Programms relevant, sodass diese später noch optimiert werden können. Dies kann im Falle von Python beispielsweise mit *Numba* (siehe Abschnitt 2.3.3) geschehen. Ein kompiliertes Programm ist auf der anderen Seite besser optimiert und läuft dadurch in den meisten Fällen merklich schneller als ein interpretiertes Programm.

Python bietet die Möglichkeit, sogenannte Dekorierer zu definieren und zu verwenden. Diese Dekorierer sind Modifikatoren für Funktionen, die die Funktionsweise einer dekorierten Funktion verändern. So kann beispielsweise eine Funktion für einen Algorithmus mit einem Dekorierer versehen werden, der diese so abändert, dass zusätzlich zur Ausführung des Algorithmus noch dessen Laufzeit gemessen wird.

Für Python existieren bereits viele Programmpakete von unabhängigen Entwicklern, die vielerlei Funktionen bieten. Unter anderem finden sich so Module für die effiziente Verarbeitung von Daten und die Laufzeitoptimierung des eigenen Codes, die in dieser Projektgruppe verwendet und im Folgenden erläutert werden.

2.3.2 NumPy

Numerical Python (*NumPy*) [30] ist eine Pythonbibliothek, welche effiziente Implementierungen für Arrays und dessen Operationen anbietet. Im Rahmen der Projektgruppe sind *NumPy*-Arrays ein wichtiger Bestandteil für die effiziente Implementierung von Hashtabellen. Der Unterschied zwischen Python Listen und *NumPy* Arrays ist, dass diese effizienter sind, sowohl beim Speicherplatzverbrauch, als auch bei der Performance. *NumPy* besitzt dabei optimierte Funktionen für verschiedene (arithmetische-, logische-) Operationen, die auf folgenden zwei Konzepten basieren. Zum einem die **Vectorization**, die dafür sorgt, dass keine Schleifen benötigt werden, da diese ineffizienter sind, aufgrund von viel Overhead bei jedem Schleifendurchlauf. In *NumPy* werden die nicht existierenden Schleifendurchläufe durch direkte Indexierung der einzelnen Elemente im optimierten kompilierten C-Code realisiert. Dadurch können Iterationen über Listen, die vorher über mehrere Zeilen gingen, nun in weniger Zeile zusammengefasst werden. Somit wird der Code auch übersichtlicher und weniger fehleranfällig, da im Allgemeinem weniger Zeilen Code vorhanden sind. Das zweite Konzept ist unter Nutzung von `ufuncs`[5] das **Broadcasting**, dies ist die elementweise Ausführung für arithmetische, logische und bitweise Operatoren. *NumPy*-Arrays müssen dabei auch nicht mehr dieselbe Länge besitzen. Diese ist beispielsweise nützlich bei der Multiplikation mit Skalar.

2.3.3 Numba

Zusätzlich, da Python eine interpretierte Sprache ist, wird das Paket *Numba* [12] als *Just-in-time-Compiler* für Python genutzt, um die Laufzeit des damit kompilierten Codes zu verringern. Dabei wird der Python-Code zur Laufzeit in Maschinencode übersetzt, der die gleichen Ergebnisse liefert, allerdings schneller ausgeführt werden kann. Die Übersetzung findet beim ersten Aufruf einer Funktion statt. Darauffolgende Aufrufe greifen dann auf die bereits übersetzte Funktion zurück. Um dies zu ermöglichen, setzt *Numba* auf *LLVM*[14], hier dient es zum Kompilieren des Python-Codes, wobei das Projekt genauer im Kapitel 2.3.4 beschrieben wird. *Numba* dient bei der Benutzung von *NumPy*-Arrays, Funktionen und Schleifen zur Optimierung der Laufzeit. Dies geschieht durch Typinferenzen und durch automatisch generiertem Maschinencode, welcher kompiliert ist, statt wie in Python üblich interpretiert. Dafür werden eine Reihe von Dekorierern mit Optionen, welche an Funktionen geschrieben werden können, bereitgestellt, um das Kompilieren zu ermöglichen. Dekorierer sind beispielsweise:

- `@jit`
- `@njit`,
- `@vectorize`,
- `@jitclass`.

Der Dekorator `@njit` ist eine Kurzform des Dekorators `@jit(nopython=True)`. Hierbei gibt die Option `nopython=True` an, dass man die Funktion kompilieren möchte, sodass der **Python** Interpreter nicht genutzt wird. Zusätzlich existiert die Option `parallel=True`,

[5] `https://numpy.org/doc/stable/reference/ufuncs.html`

welches bei parallelisierbarem Code die Möglichkeit bietet, diesen automatisch zu kompilieren und versucht diesen auf mehrere Kerne zu optimieren. Mittels des Dekorierers `@vectorize` wird die Funktionalität von `ufuncs` aus *NumPy* implementiert. Diese dienen der elementweisen Ausführung von Operationen auf *NumPy*-Arrays. Ein weiterer Dekorator ist `@jitclass`, welcher dem Kompilieren von Klassen dient. In Abbildung 2.4 wird das grundsätzliche Vorgehen von *Numba* schematisch erklärt.

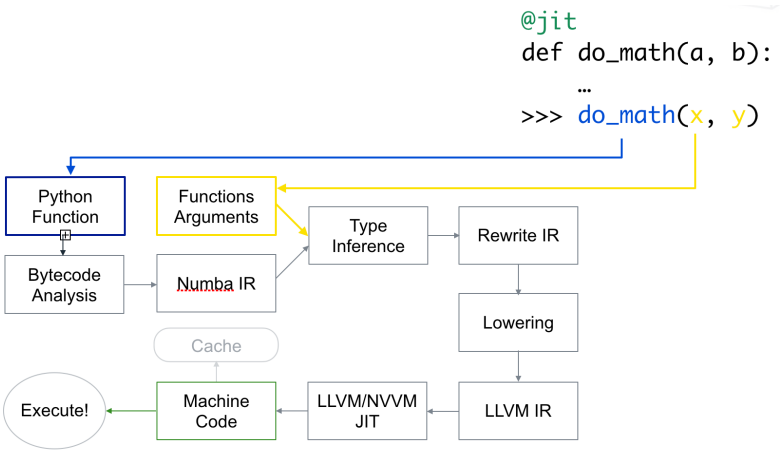


Abbildung 2.4: Funktionsweise von Numba [20]

Dafür wird der Dekorator `@njit` oberhalb der Funktionsdefinition geschrieben, um die Information, dass die Funktion kompiliert werden soll, zu übergeben. Wenn nun die Funktion das erste Mal aufgerufen wird, dann wird der Funktionsrumpf als Python-Bytecode analysiert und in eine *Numba* Zwischensprache umgewandelt. Zusätzlich werden die Typen der Argumente aus dem Funktionsaufruf genutzt, um mit der *Numba* Zwischensprache eine Typinferenz durchzuführen, um im Funktionsrumpf Typen zuzuordnen. Danach wird aus dem nun typannotierten Code eine für *LLVM* kompilierbarer Code generiert. Nun kann der *Just-in-Time Compiler* von *LLVM* genutzt werden, um Maschinencode zu generieren, welcher gecached werden, falls wieder dieselben Typen als Argumente übergeben werden. Danach kann die kompilierte Funktion ausgeführt werden. Bei der ersten Ausführung mit neuen Typen existiert ein Overhead, aber weitere Ausführungen nutzen direkt die kompilierte Variante.

2.3.4 LLVM

Bei der Implementierung folgender Projekte wird unter anderem auf Low Level Virtual Machine (*LLVM*) gesetzt. *LLVM* ist ein eigenständiger *Compiler*, welcher *Front-Ends* für unterschiedliche Programmiersprachen und *Back-Ends* für die gängigen Prozessorarchitekturen besitzt. Dadurch wird beispielsweise eine breite Anwendbarkeit eines einzelnen Optimierungspasses über mehrere Sprachen und Architekturen möglich, ohne diesen Pass jeweils anpassen zu müssen. Dies ist möglich, da *LLVM* den Code zuerst mit einem *Front-End* in die sogenannte Intermediate Representation (IR) übersetzt, welche daraufhin angepasst und optimiert werden kann. Anschließend wird der Code in IR von einem *Back-End* in die entsprechende Maschinensprache des Zielsystems übersetzt. Es existieren *Front-Ends* für die Sprachen C, C++, Haskell, Java, Python und viele Weitere. Unterstützte Zielarchitekturen sind beispielsweise x86, AMD64, PowerPC oder auch ARM [13].

In Abbildung 2.5 ist dieser Ablauf beispielhaft für Code in C, mittels des zugehörigen *Front-Ends clang*, und einer beliebigen Zielarchitektur dargestellt. Dabei übersetzt zuerst das *Front-End* den Code von C in die IR. Daraufhin werden opt-Pässe angewendet, welche

Abbildung 2.5: Beispielablauf mit Code in C[15]

den Code modifizieren, um diesen zu optimieren oder anderweitig anzupassen. Dies kann auch mehrfach erfolgen, um beispielsweise zuerst eine Optimierung durchzuführen und danach eine Anpassung zu tätigen, welche durch die Optimierung beeinflusst werden würde. Opt-Pässe sind vergleichbar mit den vom GCC-Compiler bekannten Optimierungsoptionen, welche in Sammlungen wie O0 bis O3 genutzt werden. Beispielsweise lassen sich durch Nutzung des opt-Passes *-die* (Dead Instruction Elimination) ungenutzte Instruktionen entfernen und somit das Vorkommen von totem Code verringern. Eine Weiterentwicklung dieses opt-Passes ist *-dce* (Dead Code Elimination), welcher weitere Instruktionen erkennt und löscht, die nach der Anwendung von -die ungenutzt wären. Neben opt-Pässen, die der Transformation des Codes dienen, existieren auch opt-Pässe, welche nützliche Funktionen bereitstellen. Ein Beispiel dafür ist *-view-cfg*, womit der Kontrollflussgraph für den behandelten Code, mittels der Software Graphviz [7], erstellt und angezeigt werden kann. Die Nutzung einer strengeren Optimierung kann allerdings mit einer verlängerten Laufzeit der Optimierung des Codes einhergehen. Nachdem der Code in IR wunschgemäß angepasst wurde, wird der Code schließlich mithilfe des Back-Ends in Maschinencode des genutzten Systems übersetzt.

Da in den folgenden Projekten Python genutzt wird, wird zur Verwendung von *LLVM* auf *llvmlite*, einen Python *Wrapper* für die *LLVM* C/C++-API, zurückgegriffen. Das Paket *llvmlite* ist bereits in *Numba* enthalten und ermöglicht durch die Verwendung eines Dekorierers einzelne Funktionen mit *LLVM*, anstatt des normalen Python *Interpreters*, kompilieren zu lassen. Durch die abweichende Kompilierung des Codes wird ein Laufzeitgewinn einzelner Funktionen erzielt, welche nur auf *NumPy* und weitere von *Numba* unterstützte Bibliotheken zurückgreifen. Somit kann nicht der sämtliche Code durch die Verwendung des Dekorierers beschleunigt werden, da *Numba* nur eine begrenzte Menge von Bibliotheken unterstützt.

2.3.5 Workflow-Management mit Snakemake

Zur einfachen reproduzierbaren Anwendung wurde das Workflow-Management-System *Snakemake* [10] verwendet.

In einer *Snakemake*-Datei werden dazu Regeln definiert, die beschreiben, wie aus Eingabedateien Ausgabedateien erzeugt werden.

Eine Regel weist so grundlegend einen Namen, eine Eingabedatei, eine Ausgabedatei und einen Shell-Befehl auf. Mittels dem Befehl `snakemake RULENAME` kann eine Regel ausgeführt werden. Für die Ein- bzw. Ausgabedatei wird der jeweilige Pfad angegeben. Hier besteht ebenso die Möglichkeit mehrere Dateien festzulegen. Der Shell-Befehl dient zur Generierung der Ausgabedatei aus der Eingabedatei, wobei auch Verweise auf Skripte möglich sind.

Über diese grundlegenden Optionen hinaus verfügt *Snakemake* über eine Vielzahl von Optionen, die Regeln angehängt werden können. So kann beispielsweise die Anzahl der Threads, das Cache-Verhalten oder die Nutzung des Speichers reglementiert werden. Mittels der Option `-cores, -jobs, -j` kann die zu nutzende CPU Kern Anzahl festgelegt werden, auf denen gerechnet werden soll bzw. auf denen Jobs parallel ausgeführt werden. Wird `all` als Parameter übergeben, werden alle zur Verfügung stehenden CPUs genutzt.

Darüber hinaus kann mittels `-dryrun, -n` getestet werden, ob der Workflow richtig definiert wurde. So kann bereits vor der Ausführung die Syntax eines Snakefile validiert werden. Ein weiteres hilfreiches Entwicklungswerkzeug stellt die Option `-dag` dar. Mithilfe dieser wird der Graph der Regeln zurückgegeben. Der Befehl `snakemake -forceall -dag | dot -Tpdf > dag.pdf` liefert darüber hinaus in Form der `dag.pdf` eine aufbereitete Visualisierung dieses Regelgraphen. Des weiteren kann ein *Conda*-Environment zur Bearbeitung der Jobs genutzt werden. Hierzu wird die Option `-use-conda` angeführt.

Auch Log-Dateien können festgelegt werden, in denen Informationen zur Ausführung festgehalten werden. Die Benchmark Option ermöglicht es zudem die Laufzeiten von Regeln festzuhalten. Hierbei wird eine Ausgabedatei übergeben, in der die Ergebnisse gespeichert werden.

Neben den Regeln können in der *Snakemake*-Datei Variablen, die wiederum innerhalb der Regeln genutzt werden können, definiert werden. Außerdem können einer Regel Parameter bzw. Wildcards übergeben werden. Dies erlaubt die Abstraktion von Regeln, sodass die Wiederverwendung von diesen ermöglicht wird.

2.3.6 Git und GitLab

Zur gemeinsamen Implementierung kam das verteilte Versionsverwaltungstool *Git* zum Einsatz. *Git* bietet die Möglichkeit, Änderungen, die im Laufe des Projektes durchgeführt wurden, anhand einer Historie nachzuvollziehen. Diese Snapshots, also Momentaufnahmen, werden als Commits bezeichnet. Nutzer können eine Vielzahl von Änderungen in einem Commit zusammenfassen und diese anhand der einem Commit zugehörigen Message beschreiben. Dies ermöglicht eine kollaborative Arbeit, da zu jedem Zeitpunkt erfasst werden kann, welcher Nutzer welche Änderungen und mit welcher Intention erstellt hat.

Die erstellten Commits bauen sich dabei als Commit-Historie auf, die als Branch bezeichnet wird. Dabei ist es möglich, einen Fork von einem Branch zu erstellen. Dieser zweite Branch setzt an dem letzten Commit des ursprünglichen Branch an, an dem er erstellt wurde. Somit können ab diesem Snapshot Änderungen vorgenommen werden, ohne weitere Änderungen bzw. Commits, die zukünftig an den ursprünglichen Branch angehängt werden, zu berücksichtigen. Einzelne Features beispielsweise können auf einen dafür bestimmten Branch entwickelt und auf ihre Funktionsweise überprüft werden. In einem darauffolgenden Schritt, dem sogenannten Merge, werden diese dann mit dem Master-Branch zusammengeführt. Bei einem Merge ist stets darauf zu achten, dass ein funktionsfähiger Stand erstellt wird, sodass die weitere Nutzung auch für andere Mitglieder des Projektes gewährleistet ist. Der Master-Branch aggregiert dabei alle erstellten Features. Neben dem Master-Branch können noch weitere Branches z.B. für Features, Testings oder jeweilige Releases bestehen.

Die gemeinsame Arbeitsweise mit *Git* bzw. die Organisation des Projektes zur Verwendung von unterschiedlichen Repositories und Branches für Teilaufgaben wurde jeweils gemeinschaftlich abgestimmt.

Neben der reinen Versionsverwaltung wurde die Webanwendung *GitLab* eingesetzt. *GitLab* bietet weitere Möglichkeiten, die die kollaborative Arbeit vereinfachen bzw. diese effizienter gestalten.

Hier wurde insbesondere der Issue-Tracker zu Zwecken des Projektmanagements eingesetzt. Einzelne Aufgaben bzw. Issues können hier explizit festgehalten werden. In Listen oder Kanban-Boards verwaltet, kann der Entwicklungsfortschritt dargelegt werden. Ein Kaban-Board verfügt zumeist über die drei Abschnitte „ToDo“, „Doing“ und „Done“. Weitere Abschnitte können dabei beliebig hinzugefügt werden. Dabei sind Abschnitte wie „Review“ oder „Testing“ üblich. Den einzelnen Abschnitten werden Issues zugeordnet, sodass Teammitgliedern stetig der Status, in dem sich ein Issue befindet, angezeigt wird.

Eigens erstellte Label können die Issues zusätzlich in einen entsprechenden Kontext setzen. So wurden Issues mittels Labels beispielsweise ihren Teilaufgaben zugeordnet. Zudem besteht die Option, Kommentare zu nutzen, um anderen Entwicklern zusätzliche Informationen bereitzustellen oder diese generell zur Kommunikation zu nutzen.

Des Weiteren bietet *GitLab* ein Werkzeug zur kontinuierlichen Integration, also dem Zusammenführen des Codes und automatische Ausführung von Kompilierung und Tests. In den einzelnen Repositories sind Dateien mit dem Namen `.gitlab-ci.yml` hinterlegt, die Workflows definieren, welche nach einem Push auf das jeweilige Repository ausgeführt werden. Hierbei kommt Docker zum Einsatz. Je Prozess wird ein Docker-Image geladen und entsprechende Schritte, die zuvor bestimmt wurden, durchlaufen. Im Projekt wurde so zum Beispiel kontinuierlich die Codeformatierung geprüft. Ebenso wurden vordefinierte Test durchlaufen sowie die aktuelle Testabdeckung berechnet und eine entsprechende Dokumentation automatisch generiert.

2.3.7 Testing

Um die Korrektheit des Codes zu gewährleisten, orientiert sich die Projektgruppe an dem Entwicklungsparadigma des Test Driven Developement[2] (zu deutsch *testgetriebene Entwicklung*), welches im folgenden durch das Akronym TDD abgekürzt wird. Beim TDD werden Tests geschrieben, noch bevor die Funktionalität selber implementiert wurde, um sicherzustellen, dass der entwickelte Code unmittelbar auf seine Korrektheit überprüft und eine konstante Korrektheitsprüfung aufrechterhalten wird. Konkret wird beim TDD in den folgenden Schritten vorgegangen:

1. Schreibe Tests
2. Teste, ob die Tests scheitern können
3. Implementiere die Funktion zu den Tests
4. Teste und debugge die Funktion ggf.
5. Refactoring
6. Gehe zu Schritt 1

Dieses Paradigma bietet eine Reihe an Vorteilen, da es von den Entwicklern erzwingt, erst Aufrufe der Funktionen, die zu implementieren sind, durchzugehen und sich zu überlegen, welches Verhalten die Funktionen zeigen sollen, was einerseits sicherstellt, dass die Funktion der Methode erst in einem praktischen Kontext definiert wird und außerdem, dass sie von vorne herein auf ihre Korrektheit hin überprüft wird. Ein großes Problem bei diesem Paradigma ist hingegen, dass die Implementierung der Tests viel Zeit im Vorhinein in Anspruch nimmt. Außerdem kann das frühe Schreiben der Tests in Projekten, in denen die Anforderungen nicht ganz klar sind oder sich oft ändern, schwierig sein bzw. ein häufiges Ändern der Tests erfordern.

Wir haben uns dazu entschieden, die Tests für unseren Python-Code mit dem *pytest*[11] Framework umzusetzen, da dieses eine intuitive Syntax hat und eine Reihe an umfangreichen Komfortfunktionen für das Schreiben der Tests anbietet, welche im vorinstallierten unittest Paket fehlen.

Mit *pytest* können Tests einfach in beliebigen Python-Dateien geschrieben werden, welche vom Konsolentool im lokalen Verzeichnis und allen Unterverzeichnissen automatisch gefunden werden, wenn sie das Präfix `test_` oder das Suffix `_test` haben. In solchen Dateien werden dann Funktionen, die ebenfalls das Präfix `test_` oder Suffix `_test` haben,

automatisch von *pytest* aufgerufen und Tests, die bei ihrer Ausführung Exceptions werfen oder Assertions nicht erfüllen, als Fehler abgefangen.

Gegenüber den Assertions, die über das in Python vorinstallierte unittest Framework zur Verfügung stehen, bieten die `assert` Anweisungen den Vorteil, dass sie zwar beliebige Boolesche Ausdrücke akzeptieren, aber im Falle eines Fehlers bei der Auswertung die einzelnen Komponenten der Ausdrücke auseinandernehmen und so ein schnelleres Verständnis und einen damit verbundenen effizienteren Workflow erlauben.

Eine weitere Komfortfunktion von *pytest* ist die Existenz von Fixtures. Fixtures sind Funktionen, welche mit dem Dekorator `pytest.fixture` versehen wurden und dem Zweck dienen, gleichbleibende Testspezifische Umgebungen aufzusetzen. Eine Fixture kann an einer Stelle im Code definiert werden und muss dann nur noch bei den Tests, die sie benutzen sollen, mit dem Namen der Fixture als Parameter übergeben werden, was darin resultiert, dass vor dem Test eben diese Fixture aufgerufen wird und ihr Rückgabewert im Test als Parameter zur Verfügung steht, um z.B. auf eine Datei, die per Rückgabe definiert ist, zuzugreifen oder andere Objekte zur Verfügung zu haben, die viele Tests wiederholt im gleichen Zustand benötigen.

Noch eine Komfortfunktion von *pytest* ist die Parametrisierung von Tests, welche es erlaubt, denselben Code für mehrere Tests mit einer Reihe von Parametern auszuführen. Dazu wird der Dekorator `pytest.mark.parametrize` gefolgt von einer Liste von Variablen, welche als Parameter der Test-Funktion benutzt werden, definiert. Auf die Liste der Variablen folgt schließlich eine Liste von Belegungen dieser Variablen, für welche der Test schließlich ausgeführt werden soll.

Tests, die aus einem definierten Grund nicht oder nur unter bestimmten Bedingungen ausgeführt werden sollen, können mit dem Dekorator `pytest.mark.skip` versehen werden. Mittels `pytest.mark.skipif(Bedingung)` kann dies an eine Bedingung geknüpft werden. Dies erlaubt es, Tests potentiell auf verschiedene Hardwaresysteme oder verschiedene Installationen anzupassen.

Schließlich können Tests auf das Werfen von Exceptions für ungültige Eingaben oder dergleichen mithilfe des Dekorators `pytest.mark.xfail(raises=Exception-Typ)` umgesetzt werden, welcher den Test nur akzeptiert, wenn die geforderte Exception wirklich geworfen wird.

2.3.8 Dokumentation mit Sphinx

Sphinx [25] ist ein Tool, mit dem es einfach ist, intelligente und schöne Dokumentationen zu erstellen. Ursprünglich wurde es entwickelt, um Python Dokumentationen zu erstellen. Es eignet sich gut, um Softwareprojekte zu dokumentieren. Dabei werden aus den Quellcode-Dateien (beispielsweise .py-Dateien) die Doc-Strings automatisch erkannt und deren Inhalt in .rst-Dateien übertragen. Diese enthalten zusätzlich noch weitere Informationen wie z.B. die hierarchische Struktur des Projektes. Dieses Format ist Auszeichnungssprache, die besonders lesbar ist. Durch ihren einfachen Aufbau lassen sich .rst-Dateien einfach in andere Dateiformate umwandeln. Im Anschluss werden je nach Ausgabeformat (z.B. *HTML*) die Informationen aus den .rst-Dateien entnommen und daraus die Ausgabe-Dateien erstellt. Dabei wird automatisch das Grundgerüst generiert, das die Referenzen und Verweise zwischen den einzelnen *HTML*-Seiten setzt. So kann bequem zwischen den Seiten navigiert werden. Zusätzlich wird eine `index.html` erstellt, die als Hauptseite dient. Auf dieser können allgemeine Information zum Projekt ergänzt werden, wie beispielsweise eine Projektbeschreibung oder eine Installationseinleitung. Die anderen Ausgabe-Dateien repräsentieren die einzelnen Quellcode-Dateien. In diesen werden die Inhalte der Doc-Strings angezeigt und einzelne Code-Segmente hervorgehoben. Außerdem wird automatisch eine Suchfunktion hinzugefügt, mit der nach Funktionen oder einzelnen Begriffen gesucht werden kann.

```yaml
name: lcfilter
channels:
  - bioconda
  - conda-forge
dependencies:
  - python=3.8
  - numpy
  - numba
  - pytest
  - pytest-cov
  - matplotlib
  - snakemake
  - black
  - pre-commit
```

Listing 2.2: Inhalt der environment.yml des Low Complexity Filters.

2.3.9 Conda

Conda[6] ist eine Open-Source-Software zur Verwaltung von Softwarepaketen und Umgebungen. Installiert werden kann *Conda* entweder als Minimalversion *Miniconda* oder mit einer großen Zahl an vorinstallierten Paketen als *Anaconda*. *Conda* als Paketverwaltung kann dazu genutzt werden, Pakete für viele verschiedene Programmiersprachen herunterzuladen und zu installieren. Dabei kann festgelegt werden, welche Versionen der Pakete gewünscht sind. Außerdem überprüft *Conda* bei der Installation grundsätzlich die Kompatibilität der installierten und zu installierenden Pakete. Zu den unterstützen Programmiersprachen gehören neben Python auch R, Java, C/C++ und weitere Sprachen.

In seiner Funktion als Umgebungsverwaltung können mit *Conda* verschiedene Umgebungen erstellt und verwaltet werden, die jeweils unterschiedliche installierte Softwarepakete enthalten können. So ist es beispielsweise auch möglich, eine Umgebung für Python 2 und eine andere Umgebung für Python 3 zu verwenden. Zwischen den verschiedenen Umgebungen kann einfach gewechselt werden, die enthaltenen Pakete stehen dann sofort in ihren jeweiligen Versionen zur Verfügung. Spezifikationen für verschiedene Umgebungen können in Dateien festgehalten werden, sodass die Festlegung der genutzten Pakete innerhalb einer Arbeitsgruppe einfacher umzusetzen ist. In Listing 2.2 ist der Inhalt der Datei `environment.yml` zu sehen, die für den Low Complexity Filter (siehe Kapitel 4) verwendet wird. In der ersten Zeile wird der Name der Umgebung festgelegt. Dann werden die verschiedenen *channels* festgelegt, über die Pakete gesucht und installiert werden sollen. Unter `dependencies` werden die zu installierenden Pakete angegeben. Die Versionsnummer kann ebenfalls festgelegt werden, wie das bei Python geschehen ist. Im Rahmen der Projektgruppe wurde *Conda* zur Verwaltung der benötigten Pakete für die Ausführung der verschiedenen entwickelten Programme verwendet.

[6] `https://docs.conda.io/en/latest/`

Kapitel 3 — Miniprojekte

Zu Beginn befasste sich die Projektgruppe mit drei kleineren Projekten, im Folgenden Miniprojekte genannt. Die Miniprojekte dienten einerseits zur Förderung der kollaborativen Arbeit und Vertiefung der erlernten Grundlagen. Andererseits konnten durch diese bereits grundlegende Funktionen implementiert werden, die im späteren Verlauf wiederverwendet und verbessert werden konnten. In diesem Kapitel werden die Miniprojekte vorgestellt und deren Ergebnisse zusammengetragen.

Das erste Miniprojekt befasste sich mit dem Lesen und der Verarbeitung von Genomdaten, welche im *FASTQ*-Format vorliegen. Dafür wurden Funktionen implementiert, welche alle *FASTQ*-Reads einer bzw. mehrerer Dateien einlesen und diese dann hinsichtlich unterschiedlicher Merkmale analysieren. Diese vorerst einfachen Merkmale waren beispielsweise die Länge der Sequenz oder das Auftreten einer bestimmten Basenfolge innerhalb der Reads.

Das zweite Miniprojekt befasste sich mit einem neuen Feature des Pakets *Numba*, den JIT-kompilierten Klassen. Dabei wurden Implementierungen von Klassen und Funktionen, die jeweils JIT-kompiliert wurden, gegenüber gestellt und hinsichtlich ihrer Laufzeit gemessen. Somit konnte eine Entscheidung getroffen werden, welche der beiden Herangehensweisen in den folgenden Projekten verwendet werden soll.

Im dritten Miniprojekt wurde eine effiziente Verarbeitung von k-meren, also von Stücken einer Sequenz der Länge k, angestrebt. Dazu wurden unterschiedliche Lösungsansätze gesammelt, gemessen und schließlich abgewogen, welcher Lösungsansatz im weiteren Verlauf genutzt werden soll.

3.1 Codegenerierung für FASTQ-Dateiverarbeitung

Die Ergebnisse der Miniprojektgruppe Codegenerierung für *FASTQ*-Dateiverarbeitung werden in diesem Abschnitt vorgestellt. Das Ziel der Gruppe war es, *FASTQ* Dateien einzulesen und jeden darin enthaltenen Read zu verarbeiten. Dabei wurde gemessen, ob es einen Laufzeitunterschied von verschiedenen Implementierungen gibt. Diese wurden evaluiert und daraufhin abschließend eine Entscheidung getroffen, welche Implementierung weiter verfolgt werden soll.

3.1.1 Zielsetzung

Das konkrete Ziel der Miniprojektgruppe besteht darin, *FASTQ* Dateien sowohl als Single-Read als auch als Paired-Read, zu verarbeiten. Dabei sollen die Performanz von generiertem und nicht generiertem Code, sowie die parallele Verarbeitung auf mehreren Threads gemessen werden.

Zur Überprüfung der Geschwindigkeit werden einfache Varianten einer Verarbeitung implementiert und die Ergebnisse der verschiedenen Implementierungen verglichen. Zur Evaluierung der Laufzeiten wurden folgende Methoden implementiert:

- Mustersuche im Read,
- Häufigkeiten der einzelnen Basen im Read,
- Längenbestimmung der Reads.

3.1.2 Ansätze

Die Implementierung sollte mittels der Programmiersprache Python [9] durchgeführt werden. Zu Beginn wurden unterschiedliche Implementierungsmöglichkeiten mit Hilfe von verschiedenen Python Paketen oder auch anderen Programmiersprachen gesammelt. Die folgenden Implementierungen wurden dabei diskutiert:

- Python
- *Rust* [28]
- *Biopython* [4]
- *Seq* [27]

Die Programmiersprache Seq wurde untersucht, da diese Sprache sich für Sequenzanalysen bot und eine schnellere Laufzeit erwartet worden ist. Aufgrund dessen, dass die anderen Miniprojektgruppen ihre Implementierung in Python durchführten und wir bei einer späteren Einbindung Kompatibilitäts-Schwierigkeiten befürchteten, wurde der Implementierung in der Programmiersprache Seq relativ kurz nachgegangen. Insbesondere gäbe es Kompatibilitätsprobleme, da Seq nur für Unix-Systeme verfügbar ist und wir Windows Nutzer in der Projektgruppe haben. Zeitnah wurde sich gegen die Anwendung von Biopython entschieden, da weitere Paketabhängigkeiten vermieden werden sollten und Kompatibilitätsprobleme mit *Numba* und *NumPy* befürchtet wurden. Kleinere Laufzeitvergleiche dieser Optionen mit Python wurden schon in einem sehr frühen Stadium durchgeführt und verworfen, sodass für diese Implementierungen keine Benchmarks vorliegen. Insbesondere war die Möglichkeit, die Verarbeitung zu parallelisieren, bei dem derzeitigen Wissensstand schwieriger zu realisieren.

Es folgte die Erstellung eines Prototypen in Python, um erste Benchmarking Ergebnisse zu erhalten. Zusätzlich wurde die Implementierung mit Rust länger verfolgt, da Felix Erfahrungen in Rust besitzt und in kürzester Zeit einen Prototypen implementieren konnte. Bei Rust handelt es sich um eine schnelle und typsichere Compilersprache. Hierdurch versprachen wir uns einen Performanzgewinn, welcher letzten Endes nicht erzielt wurde, wie aus Abbildung 3.1 hervorgeht. Dort wurde die Performanz der Python und Rust Implementierung für die Operationen zum Zählen der Häufigkeit der einzelnen Basen und der Längenbestimmung der Reads illustriert. In Abbildung 3.1 wird gezeigt, dass die Implementierung mit Rust ungefähr um den Faktor 2 langsamer ist als die Implementierung mit Python. Dies gilt sowohl für die Methode zum Zählen der Häufigkeiten der einzelnen Basen als auch für die Längenbestimmung der Reads. Aus diesem Grund wurde die Implementierung mit Rust nicht weiter verfolgt. Die Messung erfolgte mit dem derzeitig vorhandenen Prototypen der Implementierung in Python. Auf die endgültige Implementierung mit Codebausteinen wird in Kapitel 3.1.4 näher eingegangen. Außerdem wurde die Verarbeitung der Reads beim Einlesen realisiert, wodurch das gesamte Projekt in Rust implementiert werden müsste. Somit wurde von allen Teammitgliedern die Implementierung in Python verfolgt. Zur Optimierung der Laufzeit wurden die Python Pakete *Numba* [12] *NumPy* [30] genutzt.

3.1.3 Xengsort

Das Einlesen von *FASTQ*-Dateien basiert auf der Implementierung von *Xengsort* [32]. *Xengsort* ist ein Tool, welches eine effiziente Zuordnung von DNA Sequenzen bei Xenotransplantationen als Ziel hat. Dies wird als *xenograft sorting problem* beschrieben. Bei einer Xenotransplantation (*xenograft*) handelt es sich um eine Transplantation, bei der

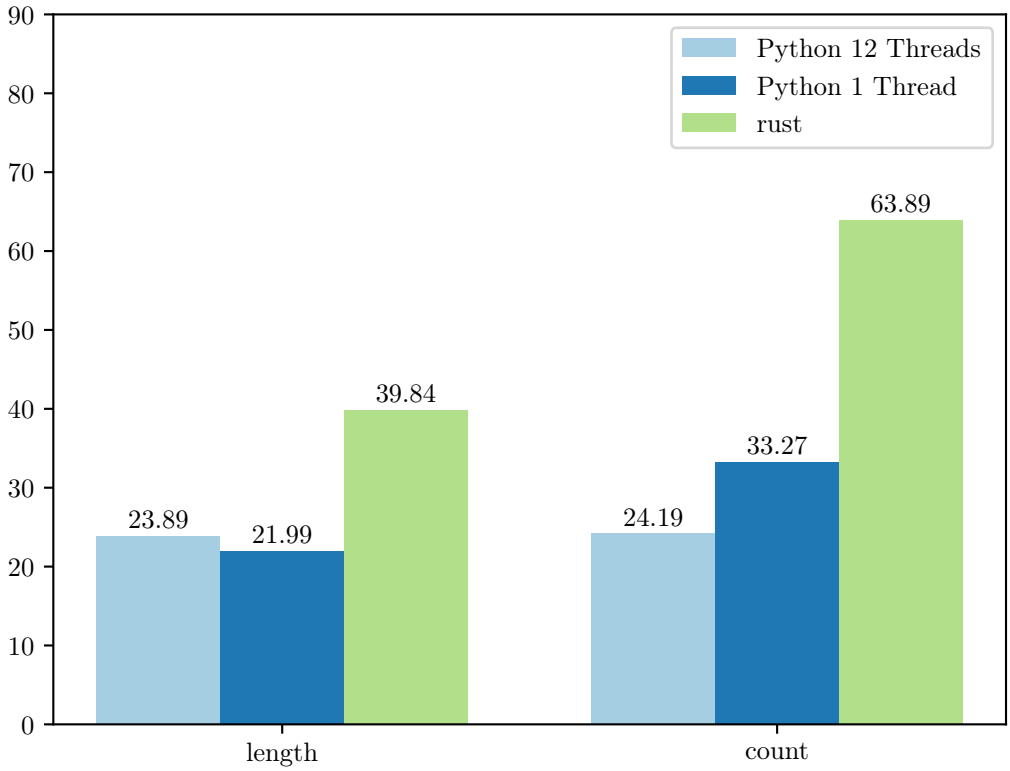


Abbildung 3.1: Laufzeitvergleich zwischen einer frühen Python Implementierung mit einem und 12 Threads und der Implementierung in Rust

Spender und Empfänger unterschiedlichen biologischen Spezies angehören. *Xengsort* vergleicht beispielsweise eingegebene DNA Sequenzen jeweils mit vorher geladenen Sequenzen von Mensch und Maus. Darauf basierend wird eine Aussage getroffen, ob die Sequenz dem Menschen, Mäusen, beiden oder keiner Gattung zugeordnet werden kann oder auch keine Aussage darüber getroffen werden kann. Die Sequenzen werden dann in dementsprechende Ausgabedateien geschrieben, die den fünf vorher genannten Zuordnungsarten entsprechen. *Xengsort* kann sowohl auf Dateien im *FASTA*-, als auch im *FASTQ*-Format arbeiten. Dieses Miniprojekt befasst sich allerdings nur mit der Implementierung für Eingabedateien im *FASTQ*-Format. Das Grundkonstrukt des Einlesens beruht auf zwei Arrays, welche eine zuvor festgelegte Größe besitzen, die mit der angegebenen Threadanzahl multipliziert wird. Das erste Array (*buf*) dient als Puffer, in den sämtliche Zeichen der Eingabedateien als Byte gelesen werden, bis dieser voll ist. Anschließend werden im zweiten Array (*linemarks*) Indizes des Puffers für jeden Eintrag in der Datei festgehalten. Dieses Array ist eine Matrix der Größe 4 x Anzahl der Sequenzen, um folgende Indizes für jeden Eintrag zu enthalten:

- Beginn der Sequenz
- Ende der Sequenz
- Beginn des Eintrags
- Ende des Eintrags

Zusätzlich besitzt *Xengsort* die Funktion `quick_dna_to_2bits`, welche eine einfache Vorsortierung ermöglicht. Dabei werden die in den Puffer geladenen Byte-Werte der Basen auf Werte von 0 bis 3 gesetzt, um diese danach schneller identifizieren und verarbeiten zu können, da beispielsweise auftreten könnte, dass die Eingabedatei in der Sequenz kleine Zeichen anstatt großer Zeichen enthält.

- A bzw. a: 0
- C bzw. c: 1
- G bzw. g: 2
- T bzw. t: 3 (DNA)
- U bzw. u: 3 (RNA)

Alle anderen Einträge einer Sequenz, die beispielsweise als N angegeben werden, werden in diesem Schritt auf den Standardwert 4 gesetzt. Im weiteren Verlauf dieses Miniprojektes werden allerdings hauptsächlich die Werte 0 bis 3 betrachtet.

Da die Eingabedateien in den meisten Fällen größer als der Puffer sind und möglichst Arbeitsspeicher während der Ausführung gespart werden soll, wird die Eingabedatei nur eingelesen, bis der Puffer voll ist und der Inhalt des Puffers anschließend verarbeitet. Diese beiden Schritte werden wiederholt, bis die Eingabedatei vollständig verarbeitet wurde. Die Kombination aus dem gefüllten Puffer *buf* und den zugehörigen Indizes in *linemarks* als Paar wird *chunk* genannt. Besonders aufgrund der Limitierung der Puffergröße kommt es vor, dass ein Eintrag unvollständig im Puffer sein kann. Dies wird abgefangen, indem nur vollständige Einträge mit allen vier Einträgen in *linemarks* verarbeitet werden. Sollte also ein unvollständiger Eintrag am Ende des Puffers existieren, wird dieser ignoriert und beim nächsten Durchlauf als erster Eintrag in den Puffer gelesen. Zusätzlich ermöglicht diese Herangehensweise eine Parallelisierung bei der Verarbeitung der zuvor gelesenen Daten, indem die Puffer jeweils aufgeteilt und gleichzeitig verarbeitet werden.

3.1.4 Implementierung

Die erste grundlegende Idee für die Generierung war die Nutzung von Strings, welche als Parameter übergeben werden und die Methode für die Verarbeitung der *FASTQ*-Dateien beinhalten. Diese Idee wurde in der endgültigen Version verworfen und die Methoden und Codebausteine werden nun an den geeigneten Stellen in der Methode zur Verarbeitung implementiert. Dabei wird eine Funktion `process` genutzt, welche die Verarbeitung auf den Teilsequenzen der *FASTQ*-Datei durchführt. Trotzdem soll hier die erste Implementierung mittels Strings für die Codebausteine erläutert werden, da dies die grundlegende Idee für die Implementierung beinhaltet. Die Ausführung von Codebausteinen, welche in Strings abgespeichert werden, wie in Listing 3.1 und 3.2 beschrieben, wird durch Nutzung von `exec`[1] ausgeführt. Hierbei beschreibt in Listing 3.1 der String `init_count` die Initialisierung der Verarbeitung zum Zählen der Häufigkeiten von Basen. Dabei wird sich allerdings nur auf $ACGT$ beziehungsweise $ACGU$ beschränkt, sämtliche abweichenden Basen werden ignoriert. Dafür wird ein Array der Größe 4 angelegt und mit 0 initialisiert, worin später die Häufigkeiten von den Basen hochgezählt werden. Da die Basen, wie in Kapitel 3.1.3 beschrieben, bereits vorher mithilfe der Funktion `quick_dna_to_2bits` auf die Werte 0 bis 3 gesetzt werden, lassen diese sich effizient in einem Array der Größe 4 abspeichern.

In Listing 3.2 wird der String `process_count` angelegt, welcher die Methode `processor` definiert. Diese Methode dient dem Zählen der Häufigkeiten der Basen. Dabei enthält der

[1] `https://docs.python.org/3/library/functions.html`

```
1 init_count = """
2 counts = np.zeros(4, np.uint64)
3 """
```

Listing 3.1: Initialisierungs-String für die Zählung der Häufigkeit der einzelnen Basen

Parameter `buf` die Zeichen der ersten *FASTQ*-Datei und der Parameter `buf2` die Zeichen der zweiten *FASTQ*-Datei, falls es sich um eine Paired-End-Datei handelt. Dafür wird über den Chunk, welcher übergeben wird, iteriert und jede Sequenz mittels der Funktion `quick_dna_to_2bits` in eine 2-Bit Darstellung umgewandelt. Anschließend wird über die einzelnen Basen der Sequenz iteriert und das Vorkommen der Basen gezählt. Dies wird zunächst für die erste Datei durchgeführt und falls es eine weitere Datei gibt, wird für diese ebenfalls jede Sequenz umgewandelt und danach das Vorkommen der Basen gezählt. Beim Zählen wird die Position der Basis im vorher initialisierten Array um den Wert 1 erhöht. Nach der Verarbeitung wird im String `output_count` in Zeile 17 das Array `counts` ausgegeben.

```
1  process_count="""
2  @njit(nogil=True)
3  def processor(buf, linemarks, buf1=None, linemarks1=None):
4      n = linemarks.shape[0]
5      for i in range(n):
6          sq = buf[linemarks[i, 0]:linemarks[i, 1]]
7          quick_dna_to_2bits(sq)
8          for acgt in buf[linemarks[i][0]:linemarks[i][1]]:
9              counts[acgt] += 1
10         if not buf1 is None:
11             sq1 = buf1[linemarks1[i, 0]:linemarks1[i, 1]]
12             quick_dna_to_2bits(sq1)
13         for acgt1 in buf1[linemarks1[i][0]:linemarks1[i][1]]:
14             counts[acgt1] += 1
15     return counts
16 """
17 output_count = """
18 print(counts)
19 """
```

Listing 3.2: Verarbeitungs- und Ausgabe-Strings für die Zählung der Häufigkeit der einzelnen Basen

Um das Auftreten eines Musters in den Sequenzen zu zählen, wurden ebenfalls Codebausteine geschrieben, welche in Listing 3.4 enthalten sind. Dafür ist die Initialisierung in Listing 3.3 notwendig, in welcher verschiedene Varianten der Darstellung der Basen in Zahlen umgewandelt werden. Dies wurde zunächst naiv implementiert und wird in späteren Projekten mit schnelleren Methoden realisiert. Danach wird der zu suchende String in ein *NumPy*-Array abgespeichert, um eine Überprüfung per Sliding-Window zu ermöglichen.

In Listing 3.4 wird der `processor` definiert, welcher zusätzlich ein Parameter `pattern` erhält, das zuvor beim Aufruf der Generierungsfunktion übergeben wurde. Die Sequenzen werden wieder mit der Funktion `quick_dna_to_2bits` verarbeitet und die Länge des Musters, welches gesucht wird, bestimmt. Danach wird mittels der Methode `rolling_window`

```
1  init_pattern = """
2  counts = np.zeros(4, np.uint64)
3  pattern = pattern.replace("a","0")
4  pattern = pattern.replace("A", "0")
5  # ... other base translations
6  patTmp = []
7  for char in pattern:
8      patTmp.append(int(char))
9      pat = np.array(patTmp)
10
11 """
```

Listing 3.3: Initialisierungs-String für die Mustersuche im Read

das Muster über die Sequenz iteriert und überprüft, ob es dem Muster entspricht. Daraus erhalten wir Boolean-Arrays, welche über `x` zugegriffen werden, welche jeweils die Länge des Musters besitzen und nur vollständig `True` sind, wenn alle Basen im Rolling-Window dem des Musters entsprechen. Mithilfe der *NumPy*-Funktion `np.all` kann dadurch effizient überprüft werden, ob zu diesem Zeitpunkt alle Positionen des Arrays `x` den Wert `True` enthalten und somit das Muster gefunden wurde. Dabei wird sowohl das Auftreten des Musters in dem Read gezählt, als auch die Häufigkeiten des nicht Vorhandenseins. Dies geschieht im Fall von zwei Eingabedateien getrennt, weshalb das Array erneut die Größe 4 besitzt. Nach der Verarbeitung steht im Array an den Positionen 0 und 2 die Häufigkeiten des Aufkommens des Musters in der Datei und an den Positionen 1 und 3 die Anzahl der Reads, in der das Muster nicht vorkam. Dies wird durch den String in Zeile 29 ausgegeben.

Aufgrund der simplen Implementierung, welche lediglich die Länge der einzelnen Sequenzen festhält und deren Auftreten zählt, und den Überschneidungen mit der Methode zum Zählen der Häufigkeiten der Basen wird auf eine nähere Erläuterung verzichtet. Der Codebausteine zur Verarbeitung entspricht einer Vereinfachung des Codebausteins in Listing 3.2.

Allgemein können Strings einer Methode zur Generierung eines *FASTQ*-Readers übergeben werden. Diese Strings werden in einen bestehenden *FASTQ*-Reader eingebaut, sodass dieser die Verarbeitung, wie in den Strings definiert, durchführt. Im Listing 3.5 wird in Zeile 1 die Funktion `make_process_read_from_fastq` definiert. Dieser Funktion kann zusätzlich zur Anzahl der Threads, der Angabe, ob es sich um eine Paired-End-Datei handelt, die jeweiligen Puffer- und Chunkgrößen, mit denen gearbeitet werden soll, übergeben werden. Zusätzlich zu den zuvor beschriebenen String Variationen können ebenfalls Strings für die Verarbeitungsschritte eines gesamten Chunks übergeben werden. In Zeile 6 wird der String `init` mittels `exec` ausgeführt und im globalen Namespace geladen. Danach wird in Zeile 7 die Definition der Funktion `processor` ausgeführt und aufrufbar gemacht. Es wird überprüft, ob es sich um eine Paired-End Datei handelt und anschließend die passende Einlesemethode ausgewählt. Danach wird durch die Chunks iteriert und in Zeile 14 `processor`, welcher in Zeile 7 definiert wurde, ausgeführt. Nach der Verarbeitung der Chunks werden, falls ab Zeile 16 definiert, eine mögliche Ausgabe oder weitere Verarbeitung durchgeführt. Zum Schluss wird die im String `output` definierte Ausgabe in Zeile 21 ausgeführt.

```
process_pattern="""
@njit(nogil=True)
def processor(buf, linemarks, buf1=None, linemarks1=None, pattern):
    counts = np.zeros(4, np.uint64)
    n = linemarks.shape[0]
    for i in range(n):
        sq = buf[linemarks[i, 0]:linemarks[i, 1]]
        sq1 = buf1[linemarks1[i, 0]:linemarks1[i, 1]]
        quick_dna_to_2bits(sq)
        quick_dna_to_2bits(sq1)
        patLen = pattern.shape[0]
        contained = False
        contained1 = False
        for x in (rolling_window(sq, patLen) == pattern):
            if np.all(x):
                counts[0] += 1
                contained = True
        for x in (rolling_window(sq1, patLen) == pattern):
            if np.all(x):
                counts[2] += 1
                contained1 = True
            if not contained:
                counts[1] += 1
            if not contained1:
                counts[3] += 1
    return counts
"""

output_pattern = """
print(counts)
"""
```

Listing 3.4: Verarbeitungs- und Ausgabe-Strings für die Mustersuche im Read

3.1.5 Benchmarks

Im Folgenden werden die Ergebnisse der Benchmarks illustriert. Die Messungen wurden auf online bezogenen *FASTQ* Dateien[2] ausgeführt, die bei der Sequenzierung der DNA einer Maus ermittelt wurden. Diese besitzen einzeln als Single-File eine Größe von 16 GB entpackt und 4 GB verpackt und somit bei Verwendung beider Paired-Files eine Größe von 32 GB bzw. 8 GB. Dabei wurde jeder Test 10 mal durchgeführt und das arithmetische Mittel der Laufzeit berechnet. Die Tests wurden sowohl auf einem Thread, als auch parallel auf 12 Threads ausgeführt und dokumentiert. Die Messungen wurden auf einem System mit folgender Hardware durchgeführt: Ryzen 5 3600x (6x 3,8 GHz, 12 Threads), 2x 8 GB DDR4 Speicher mit 3000 MHz, Intel 660P NVMe SSD(1800 MB/s read/write). Als Betriebssystem wurde macOS Catalina 10.15.4 genutzt.

In Abbildung 3.2 wird der Vergleich zwischen dem Einlesen von entpackten *FASTQ*-Dateien und verpackten *FASTQ*-Dateien gezeigt. Dabei wurden die verpackten *FASTQ* Dateien mit *zcat* oder *gzcat* mittels Aufruf in der Kommandozeile übergeben. Das Stück-

[2] https://www.ebi.ac.uk/ena/browser/view/SRR9130495

```
def make_process_read_from_fastq(threads, pairs, init, process,
↪  chunk_handling, chunk_output, output, bufsize=2**23,
↪  chunkreads=2**23//200):
    ...
    def process_read_from_fastq(fastq):
        with ThreadPoolExecutor(max_workers=threads) as executor:
        #initialization depending on mode and paired
        exec(init, globals())
        exec(process, globals()) #loads definition into global space
        #... case for Paired-End reading
            readMethod = fastq_chunks(fastq, bufsize=bufsize * threads,
            ↪  maxreads=chunkreads * threads)
        #processing depending on mode and paired
        for chunk in readMethod:
            borders = get_border(chunk[1], threads)
            #...case for Paired-End processing
                futures = [executor.submit(processor, chunk[0],
                ↪  chunk[1][borders[i]:borders[i + 1]]) for i in
                ↪  range(threads)]
            #handling of each result
            for i in as_completed(futures):
                result = i.result()
                exec(chunk_handling)
                exec(chunk_output)

        exec(output, globals())
        return counts

    return process_read_from_fastq
```

Listing 3.5: Methode zum Generieren zur Verarbeitung von FASTQ-Dateien

weise Lesen aus den Dateien erfolgte also durch das System und nicht durch das Programm. Hierbei wurde kein signifikanter Unterschied zwischen der Nutzung von *zcat* und *gzcat* festgestellt, während der Unterschied zu entpackten Dateien signifikant ist. Das Einlesen und Verarbeiten der komprimierten Dateien dauert in beiden Fällen auf einem Thread circa 65 Sekunden und auf 12 Threads circa 55 Sekunden. Der Aufruf mit entpackten Dateien ist in beiden Fällen um circa den Faktor 2 schneller. Die Laufzeit mit einem Thread beträgt 32 Sekunden und mit 12 Threads 24 Sekunden. Durch die Verwendung mehrerer Threads wurde ein größerer Laufzeitgewinn erwartet, allerdings betrifft dies nur die Verarbeitung der Daten, da das Einlesen nicht parallelisiert erfolgt. Somit ist die Laufzeit zum Teil an die Lesegeschwindigkeit gebunden. Zusammenfassend lässt sich daraus schließen, dass der Aufruf mit entpackten Dateien um den Faktor 2 schneller ist, als der Aufruf der verpackten Dateien. Dies führt zu unterschiedlichen Laufzeiten bei der Verarbeitung größerer *FASTQ* Dateien. Allerdings muss auch beachtet werden, dass der Speicherverbrauch der Dateien auf der Festplatte mit komprimierten Dateien weitaus niedriger ist und das Entpacken vor dem Aufruf weitere Zeit in Anspruch nimmt. Ebenso könnte durch das vorherige Entpacken der Dateien insgesamt Zeit gespart werden, falls das Programm auf diesen Dateien so oft aufgerufen wird, dass die Laufzeitersparnis die Zeit des Entpackens übertrifft.

Die Ergebnisse der Benchmarks der verschiedenen Prozesse für jeweils generierten und

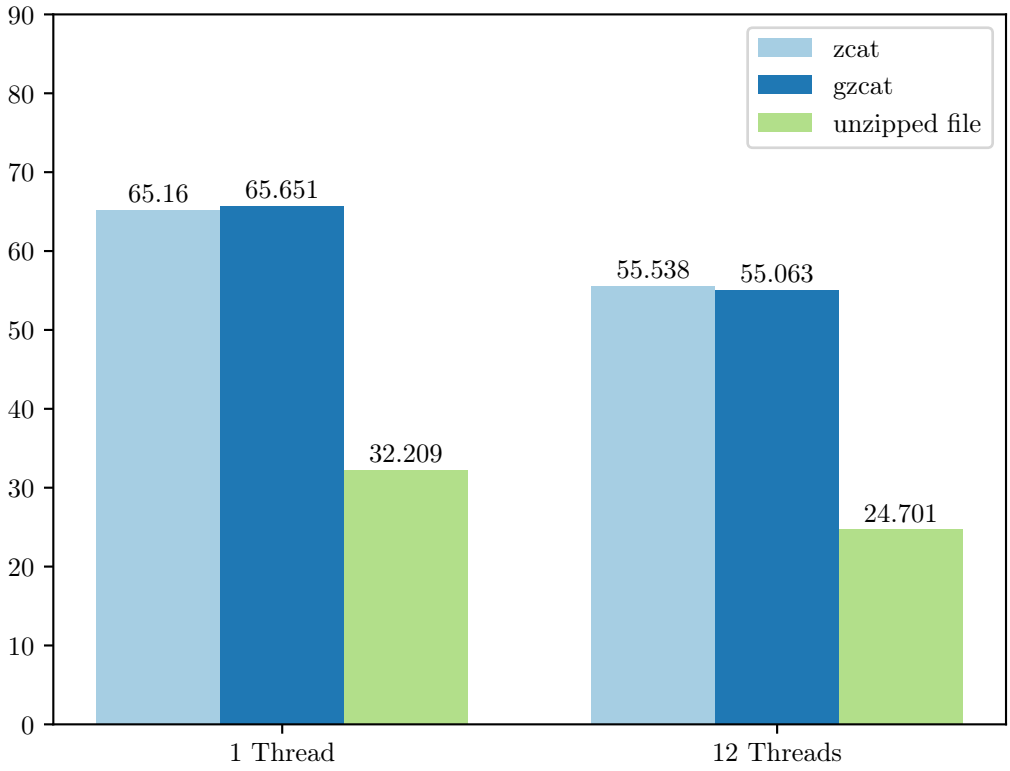


Abbildung 3.2: Laufzeitvergleich vom Einlesen zwischen entpackten und unverpackten *FASTQ*-Dateien, auf einem und 12 Threads

nicht-generierten Code sind im Folgenden aufgeführt. Abbildung 3.3 zeigt die durchschnittlichen Laufzeiten der drei Methoden zur Mustersuche nach dem zuvor übergebenen Parameter *TGCC* (pattern), Bestimmung der Häufigkeiten der einzelnen Basen (count) und der Längenbestimmung jeder Sequenz (length). Hierbei wurden die Tests sowohl auf Single-End-Reads, als auch auf Paired-End-Reads durchgeführt. Bei der Mustersuche wurde dabei die Anzahl der Treffer und Nicht-Treffer gezählt, sowohl für Single-End-Reads, als auch Paired-End-Reads. Es wurden für jede Kombination aus Methode, verwendete *FASTQ* Dateien und Art des vorliegenden Codes, 10 Durchläufe ausgeführt. Dabei ist festzustellen, dass es keinen nennenswerten Unterschied zwischen generiertem und nicht generiertem Code gibt. Hierfür sind die Laufzeiten für die einzelnen Methoden sehr nah beieinander und unterscheiden sich nur um weniger als 5%. Die Laufzeiten zwischen Single-End-Reads und Paired-End-Reads unterscheiden sich jeweils um circa den Faktor 2. Die Einbindung von Code mittels Strings und die Ausführung dieser durch `exec` ergeben also keine nennenswert erhöhten Laufzeitkosten.

3.1.6 Fazit

Als Ergebnis der Miniprojektgruppe lässt sich festhalten, dass ein Programm erstellt wurde, mit dem *FASTQ* Dateien eingelesen werden können, sowohl als Single-End-Reads als auch als Paired-End-Reads. Dabei können die Dateien entpackt vorliegen oder auch komprimiert übergeben werden. Dies erfolgt per Aufruf in der Kommandozeile, je nach Betriebssystem, mittels beispielsweise *zcat* auf Unix-Systemen. Auf den Reads können dann verschiedene

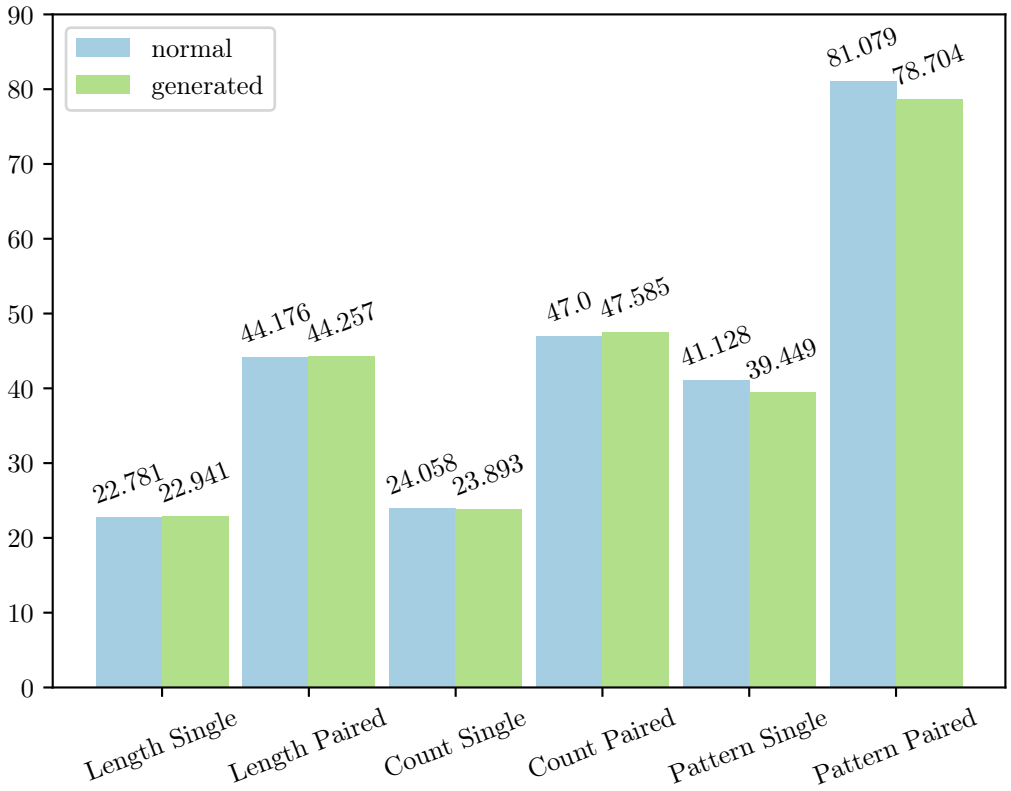


Abbildung 3.3: Laufzeitvergleich zwischen generiertem und nicht-generiertem Code für die drei Methoden zur Mustersuche, Bestimmung der Häufigkeiten der einzelnen Basen und der Längenbestimmung für Single-End-Reads und Paired-End-Reads

Methoden ausgeführt werden. Dies umfasst zum einen eine Mustersuche, mit der nach einem Teilstring gesucht werden kann, die vorher übergeben wird. Außerdem existieren Methoden zur Bestimmung der Häufigkeit der einzelnen Basen und zur Längenbestimmung des Reads. Die Ergebnisse der Benchmarks Abschnitt 3.1.5 haben gezeigt, dass es keinen großen Performanzunterschied zwischen generiertem und nicht generiertem Code gibt. Für die Übersichtlichkeit in der weiteren Implementierung wurde von der Nutzung von Strings in den Aufrufen von `make_process_read_from_fastq` abgesehen und die Strings werden für weitere Anwendungen direkt in der Funktion implementiert.

Zusammenfassend wird das Aufrufen des Programms mit entpackten und verpackten Dateien im kommenden Projekt unterstützt und optimiert. Die Ergebnisse der Miniprojektgruppe dienen im weiteren Verlauf der Projektgruppe als Grundgerüst zum Einlesen und Verarbeiten von *FASTQ* Dateien. Diese werden im nächsten Schritt bei der Implementierung eines low complexity Filters verwendet.

3.2 JIT-Kompilierte Klassen und Funktionen

Die Python Bibliothek *Numba* [12] bietet die Möglichkeit, Teile von Pythonprogrammen mittels JIT-Kompilierung in Maschinencode zu übersetzen. Zum Einsatz kommt dabei das Compilerprojekt *LLVM* [13]. Der größte durch *Numba* gewonnene Vorteil ist eine Geschwindigkeitssteigerung des ausgeführten Codes. Dabei ist die direkte Ausführung von Maschinencode durch die CPU in der Regel schneller als durch den Python-Interpreter ausgeführter Python-Code.

Ein neues Feature von *Numba* sind JIT-kompilierte Klassen. Zuvor wurde lediglich das Kompilieren einzelner Funktionen unterstützt. Bei den JIT-kompilierten Klassen handelt es sich jedoch zu diesem Zeitpunkt noch um ein experimentelles Feature, dessen Implementierung noch nicht vollständig abgeschlossen ist und welches Änderungen unterliegen kann. Deshalb soll im Folgenden untersucht werden, ob ein Einsatz von JIT-kompilierten Klassen erfolgen kann und unter welchen Bedingungen dieser erfolgen sollte.

3.2.1 Motivation

Eine Klasse als Hashtabelle zu verwenden bietet verschiedene Vorteile. Eine bessere Codestruktur steht hierbei im Vordergrund, da diese die Entwicklung vereinfacht. So lassen sich Funktionen innerhalb einer Klasse in Form von Methoden nach ihrem Zweck gruppieren. Weiter ermöglicht die Verwendung einer Klasse eine bessere Kapselung und die dem Nutzer bereitgestellte Schnittstelle kann vereinfacht werden. Das explizite Übergeben von bestimmten Argumenten an einzelne Funktionen kann durch die implizite Übergabe von Klassenattributen ersetzt werden. Im Falle einer einfachen Hashtabelle wäre dies etwa das Array, in dem der Inhalt der Tabelle gespeichert wird, welches bei jedem Zugriff auf die Hashtabelle explizit übergeben werden müsste.

Bei Performance-kritischem Code stellt sich jedoch die Frage, ob durch den Einsatz von Klassen gewonnene Vorteile sich negativ auf die Performance auswirken. Um dies näher zu erörtern, erfolgt die Implementierung einer einfachen Hashtabelle. Umgesetzt wird zum einen eine Hashtabelle unter der Nutzung von JIT-kompilierten Klassen und eine weitere Hashtabelle, die lediglich JIT-kompilierte Funktionen einsetzt. Beide Hashtabellen-Implementierungen sollen dabei die gleichen Charakteristiken aufweisen, wie etwa die gewählte Einfügestrategie. Der Unterschied zwischen beiden Implementierungen soll sich soweit wie möglich nur auf die Wahl zwischen JIT-Klassen und JIT-Funktionen beschränken.

Auf diese Weise soll es dann im zweiten Schritt möglich sein die von den beiden Implementierungen gezeigte Performance zu vergleichen. Das Hauptaugenmerk liegt dabei auf dem Zeitbedarf der Methoden bzw. Funktionsaufrufe. Zusätzlich soll aber auch der Speicherbedarf betrachtet werden, um Unterschiede beim Speicherbedarf zwischen den beiden Implementierungen feststellen zu können. Zu diesem Zweck erfolgt der Einsatz von Benchmarks, die verschiedene Zugriffe auf die Hashtabelle simulieren, sodass ein Vergleich der beiden Implementierungen möglich ist.

3.2.2 Vorgehen

Zu Beginn erfolgt die Implementierung einer Hashtabelle auf zwei Arten: Einerseits durch JIT-kompilierte Klassen und andererseits durch JIT-kompilierte Funktionen.

Die beiden Hashtabellen sind auf einfache Art und Weise implementiert. Auf komplexe Algorithmen wird verzichtet, da dies nicht der Fokus der Untersuchung ist. Die verwendete Hashfunktion leitet durch einfache Multiplikation des Schlüsselwertes mit einer Primzahl einen Hashwert ab. Zur Suche innerhalb der Tabelle wird offene Adressierung mit linearem Sondieren [5, S. 239] verwendet.

Die beiden Implementierungen sind bei Betrachtung des Python-Codes soweit wie möglich identisch. Das heißt, dass die einzigen Unterschiede durch die Art der Implementierung gegeben sind. So erfolgt der Speicherzugriff innerhalb der Klassen auf ein im Objekt gespeichertes Array, während für Funktionen das Array als Argument übergeben wird und der Zugriff dementsprechend so auf dieses erfolgt. Auf diese Weise wird sichergestellt, dass beide Varianten beim Aufruf mit den gleichen Operationen das gleiche Verhalten haben. Speicherzugriffe und Funktionsaufrufe sind also bei beiden Implementierungen gleich und es wird ein besserer Vergleich der JIT-Kompilierung durch *Numba* möglich.

Die beiden Codebeispiele Listing 3.6 und Listing 3.7 zeigen jeweils die beiden Implementierungen der Einfügeoperation einerseits als Methode der Klasse und andererseits als alleinstehende Funktion. Die zu erkennenden Unterschiede sind der Speicherzugriff auf das Array der Hashtabelle. Im Fall der Klasse muss dem Zugriff das Schlüsselwort `self` vorangestellt werden, um den Zugriff auf ein Attribut der Klasse zu signalisieren. Ähnlich ist dies beim Aufruf der Hilfsfunktion zu Bestimmung des Startindexes für einen Schlüssel innerhalb der Tabelle. Für die Klasse erfolgt dies durch einen Methodenaufruf in Zeile 4 von Listing 3.6, im Fall der Funktion durch einen einfachen Funktionsaufruf in Zeile 6 von Listing 3.7 einer weiteren JIT-kompilierten Funktion.

```
1  def insert(self, key: types.uint64, value: types.uint64):
2      if key == 0:
3          return
4      index = self.index(key)
5      for i in range(index, self.size):
6          if self.array[i][0] == 0 or self.array[i][0] == key:
7              self.array[i][0] = key
8              self.array[i][1] = value
9              return
10     for i in range(0, index):
11         if self.array[i][0] == 0 or self.array[i][0] == key:
12             self.array[i][0] = key
13             self.array[i][1] = value
14             return
```

Listing 3.6: Implementierung der Einfügeoperation der Klasse

Für die Benchmarks wird eine Datei mit zufälligen Daten für Operationen auf der Hashtabelle generiert. Für die Generierung kann ein Seed angegeben werden, um den Datensatz für einen Durchlauf reproduzieren zu können. Unterschiedliche Optionen bei der Datengenerierung ermöglichen es, verschiedene Nutzungsszenarien der Hashtabelle zu simulieren. So kann ein initialer Füllgrad für die Hashtabelle angegeben werden. Dementsprechend wird dann eine Menge von Einfügeoperationen generiert, mit denen die Hashtabelle befüllt werden kann. Anschließend kann durch die Angabe einer Wahrscheinlichkeitsverteilung für die unterstützten Operationen Einfügen, Suchen und Löschen eine Menge von Operationen erzeugt werden, wobei die relative Häufigkeit der einzelnen Operationen der angegebenen Verteilung entspricht. Zudem kann auch die Verteilung der Zugriffe auf existierende und nicht existierende Schlüssel innerhalb der Tabelle bestimmt werden. So kann etwa simuliert werden, dass die eine Hälfte der Suchoperationen erfolgreich sein soll und die andere Hälfte nicht.

Mithilfe der Daten kann schließlich die Nutzung einer Hashtabelle simuliert werden. Dazu werden mehrere Benchmarkdurchläufe durchgeführt. Als Erstes wird der aktuelle Speicherbedarf des ausführenden Python-Prozesses als m_0 bestimmt. Zum Einsatz kommt

```python
@njit
def jit_insert(array: np.array, key: types.uint64, value: types.uint64):
    if key == 0:
        return
    size: types.uint64 = array.shape[0]
    index = jit_index(key, size)
    for i in range(index, size):
        if array[i][0] == 0 or array[i][0] == key:
            array[i][0] = key
            array[i][1] = value
            return
    for i in range(0, index):
        if array[i][0] == 0 or array[i][0] == key:
            array[i][0] = key
            array[i][1] = value
            return
```

Listing 3.7: Implementierung der Einfügeoperation als alleinstehende Funktion

hierbei das Python-Modul `ps_util`, welches eines plattformunabhängigen Zugriff auf den Speicherbedarf ermöglicht. Dann wird eine neue Instanz einer Hashtabelle initialisiert. Anschließend wird der aktuelle Zeitpunkt als t_0 gespeichert. Die Bestimmung der Zeit erfolgt mit Hilfe der Funktion `time.perf_counter_ns()` aus der Python-Standardbibliothek. Jene liefert die aktuelle Zeit in Nanosekunden seit einem nicht definierten Referenzzeitpunkt, sodass nur aus zwei Werten gebildete Differenzen sinnvoll sind[3]. Dadurch eignet sich die Funktion um die zwischen zwei Programmpunkten vergangene Zeit zu ermitteln. Nun werden die Operationen auf der Hashtabelle ausgeführt. Nach Abschluss der Ausführung werden Speicherbedarf und Zeitpunkt als m_1 bzw. t_1 gespeichert. Im Anschluss können nun Speicher- und Zeitdifferenz als $m_\Delta = m_1 - m_0$ bzw. $t_\Delta = t_1 - t_0$ bestimmt werden. Die Ausführung und Bestimmung der Differenzen wird dabei mehrere Male wiederholt, sodass sich je eine Messreihe von Speicher- und Zeitdifferenzen ergibt. Zu beachten ist hierbei, dass die ersten Ausführungen allerdings verworfen werden und als „Warmup-Durchläufe“ gewertet werden, um eine Verfälschung der Ergebnisse zu vermeiden. Dies ist gerade deshalb wichtig, da die Funktionen bzw. Klassen während der Nutzung von *Numba* JIT-kompiliert werden und dadurch der erste Aufruf eine längere Zeit in Anspruch nehmen kann (siehe Kapitel 2.3.2). Die Messreihen können dann einer einfachen statistischen Analyse unterzogen werden. So lassen sich jeweils das arithmetische Mittel, die Standardabweichung, Minimal- und Maximalwert ermitteln.

Alle Schritte von der Generierung der Testdaten über die Ausführung der Benchmarks und die Visualisierung werden durch *Snakemake* (siehe Kapitel 2.3.5) gesteuert. Neben Parametern wie den Wahrscheinlichkeiten und der Größe ist die Auswahl und Zusammenfassung mehrerer thematisch zusammengehöriger Testläufe dabei der einzige Schritt mit eigens dazu hard-codierter Logik. Der Austausch der Daten zwischen den Schritten erfolgt durch Pythonobjekte, die mit `pickle` serialisiert wurden. Um ein faires und wiederholbares Ergebnis zu erhalten, wird der gesamte Ablauf durch Angabe des Kommandozeilenarguments `--cores 1` auf nur einem Prozessorkern ausgeführt. Ansonsten würden mehrere Benchmarks zum gleichen Zeitpunkt ausgeführt, wobei eine gegenseitige Beeinflussung nicht ausgeschlossen werden kann.

[3] `https://docs.python.org/3/library/time.html#time.process_time_ns`

3.2.3 Ergebnisse

Bei der Ausführung der beschriebenen Benchmarks wurden Performanceunterschiede zwischen den beiden Implementierungen festgestellt. Dazu werden verschiedene Datensätze generiert, um so unterschiedliche Benchmarks auszuführen. Anschließend wird zwischen zwei Ausführungsarten unterschieden. Zum einen können die Zugriffe auf die Hashtabelle aus einem mit *Numba* JIT-kompilierten Kontext erfolgen und anderseits kann es sich um einen normalen Python-Kontext handelt. In dem JIT-kompilierten Kontext kann *Numba* weitere Optimierungen vornehmen.

Die Version von Python und der verschiedenen Python-Pakete, die zur Ausführung der Benchmarks verwendet wurden, sind in Tabelle 3.1 zu finden.

Paket	Version
python	3.8.3
numba	0.49.1
numpy	1.18.1
llvmlite	0.32.0

Tabelle 3.1: Python und Paket-Versionen

Die Betrachtung der Ergebnisse bezüglich des Speicherbedarfs liefert keine wichtigen Erkenntnisse. Der Speicherbedarf entspricht ungefähr dem zu erwartenden Bedarf zur Speicherung der Hashtabelle. Ein großer Unterschied zwischen JIT-kompilierten Klassen und Funktionen zeigt sich nicht.

Bei Betrachtung der Zeit erweisen sich einzelne JIT-kompilierte Funktionen allgemein als besser gegenüber den Klassen. In einem JIT-kompilierten Kontext benötigte die Hashtabelle als Klasse für Ausführung des gleichen Benchmarks jeweils etwa 2-3% mehr Zeit. Abbildung 3.4 zeigt die benötigte Zeit für verschiedene Hash-Tabllengrößen. Dabei wird die Hashtabelle zunächst zu 50% gefüllt, anschließend werden durchmischte Operationen ausgeführt. Bei den Operationen handelt es sich zu 80% um Suchoperationen, zu 10% um Einfügeoperationen und 10% um Löschoperationen. Die Wahrscheinlichkeit, einen bereits vorhandenen Schlüssel zu betreffen, ist 20% beim Einfügen und je 80% beim Suchen und Löschen. Diese Werte erreichen die kritische Auslastung des Addressierungsverfahrens (50%) und erzeugen auch Sonderfälle, wie die Suchen nach einem nicht vorhandenen Schlüssels.

Abbildung 3.5 zeigt Benchmarks für die einzelnen Hashtabellenoperationen, ausgeführt in einem JIT-kompilierten Kontext. Da die einzelnen Operationen getrennt getestet werden, sind die Tabelle je so vorbereitet, dass alle Einfügeoperationen neue Einträge erzeugen und alle Such- und Löschoperationen ihr Schlüssel finden. Hier zeigt sich, dass Einfügeoperationen bei der Klasse 4% schneller sind als bei den Funktionen, Such- und Löschoperationen jedoch 8% bzw. 12% langsamer sind.

In einem normalen Python-Kontext wurden größere Performanceunterschiede zwischen Klassen und Funktionen nachgewiesen, wie Abbildung 3.6 zeigt. Hierbei benötigen Einfügeoperationen bei Klassen etwa 20% mehr Zeit als bei Funktionen, Suchoperationen etwa 12% und Löschoperationen etwa 13%.

3.2.4 Fazit

Die Ergebnisse der Benchmarks und Rahmenbedingungen betrachtend, müssen vor dem Einsatz von Klassen verschiedene Dinge beachtet werden. Es sollte eine Abwägung der sich aus dem Einsatz ergebenden Vorteile und Nachteile erfolgen.

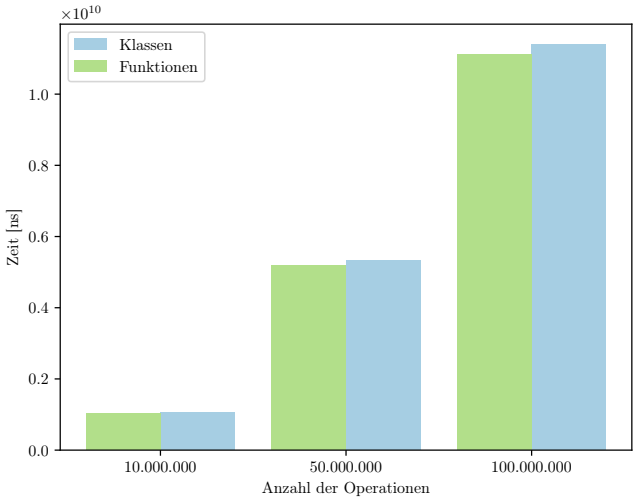


Abbildung 3.4: Benchmark gemischter Operationen (JIT)

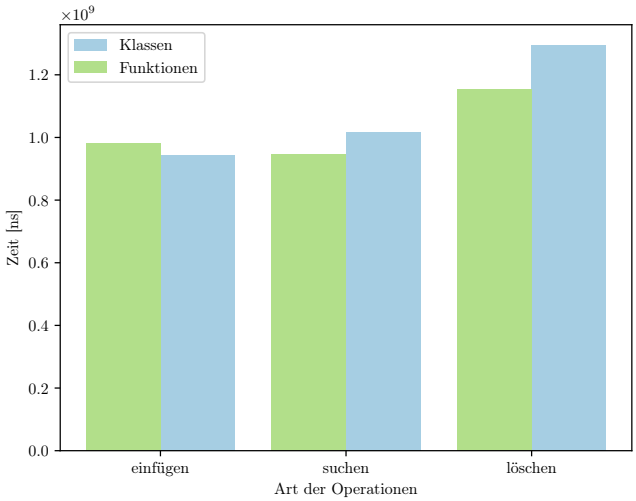


Abbildung 3.5: Benchmark Einzeloperationen (JIT)

Klassen sorgen für eine bessere Strukturierung des Codes durch die Gruppierung der Funktionen und Attribute. Dadurch erhöht sich die Lesbarkeit und Wartbarkeit des Codes. Allerdings haben die Klassen verglichen mit den Funktionen allgemein eine schlechtere Laufzeit, die in einem normalen Python-Kontext noch merklich höher ausfällt. Außerdem sind die Klassen noch ein experimentelles Feature von *Numba*. Das heißt, dass noch einige Features für Klassen nicht implementiert sind, wie zum Beispiel GPU-Support, welcher möglicherweise geeignet wäre, um die Verarbeitung und das Hashing von k-meren zu parallelisieren. Da sich JIT-kompilierte Klassen noch in der Entwicklung befinden, könnte solche Features in der Zukunft noch hinzugefügt werden.

Abschließend lässt sich sagen, dass es für zeitkritische Anwendungen sinnvoller ist, die Funktionen zu nutzen, in einem JIT-kompilierten Kontext jedoch die Klassen fast so gut abschneiden, wie die Funktionen. Da die geplante Anwendung zwar leistungsorientiert, aber nicht zeitkritisch ist, sollte auch der Einsatz von Klassen in Betracht gezogen werden.

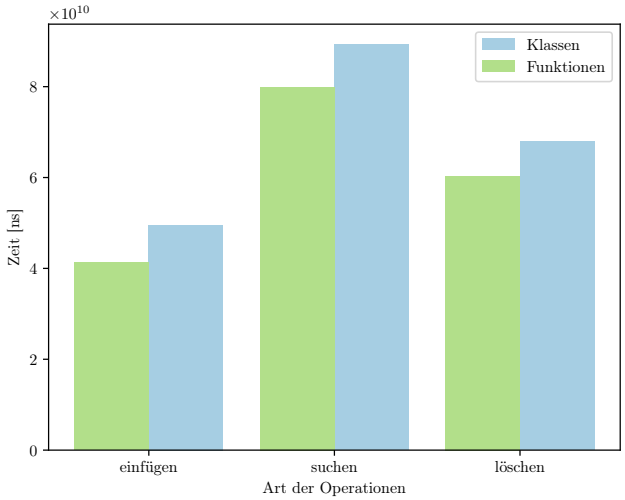


Abbildung 3.6: Benchmark Einzeloperationen

3.3 k-mer Processing

Ein wichtiger Bestandteil der in dieser Projektgruppe verwendeten Ansätze zur automatisierten Verarbeitung von genetischen Daten besteht darin, diese einzulesen, in k-mere zu unterteilen und Funktionen auf diese k-mere anzuwenden. In dem Miniprojekt k-mer Processing wurde untersucht, wie die Berechnung und Verarbeitung der k-mere aus einer Gensequenz für beliebige Funktionen möglichst effizient umgesetzt werden kann. Hierzu werden zunächst die Grundlagen und die Zielsetzung der effizienten Verarbeitung von k-meren besprochen, worauf folgend die Lösungsansätze und die daraus resultierenden Erkenntnisse, sowie die implementierte Umsetzung vorgestellt wird.

3.3.1 Verarbeitung von genetischen Daten

Die Grundlagen der genetischen Daten werden im Kapitel 2.1.1 näher erörtert. Im Folgenden soll auf Basis dieses Wissens auf die Verarbeitung von genetischen Daten eingegangen werden.

Zur schnelleren Verarbeitung ist es üblich, ein k-mer aus DNA-Basen, wie zum Beispiel `GATCTAG`, zu codieren. Da es genau vier verschiedene Zeichen zu codieren gibt, ist folgende Zuordnung möglich: $\{A : 0,\ C : 1,\ G : 2,\ T : 3\}$ oder in Bits: $\{A : 00,\ C : 01,\ G : 10,\ T : 11\}$. Neben den vier Buchstaben für die Basen kommen in Dateien mit DNA-Daten auch andere Zeichen vor, so zum Beispiel `N` für eine unbekannte Base, `S` für `C` oder `G`, oder `W` für `A` oder `T`. Die sinnvolle Bearbeitung solcher nicht-eindeutigen Basenzeichen hätte den Rahmen des Miniprojekts übertreten, weshalb diese überlesen werden. Ein k-mer wird codiert, indem jeder Buchstabe durch die entsprechenden zwei Bits ersetzt wird. Das Ergebnis ist eine Bitfolge, die als positive, ganze Zahl interpretiert werden kann. Die DNA-Sequenz `ACGT` lässt sich also als Bitfolge 00011011 bzw. als Ganzzahl 27 repräsentieren. Die Repräsentation als Ganzzahl enthält im Gegensatz zur Buchstaben- bzw. Bitfolge keine Information über führende A's bzw. Nullen. Bei der Decodierung einer Ganzzahl muss daher die Länge der resultierenden DNA-Sequenz bekannt sein.

DNA-Sequenzen, wie sie von Maschinen zur Sequenzierung ausgegeben werden, sind unterteilt in sogenannte Reads. Die Länge der Reads hängt von der verwendeten Technologie zur Sequenzierung ab, beträgt aber meist weniger als 1000 Basen. Eine Ausnahme bildet hier Nanopore Sequencing, welches Reads von Längen deutlich über 1000 Basen

produziert. Aus diesen Reads werden die k-mere zur weiteren Verarbeitung generiert.

3.3.2 Zielsetzung

Das Ziel des Miniprojekts war es, einen k-mer Processor Generator zu implementieren, der für eine gegebene Funktion einen k-mer Processor generiert. Der resultierende k-mer Processor sollte dann für eine gegebene DNA-Sequenz alle k-mere berechnen, codieren und die Funktion auf die codierten k-mere anwenden.

Außerdem sollte es eine Option geben, die es dem Nutzer erlaubt, anzugeben, ob das reverse Complement der k-mere ebenfalls berechnet und verarbeitet werden sollen.

Die Implementierung soll dabei auf ihre Laufzeit optimiert werden. Hierzu sollen unterschiedliche Ansätze der Umsetzung generiert werden und auf ihre Laufzeit untersucht werden. Die aus diesen Benchmarks resultierenden Ergebnisse sollen die Grundlage für die finale Implementierung darstellen. Die Implementierung soll mittels der Programmiersprache Python durchgeführt werden.

3.3.3 Lösungsansätze

Im Folgenden werden die Lösungsansätze zu Berechnung und Codierung der k-mere einer eingelesenen DNA-Sequenz, der Bildung von reverse Complements und die Benchmarks dessen erläutert.

Berechnung der k-mere

Die Berechnung, Codierung und Verarbeitung der k-mere sollen im k-mer Processor parallel geschehen. Dazu wird von der Eingabesequenz zunächst das erste k-mer berechnet, codiert und die gegebene Funktion darauf angewendet. Anschließend wird über die restlichen Zeichen iteriert und jeweils durch Bitoperationen die vorderste Base aus dem codierten k-mer heraus- und die nächste Base hineingerechnet. Welche Methoden zur Bestimmung des Bitcodes einer Base untersucht wurden, wird im Folgenden vorgestellt.

Ansätze k-mer-Codierung

Für die Codierung wurden Kombinationen verschiedener Eingabetypen und Übersetzungsmethoden implementiert. Als Eingabetypen wurden Strings, *NumPy*-Arrays und Bytes verwendet. Bytes als Eingabetyp können verwendet werden, indem ein Eingabe-String als Bytes interpretiert wird. In Python ist dies möglich durch `input_string.encode()`. *NumPy*-Arrays können erhalten werden, indem ein Eingabe-String in Bytes umgewandelt und daraus ein *NumPy*-Array erzeugt wird. Dies wird durch `np.frombuffer("GATTACA".encode(), dtype=np.uint8)` bewerkstelligt. Die Übersetzungsmethoden unterscheiden sich bezüglich der Zuweisung von DNA-Basen zur Zwei-Bit-Codierung:

1. **Unterscheidung der Basen durch If-Blöcke:** Jede Base wird separat eingelesen und anhand von If-Blöcken ihrer jeweiligen binären Repräsentation zugeordnet. Mittels bitweisem Shifting um zwei Stellen wird jede Base an die Gesamtsequenz angehängt. Listing 3.8 zeigt exemplarisch die Implementierung mittels If-Blöcken für den Eingabetypen String.

2. **Unterscheidung der Basen durch ein Array:** Ein *Numpy*-Array der Größe 256 enthält 2-Bit-Codierungen für alle ASCII-Zeichen. Die entsprechenden Bits für eine Base können durch einen direkten Lookup im Array festgestellt werden. Listing 3.9 zeigt exemplarisch die Implementierung durch ein Array für den Eingabetypen Byte.

```
res = 0

for character in seq:
    if character == "A":  # A
        res = (res << 2) | 0b00
    elif character == "C":  # C
        res = (res << 2) | 0b01
    elif character == "G":  # G
        res = (res << 2) | 0b10
    elif character == "T":  # T
        res = (res << 2) | 0b11
```

Listing 3.8: Encodierung druch If-Blöcke für Strings als Eingabetyp

```
res = 0
for character in seq:
    currentMappingResult = sequence_encode_lookup_array1[(character)]
    if currentMappingResult != 255:
        res = (res << 2) | currentMappingResult
```

Listing 3.9: Encodierung durch ein Array für Byte als Eingabetyp

3. **Unterscheidung von je 2 Basen durch ein Array:** Ein NumPy-Array der Größe 65536 wird verwendet, um je 2 ASCII-Zeichen gleichzeitig zu übersetzen. Dies soll die Zahl der Speicherzugriffe reduzieren, um Zeit zu sparen. Im Abschnitt Ausblick wird auf eine mögliche Strategie zur Verringerung des benötigten Speichers für die Tabelle eingegangen. Listing 3.10 zeigt exemplarisch die Implementierung von je 2 Basen durch ein Array für den Eingabetype Array.

Aus den Kombinationen resultieren neun unterschiedliche Implementierungen der DNA-Codierung, die auf ihre Laufzeit untersucht wurden. Die Ergebnisse dazu sind im Abschnitt Benchmarks näher beschrieben.

Ansätze Reverse-Complemt-Bildung

Für die Bildung der umgekehrten Sequenzen für als Integer codierte k-mere wurden wie bereits bei der k-mer Codierung mehrere Ansätze entwickelt, welche im Folgenden vorgestellt werden. Da die Funktion in unserer Anwendung ausschließlich benutzt wird, um für bereits codierte Integer die umgekehrte Sequenzen zu ermitteln, bieten beide Funktionen nur diesen Eingabe- und Ausgabetypen an. Soll für einen k-mer in String-Darstellung oder einer anderen codierten Darstellung die umgekehrte Sequenz ermittelt werden, so muss dieser erst codiert werden, dann für diese Darstellung die umgekehrte Sequenz ermittelt werden und das Ergebnis dann wieder decodiert werden.

1. **Bildung der umgekehrten Sequenzen für codierte Sequenzen mithilfe von Strings:** Für die Komplement-Bildung wird aufgrund der einfachen Implementierung bei Integern auch intern mit den übergebenen codierten Integern gearbeitet, da die Codierungen einzelner Basen durch 2-Bit und die Wahl der Werte für die einzelnen Basen also, z.B. A: 00 und T: 11, es erlaubt, das Komplement direkt über das Bit-Komplement zu bestimmen. Für die Bildung der umgekehrten Sequenz einer Reihenfolge wird hingegen eine Transformation vom codierten Integer in die binäre Version des Strings vorgenommen und jede Base einzeln von hinten nach vorne an

```
local_array = None
sequence_length = len(seq)

res = 0

if (sequence_length & 1) == 1:
    current_mapping_result = sequence_encode_lookup_array1[seq[0]]

    if current_mapping_result != 255:
        res = current_mapping_result
    local_array = seq[1:].view(np.uint16)
else:
    local_array = seq.view(np.uint16)

for character in local_array:
    currentMappingResult = sequence_encode_lookup_array2[character]


    number_of_encoded_bases = ((currentMappingResult & 0b11_00_00) >> 4)

    res = (res << number_of_encoded_bases * 2)

    res = (res | (currentMappingResult &
            ((1 << (number_of_encoded_bases * 2)) - 1)))
```

Listing 3.10: Encodierung durch ein Array für Byte als Eingabetyp

einen neuen String konkateniert, der schließlich in die codierte Integer-Repräsentation des Ergebnisses umgewandelt wird.

2. **Bildung der umgekehrten Sequenzen für codierte Sequenzen mithilfe von Integer-Operationen:** Ebenso wie bei der Bildung der umgekehrten Sequenzen mithilfe von Strings wird die Komplement-Bildung per Bit-Komplement durchgeführt, also direkt auf dem übergebenen codierten Integer gearbeitet. Für die Bildung der umgekehrten Reihenfolge wird allerdings ein *Numpy*-Array benutzt, das jedes Byte an möglichen Sequenzdaten in umgekehrter Reihenfolge speichert. Mithilfe dieses Arrays kann für ein k-mer jedes Byte einzeln umgedreht werden, indem der Wert an der Stelle der einzelnen Bytes des k-mers im Array ermittelt wird. Anschließend wird die Reihenfolge der so ermittelten invertierten Bytes über eine Reihe von Bit-Shift-Operation umgedreht.

3.3.4 Umsetzung

Im Weiteren wird auf die Umsetzung des k-mer Processors eingegangen. So wird die Implementierung des k-mer Processors selbst erläutert und die Benchmarks vorgestellt. Zur Verwendung von *Numba* wird auf Abschnitt 2.3.2 in den Grundlagen verwiesen.

k-Mer Processor

Zur Umsetzung des k-mer Processors wurde die Dekoratorklasse `@process_kmers` umgesetzt. Dieser wird die k-mer Länge sowie das optionale Bool-Flag, das bestimmt, ob auch

das reverse Komplement genutzt wird, übergeben. Die Dekoratorklasse erstellt mittels der dekorierten Funktion einen k-mer Processor und gibt diesen zurück.

```
if k <= 0:
    raise ValueError("Error: The parameter k must be 1 or bigger!")

def kmer_processor(arr: np.ndarray, args: T = None) -> T:

    if k > len(arr):
        return args

    a = _dna_encode_ndarray_array_2_at_a_time(arr[0:k])
    args = f(a, args)
    if include_rev_compl:
        args = f(rev_compl(a, k), args)
    for i in range(k, len(arr)):
        elem_int = sequence_encode_lookup_array2[arr[i]]
        if elem_int <= 0b11:
            a = a & (1 << 2 * (k - 1)) - 1  # entferne die ersten beiden
            ↪ Bits
            a = (a << 2) | elem_int  # füge die zwei Bits für die nächste
            ↪ Base hinzu
            args = f(a, args)
            if include_rev_compl:
                args = f(rev_compl(a, k), args)
    return args


return kmer_processor
```

Listing 3.11: Implementierung des k-mer Processors

Die Implementierung des `kmer_processor` ist in Listing 3.11 zu sehen. Bei der Erstellung des k-mer Processor wird zunächst das festgelegte k auf Plausibilität überprüft. Das k sollte größer als Eins sein, sonst wird ein `ValueError` erzeugt. Falls k größer ist als die Länge der Sequenz, so wird diese Sequenz übersprungen und die Argumente unverändert wieder ausgegeben. Nachfolgend wird das erste k-mer aus der übergebenen DNA-Sequenz extrahiert, mittels der aus den Benchmarks ermittelte Encodierungsmethode encodiert und der annotierten Funktion zur weiteren Verarbeitung zurückgegeben. Wurde im Dekorator das reverse Komplement verlangt, wird auch das reverse k-mer der annotierten Funktion zur weiteren Verarbeitung zurückgegeben. Im Folgenden werden die weiteren k-mere der Sequenz berechnet. Dazu wird mittels Bit-Verschiebung die vorderste Base aus dem zuletzt berechneten codierten k-mer herausgerechnet und die nächstfolgende Base anschließend hineingerechnet. Dies wird solange fortgeführt, bis das Ende der Sequenz erreicht und alle k-mere berechnet sind. Zur Illustration der Umsetzung des k-mer Processors wurde die Funktion `get_kmers` implementiert. Jedes k-mer wird dabei gezählt und die vorkommenden k-mere mit ihren Häufigkeiten ausgegeben.

3.3.5 Benchmarks

Im Folgenden werden die Benchmarks der unterschiedlichen Encodierungsmethoden erörtert, dazu wird die Methodik der Messung erläutert und die Ergebnisse vorgestellt.

Um valide Ergebnisse zu erhalten, sollte für den Benchmark eine reale Anwendung simuliert werden. Auf eine Referenz DNA-Sequenz wurde verzichtet, um mögliche Verzerrungen, die durch diese auftreten könnten, ausschließen zu können. Dies konnte durch Anwendung auf mehrere generierte DNA-Sequenzen sichergestellt werden. Zur Generierung wurde in erster Instanz mithilfe von *NumPy* ein String-Array erzeugt. Dazu wurde die Funktion `np.random.choice(BASES, p=HOMOSAPIENSPROBS, size=n)` genutzt. Diese erhält als ersten Parameter ein eindimensionales Array mit den möglichen Basen, die in der DNA-Sequenz auftreten können, als zweiten Parameter ein eindimensionales Array mit den jeweiligen Wahrscheinlichkeiten des Auftretens dieser Basen und als dritten Parameter die gewünschte Länge des Arrays. Das resultierende String-Array dient als Grundlage für die Benchmarks der Encodings via Strings. Anhand des String-Arrays wird mithilfe der Funktion `encode()` das Byte-Array der DNA-Sequenz erstellt, welches für die Benchmarks der Encodings via Bytes genutzt wird. Analog dazu wird durch die Verwendung der Funktion `np.frombuffer(bytearray, dtype=np.uint8)` das Byte-Array genutzt und mittels des dtype `np.uint8` das `np.ndarray`, das die jeweiligen Integerwerte der Basen enthält, erstellt. Das resultierende Array wird wiederum für die Benchmarks der Encodings via *NumPy*-Array genutzt. Diese Typ-Konvertierungen geschehen im Rahmen des Benchmarks jedes Mal bei Aufruf einer Encoding-Funktion, sodass die Zeit für die Konvertierung in der Zeitmessung der Encoding-Methoden berücksichtigt wird.

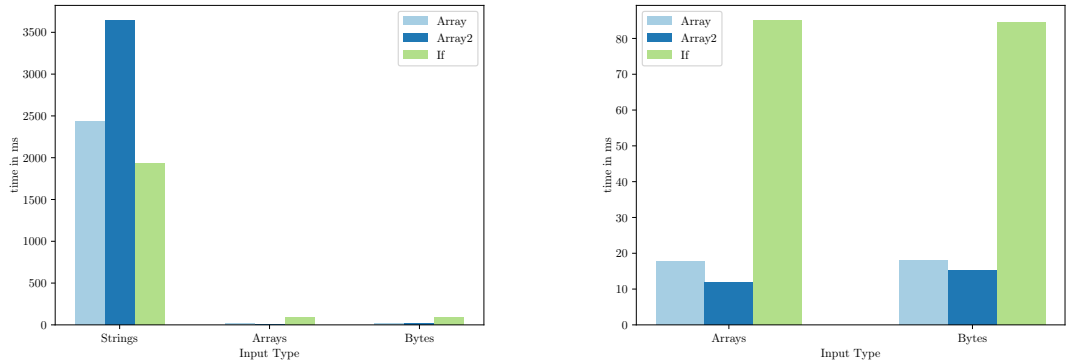
Somit wurde als Datensatz für die Benchmarks eine DNA-Sequenz, die dem menschlichen Genom angenähert ist, generiert. Dazu wurde ein jeweiliges Array von 10 Millionen Basen mit ihrer Wahrscheinlichkeit des Auftretens beim Menschen generiert. A und T treten dabei jeweils mit einer Wahrscheinlichkeit von ca. 30% und G und C jeweils mit einer Wahrscheinlichkeit von ca. 20% auf. Diese Sequenz wurde im Ganzen codiert, um ausreichend Laufzeiten der einzelnen Codervorgänge für den Vergleich zu erhalten.

Die Messungen wurden pro Kombination aus Eingabetyp und Encoding-Methode mit jeweils 100 Encoding-Operationen auf allen 27-meren durchgeführt. Im Ergebnis werden die durchschnittlichen Laufzeiten der Encodierungen in Millisekunden verglichen. Da die Standardabweichungen keine weiteren Erkenntnisse hervorbrachten, werden diese nicht weiter betrachtet.

Zur Implementierung des k-mer Processors wurde der Just-in-time-Compiler *Numba* verwendet. Dieser übersetzt annotierten Quellcode bei der ersten Ausführung zu Maschinencode, sodass darauf folgende Ausführungen beschleunigt werden. Aufgrund dieser Tatsache wurde dem Benchmark ein Pre-Processing vorgelagert, um Verfälschungen der Laufzeitmessung, die aus der Übersetzung des Quellcodes zu Maschinencode resultieren, zu vermeiden. Dieses Pre-Processing führt alle für den Benchmark relevanten Funktionen einmal aus, sodass diese vor dem Benchmark mit der Just-in-time-Compilierung compiliert werden.

Abbildung 3.7a zeigt die Laufzeitmessungen für das Encoding gruppiert nach Eingabetypen. Die verschiedenen Codierungsmethoden sind farblich markiert, wie in der Legende abzulesen. Die Visualisierung veranschaulicht insbesondere die hohen Laufzeiten der Implementierungen, die Strings als Eingabetyp für das Encoding verwenden im Vergleich zu den Implementierungen der anderen Eingabetypen. So weisen die String Implementierungen Laufzeiten von mindestens über 1000 ms auf. Bei dem Benchmark vorangegangenen Tests auf kurzen DNA-Sequenzen wurde dieser Unterschied nicht deutlich. Somit wird auch die Relevanz der Anwendung des Benchmarks auf längeren DNA-Sequenzen verdeutlicht. Aufgrund der hohen Laufzeit werden die drei Methoden im Folgenden nicht weiter

betrachtet.



(a) Mit Strings. **(b)** Ohne Strings.

Abbildung 3.7: Laufzeitvergleich für das Encoding. Gruppiert nach Eingabedatentyp, farbliche Zuordnung nach Encoding-Methode.

Abbildung 3.7b zeigt dieselben Laufzeitmessungen, allerdings ohne die Werte für den Eingabetyp String, sodass Unterschiede zwischen den Encoding-Methoden für Arrays sowie Bytes deutlicher betrachtet werden können. Hier wird deutlich, dass If-Blöcke zur Übersetzung einen Nachteil in der Laufzeit gegenüber dem Array und dem 2-Basen-Array aufweisen. Diese Lösungsansätze weisen eingabetypübergreifend Laufzeiten von ca. 84 ms auf. Implementierungen mittels Arrays schneiden im Vergleich mit den If-Blöcken mit Laufzeiten kleiner 20 ms deutlich effizienter ab, jedoch wird kein Unterschied zwischen den Eingabetypen ersichtlich. Die Implementierung mittels 2-Basen-Arrays schneiden im gesamten Vergleich am schnellsten ab und es liegt zudem ein Unterschied zwischen den Eingabetypen vor. Die Verwendung von Bytes als Eingabetyp ergab dabei eine durchschnittliche Laufzeit von 11,78 ms, während *NumPy*-Arrays eine durchschnittliche Laufzeit von 15,25 ms erzielten und damit etwas langsamer waren.

Für die Funktionen für die Bildung vom reverse Complement wurde auf Benchmarks verzichtet, da der Performance-Unterschied zwischen der Funktion, die intern in einen String konvertiert und dann wieder in die kodierte Fassung zurück umwandelt, gegenüber der Funktion, die nur mit der Integerrepräsentation arbeitet, so gravierend zugunsten der Letzteren ausfiel.

3.3.6 Ausblick

k-mer Codierung

Zum aktuellen Zeitpunkt wurde auf dem Array für die Codierung zweier Basen auf einmal bezüglich des benötigten Speichers recht naiv umgesetzt. Obwohl lediglich die Zeichen A, C, G und T und in Einzelfällen andere Großbuchstaben tatsächlich benutzt werden, speichert der Array $256 \cdot 256 = 65536$ Einträge. Aufgrund des exponentiellen Wachstums des Übersetzungs-Arrays würde dies bei 3 Basen zu $256^3 = 16,777,216$ und bei 4 Basen zu $256^4 = 4,294,967,296$ Byte Speicherverbrauch führen, wobei beides zu einem deutlich größeren Speicherverbrauch führen würde als durch die z.B. $20^4 = 160000$ sinnvollen Kombinationen gerechtfertigt wäre. Denkbar wäre also für die Kompression am Beispiel der 2 Basen Codierung folgende Strategie:

1. Von der 2-Byte Variable, die die zu codierenden Zeichen speichert, wird $65 + 256 \cdot 65$ abgezogen. Einzeln betrachtet speichern die beiden Bytes dann je Zahlen von 0 bis 19, also A bis T.

2. Die beiden Bytes können dann per Bit-Shift und Bit-UND Operationen in zwei Variablen isoliert werden und anschließend kann das obere Byte mit 20 multipliziert auf das untere aufaddiert werden.

3. Schließlich wird der Lookup in dem nun 20^2 Byte großen Array ein Lookup für die ermittelte Zahl durchgeführt.

Während dieses Verfahren den benötigten Speicher deutlich reduziert, ist das Wachstum weiterhin exponentiell und es müsste experimentell festgestellt werden, bei welcher Anzahl übersetzter Basen der Performance-Gewinn durch die Abnahme an Speicherzugriffen durch den Performance-Verlust durch die eventuell steigende Anzahl an Cache-Misses eingeholt wird. Auch ist zu beachten, dass bei diesem Verfahren zusätzliche Rechenoperationen in Form der Subtraktionen und Bit-Operationen notwendig sind.

3.3.7 Fazit

Im Miniprojekt k-mer Processor wurde ein Tool zur effizienten Verarbeitung von k-meren von DNA-Sequenzen erstellt. Hierzu wurden Lösungsansätze für die Encodierung von DNA-Sequenzen mittels Kombinationen von Eingabetypen und Methoden der Implementierung erarbeitet und diese mittels eines Benchmarks auf ihre Laufzeit überprüft. Die Implementierungen mit Strings als Eingabetyp stellten sich dabei insbesondere bei der Anwendung auf lange DNA-Sequenzen als ineffizient in Bezug auf ihre Laufzeit dar. Hier zeigte sich die Übersetzung mittels des 2-Basen-Arrays auf dem Eingabetyp *NumPy*-Array als effizienteste Lösung hinsichtlich der Laufzeit, gefolgt von der Methode mit einem Array für einzelne Basen. Dass sich die 2-Basen-Array Methode als schneller erweist als die 1-Basen-Array Methode, ist nicht überraschend, allerdings ist hier anzumerken, dass mit höherer Zahl an Basen, die zeitgleich codiert werden sollen, die Größe des Arrays rapide steigt. Aus Gründen des Speicherbedarfs ist es daher nicht möglich, einfach ein Array für eine hohe Zahl an Basen anzulegen. Zudem benötigt der k-mer Processor eine Codierungsmethode für jeweils eine einzelne Base, sodass die Nützlichkeit von Arrays für mehr als 1 Base begrenzt ist. Des Weiteren wurde die Dekoratorklasse `kmer_processor` erstellt. Dieser ermöglicht es eine entsprechend annotierte Funktion auf jedes k-mer einer DNA-Sequenz auszuführen. Der Dekorator dient somit im weiteren Verlauf des Projektes zur effizienten Verarbeitung von k-meren.

Kapitel 4 — Low Complexity Filter

Unter Einbeziehung der Ergebnisse der vorangegangenen Miniprojekte wurde ein Tool entwickelt, das *reads* einer *FASTQ*-Datei bezüglich ihrer Komplexität klassifiziert. Im Folgenden werden Zielsetzung und Vorgehen beim Implementieren dieses Tools erläutert und auf die Ergebnisse der Klassifizierungsmethoden eingegangen.

4.1 Zielsetzung

Ziel des Projektes ist die Erstellung eines Filter-Werkzeugs, mit dem *reads* aus einer *FASTQ*-Datei anhand ihrer Komplexität klassifiziert werden. Genome bzw. DNA-Sequenzen können Regionen mit geringer Komplexität und sich wiederholenden Folgen von Basen aufweisen. *Reads* können aufgrund der geringen Komplexität zufällig als Verwandte eingestuft werden. Bei der weiteren Verarbeitung können diese Regionen so zu Verzerrungen führen.

Der Low Complexity Filter dient dazu, diese Regionen zu ermitteln und herauszufiltern. Hierbei sollen die *reads* jeweils als *read* mit hoher oder geringer Komplexität klassifiziert und in die entsprechende Ausgabedatei geschrieben werden. Dazu wird ein Schwellwert benutzt, mit dem die einzelnen *reads* in wenig bzw. hoch komplex kategorisiert werden. Hierzu müssen ein passendes Maß für die Komplexität und ein geeigneter Schwellwert gefunden werden.

4.2 Problembeschreibung

Die Aufgabe des Low Complexity Filters besteht darin, jeden *read* einer DNA-Sequenz aus einem *chunk*, einer Klasse anhand von zuvor bestimmten Schwellwerten zuzuordnen. Dabei wird auf eine Verarbeitung der DNA-Sequenzen in *chunks* gesetzt, da somit ein Kompromiss zwischen einzelnem Einlesen und dem Einlesen der ganzen *FASTQ*-Datei eingegangen wird. Die Verarbeitung einer ganzen *FASTQ*-Datei ist aufgrund ihrer Größe und der somit entstehenden Speicherlast generell nicht erstrebenswert oder auch, je nach benutztem System, nicht ohne Nutzung von Auslagerungsdateien möglich.

Der Lösungsansatz dieses Projektes zu diesem Klassifizierungsproblem basiert auf einer Zählung der k-mere der einzelnen *reads*. In diesem Kapitel beziehen sich alle Beispiele und Ergebnisse auf 5-mere. Diese sind kurz genug, dass ein bedeutender prozentualer Anteil der $4^5 = 1024$ Varianten in einem *read* vorkommen kann und Häufungen von ähnlichen Abschnitten deutlich werden.

Angenommen die Anzahl der Vorkommen jedes k-meres in einem *read* wird in einem *array counts* vom Typ `int32[1024]` gezählt (während der Entwicklung wurden über alle *reads* aggregiert, weshalb dieser Datentyp gewählt wurde). Anhand dessen wird ein *read* klassifiziert und in die entsprechende Ausgabe-Datei geschrieben. Die Klassifizierung der *reads* basiert auf folgenden zwei Statistiken:

- die Häufigkeit des häufigsten k-mers in einem *read*

$$\text{max-k-mer}(counts) = \left\{ \frac{\max(counts)}{\text{sum}(counts)} \right. \tag{4.1}$$

- die Anzahl der verschiedenen k-mere in einem *read* (Divisor wird durch Zahl der möglichen k-mere begrenzt)

$$\text{unique-k-mer}(counts) = \left\{ \frac{\sum_{i \in counts} \min(1, i)}{\min(\text{sum}(counts), \text{length}(counts))} \right. \tag{4.2}$$

Um für *reads* verschiedener Länge einheitliche Vergleichswerte zu bilden, werden die Statistiken auf den Wertebereich $[0, 1]$ skaliert. Dies wird folgendermaßen berechnet: Die obere Schranke von max-k-mer ist 1, welche genau dann erreicht wird, wenn ein *read* nur aus Wiederholungen eines k-mers besteht. Die untere Schranke ist $1/\,\text{sum}(counts)$ und wird erreicht auf, wenn jedes auftretende k-mer einmalig ist. Bei unique-k-mer hingegen wird der Höchstwert 1 erreicht, wenn entweder alle aufgefunden k-mere einmalig sind, oder der *read* länger als 1024 5-mere ist und jedes davon mindestens ein mal enthält. Auch hier wird die untere Schranke $1/\,\text{sum}(counts)$ erreicht, wenn der *read* aus Wiederholungen eines 5-mers besteht. Diese Werte dieser Funktionen sind allerdings noch nicht vergleichbar für *reads* mit verschiedenen Längen, da die unteren Grenzwerte abhängig von der Gesamtanzahl an k-meren ist. Dies führt beispielsweise dazu, dass der bestmögliche max-k-mer-Wert eines *reads* mit 20 k-meren $0,05$ ist, während der bestmögliche Wert eines *reads* mit 100 k-meren $0,01$ ist. Um einen Satz an Grenzwerten über *reads* verschiedener Länge nutzen zu können, wird der Wertebereich der Funktionen auf das Intervall $[0, 1]$ normiert, indem Dividend und Divisor um je eins verringert werden. Dadurch wird der kleinste erreichbare Wert (von max-k-mer und unique-k-mer) $1/\,\text{sum}(counts)$ auf 0 abgebildet und der höchste erreichbare Wert bleibt bei 1. Außerdem werden *reads* mit zehn k-meren oder weniger direkt auf den Wert für geringe Komplexität abgebildet, da sie sehr wahrscheinlich aus *reads* mit sehr vielen undefinierten Stellen stammen.

Dies sind die finalen Funktionen mit den genannten Anpassungen, welche für den Rest des Kapitels verwendet werden:

$$\text{max-score}(counts) = \begin{cases} \frac{\max(counts)-1}{\text{sum}(counts)-1} & \text{if } \text{sum}(counts) > 10 \\ 1 & \text{if } \text{sum}(counts) \leq 10 \end{cases} \tag{4.3}$$

$$\text{unique-score}(counts) = \begin{cases} \frac{-1+\sum_{i \in counts} \min(1,i)}{\min(\text{sum}(counts),\text{length}(counts))-1} & \text{if } \text{sum}(counts) > 10 \\ 0 & \text{if } \text{sum}(counts) \leq 10 \end{cases} \tag{4.4}$$

Um nun die Kategorisierung in *low-* und *high-complexity* umzusetzen, muss der Zusammenhang zwischen den beiden verwendeten *scores* untersucht werden und daraus eine Filterfunktion samt Schwellwert gebildet werden.

4.3 Aufbau

Im Folgenden wird der Aufbau des Low Complexity Filter Projektes näher erläutert. Hierzu soll zunächst die Integration der Miniprojekte erläutert werden. Dabei wird auf die Verwendung des k-mer Processors, sowie das Einlesen der *FASTQ*-Dateien eingegangen. Des Weiteren wird die Handhabung des *command line interface*, die Verwendung der Paketverwaltung mittels *Conda*, die Code-Formatierung, sowie die Verwendung von *Snakemake* zur reproduzierbaren Anwendung des Moduls erörtert. Bei der Erstellung der Projektstruktur und der Anwendung von Programmier-Werkzeugen wurde der Fokus auf eine effiziente, kollaborative Arbeit gelegt.

4.3.1 Zusammenführung der Projekte

Ein großer Teil des Projektes beinhaltete die Zusammenführung der Miniprojekte bzw. die Nutzung des k-mer Processors im zu generierenden Code und dem Einlesen der *FASTQ*-Dateien.

Verwendung des k-mer Processors

Sowohl das *FASTQ*-Processing als auch das k-mer Processing haben auf die Verarbeitung von DNA-Sequenzen als *NumPy*-Array gesetzt. Somit war die Verwendung des k-mer Processors innerhalb des zu generierenden Codes ohne weitere Aufwände möglich. Mittels eines Dekorators konnte der k-mer Processor zur Ermittlung der k-mere und der damit verbundenen Filterung auf diese k-mere angewandt werden. Des Weiteren werden auch Paare von *FASTQ*-Dateien unterstützt.

Einlesen der FASTQ-Dateien

Zum Einlesen der *FASTQ*-Dateien wurden einzelne Funktionen aus dem Paket *Xengsort* übernommen und angepasst. Dabei wird die einzulesende *FASTQ*-Datei, die Größe des Puffers und die maximale Anzahl an *reads* übergeben. Als Rückgabe wird ein Tupel vom Puffer und *Linemarks* zurückgegeben. Der Puffer enthält alle eingelesenen Zeichen als Bytewert. *Linemarks* ist ein zweidimensionales *array* in dem für *reads* der Anfangs-Index der Sequenz, der End-Index der Sequenz, der Anfangs-Index des *reads* sowie der End-Index des *reads* zugeordnet ist. Im Fall der Prozessierung von zwei *FASTQ*-Dateien wird ein zusätzlicher Puffer sowie die zugehörigen *linemarks* zurückgegeben.

4.3.2 Command Line Interface

Zur einfachen Verwendung des Modules wurde ein Argument-Parser genutzt, der ein nutzerfreundliches *command line interface* bereitstellt. Hierbei wurde auf das Paket argparse[1] zurückgegriffen. Dieses bietet eine automatische Generierung von Hilfs- bzw. Fehlernachrichten bei dem Aufruf der Anwendung durch den Nutzer, sodass diesem die Nutzung auch ohne detaillierten Blick in die Dokumentation gewährleistet ist. Das Modul verfügt über die drei Subargumente `process`, `statistics` und `visualize`, die auf die jeweilige main-function verweisen. Mit `process` werden die *reads* aus Eingabedateien gelesen und nach Komplexität gefiltert in mehrere Ausgabedateien geschrieben, welches die Hauptaufgabe des Programms ist. Der Befehl `statistics` schreibt nicht die *reads* als Ausgabe, sondern notiert die Werte der genutzten Statistiken und die Häufigkeit jedes k-mers pro *read*. Diese Statistiken dienen als Eingabe, um mit dem `visualize` Befehl mehrere Darstellungen der Verteilung zu generieren, darunter Histogramme und Streudiagramme. Tabelle 4.1 zeigt die Liste der Kommandozeilenargumente, die beim Aufruf des Programms übergeben werden können bzw. müssen. Wobei darauf hinzuweisen ist, dass `--fastq` nur in `process` und `statistics` und `--stats` nur in `visualize` vorhanden ist.

Option	Beschreibung
`--fastq`	legt die Eingabe FASTQ Datei fest (nur in `process` und `statistics` vorhanden)
`--stats`	legt die einzulesende Json-Datei fest (nur in `visualize` vorhanden)
`--out`	legt das Ausgabeverzeichnis fest
`--threadcount`	gibt an wie viele Threads genutzt werden sollen
`--kmer`	bestimmt die Größe der k-mere
`--pairs`	legt die FASTQ Datei mit gepaarten bzw. gekoppelten Enden fest
`--rev`	bestimmt ob komplementäre k-mere enthalten sein sollen

Tabelle 4.1: Liste der Kommandozeilenargumente

[1] `https://docs.python.org/3/library/argparse.html`

4.3.3 Conda

Zur Paketverwaltung wurde wie bereits in den vorangegangenen Miniprojekten *Conda* genutzt. Eine Erläuterung zu *Conda* ist in den Grundlagen in Abschnitt 2.3.9 zu finden. In der Datei environment.yml sind die vorausgesetzten Pakete und die durch *Conda* zu verwendenden *channel* definiert, wobei die Datei durch *Conda* genutzt wird, um die entsprechende Umgebung zu generieren. Neben der lokalen Anwendung von *Conda* wird die Umgebung ebenfalls im Docker der *GitLab-CI* genutzt, um auch dort für den Build und die Tests die richtigen Abhängigkeiten zu garantieren.

4.3.4 Code-Formatierung

Um die Lesbarkeit des Programmcodes zu verbessern und dem damit einhergehenden höheren Verständnis des Codes, sowie der effektiven Kommunikation im Team wurde eine Code-Formatierung eingesetzt. Das Tool Black[2] dient dabei als Grundlage für den Codestil. Mit pre-commit wird ein *Git*-Hook Skript genutzt, welches die Code-Überprüfung mit Black vor jedem *commit* vornimmt. Der *hook* ist in der Konfigurationsdatei `pre-commit-config.yaml` beschrieben. Somit ist eine stetige, einheitliche Code-Formatierung zu jedem Zeitpunkt auf dem *repository* gewährleistet.

4.3.5 Snakemake

Auf das *workflow management system Snakemake* wurde bereits im Kapitel Grundlagen 2.3.5 eingegangen. Im Folgenden werden die Regeln, die in diesen Modul festgelegt sind, beschrieben:

- `process_mouse_exomes` ruft den Filterprozess auf (time misst dabei die Laufzeit)
- `calculate_statistics` berechnet Statistiken und schreibt diese in eine pickle-Datei
- `visualize_statistics` visualisiert die Werte aus den zuvor berechneten Statistiken
- `classify_statistics` ermittelt aus den zuvor berechneten Statistiken die Werte zur Klassifikation bzw. Bestimmung des Schwellwerts

4.4 Implementierung des Filters

Basierend auf den Statistiken aus Abschnitt 4.2 kann die eigentliche Filterfunktion gebaut werden, welche einen *read* als *low-* oder *high-complexity* klassifiziert. Die Filterfunktion gewichtet beide Statistiken (max-score und unique-score), um einen möglichst robusten Entscheider zu bilden. Dazu werden die Verteilungen der Statistiken und ausgewählte Beispiele empirisch untersucht. Neben der Kategorisierung in *low-* und *high-complexity* ergibt sich für das paarweise Filtern von *reads* die dritte Kategorie *mixed*. Es entstehen möglicherweise Grenzfälle bei Readpaaren, bei denen beispielsweise ein *read low-complexity* ist, jedoch der Zugehörige nicht. Solche Paare werden dann der dritten Kategorie *mixed* zugewiesen. So lassen sich die einzelnen *reads* in drei Kategorien einordnen:

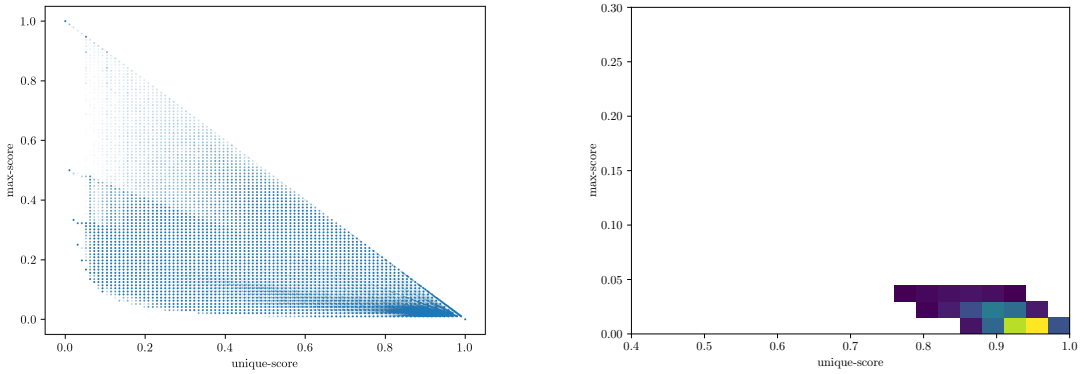
- *high-complexity reads/pairs*
- *low-complexity reads/pairs*
- *mixed reads/pairs*

[2] `https://github.com/psf/black`

Die folgenden Abschnitte beschreiben zwei Ansätze zur Bestimmung der Gewichtung und Schwellwerte. Der erste Ansatz bezieht sich auf die Verteilung der *reads* im Bezug auf die Statistiken und der Zweite auf konkrete Beispiele. Durch spätere Experimente stellte sich allerdings heraus, dass der erste Ansatz viele *reads* fälschlicherweise ausfiltert, weshalb in der anschließenden Evaluation in Abschnitt 4.5 allerdings nur Letzterer benutzt. Die Verteilungen der Statistiken sind trotzdem zur Einschätzung des zweiten Ansatzes relevant.

4.4.1 Bestimmung von Schwellwerten in Histogrammen

In einem ersten Ansatz wird zunächst für jeden *read* aus einem Test-Datensatz (das Genom einer Maus, 59,7 Millionen *reads*) der unique-score und der max-score Wert ermittelt. Bei Betrachtung der Verteilung der *reads* in Abbildung 4.1a ist eine starke negative Korrelation zwischen diesen beiden Werten zu sehen, deren Korrelationskoeffizienten nach Pearson bei -0,79 liegt. Abbildung 4.1b zeigt durch ein 2D-Histogramm in welchem Bereich die Werte



(a) Streudiagramm der *reads* nach Scores (59,7 Millionen *reads*, Alphawert 0.015)

(b) 2D Histogramm der *reads*. Nur Flächen mit mehr als zehn *reads* sind gefärbt. Helligkeit zeigt Dichte an.

Abbildung 4.1: Verteilung der *reads* auf den max-score und unique-score als Streudiagramm 4.2 und 2D Histogramm 4.1b. Die überwiegende Mehrheit der *reads* einen sehr geringen unique-score und einen hohen max-score hat.

konzentriert sind, wobei hier nur Felder mit mindestens 10 Elementen aufgeführt sind. Es wird deutlich, dass die Mehrzahl der *reads* in einem kleinen Bereich mit einem unique-score Wert in [0.7,1] und einen max-score Wert in [0, 0.05] liegen. Tests mit generierten *reads*, die sich aus wiederholenden Mustern von unterschiedlicher Größe zusammensetzen, zeigen, dass der max-score Wert diesen Schwellwert selten überschreitet, da bei solchen Mustern immer mehrere k-mere gemeinsam gehäuft vorkommen, was den maximalen Anteil eines einzelnen k-mers reduziert. Der Wert für unique-score hingegen kann zuverlässig zum Herausfiltern der generierten Beispielreads genutzt werden.

Motiviert durch diese Einsichten bietet sich eine Trennung der *reads* durch eine Hyperebene im Diagramm an, die in erster Linie an unique-score ausgerichtet ist, aber auch max-score mit einbezieht. Die Formel, um zu entscheiden, ob ein *read* auf der rechten Seite der Hyperebene ist (*high-complexity*), lautet unique-score $-$ max-score $> t$, wobei t der Schwellwert ist. Um den Bereich mit hoher Dichte zu erfassen, wird der Schwellwert im ersten Ansatz auf 0.75 festgelegt. In Abbildung 4.2 sind die auf diese Weise als *high-complex* eingestuften Werte blau und die als *low-complexity* eingestuften rot eingefärbt.

Im Umkehrschluss ergibt die Summe der klassifizierten *reads* die Gesamtanzahl der eingelesenen *reads*. Damit liegen im Beispiel aus Abbildung 4.2 84% der *reads* in *high*,

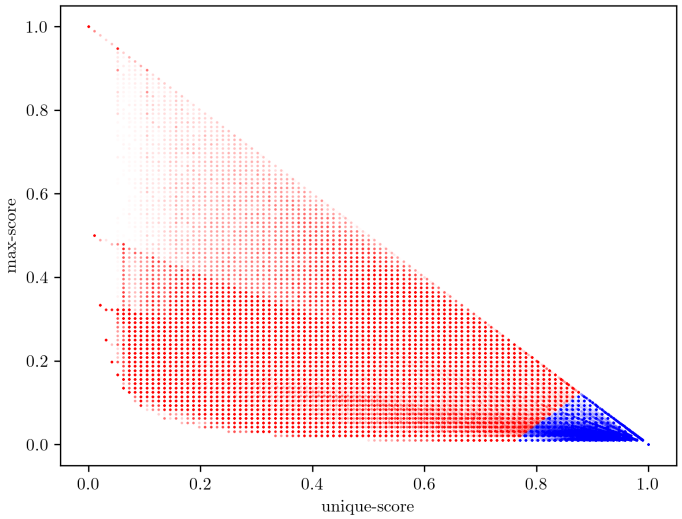


Abbildung 4.2: Kategorisierung in *low-complexity reads* (rot) und *high-complexity reads* (blau)

8% in *mixed* und 8% in *low*. Es wird deutlich, dass ein nicht unbedeutender Anteil der Kategorie *mixed* zugeordnet wurde.

Da dieser Ansatz zur Klassifizierung nur auf der Verteilung der *scores* basiert, können kaum qualitativen Aussagen über das Filterverhalten gemacht werden. Es zeigte sich jedoch, dass viele subjektiv als komplex eingestufte *reads* aussortiert wurden, weshalb der Ansatz im nächsten Abschnitt schlussendlich bevorzugt wurde. Mehrere Beispiele, welche im folgenden Unterkapitel genannt werden, fallen in diese Kategorie.

4.4.2 Bestimmung von Schwellwerten anhand von Beispielen

In einem zweiten Ansatz wurde ein exemplarischer Datensatz von *reads* von Hand erstellt, um die Filterfunktion aus dem ersten Ansatz gezielt an relevanten Randfällen auszurichten. Als Randfälle werden *reads* konstruiert, welche entweder einen signifikanten Bereich mit sehr geringer Komplexität enthalten (z.B. die eine Hälfte besteht nur aus A), mehrfach wiederholende Sequenzen verschiedener Längen oder eine Mischung aus beiden. Die Klassifizierung der Beispiele erfolgt manuell und gezielt großzügig. Dadurch soll die Zahl der *false negatives* (fälschlicherweise als *low-complexity* eingeordnet) verringert werden. In Abbildung 4.3a ist der generierte Datensatz dargestellt. Wobei rote Punkte als *low-* und blaue Punkte als *high-complexity* zugeordnet werden sollten.

Aus den dargestellten Punkten wurde von Hand eine Hyperebene konstruiert, welche die getesteten Punkte entsprechend ihrer vorherigen Klassifizierung trennen soll. Die Formel $\text{unique-score} - \text{max-score} \cdot 1.25 > t$ mit Schwellwert $t = 0.15$ wird zur Klassifizierung der Kategorie *high-complexity* verwendet. Abbildung 4.3b zeigt wie die Handgebauten *reads* klassifiziert werden, wobei folgende Randfälle durch ihre Nähe zum Übergangsbereich besonders zur Entscheidung des Schwellenwertes beigetragen haben:

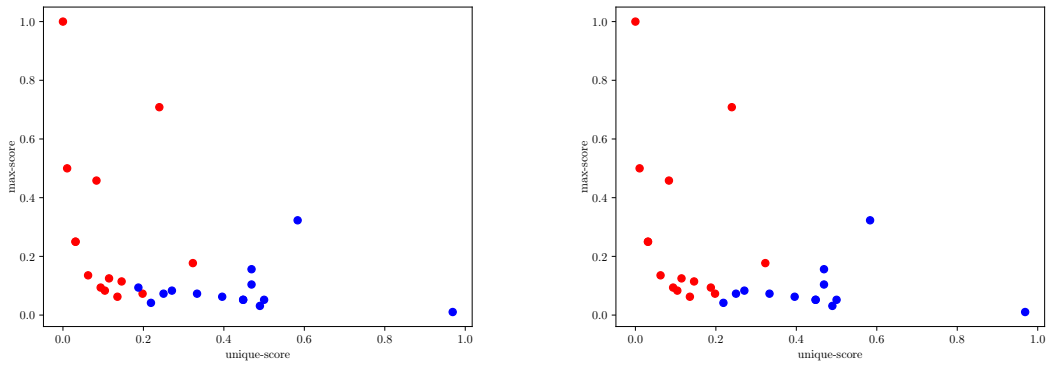
1. Randfall
   AGCTCTAGGCAGCTCTAGGCAGCTCTAGGCAGCTCTAGGCAGCTCTAGGC
   AGAGTTCCGGAGAGTTCCGGAGAGTTCCGGAGAGTTCCGGAGAGTTCCGGA
   Die Sequenz besteht aus zwei unterschiedlichen, sich wiederholenden 10 Basenpaaren,

sodass es sich mit den Werten [0.22, 0.04] im unteren, linken Bereich der blauen Region befindet.

2. Randfall
   TATAGCGATCGGGATCTAGGCTATGCTAGTATACGTCTAGATCGTACGAT
   GCTGATCTAGTACGCAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
   Eine Sequenz, bei der ein Drittel der im *read* vorkommenden Basen aus *A* besteht. Dieser *read* hat den höchsten getesteten max-*k*-mer Wert und befindet sich an der Position [0.58, 0.32]. Er bildet einen Orientierungspunkt für die Trennlinie.

3. Randfall
   AAAATTTTAAAATTTTAAAATTTTAAAATTTTAAAATTTTCCCCGGGGCC
   CCGGGGCCCCGGGGCCCCGGGGCCCCGGGGCCCCGGGGCCCCGGGGCCCCG
   Diese Sequenz ist offensichtlich nicht sehr komplex, verfügt allerdings über Werte von [0.2, 0.07]. Die Trennlinie wurde daher gezielt so gezogen, dass sie diesen *read* als *low-complexity* einstuft, wobei auch der nächste *read* mit als *low-complexity* eingestuft wurde.

4. Randfall
   ATCGTAGCTAGCTGCTACGAATCGTAGCTAGCTGCTACGAATCGTAGCTA
   GCTGCTACGAATCGTAGCTAGCTGCTACGAATCGTAGCTAGCTGCTACGAA
   Die Sequenz setzt sich aus fünf Wiederholungen der 20 Basenpaar großen Sequenz: *ATCGTAGCTAGCTGCTACGA* zusammen. Dieser *read* wurde erstellt, um als *high-komplexity* eingestuft zu werden, ist jedoch mit Position [0.19, 0.09] zwischen mehreren anderen Beispielen, die klar *low-complexity* sind und ist daher ein *false negative* klassifiziertes Beispiel.



(a) Manuelle Kategorisierung **(b)** Kategorisierung durch Hyperebene

Abbildung 4.3: Kategorisierung eines handgebauten Testdatensatzes. *Reads*, die von der verwendeten Methode als komplex eingestuft werden, sind blau, die nicht komplexen rot. Abbildung 4.3a zeigt die manuelle Einteilung des Testdatensatzes. Abbildung 4.3b zeigt die Einteilung mittels der Hyperebene, welche durch die Funktion aus Abschnitt 4.4.2 bestimmt wird.

Abbildung 4.4 zeigt das Resultat nach Anwendung des Filters mit den ermittelten Schwellwerten auf einen 500000 *reads* langen Ausschnitt des Genoms aus dem ersten Ansatz. Es wird nur diese zufällige Stichprobe abgebildet, da es trotz sehr starker Transparenz nicht möglich ist, den gesamten Datensatz so darzustellen, dass die Dichte der Streuung sichtbar wird. Im Vergleich zu Abbildung 4.3b wird viel deutlicher, dass der gewählte

Grenzwert den Bereich der höchsten Dichte großteils umfasst. Bei der Anwendung wird lediglich nur 1% der gefilterten *reads* der Kategorie *low-complexity* oder *mixed* zugewiesen. Insgesamt sollte dieser Filter eine geringe Anzahl *false negatives* hinsichtlich der Klassifizierung von *reads* der Kategorie *high-complexity* aufweisen.

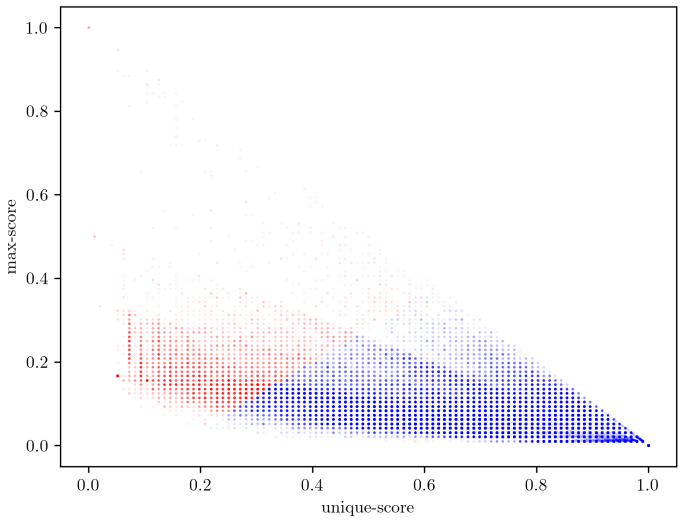


Abbildung 4.4: Kategorisierung von 500000 *reads* nach Anwendung des Filters in *low-complexity reads* (rot) und *high-complexity reads* (blau). Der Alphawert beträgt 0.05.

4.5 Evaluation

Um die Klassifizierung, die *lcfilter* (die Implementierung des Low Complexity Filters) vornimmt, hinsichtlich ihrer Genauigkeit zu beurteilen, wäre eine große Menge an vorklassifizierten realistischen Testdaten vonnöten. Da eine solche Menge nicht vorliegt und die gewünschte Klassifizierung je nach Anwendungsfall verschieden aussehen kann, bleibt nur ein relativer Vergleich mit etablierten Methoden zur Filterung von Sequenzen.

Als Vergleich dient PRINSEQ[3][23], welches seinerseits zwei Verfahren zur Filterung von *reads* mit geringer Komplexität implementiert. Die erste Methode basiert auf dem DUST Algorithmus[26, 18], welcher im populären Werkzeug BLAST[4][1] Verwendung findet. Die zweite Methode basiert auf der Block-Entropie von Wörtern[23] (k-mere).

Um zu testen, inwiefern die Mengen der aussortierten *reads* von *lcfilter* und PRINSEQ übereinstimmen, sind die Grenzwerte von PRINSEQ so eingestellt, dass die Anzahl der aussortierten *reads* möglichst identisch zu der von *lcfilter* ist. Dabei dient wieder das zuvor genutzte Mausgenom als Datensatz. Hierbei werden die Readpaare nicht berücksichtigt, weshalb es keine *mixes* Klassifizierung gibt. Der resultierende Grenzwert für DUST liegt bei 14 und für die Entropie bei 52. Der Standardgrenzwert von *lcfilter* ist 15. Bei diesen Grenzwerten filtert *lcfilter* 492999 von 59784629 *reads*, was $0,824\%$ entspricht, von denen 383403 *reads* voneinander verschieden sind. Mit der DUST Einstellung werden 441502 verschiedene *reads* gefiltert, mit der Entropie hingegen 385872. Tabelle 4.2 vergleicht die aussortierten Mengen jeder Methode anhand der *IoU scores* (*Intersection over Union*). Wie darin zu erkennen ist, hat die Menge der von *lcfilter* aussortierten *reads* große Über-

[3] `http://prinseq.sourceforge.net/`
[4] `https://blast.ncbi.nlm.nih.gov/Blast.cgi`

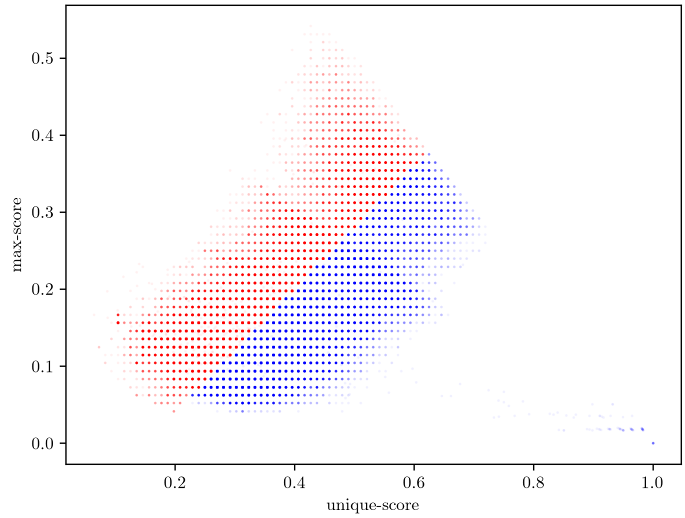


Abbildung 4.5: Verteilung der *low-complexity reads*, die nur durch lcfilter (blau) oder Entropie (rot) gefiltert wurden (Alphawert 0.05). Im Bild nicht zu sehen ist die Region der von beiden gefilterten *reads* im linken Bereich. Insgesamt befinden sich fast alle von nur einer Methode gefilterten Punkte in einem großen Cluster. Daher sind die wenigen Punkte im unteren rechten Bereich interessant, die aus einer uneinheitlichen Behandlung von *reads* mit vielen nicht definierten Basenpaaren resultiert.

schneidungen mit den aussortierten *reads* der Entropie-Methode. Die Überschneidungen mit den *reads* der DUST-Methode sind hingegen deutlich geringer.

Methode	Entropie	DUST
lcfilter	0,68	0,51
Entropie	1	0,60

Tabelle 4.2: Der *IoU score* zwischen den Mengen der aussortierten *reads* der jeweiligen Methoden

In Abbildung 4.5 wird die Verteilung der *reads* dargestellt, die nur von je *lcfilter* oder Entropie als *low-complexity* klassifiziert wurden. Darin ist der Großteil der gemeinsam erkannten *reads* ausgeblendet, der im Bild den linken Bereich ausfüllen würde. Es ist zu erkennen, dass beide Bereiche zusammen einen *cluster* bilden, der um die Trennlinie der Verteilungsfunktion liegt. Dies bestätigt, dass die Klassifizierung anhand einer Hyperebene zwischen unique-score und max-score vergleichbar mit bestehenden Methoden ist. In der Abbildung sind im linken unteren Bereich einige Punkte zu sehen, die dieser Aussage zu widersprechen scheinen. Diese sind *reads*, die zu einem großen Teil (bis zu 30%) aus nicht lesbaren Basenpaaren „N“ bestehen, beispielsweise dem *read*: CTGGCTCTCTACAGCCTCGCCATGGC-CATGAATGGCCTCATCATCTTCATCACATGNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNANN

In *lcfilter* werden diese Stellen ignoriert, wodurch sie effektiv eine kürzere Länge haben, aber trotzdem hohe Werte erreichen, während PRINSEQ diese nicht ignoriert, sondern in die Qualitätsberechnung mit einbezieht.

Laut der Dokumentation von PRINSEQ[5] sind Filtergrenzen von 7 für DUST und 70

[5] `http://prinseq.sourceforge.net/manual.html#DPFILTER`

lcfilter 1T.	lcfilter 4T.	DUST	Entropie
11:19	10:02	55:22	57:37

Tabelle 4.3: Die Laufzeiten der Tools in Minuten. DUST und Entropie sind die zwei Filter-Modi von PRINSEQ. lcfilter wurde im Rahmen dieses Projektes erstellt. Verarbeitet wurde eine 20 GB große FASTQ Datei. PRINSEQ kann nur einen *thread* benutzen, während lcfilter auch mehrere *threads* nutzen kann. In der Tabelle sind daher Laufzeiten für je einen und vier genutzte *threads* aufgeführt (1T bzw. 4T).

für Entropie üblich, während eine Entropiegrenze zwischen 50 und 60 konservativ ist. Im Vergleich dazu müssen die anhand der Größe gewählten Vergleichswerte und damit auch der Standardgrenzwert von *lcfilter* als sehr konservativ angesehen werden. Die bedeutet, dass wenige komplexe *reads* fälschlicherweise aussortiert werden (also eine hohe Präzision beim Erkennen der *low-complexity*).

Der Zeitunterschied bei der Ausführung der Programme war erheblich. Tabelle 4.3 zeigt die Laufzeiten. Da PRINSEQ in Perl geschrieben ist, braucht es zur Verarbeitung der Testdatei erheblich länger, als der *Numba*-kompilierte Pythoncode von *lcfilter*. Zudem kann *lcfilter* durch aufteilen der Eingabedateien in *chunks* die Verarbeitung der k-mere und die Anwendung der Filterfunktion auf mehrere parallel laufende *threads* verteilen, um die verfügbaren Rechenkerne besser auszunutzen. PRINSEQ bietet diese Möglichkeit nicht. Im getesteten Beispiel wirken sich die zusätzlichen *threads* allerdings nur gering auf die Gesamtlaufzeit (siehe Tabelle 4.3) aus.

Insgesamt funktioniert die untersuchte Methode des Filterns nach Komplexität anhand der Statistiken unique-score und max-score gut. Die Filterergebnisse sind vergleichbar mit anderen Methoden, die den Stand der Technik repräsentieren. Die Standardgrenzen, die anhand von Beispielen konstruiert wurden, sind verglichen mit den Standards anderer Methoden allerdings konservativ und können weiter verschärft werden. Die Ausführungsgeschwindigkeit der Implementierung ist deutlich höher als die der verglichenen Tools.

Kapitel 5 — Konzepte für die Hashtabelle

In diesem Kapitel wird das Konzept für eine Cuckoo-Hashtabelle zur Speicherung von k-meren als Schlüssel, welchen Bitsequenzen mit maximaler Länge von 64-Bit zugeordnet werden können, beschrieben. Hierbei kommt das (h, b) Cuckoo-Hashing zum Einsatz, sodass eine konfigurierbare Anzahl von Hashfunktionen sowie Anzahl Slots je Bucket verwendet wird. In dem Konzept werden Ideen für die spätere Implementierung gesammelt. Zunächst werden die Möglichkeiten für die Konfiguration der Hashtabelle umfänglich erläutert und die Generierung einer Hashtabelle zur weiteren Nutzung beschrieben. Bei der Generierung wird der Speicher der Hashtabelle reserviert und alle Funktionen zur Nutzung der Hashtabelle wie z.B. Einfügen und Suchen kompiliert. Die Funktionen werden dabei mittels *Numba* Just-in-time kompiliert. Dabei werden zur Laufzeit übergebene Parameter als Konstanten bei der Kompilierung genutzt. Hierdurch können Optimierungen vom Compiler durchgeführt werden, welche bei der Ahead-of-time Kompilierung nicht möglich wären. Des Weiteren wird der Aufbau der Schnittstelle erklärt, die dem Nutzer Interaktionen mit der Hashtabelle ermöglicht. Abschließend wird auf die Möglichkeiten der Nutzung benutzerdefinierter Module zur Berechnung der Werte sowie der Nutzung benutzerdefinierter Hashfunktionen, die von der Hashtabelle genutzt werden sollen, eingegangen.

Ferner werden hierzu auch Umsetzungsvorschläge gegeben. Somit soll das Konzept eine Grundlage für die spätere Implementierung darstellen.

5.1 Konfiguration

Da der Einsatz von Cuckoo-Hashing in Abhängigkeit von beispielsweise der Anzahl der gewählten Hashfunktionen oder der Anzahl der Elemente pro Bucket verschiedenartig ausfällt, soll die Hashtabelle konfiguriert werden könne. Hierzu können Parameter genutzt werden, die im Folgenden erläutert werden.

Die Konfiguration erfolgt entweder auf Basis einer Konfigurationsdatei oder mittels generierter Konfigurationen für die Hashtabelle. Konfigurationsdateien bieten dabei den Vorteil, dass sie wiederverwendbar sind und unabhängig von dem Programmcode gespeichert und geladen werden können.

Im Folgenden wird auf die einzelnen Parameter, ihre Funktion und ihre Bedeutung eingegangen, als auch die Auswirkungen auf das Verhalten der Hashtabelle. So werden bei der Beschreibung der Parameter neben den konzeptionellen Punkten auch Erkenntnisse aus der Implementierung dargelegt.

Im Anschluss soll die Verwendung erläutert werden, so wird auf die optionale Möglichkeit und die Umsetzung der Konfiguration mittels Konfigurationsdatei eingegangen.

5.1.1 Parameter

Folgend ist eine Liste aller zur Verfügung stehenden Konfigurationsparameter und ihre jeweilige Funktion bzw. eine erläuternde Beschreibung dieser aufgeführt:

`dimensions` Der Konfigurationsparameter dimensions umfasst alle grundlegenden Konfigurationsparameter, die auf die Struktur der Hashtabelle Einfluss nehmen. Dimensions weist die Subparameter n, load_factor, bucket_size und k-mer Größe auf, die im Folgenden erläutert werden.

`n` Gibt die Anzahl der möglichen Schlüssel, die in der Hashtabelle gespeichert werden sollen, an. Je größer das gewählte n, desto größer ist der Speicherbedarf der Hashtabelle.

`load_factor` Für den load_factor können somit Werte zwischen 0 und 1 angegeben werden. Aus dem load_factor ergibt sich die Kapazität $c = n/load\_factor$ der Hashtabelle. Ein niedriger load_factor bedeutet, dass Funktionen auf der Hashtabelle, wie Einfügen und Suchen schneller ablaufen, jedoch wird auch mehr Speicher benötigt. Bei der Nutzung von zwei Hashfunktionen haben sich Werte zwischen 0.8 und 0.95 als empfehlenswert herausgestellt, um einen Ausgleich zwischen Speicherbedarf und Geschwindigkeit zu erzielen. Wobei der Wert je nach Anforderung in der Nutzung schwanken kann.

`bucket_size` Die bucket_size gibt an, wie viele Schlüssel-Werte-Paare mit dem gleichen Hashwert an der gleichen Position in der Hashtabelle gespeichert werden können. Beim normalen Cuckoo-Hashing wird immer ein Wert pro Hashposition gespeichert, dies wird mit der Angabe einer bucket_size von 1 erreicht. Bei der Suche bewirkt eine niedrige bucket_size ein schnelles Auffinden des Elements an der Hashposition, weil nur wenige Werte verglichen werden müssen. Jedoch werden diese aufgrund der geringen Anzahl an verfügbaren Positionen schneller gefüllt, sodass mehr Verschiebungen beim Einfügen mittels Random Walk Strategie nötig sind. Ebenso erlaubt eine niedrige bucket_size nur niedrige Füllraten. Eine größere bucket_size hat den gegenteiligen Effekt und führt so zu weniger Verschiebungen beim Einfügen und jedoch auch zu mehr Vergleichen bei der Suche. Bei der Umsetzung haben sich Werte zwischen 4 und 16 als sinnvoll ergeben. Der genaue Wert hängt wiederum von den Anforderungen ab.

k-`mer Größe` Da die Hashtabellen primär auf den Einsatz im Kontext der Verarbeitung von DNA-Sequenzen gleicher Länge (k-mere) ausgerichtet sind, ist die Angabe der Länge dieser einzelnen Sequenzen eine Option. Die k-mer Größe des Schlüssels kann dabei zwischen $1 \leq k \leq 32$ liegen. Die resultierende Schlüsselgröße ist $2k$ Bits. In der Konfigurationsdatei wird die Größe mit dem Schlüssel `k` angegeben.

`value_module` Für die Berechnung der Werte der Hashtabelle werden Value Module verwendet. Neben den vordefinierten Value Modulen besteht die Möglichkeit, ein benutzerdefiniertes Value Module anzugeben. In der Konfiguration wird dazu das Value Module angegeben, das in der Hashtabelle verwendet werden soll. Die Angabe erfolgt per Python-Modulnamen (Beispiel: `"gecocoo.valuemodules.store"`). Zur Laufzeit kann dieses dann aufgelöst werden und das referenzierte Modul geladen werden. Optional können dabei Parameter zum Erzeugen einer Instanz des Moduls angegeben werden. Diese werden dann zur Initialisierung des Moduls verwendet.

Im Abschnitt 5.4 werden die vordefinierten Value Module vorgestellt und auf die Verwendung benutzerdefinierter Value Module eingegangen.

`hash_functions` Für das von der Hashtabelle verwendete Cuckoo-Hashing können die zu verwendeten Hashfunktionen angegeben werden. Dabei wird einerseits die Anzahl der Hashfunktionen angegeben und welche Hashfunktionen verwendet werden.

Hierbei stehen vordefinierte Funktionsfamilien wie z.B. swaplinear zur Verfügung (siehe Abschnitt 5.5). Diese werden durch Angabe von Parametern spezifiziert.

Die Hashfunktionen werden als Liste der Hashfunktionen angegeben. Hierzu wird jeweils der Name der verwendenden Hashfunktion und Parameter angegeben. Schließlich können die Hashfunktionen konkret mithilfe der Schlüssel `f1`, `f2`, `f3` usw. gewählt werden.

Neben den vordefinierten Hashfunktionen können benutzerdefinierte Hashfunktionen verwendet werden, wobei anstelle des Namens eine Referenz auf den Konstruktor der Funktion angegeben wird (Beispiel: `"gecocoo.hashing.swaplinear"`). Auf die Umsetzung benutzerdefinierter Hashfunktionen wird im Abschnitt 5.5 eingegangen.

`storage` Der Konfigurationsparameter storage legt fest, wie innerhalb der Hashtabelle gespeichert werden soll. Storage weißt die drei Subparameter mode, concurrent, disable_quotienting und disable_occupied_flag auf, die im Folgenden erläutert werden.

`mode` Legt den zu verwendenden Storage-Mode fest, wobei zwischen `Packed`, `Buckets` und `Fast` gewählt werden kann. Der Standardmodus ist `Buckets`.

Storage-Mode `Packed`, der alle Hash-Tabelleneinträge in bitgenaue Felder packt und damit so wenig Speicherplatz wie möglich verschwendet. Die Speichereffizienz hat eine schlechtere Datenausrichtung und mehr Bitoperationen zur Folge, die zum Extrahieren der Werte durchgeführt werden müssen.

Storage-Mode `Buckets` ist ein Kompromiss von Speichereffizienz und Verarbeitungsleistung. Jeder Bucket wird an 64-Bit-Adressen ausgerichtet. Dies verbessert die Verarbeitungsleistung und spart gleichzeitig etwas Speicher, indem die Slots pro Bucket eng gepackt werden.

Storage-Mode `Fast` bevorzugt die Leistung gegenüber der Speichereffizienz. Er weist eine optimierte Ausrichtung auf und führt weniger Bitoperationen durch. Er erfordert auch die Deaktivierung jeglicher Quotientierung durch die gewählten Hashfunktionen und vermeidet damit auch die Schlüsselwiederherstellung, die bei Schlüsselvergleichen mit Quotientierung bei den anderen Speichermodi durchgeführt werden.

`concurrent` Flag für den Concurrent-Support für die Hashtabelle. Dieser ermöglicht vorwiegend das gleichzeitige Einfügen von Elementen in die Hashtabelle. Die Speicheroperationen der Hashtabelle werden atomar durchgeführt, wenn der Wert `True` übergeben wird. Das atomare Ausführen der Speicheroperationen erfordert Locking, sodass es hier bei nicht erforderlicher Verwendung zu einem Overhead kommt. Sollte deaktiviert werden, wenn nur Single-Thread-Zugriff erforderlich ist (Standardwert `False`).

`disable_quotienting` Deaktiviert Quotienting, welches die gewählte Hashfunktion durchführen könnte, sodass der vollständige Schlüssel in der Hashtabelle gespeichert werden muss (Standardwert `False`). Bei `True` ist das Quotienting deaktiviert und bei `False` aktiviert. Ist das Quotienting deaktiviert, muss zum Beispiel beim Iterieren über die Hashtabelle nicht mehr der Schlüssel berechnet werden, da dieser bereits vollständig gespeichert wird.

Ist Quotienting hingegen aktiv, so wird nicht der gesamte Schlüssel eines k-mers sondern lediglich der Quotient sowie die genutzte Hashfunktion und der Wert gespeichert. Der Quotient ergibt sich aus der Division des numerischen Schlüssels und der Anzahl der verwendeten *Buckets*. Aufgrund der geringeren Anzahl an genutzten Bits im Vergleich zum vollständigen Speichern des Schlüssels verringert sich der Speicheraufwand.

`disable_occupied_flag` Flag zur Erkennung, ob ein Slot bereits belegt ist (Standardwert `False`). Das Flag kann derzeit nur im `Buckets` Modus verwendet werden.

5.1.2 Einlesen der Konfigurationsdatei

Zur Verwendung wurde es neben der programmatischen Konfiguration ermöglicht, die Konfiguration mittels Konfigurationsdatei vorzunehmen.

Ein Auszug einer exemplarischen Konfigurationsdatei kann in Listing 5.1 eingesehen werden.

```yaml
# Sample configuration file for GeCoCoo

dimensions:
    n: 100000000
    k: 31
    bucket_size: 4
    load_factor: 0.98

values:
    module_name: "gecocoo.valuemodules.store"
    parameters:
    - 32

hashfunctions:
    choices: 3
    default_hash: xorlinear

    functions:
        - name: swaplinear
          parameters:
            - 23
        - default
        - name: customhashfunction
          parameters:
            - 23
            - 32

storage:
    mode: BUCKETS
    concurrent: true
    disable_quotienting: true
```

Listing 5.1: Ausschnitt einer Konfigurationsdatei

5.2 Generierung

Zur Generierung der Hashtabelle soll eine Funktion genutzt werden. Diese erhält als Parameter die Konfiguration, auf die im Abschnitt 5.1 näher eingegangen wird. Um Fehler, die sich aus der Konfiguration während der Laufzeit ergeben könnten, vorzubeugen, werden die einzelnen Konfigurationsparameter vor der Generierung validiert. So könnten insbesondere bei den Referenzen auf die Value Module bzw. Hashfunktionen fehlerhafte Eingaben vom Nutzer getätigt werden. Die Funktion gibt ein Tupel zurück, das die Schnittstelle zur Hashtabelle darstellt. Die einzelnen Komponenten des Tupels werden im folgenden Abschnitt 5.3 erläutert. Neben dieser programmatischen Generierung der Hashtabelle kann diese auch geladen werden. Hierzu wird eine zuvor gespeicherte bzw. serialisierte Hashtabelle deserialisiert (siehe 5.3).

5.3 Schnittstelle der Hashtabelle

Grundsätzlich muss gewährleistet werden, dass der Nutzer mit der Hashtabelle stets über eine gleichbleibende Schnittstelle interagieren kann. Die Schnittstelle muss also unabhängig von jeglicher Art von Konfiguration sein, also eine konfigurations-agnostische Struktur aufweisen.

Die Schnittstelle besteht aus mehreren Bestandteilen, die jeweils verschiedene Aufgaben erfüllen. Eine Bündelung der Bestandteile zu einem einzelnen Objekt erscheint hierbei für eine vereinfachte Handhabung sinnvoll. Python bietet dafür verschiedene Möglichkeiten. Dazu zählen die Nutzung von Tupeln, benannten Tupeln und Datenklassen mit Verwendung des Dekorators `@dataclass`. Tupel sind zwar leicht erweiterbar, erweisen sich allerdings aufgrund der hohen Anzahl an Elementen und der damit verbundenen Unübersichtlichkeit als nicht praktikabel. Benannte Tupel und Datenklassen sind aus Nutzersicht etwa vergleichbar. In beiden Fällen kann auf einzelne Bestandteile eines Objekts durch einen eindeutigen Namen zugegriffen werden. Benannte Tupel erlauben es neben der Adressierung der Attribute über ihren Namen auch auf diese über einen Index zuzugreifen, so wie gewöhnliche Tupel. Datenklassen erlauben diese Art der Adressierung nicht. Jedoch erlauben diese dazwischen zu wählen, ob die Attribute nach der Erstellung geändert werden dürfen, während benannte Tupel wie auch Tupel immer unveränderbare Attribute haben. Darüber hinaus erlauben es Datenklassen, diverse Methoden für die Klasse automatisch zu generieren, allerdings umfasst diese Generierung lediglich Vergleichsfunktionen, Hashes und andere sehr generisch implementierten Funktionalitäten, die sich auf die Klasse beziehen und somit bei unserer Anwendung von keinem bedeutenden Nutzen wären. Für benannte Tupel stehen hingegen einige Funktionen zur Verfügung, die es etwa erlauben, Instanzen leicht in andere Darstellungsformen umzuwandeln. Also Dictionaries oder andere Komfortfunktionen wie etwa Instanzen zu erzeugen, die die gleichen Werte haben wie vorherige Instanzen, nur dass einzelne Attribute andere Werte speichern. Bei der Implementierung der Hashtabelle wurde so auf die Verwendung von benannten Tupeln zurückgegriffen.

Bei der Generierung der Hashtabelle wird ein Tupel zurückgegeben, das wiederum zwei benannte Tupel aufweist. Zum einen das `HashtableInterface`, das Funktionen zur Nutzung aus einem JIT-Kontext bereitstellt. Zum anderen die `HashtableUtilities`, die sämtliche Funktionen enthält, die nicht aus einem JIT-Kontext aufgerufen werden können. Diese Trennung ermöglicht den expliziten Einsatz innerhalb `@jit`-dekorierten Funktionen und gewährleistet so eine effiziente Nutzung.

5.3.1 HashtableInterface

`storage` Das Storage-Objekt stellt den Speicher der Hashtabelle dar. Das Objekt enthält den Speicher der Hashtabelle und weitere Informationen, die für die Implementierung, aber nicht für den Nutzer, wichtig sind. Dieser muss den folgenden Funktionen übergeben werden, um auf diesem Operationen auszuführen. Die Struktur des Storage-Objekts ist von der Konfiguration der Hashtabelle (siehe 5.1) abhängig. Das Objekt sollte daher wie die anderen Funktionen auch lediglich über das Interface genutzt werden. In der Implementierung handelt es sich bei dem Storage-Objekt um ein *NumPy*-Array, welches sowohl den Speicher der Hashtabelle als auch zusätzlichen Speicher für Hilfswerte wie die Schlüsselanzahl beinhaltet.

`insert_single` Mittels der `insert_single` Funktion können Schlüssel mit ihrem zugehörigen Wert der Hashtabelle hinzugefügt werden. Die Einfügefunktion bietet zugleich auch die Funktionalität an, einen Wert in der Hashtabelle zu aktualisieren. Der Funktion wird dabei der Schlüssel sowie alle Werte, die das zu nutzende Value Module benötigt,

übergeben. Zurückgegeben wird ein Boolean, so wird bei erfolgreichem Einfügen in die Hashtabelle True und bei einer Änderung False zurückgegeben. Wenn das Einfügen nicht erfolgreich ist, wird ein Fehler geworfen.

`initialize` Funktion zur Initialisierung einer leeren Hashtabelle. Die Funktion nimmt einen Schlüssel- und einen gleich großen Werte-Array von einzufügenden Elementen entgegen und fügt sie mithilfe der kostenoptimalen Einfügestrategie in die Tabelle ein. Falls die Schlüssel-Menge, die in der Tabelle gespeichert werden soll, initial bekannt ist und anschließend nur Update- oder Search-Operationen für die Werte ausgeführt werden sollen, ist der Einsatz der initialize Funktion der Nutzung von `insert_single` vorzuziehen. Die liegt daran, dass durch die kostenoptimale Einfügestrategie alle Schlüssel in Summe minimale Kosten haben. Da die Kosten sich aus dem Index der verwendeten Hashfunktion errechnen und somit in etwa der Anzahl der *cache misses* für den Zugriff auf einen vorhandenen Schlüssel gleichkommen, kann so die Zugriffszeit auf die Elemente in der Tabelle erheblich reduziert werden. Darüber hinaus garantiert die Methode ein erfolgreiches Einfügen der Schlüssel-Menge, falls dies für die gegebene Konfiguration überhaupt möglich ist. Da aber beim Einfügen neuer Schlüssel per Random-Walk bereits das Einfügen eines Schlüssels in beliebig langen Ersetzungsketten resultieren kann und dabei nicht auf die Minimierung der Kosten geachtet wird, ist der Einsatz der initialize Funktion in Situationen, in denen Schlüssel häufig neu eingefügt werden sollen, zwecklos, da die Cachefreundlichkeit rasch verloren geht.

Anzumerken ist dabei, dass die Schlüssel im übergebenen Array einzigartig sein müssen, da ansonsten mehrfach vorkommende Schlüssel nicht automatisch zu einem Eintrag in der Tabelle vereinigt werden.

`search_single` Die Funktion sucht nach einem Schlüssel in der Hashtabelle. Das Storage-Objekt sowie der Schlüssel werden der Operation dabei übergeben und der zugehörige Wert zurückgegeben. Der Rückgabewert ist dabei von dem verwendeten Value Module abhängig. Wenn der Schlüssel nicht in der Hashtabelle vorhanden ist, wird `None` zurückgegeben.

`remove_single` Die Funktion löscht einen Schlüssel aus der Hashtabelle. Das Storage-Objekt sowie der Schlüssel werden der Operation dabei übergeben und der zugehörige Wert, wenn in der Hashtabelle vorhanden zurückgegeben. Der Rückgabewert ist dabei von dem verwendeten Value Modul abhängig. Wird kein Wert in der Hashtabelle an der angegebenen Stelle gefunden, wird `None` zurückgegeben.

`clear` Eine Resetfunktion der Hashtabelle um alle Einträge, die in der Hashtabelle vorhanden sind, zu löschen. Die Werte in der Hashtabelle werden alle gleichzeitig entfernt und die Funktionen der Hashtabelle können weiterhin benutzt werden, ohne diese erneut generieren zu müssen. Als Parameter wird entsprechend das Storage-Objekt erwartet.

`key_iterator` Gibt einen Iterator zurück, der für jeden Eintrag der Hashtabelle ein Schlüssel-Wert-Paar enthält.

`get_key_count` Gibt die Anzahl an Schlüsseln zurück, die in der Hashtabelle gespeicherten sind.

5.3.2 HashtableUtilities

`serialize` Die Funktion serialisiert die Hashtabelle und schreibt diese in eine Datei. Hierzu wird als Parameter entweder der Dateiname oder ein geöffnetes Dateiobjekt erwartet.

Als weiteres wird ein Flag erwartet, die bestimmt, ob bei der Serialisierung die Ausgabedatei komprimiert werden soll, also entsprechend ein ZIP-Archiv erstellt wird. Bei der Serialisierung wird sowohl der Speicher der Hashtabelle als auch die Konfiguration dieser gespeichert, sodass diese bei der Deserialisierung vollständig wiederhergestellt werden kann.

`occupancy` Die Funktion berechnet Metadaten zur aktuell verwendeten Hashtabelle. Als Parameter wird dazu die Anzahl der Segmente übergeben, in die die Tabelle unterteilt werden soll. Diese Segmente sind gleich große zusammenhängende Folgen von Buckets. In einer Tabelle mit 30 Buckets und 3 Segmenten enthält das erste Segment die Buckets 0 bis 9, das zweite die Buckets 10 bis 19 und das dritte die Buckets 20 bis 29. Für jedes Segment ermittelt die Funktion zwei Datensätze:

- Für jede Hashfunktion, die in der Tabelle im Einsatz ist, also 0 bis $h-1$, wird ermittelt, wie viele Schlüssel aus dem Segment mit ihr in ihren Bucket eingefügt wurden. So kann für jedes Segment überprüft werden, ob eine oder mehrere Hashfunktionen häufig in jenes abbilden. Dies kann ein Indiz für eine schlecht geeignete Hashfunktion sein.
- Für jeden Slot-Index pro Bucket, also 0 bis $b-1$, wird die Anzahl der Schlüssel in diesem Segment, die einen solchen Slot-Index besetzen, ermittelt. Hat also eine Tabelle mit $b = 4$ ein Segment mit 5 Buckets von denen 4 einen Schlüssel speichern und 1 Bucket vier Schlüssel speichert, so würde für den ersten Slot-Index 5 ermittelt, weil es fünf Slots mit Schlüssel gibt und für alle anderen Slot-Indizes 1. Auch hiermit können Aussagen über die Streuung der Hashfunktionen bezogen werden.

Die so ermittelten Daten können mithilfe von Plotting-Bibliotheken visualisiert werden und so Aufschluss über die Verteilung der eingesetzten Hashfunktionen und den Füllgrad der Buckets in bestimmten Regionen der Tabelle geben. Dadurch kann die Eignung von Hashfunktionen für eine Schlüsselmenge zu einem gewissen Grad hin überprüft werden.

5.4 Value Module

Um zu einem gegebenen Schlüssel den zu speichernden Wert zu berechnen, stellt die Tabelle eine Schnittstelle für Module zur Verfügung. Diese Schnittstelle legt Funktionen fest, die bei einer Einfügeoperation aufgerufen werden. Die Module können entweder, den Anforderungen an die Schnittstelle entsprechend, vom Nutzer der Hashtabelle entwickelt werden oder der Nutzer greift auf bestehende Module zurück. Einfache und häufig benötigte Module sollten deshalb bereits mit der Hashtabelle ausgeliefert werden. So stehen die drei Value Module `counting`, `store` und `keyset` standardmäßig zur Verfügung.

Das Modul `counting` zählt jedes Auftreten von einzelnen Schlüsseln. Ein Initialwert sowie ein Schrittwert wird angegeben, sodass beim Einfügen der initiale Wert gesetzt und bei einer Änderung entsprechend dem Schrittwert inkrementiert wird, wobei dies unabhängig vom Vorzeichen der Werte ist.

Das Modul `store` speichert zu jedem Schlüssel eine angegebene Anzahl von *Bits* und dient so als simpler Speicher. Wobei die *Bits* den Funktionen zum Einfügen und Ändern als Parameter übergeben werden. Hierbei wird beim ersten Einfügen sowie bei Änderungen analog verfahren.

Das Modul `keyset` lässt die Hashtabelle wie ein einfaches *Set* fungieren. So werden lediglich die Schlüssel in der Hashtabelle gespeichert. Bei Suche in der Hashtabelle nach einem Schlüssel wird beim Value Module *keyset* bei vorhandenem Schlüssel *True* zurückgegeben.

5.4.1 Benutzerdefinierte Value Module

Neben den vordefinierten Value Modulen ist es möglich, benutzerdefinierte Module anzuwenden. Die Verwendung dieser Module wird in der Konfiguration wie in Abschnitt 5.1 beschrieben.

Das Listing 5.2 zeigt ein exemplarisches Value Module. Ein solches muss die drei Funktionen `first_insert`, `update_value` und `map_value` implementieren, die jeweils den berechneten Wert zurückgegeben und diese als Rückgabewert bereitstellen.

Die Funktion `first_insert` wird genutzt, um einen noch nicht in der Hashtabelle vorhandenen Schlüssel hinzuzufügen. Der Wert dieses Schlüssels wird berechnet und als Rückgabewert erwartet. Intern wird die *insert_ single* Methode des HashtableInterface aufgerufen und der ermittelte Wert in der Hashtabelle gespeichert. Die Funktion `update_value` wird genutzt, um einen in der Hashtabelle vorhandenen Schlüssel zu ändern. Der Wert dieses Schlüssels wird berechnet und als Rückgabewert erwartet. Die initiale Einfüge- und die Änderungsfunktion wurden voneinander getrennt, sodass die Nutzung unabhängig von ihrem Kontext erfolgen kann und Abfragen innerhalb hinfällig sind. Bei der Anwendung wird somit der Kontext direkt verdeutlicht und die Implementierung dessen findet auf der abstrahierten Ebene Anwendung.

Und die Funktion `map_value` ermittelt aus einem angegebenen Schlüssel den zugehörigen Wert und wird somit bei Aufruf der *search_ single* Methode aus dem HashtabelInterface angewandt. Wie der Wert berechnet wird, ist ein Implementierungsdetail und kann je nach Anwendung variieren.

```
1  def custom_value_module():
2      @njit(uint64(uint64))
3      def first_insert(key):
4          return custom_return
5
6      @njit(uint64(uint64, uint64))
7      def update_value(key, stored_value):
8          return custom_return
9
10     @njit
11     def map_value(key, stored_value):
12         return custom_return
13
14     return (
15         first_insert,
16         update_value,
17         map_value,
18         bits,
19     )
```

Listing 5.2: Beispielhaftes benutzerdefiniertes Value Module

5.5 Hashfunktionen

Analog zu den Value Modulen stehen dem Nutzer vordefinierte Hashfunktionen zur Verfügung, die einen Schlüssel zur Verwendung in der Hashtabelle berechnen. Hierbei stehen dem Nutzer die vordefinierten Hashfunktionen `swaplinear` und `xorlinear` zur Verfügung.

Die Verwendung dieser Hashfunktionen wird in der Konfiguration wie in Abschnitt 5.1 beschrieben.

Die Funktion `swaplinear` arbeitet, indem sie zunächst die obere und untere Hälfte der Bits des Schlüssels vertauscht und dann mit einem bestimmten Wert, dem sogenannten *Seed*, multipliziert. Die Funktion `xorlinear` funktioniert analog zu `swaplinear`, jeodch wendet diese auf den resultierenden Schlüssel noch eine *xor* Operation an, um eine bessere Verteilung insbesondere für kleine Schlüssel zu erhalten. Auf die Implementierung der Funktionen wird im Abschnitt 6.2 näher eingegangen.

5.5.1 Benutzerdefinierte Hashfunktionen

Neben den vordefinierten Hashfunktionen steht es dem Nutzer analog zu den Value Modulen frei, eigene Hashfunktionen anzugeben. Die zu verwendenden Hashfunktionen müssen dabei reversibel sein, um in die Hashtabelle eingefügte Einträge laden zu können. Eine Hashfunktion muss dazu die Funktionen `forward` und `backward` implementieren und diese mit der Anzahl an benötigten *Bits* zum Speichern des Schlüssels zurückgeben. Die `forward` Funktion berechnet aus einem Schlüssel den zugehörigen *Hash* und die `backward` Funktion berechnet entsprechend aus einem Hash den zugehörigen Schlüssel. Als Parameter erhält die Hashfunktion die Anzahl der Buckets, die verwendete k-mer Größe, einen *Seed* zur Randomisierung sowie weitere spezifische Parameter als Liste.

5.6 Verwendung

Für die Verwendung der Hashtabelle sollten die folgenden Schritte angewandt werden:

1. Konfiguration erstellen
2. Hashtabelle generieren
3. Datei oder Dateien mit der oder den Gensequenzen laden (z.B. FASTQ-Datei)
4. k-mere extrahieren, entsprechende Werte für diese generieren und die Schlüssel-Werte-Paare einfügen
5. Berechnete Hashtabelle analysieren oder anderweitig einsetzen

Sie kann beispielsweise als Referenztabelle für k-mere verwendet werden, die jedem k-mer eine Übereinstimmung mit verschiedenen Referenzgenomen zuordnet [32]. Das Vorgehen für eine solche Anwendung hat mit der Hashtabelle folgende Form:

1. Auf Basis der Hardware und des Umfangs der Genome, der Spezies nach denen das zu klassifizierende Genom klassifiziert werden soll, wird die Konfiguration definiert.
2. Auf Basis der Konfiguration wird die Hashtabelle generiert.
3. Die Dateien, die die Genome auf deren Basis die Hashtabelle erstellt werden soll enthalten, werden geladen.
4. Die Genome werden in k-mere zerlegt und für die k-mere wird ermittelt was sie für eine Beziehung zu den Spezies haben, also ob sie exklusiv in einer bzw. in mehreren Spezies vorkommen oder ob es sich um eine Adaptersequenz handelt. Diese Information wird pro k-mer kodiert und als Wert dem Schlüssel in der Tabelle zugeordnet, wobei der Schlüssel das k-mer darstellt.
5. Schließlich kann die Tabelle genutzt werden um für andere Genome Sequenzen dieser einer der Spezies aus der Tabelle zuzuordnen.

Kapitel 6 — Implementierung der Hashtabelle

Das im ersten Semester der Projektgruppe erarbeitete Konzept für eine Hashtabelle wurde bereits im Kapitel 5 vorgestellt. Im zweiten Semester wurde das Konzept präzisiert und umgesetzt. So wurde eine konfigurierbare Hashtabelle zur effizienten Speicherung von k-meren entwickelt. Im folgenden Kapitel wird auf die Umsetzung, welche die Nutzungsmöglichkeiten der Hashtabelle mittels dessen Interface, die Konfiguration und die finale Implementierung umfasst, eingegangen. Abschließend werden Ergebnisse der Benchmarks der Hashtabelle vorgestellt.

6.1 Konfiguration

Der Aufbau und die auf ihr anzuwendenden Operationen der Hashtabelle haben im Hinblick auf die Leistung sowie die Speichereffizienz Einfluss. Die Anforderungen, die an die Hashtabelle gestellt werden, variieren dabei je nach Anwendung und der damit verbundenen Zielsetzung. Ebenso werden unterschiedliche Ansprüche und Möglichkeiten zum Einsatz von Ressourcen gefordert. So kann ein möglichst geringer Speicherbedarf oder eine performante Ausführung als Ziel gesetzt werden.

Diese Eigenschaften werden daher als Parameter bei der Generierung der Hashtabelle definiert und so die Konfiguration dieser ermöglicht. Im Folgenden wird auf die einzelnen Parameter und ihre Funktion und Bedeutung eingegangen, woraufhin im Anschluss die Verwendung erläutert werden soll. Dabei wird auf die optionale Möglichkeit und die Umsetzung der Konfiguration mittels Konfigurationsdatei eingegangen.

Die zur Verfügung stehenden Konfigurationsparameter und ihre jeweilige Funktion bzw. eine erläuternde Beschreibung wurde im Abschnitt 5.1 aufgeführt.

6.1.1 Einlesen der Konfigurationsdatei

Zusätzlich zu der Möglichkeit, programmatisch die Konfiguration vorzunehmen, sollte es ermöglicht werden, eine Konfigurationsdatei zu nutzen. Dies erlaubt es dem Nutzer, Konfigurationen mit Dritten zu teilen und diese je nach Anwendungsfall durch den Austausch der Konfigurationsdatei zu variieren. Somit soll der Benutzerfreundlichkeit der Hashtabelle erhöht werden.

Als Dateiformat für die Konfigurationsdatei wurde auf das *YAML*-Format gesetzt. Dieses eignet sich aufgrund der leichten bzw. menschlichen Lesbarkeit und der Verwendung simpler Datenstrukturen wie Listen und Skalaren, die von aktuell verbreiteten Programmiersprachen genutzt werden, als Konfigurationsformat. Um Verschachtelungen im *YAML*-Format verwenden zu können, wird Einrückung im Python-Stil angewandt. Dies stellt zwar im Hinblick auf die Bedienbarkeit eine Restriktion dar, jedoch sollte diese Entscheidung für die meisten Anwender aufgrund der Verbreitung von Python vertretbar sein.

Zum Einlesen der *YAML*-Datei wurde auf den *YAML*-Parser *strictyaml* gesetzt. Dieser setzt in der aktuellen Version 0.2 auf eine Untermenge des *YAML*-Standards, welche jedoch für die intendierte Verwendung ausreichend ist. Insbesondere der typsichere Umgang von *strictyaml* ermöglicht es, Anwendungsfehler von Nutzern frühzeitig zu detektieren. Auf diese Weise wird, durch Angabe des Schemas der Konfigurationsdatei, bereits beim Einlesen der jeweilige Typ der einzelnen Konfigurationsparameter validiert.

Mittels der Funktion `load_from_yaml` aus dem Paket `gecocoo.config` kann aus der zuvor eingelesenen Konfigurationsdatei die Konfiguration geladen werden, die dann zur Generierung der Hashtabelle genutzt werden kann.

6.2 Hashfunktionen

Um eine maximale Ausnutzung des Platzes innerhalb der Tabelle zu ermöglichen, wird Quotienting genutzt. Beim Quotienting wird nicht der Schlüssel selbst in der Tabelle gespeichert, sondern nur genügend Informationen gespeichert, um zusammen mit der Adresse des Eintrages die Identität zweier Schlüssel eindeutig feststellen zu können. Um sicherzustellen, dass der Schlüssel aus diesen beiden Informationen wiederhergestellt werden kann, wird darüber hinaus vorausgesetzt, dass die Hashfunktion umkehrbar sein muss. Nach Anwendung der Hashoperation wird aus dem Hashwert durch Division mit der Anzahl der *Buckets* der Quotient als Ersatz für den Schlüssel genutzt und der Rest als Adresse des *Buckets*. In der Implementierung werden diese beiden Schritte kombiniert. Daher müssen die Hashfunktionen, welche im Folgenden beschrieben werden, nur die zwei Methoden `forward(key)` → `(rest,quotient)` und `backward(rest,quotient)` → `key` implementieren. Außerdem muss bei der Erstellung der Funktionen ermittelt werden, wie viele Bits für den Quotienten reserviert werden müssen. Durch den Endnutzer geschriebene Funktionen müssen diese beiden Funktionen unterstützen, außer es wird explizit auf Quotienting verzichtet. In diesem Fall wird der vollständige Schlüssel in der Tabelle gespeichert und die Umkehrfunktion ist somit trivial.

6.2.1 swaplinear

Diese Hashfunktion nutzt Multiplikation und Modulo, um eine gute Streuung der Schlüssel zu erreichen. Außerdem werden die obere und untere Hälfte der entstehenden Bitmuster rotiert, um die höherwertigen Bits an der Streuung zu beteiligen, da diese bei einem kleinen Adressraum insignifikant wären. Sie ist bijektiv und wurde schon in anderen Arbeiten zu Cuckoo-Hashing benutzt [33].

Die folgenden Funktionen bilden Schlüssel der Länge $2k$ auf Hashwerte ab und umgekehrt. Darin ist a ein beliebiger ungerader Wert und a' sein multiplikatives inverses mit 4^k. Die Funktion rotate_k tauscht die oberen und unteren k Bits eines Wertes.

$$\text{forward-hash}(key) = a \times \text{rotate}_\text{k}(key) \bmod 4^k \tag{6.1a}$$

$$\text{reverse-hash}(x) = \text{rotate}_\text{k}(a' \times x \bmod 4^k) \tag{6.1b}$$

Aus den Hashwerten lassen sich dann durch Division mit Rest die Quotienten und Adressen berechnen.

6.2.2 xorlinear

Diese Hashfunktion stellt eine leichte Verbesserung gegenüber der vorherigen Funktion dar, welche verschiedene Schlüssel besser trennt. Dazu wird die obere Hälfte der Schlüssel-Bits durch `xor` mit der unteren Hälfte verknüpft. Im Anschluss werden beide Hälften dann wie in Funktion 6.1 vertauscht und durch Multiplikation kodiert. Beim Umkehren wird die untere Häfte der Hash-Bits mit der oberen Hälfte (welche den unteren Schlüssel-Bits entspricht) dekodiert, um die ursprüngliche obere Hälfte des Schlüssels zu erhalten. Auf diese Weise tragen mehr Bits des Schlüssels zu den Adressbits bei, statt alle Schlüssel mit gleicher oberen Hälfte auf eine Adresse abzubilden. Dies ist besonders bei vielen kleinen Werten unter k Bits hilfreich, welche sonst oft für Kollisionen sorgen würden.

Angenommen die Funktionen upper_k und lower_k teilen das Bitmuster eines $2k$ langen

Wertes, dann ist die Hashfunktion folgendermaßen definiert:

$$\text{forwardshift}(x) = \text{lower}_\text{k}(x) \times 2^k + \text{xor}(\text{lower}_\text{k}(x), \text{upper}_\text{k}(x)) \tag{6.2a}$$

$$\text{forward-hash}(key) = a \times \text{forwardshift}(key) \bmod 4^k \tag{6.2b}$$

$$\text{reverseshift}(x) = \text{xor}(\text{lower}_\text{k}(x), \text{upper}_\text{k}(x)) \times 2^k + \text{lower}_\text{k}(x) \tag{6.2c}$$

$$\text{reverse-hash}(x) = \text{reverseshift}(a' \times x \bmod 4^k) \tag{6.2d}$$

6.3 Value Module

Je nach Anwendungsfall können die Anforderungen an den zu speichernden Werten in der Hashtabelle variieren. Um eine generische Hashtabelle, die von der spezifischen Nutzung entkoppelt ist, zu gewährleisten, kommen *Value Module* zur Berechnung der Werte zum Einsatz. In der Konfiguration der Hashtabelle können somit *Value Module* angegeben werden, um die Hashtabelle für die intendierte Nutzung zu spezifizieren.

Im Folgenden wird auf die Implementierung von *Value Module* als solches eingegangen. Darüber hinaus werden vordefinierte Value Module vorgestellt, die die Hashtabelle von Haus aus bereitstellt.

Für die Umsetzung der *Value Module* wurde die abstrakte Klasse `ValueModuleBase` implementiert. Diese implementiert die abstrakten Methoden `make_first_insert`, `make_update_value`, `make_map_value` und `value_bits`. Diese Methoden wiederum müssen von den konkreten *Value Module* implementiert werden, wobei die Umsetzung jeweils Detail des *Value Module* selbst ist. `make_first_insert` erzeugt die Funktion, die aufgerufen wird, wenn ein Schlüssel zum ersten Mal eingefügt wird. `make_update_value` erzeugt die Funktion, die aufgerufen wird, wenn ein bereits vorhandener Schlüssel weitere Male eingefügt bzw. geändert wird. `make_map_value` erzeugt die Funktion, die verwendet wird, um einen gespeicherten Wert zu einem gesuchten Schlüssel zuzuordnen. `value_bits` gibt die Anzahl der Bits zurück, die zum Speichern von Werten verwendet werden. Der Rückgabewert ist bei dieser Methode fest als `int` definiert.

6.3.1 Vordefinierte Value Module

Die Gecocoo-Hashtabelle liefert vordefinierte *Value Module* zur Verfügung. Diese sollen zum einen fertige Implementierungen für gängige Anwendungsfälle von Hashtabellen bereitstellen als auch als Vorlage für die Implementierung benutzerdefinierter *Value Module* dienen. Die Hashtabelle stellt dazu die drei *Value Module counting*, *keyset* und *store* bereit. Bei den Funktionen `first_insert`, `update_value` und `map_value` ist auf die Nutzung des `@njit`-Dekorators zu achten. Nur dadurch lassen sich alle Hashtabellenoperationen später JIT-kompilieren.

`counting` zählt die in die Hashtabelle eingefügten Schlüssel. Hierzu werden die Parameter `bits`, `initial_count` und `increment` angegeben. Der erste Parameter gibt die Anzahl der Bits, die zum Speichern des aktuellen Zählerstands verwendet werden sollen, an. Die Anzahl sollte vom erwarteten Maximalwert abhängen (Standard: 32). `initial_count` stellt den Wert für das erste Einfügen dar (Standard: 1). `increment` ist eine Ganzzahl zur Erhöhung des aktuellen Wertes, also die Schrittweite (Standard: 1)).

Die Funktion `first_insert` gibt entsprechend den `initial_count`, `update_value` den aktuellen Wert addiert mit dem Inkrement und `map_value` den aktuellen Wert zurück.

`store` speichert zu jedem Schlüssel eine festgelegte Anzahl an *bits*. Die Funktion erhält als Parameter die Anzahl der *bits*, die maximal gespeichert werden sollen (Standard: 64).

Die Funktionen `first_insert` und `update_value` geben entsprechend den zu speichernden Wert zurück, wobei `map_value` den zum Schlüssel zugehörigen Wert zurückgibt.

`keyset` nutzt die Hashtabelle als simples *Set*, wobei lediglich der Schlüssel ohne zugehörigen Wert gespeichert wird. Die Rückgabewerte der Funktionen `first_insert` und `update_value` sind somit nicht relevant und geben eine 0 zurück. `map_value` gibt $True$ zurück, wenn der Schlüssel in der Hashtabelle vorhanden ist.

Neben den hier vorgestellten vordefinierten *Value Modules* steht es dem Nutzer frei, benutzerdefinierte *Value Module* zu nutzen. Die Konfiguration dieser *Value Module* ist im Abschnitt 5.4 näher beschrieben.

6.4 Speicherabstraktion

Die Daten der Hashtabelle werden in einem *NumPy* Array kodiert. Bei der Umsetzung der Codierung gibt es allerdings viele Parameter zu berücksichtigen. Hauptsächlich muss zwischen den Kriterien Zugriffsgeschwindigkeit und Speicherbedarf abgewogen werden. Von der Tabelle werden daher drei verschiedene Speichermodi unterstützt, um möglichst jeden Anwendungsfall bedienen zu können:

1. `Fast`: In diesem Modus belegt jeder Slot der Tabelle 128 Bit (64 Bit Wert, 62 Bit Schlüssel ohne Quotienting, 2 Flags), was den Zugriff vereinfacht.

2. `Buckets`: In diesem Modus werden die Einträge eines *buckets* so dicht wie möglich gepackt und das Ende der *buckets* wird auf eine 64 Bit Grenze aufgefüllt.

3. `Packed`: Alle Einträge sind dicht gepackt.

Außerdem können weitere Zusatzoptionen aktiviert werden, welche atomare Operationen fordern oder die Codierung leerer Felder ändern. Um die Einfügeoperationen unabhängig vom Speichermodus und Zusatzoptionen zu machen, müssen alle Speichermodi einen Satz an Zugriffsfunktionen implementieren, welcher im Folgenden beschrieben wird. Anschließend wird auf die Atomizität der Zugriffe und auf Implementierungsdetails der Modi eingegangen.

6.4.1 Funktionen

Alle drei Speichermodi implementieren folgende Zugriffsfunktionen:

1. `set_empty_entry`, um einen leeren Eintrag zu überschreiben

2. `clear_entry`, um einen Eintrag zu leeren

3. `cmp_swap_entry`, um die Schlüssel und Werte eines nicht-leeren Eintrages zu ändern. Vorausgesetzt dieser wird nicht relocated.

4. `get_entry`, um den Schlüssel, Wert und Meta-Information des Eintrags zu erhalten.

5. `mark_relocating_entry`, um die Relocation-Markierung umzuschalten.

6.4.2 Implementierungen des Zugriffs

Alle Modi haben gemeinsam, dass sie mehrere vorgegebene Felder verschiedener Länge in mehrere 64 Bit Blöcke aufteilen müssen. Dazu wird bei der Generierung der Zugriffsfunktionen eine Liste der Felder entgegengenommen. Im `Fast` Modus können diese Werte einfach aufsummiert werden, um die Offsets der einzelnen Werte zu erhalten. Beim Überschreiten der 64 Bit muss eines der Felder durch gezielte Bitshifts aufgeteilt werden. Im `Buckets` Modus hingegen können die Offsets der Felder verschieden sein. Allerdings können sie dennoch vorgeneriert werden, da sie zumindest für einen Index innerhalb eines *buckets* konstant sind. Beim `Packed` Modus kann diese Annahme nicht getroffen werden, sodass alle Offsets zur Laufzeit ermittelt werden müssen.

6.4.3 Concurrency

Um mit mehreren Threads gleichzeitig auf die Tabelle zugreifen zu können, muss, unabhängig vom Einfügealgorithmus, sichergestellt werden, dass ein Eintrag immer ganz oder gar nicht geschrieben wird, sodass Schreibzugriffe atomar sind. Zu einem Eintrag gehören in der Regel entweder der Schlüssel oder dessen Quotient und ein Wert, welche zusammen länger als 64 sein können. Daher ist es nicht in allen Fällen trivial, atomare Zugriffe zu garantieren.

Im `Fast` Modus kann ein Zugriff auf die Tabelle mit atomaren Instruktionen wie `CMPXCHG16B` erfolgen, da in diesem Modus alle Einträge eine feste Länge von 128 Bits besitzen (64 Bit Wert, 62 Bit Schlüssel, 2 Flags) und an den entsprechenden Blockgrenzen im Speicher liegen. Im `Buckets`-Modus (mit aktiviertem *Concurency Support*) werden Einträge unter 128 Bit so aufgefüllt, dass sie vollständig innerhalb eines adressierbaren 64 oder 128 Bit Blocks liegen, was ähnlich wie bei `Fast` den Einsatz von atomaren Instruktionen erlaubt. Bei längeren Einträgen oder im `Packed`-Modus kann es vorkommen, dass ein Eintrag über drei 64 Bit Blöcke verteilt ist, was einen Austausch mit einer atomaren Instruktion ausschließt. Für diese Modi wird daher ein Feld mit Lock-Bits verwendet, um gleichzeitige Zugriffe zu verhindern. Dabei teilen sich die mehrere *buckets* einzelne Lock-Bits.

Dies umfasst nur die notwendigen Maßnahmen auf Ebene des Speicherzugriffs. Zusätzlich muss der Einfügealgorithmus noch dafür sorgen, dass beim Verschieben der Einträge keine Wettlaufsituationen entstehen. Dies wird in Abschnitt 6.5.1 behandelt.

6.5 Hashtabellenoperationen

Auf der Hashtabelle sind verschiedene Operationen definiert, die es ermöglichen, Daten der Hashtabelle hinzuzufügen, die Daten in der Hashtabelle abzufragen, zu aktualisieren und zu löschen. Die Operationen auf der Hashtabelle können über das `HashtableInterface` aufgerufen werden.

Mit der `insert_single` Funktion werden einzelne Werte mit der Random Walk Strategie der Hashtabelle hinzugefügt oder die Werte in der Hashtabelle werden aktualisiert, falls der Schlüssel bereits gespeichert ist.

Die `initialize` Funktion fügt mehrere Werte in eine leere Hashtabelle ein und verwendet dabei die kostenoptimale Einfügestrategie.

Mit den Funktionen `search_single` und `remove_single` können einzelne Werte der Hashtabelle über den Schlüssel abgefragt bzw. gelöscht werden.

6.5.1 Random Walk

Die Funktion `insert_single` verwendet die Random Walk Strategie, um einzelne Werte der Hashtabelle hinzuzufügen. Aufrufe der Funktion können parallel erfolgen, sodass beim

Einfügen parallele Veränderungen der Hashtabelle behandelt werden müssen. Grundsätzlich sind alle Speicherzugriffe atomar, sodass hauptsächlich das gleichzeitige Verschieben von Werten beachtet werden muss.

Als Grundlage dazu dient das Verfahren von Nguyen und Tsigas [19], welches Markierungen (genannt relocation) nutzt, um die verschobenen Elemente vor Lesen und Schreiben zu sichern. Außerdem nutzt es Zähler, um einer Suche zu ermöglichen, eine gleichzeitige Verschiebung zu erkennen. In der Arbeit beweisen die Autoren, dass ihr Verfahren eine Linearisierung aller Operationen ermöglicht.

In diese Arbeit werden Zähler aus Platzgründen jedoch nicht umgesetzt, was dazu führen kann, dass eine Suche das Ergebnis fälschlicherweise nicht findet. Dies tritt auf, wenn ein Eintrag während einer Suche (oder beim Löschen) von einem noch nicht abgesuchten Slot auf einen schon besuchten Slot verschoben wird und keine Markierung mehr gefunden wird, welche die Suche neu starten würde. Im Hinblick auf den Verwendungszweck (erst alle k-mere einfügen, dann suchen) in Verbindung mit dem geringen Risiko (Suche muss langsamer als Einfügen sein) des Eintretens ist abzusehen, dass die Ersparnis wichtiger ist. Das Einfügen hingegen ist gegen vergleichbare Probleme geschützt, wie im Folgenden beschrieben wird.

Da für das Einfügen einzelner Werte und das Aktualisieren dieser nur die Random Walk Strategie verwendet wird, ist die Funktion `insert_single` ein Synonym für `insert_random_walk`. Die Funktion `insert_random_walk` lässt sich grundsätzlich in drei Schritte unterteilen.

1. **Schlüssel aktualisieren**. Zuerst wird überprüft, ob der Schlüssel bereits in der Tabelle gespeichert ist. Hierbei werden die Buckets die durch die verschiedenen Hashfunktionen bestimmt werden, nach dem Schlüssel durchsucht. Konnte der Schlüssel gefunden werden, so wird der neue zu speichernde Wert berechnet und dann wird dieser aktualisiert. Diese Abfragen sind nur dann erfolgreich, wenn kein betrachteter Wert in der Hashtabelle gerade verschoben wurde. Ansonsten wird dieser Schritt wiederholt.

2. **Schlüssel einfügen**. Konnte der Schlüssel nicht in der Tabelle gefunden werden, so wird eine freie Position in der Tabelle gesucht, um den Schlüssel einzufügen. Falls eine freie Position gefunden wurde, wird der Eintrag an die Stelle geschrieben. Danach werden alle anderen möglichen Plätze für den Schlüssel überprüft, um festzustellen, ob parallel der gleiche Wert in die Tabelle eingefügt wurde. Sollte dies der Fall sein, so muss der gerade eingefügte Wert wieder entfernt werden und das Einfügen wird neu gestartet.

3. **Schlüssel verschieben**. Existiert keine freie Position in der Tabelle, werden Einträge in der Tabelle verschoben, sodass eine freie Position für den Schlüssel entsteht. Dies geschieht, indem zuerst iterativ eine Reihe von Verschiebungen gesucht wird, sodass bei der letzten Verschiebung der Schlüssel auf eine freie Position verschoben wird. Wenn diese Verschiebungen gefunden wurden, werden die Schlüssel verschoben und das Einfügen startet erneut, allerdings existiert jetzt eine freie Position für den einzufügenden Schlüssel.

Schlüssel Aktualisieren

Die in Listing 6.2 gezeigte Funktion `insert_into_existing_slot` aktualisiert den Wert für einen bereits existierenden Schlüssel. Dafür wird zunächst nach einem vorhandenen Eintrag mit der Funktion `find_by_key` gesucht. Wurde ein entsprechender Eintrag gefunden, wird der neue Wert durch die Funktion `update_value` vom *Value Module* ermittelt

```
def insert_random_walk(key, ...):
    while True:
        # 1. Schlüssel Aktualisieren
        if insert_into_existing_slot(key, ...):
            return False

        # 2. Schlüssel Einfügen
        empty_slot_found, insert_success = insert_into_empty_slot(key, ...)

        if insert_success:
            increase_key_count(...)
            return True

        # 3. Schlüssel Verschieben
        if not empty_slot_found:
            path_found, path_length = \
                find_relocation_path(relocation_path, key, ...)

        if not path_found:
            raise Exception("failed to find replacement path")

        execute_replacement_path(relocation_path, path_length, ...)
```

Listing 6.1: Implementierung Einfügen Random Walk

und mittels `compare_swap_entry` aktualisiert. Wurde der Eintrag erfolgreich aktualisiert, gibt die Funktion `True` zurück. Die Funktion `find_by_key` gibt als Rückgabewert boolsche Werte, ob ein Eintrag gefunden wurde und dieser sich in einem relocation Zustand befinde, als auch Informationen über den Eintrag selbst zurück. Wurde ein entsprechender Eintrag gefunden, der sich nicht im relocation Zustand befindet, wird er wie eingangs beschrieben aktualisiert. Wurde jedoch ein Eintrag gefunden, der sich im relocation Zustand befindet, wird der eingangs beschriebene Prozess in einer Schleife so lange durchlaufen, bis der Eintrag wieder verfügbar ist. Somit wird aktiv auf den Eintrag gewartet.

Schlüssel Einfügen

Die Funktion `insert_into_empty_slot` in Listing 6.3 ermittelt einen freien Slot und fügt den Eintrag ein. Rückgabe sind zwei boolsche Werte die anzeigen, ob ein freier Slot gefunden wurde und dieser erfolgreich befüllt wurde. Zunächst wird nach einem freien Slot gesucht. Die Funktion `find_empty_slot_for_key` gibt als Rückgabewert einen Boolean, der angibt, ob ein Eintrag gefunden wurde und den Eintrag als solchen zurück. Wurde kein freier Slot gefunden, wird `(False, False)` zurückgegeben und der Schritt abgebrochen.

Wurde hingegen ein freier Slot gefunden, wird mittels `first_insert` aus dem *Value Module* der Wert für das erste Einfügen ermittelt. Anschließend wird der Wert eingefügt. Sollte diese Operation fehlschlagen, wird auch hier `(False, False)` zurückgegeben, da der Slot in der Zwischenzeit belegt worden sein muss.

Bei erfolgreichem Einfügen wird die Hashtabelle auf Duplikate des Schlüssels überprüft. Dies könnte der Fall sein, wenn dieser parallel eingefügt wurde. Beim Finden eines oder mehrerer Duplikate wird der Wert wieder aus der Hashtabelle entfernt und `(True, False)` zurückgegeben. Der Wert ist in der Zwischenzeit als relocation markiert und ist damit

```
def insert_into_existing_slot(key, ...):
    found = False
    relocating = True
    while found or relocating:
        (found, relocating, index, slot, old_value, ...) = \
            find_by_key(key, ...)

        if found:
            new_value = update_value(key, old_value,...)

        if compare_swap_entry(index, slot, new_value, ...):
            return True

    return False
```

Listing 6.2: Implementierung Schlüssel Aktualisieren

ungültig. Dies garantiert einen konsistenten Zustand der Tabelle, indem die betroffenen Einfügeoperationen wiederholt werden, bis alle Aktualisierungen in einer seriellen Folge durchgeführt wurden.

Sollten keine Duplikate vorhanden sein, war das Einfügen des Schlüssels erfolgreich und es wird `(True, True)` zurückgegeben.

```
def insert_into_empty_slot(key, ...): # -> empty_slot_found, insert_success
    found, index, slot = find_empty_slot_for_key(key, ...)

    if not found:
        return False, False
    else:
        value = first_insert(key, ...)
        if not set_empty_entry(key, value, index, slot, ...):
            return False, False

        if check_for_duplicates_and_remove_if_present(key):
            return True, False

    return True, True
```

Listing 6.3: Implementierung Schlüssel Einfügen

Schlüssel Verschieben

Wenn keine freie Position in der Hashtabelle gefunden wurde, werden Schlüssel verschoben, um eine freie Position zu schaffen. Dies geschieht in zwei Schritten. Zuerst wird eine Reihe von Verschiebungen berechnet, welche eine freie Position für den einzufügenden Schlüssel schaffen würde. Der Pfad der Verschiebungen wird dabei markiert, um Concurrency Probleme zu vermeiden, indem markierte Slots vor ihrer Verschiebung nicht gelesen oder verändert werden. Die entsprechenden Suchen und Einfügeoperationen in bestehende Slots warten aktiv, bis die Markierung auf ihren möglichen Slots aufgehoben wird. Zuletzt werde diese Verschiebungen ausgeführt und Markierungen gelöscht.

Verschiebungen berechnen Um eine mögliche Verschiebung zu berechnen, wird zuerst Speicher allokiert, damit die Verschiebungen gespeichert werden können. Es werden maximal 1000 Verschiebungen durchgeführt. Die Berechnung beginnt mit dem einzufügenden Schlüssel. Nun wird für den Schlüssel eine freie Position gesucht. Kann keiner gefunden werden, wird eine zufällige Position gewählt, an die der Schlüssel geschrieben werden kann. Diese Positionen werden durch die Hashfunktionen bestimmt. Nun wird entweder die freie oder die zufällige Position gespeichert. War die Position frei, so wird der Pfad beendet. Ansonsten wird nun der Schlüssel betrachtet, der an der zufälligen Position gespeichert ist. Für diesen wird nun eine neue Position gesucht. Falls nach 1000 Verschiebungen keine freie Position gefunden wurde, schlägt das Einfügen fehl und es wird ein Fehler ausgegeben.

```
1  def find_relocation_path(path, key, ...): # -> path_found, path_length
2      for i in range(len(path)):
3          # First try finding an empty slot
4          (found_empty, relocate_index, relocate_slot) = \
5              find_empty_slot_for_key(key, ...)
6          occupied = False
7
8          if not found_empty:
9              (occupied, relocate_index, relocate_slot, relocate_key) = \
10                 find_random_slot_for_key(key)
11
12         path[i] = create_path_entry(relocate_index, relocate_slot, ...)
13
14         if not occupied:
15             return True, i + 1
16         else:
17             key = relocate_key
18
19     return False, 0
```

Listing 6.4: Implementierung Verschiebungspfad finden

Verschiebungen durchführen Wurde ein Pfad zur Verschiebung gefunden, können die Elemente des Pfads rückwärts um eine Position verschoben werden. Das heißt, zuerst wird der Wert, der in der Hashtabelle an der Position des vorletzten Elements im Pfad steht, an die Position vom letzten Element verschoben. An der Position des letzten Elements ist in der Hashtabelle kein Element gespeichert. So wird die Position vom vorletzten Element frei. Dieses Vorgehen wird für die anderen Elemente im Pfad wiederholt, bis das erste Element im Pfad verschoben wird. Wenn dieses Verschieben erfolgreich war, existiert ein freier Platz in der Hashtabelle, wo der einzufügende Schlüssel eingefügt werden kann. Wichtig ist, dass sich die Elemente auf dem Pfad nicht verändert haben. Sollte dies passiert sein, muss ein neuer Pfad gefunden werden. Falls ein Element auf dem Pfad nicht mehr existiert und der Platz in der Tabelle frei ist, können die Verschiebungen weiter durchgeführt werden, da für das nächste Element auf dem Pfad wieder ein freier Platz existiert. Wenn alle Verschiebungen abgeschlossen sind, wird erneut versucht, den Schlüssel einzufügen.

Ein einzelner Schritt bei der Ausführung des Verschiebepfads besteht grundsätzlich daraus, dass ein Schlüssel zunächst an seine neue Position kopiert wird und anschließend an der alten Position gelöscht wird. Ein Schlüssel wird also niemals gänzlich aus der Tabelle entfernt, sondern kann sich nur für kurze Zeit doppelt in der Tabelle befinden. Dadurch

ist sichergestellt, dass ein gleichzeitiger Einfügeprozess den Schlüssel nicht erneut einfügt, sondern mindestens einen der existierenden Schlüssel finden kann. Dabei können einige weitere Wettlaufsituationen auftreten, die von dem Verfahren entweder verhindert werden müssen oder zu einem Fehlschlag der Verschiebung führen. Bei einem Fehlschlag kann die Einfügeprozedur erneut ausgeführt werden. Die Konsistenz der Tabelle bleibt jedoch erhalten. Die Situationen lauten wie folgt:

1. Der Schlüssel an der alten Position wurde durch einen anderen ausgetauscht.

2. Der Schlüssel an der alten Position wurde aktualisiert.

3. Die Zielposition des Schlüssels ist bereits durch einen anderen Schlüssel belegt.

Die Situationen 1 und 2 werden verhindert, indem der Schlüssel an der Ursprungsposition als *relocation* markiert wird. Damit ist sichergestellt, dass der Schlüssel nicht aktualisiert wird und sich auch der anhand des Verschiebepfads erwartete Schlüssel dort befindet. Als nächstes wird der Schlüssel an die Zielposition kopiert. Die Operation schlägt fehl, wenn der Platz bereits belegt ist (Situation 3). Im Erfolgsfall muss lediglich der Schlüssel an der alten Position entfernt werden. Bei einem Fehlschlag wird die Markierung als *relocation* vom Schlüssel an der Ursprungsposition wieder aufgehoben. Die Annahme, dass die Zielposition leer ist, ist verletzt worden und somit ist der Verschiebepfad nun ungültig. In diesem Fall muss die Einfügeprozedur wiederholt werden.

```
def execute_replacement_path(path, path_length, ...): # -> success
    for i in range(1, path_length):
        target_path_entry = path[path_length - i]
        source_path_entry = path[path_length - i - 1]

        (occupied, ...) = get_entry(source_path_entry, ...)

        if not occupied:
            continue

        if not mark_relocating_entry(source_path_entry, True, ...):
            return False

        relocate_key = source_path_entry.key

        if not set_empty_entry(target_path_entry.location,
        source_path_entry.value, ...):
            while not mark_relocating_entry(source_path_entry, False, ...):
                pass

            return False

        while not clear_entry(source_path_entry.location):
            pass
    return True
```

Listing 6.5: Implementierung Verschiebungen ausführen

6.5.2 Cost Optimal

Im Folgenden wird die Implementierung der kostenoptimalen Einfügestrategie für das Projekt diskutiert, deren theoretische Grundlage im Grundlagen-Kaptiel vorgestellt wurde.

Bei der Implementierung haben wir uns weitestgehend an der im Paper [33] beschriebenen Strategie orientiert, welche ohne weitere Kompression eine bessere Performance, dank Cachefreundlichkeit, bei höherem Speicherverbrauch ermöglicht. Weiterhin haben wir uns dagegen entschieden, den Algorithmus so zu implementieren, dass er für eine nicht leere Tabelle richtig arbeitet, da dies lediglich, wie im Paper beschrieben, Performance-Nachteile mit sich bringt und mit dem gleichen Speicherverbrauch einhergeht. Im Folgenden sollen die einzelnen Schritte des Algorithmus diesbezüglich konkreter erläutert werden:

Datenstrukturen und Vorbereitung

Da der Algorithmus für große Datenmengen vorgesehen ist und somit möglichst schneller Random-Access sehr von Vorteil ist, wurden alle Hilfsinformationen zur Berechnung der optimalen Belegung in *NumPy* Arrays abgelegt. Da die darin zu speichernden Daten stets aus einer bekannten Menge stammen, kann der Speicher darin verdichtet genutzt werden, indem die Einträge nicht etwa 32 oder 64 Bit einnehmen, sondern nur die exakt benötigte Menge an Bits. Die im Paper[33] beschriebenen benötigten Arrays, welche alle bei der Implementierung auf diese Weise umgesetzt wurden, sind:

- Ein Assignment-Array, das die Hash-Wahl für jeden Schlüssel speichert. Dementsprechend braucht es so viele Einträge, wie es Schlüssel gibt und $\log_2$ von der Zahl der Hashfunktionen plus 1 große Einträge.
- Ein Bucket-Füllstand Array, das speichert, wie viele Assignments jeden Bucket zum Ziel haben. Dies wird benötigt, um während des Algorithmus herauszufinden, ob der Bucket schon voll ist. Dementsprechend braucht es so viele Einträge, wie es Buckets gibt und $\log_2$ von der Zahl der Bucket-Größe plus 1 große Einträge.
- Drei Arrays für die ausgehenden Kanten aus jedem Bucket. Die ersten beiden speichern jeweils den Schlüssel und den Hash-Index, welcher auf den Bucket verweist. Das Dritte speichert den Start-Index für den jeweiligen Bucket. Dementsprechend haben die ersten beiden Arrays so viele Einträge wie das Produkt aus der Schlüsselanzahl und der Hashfunktionsanzahl. Das Dritte hat so viele Elemente wie es Buckets gibt plus eins, da für jeden Bucket der Startindex am Index b im dritten Array steht und der Endindex am Index $b+1$ und davon 1 abgezogen werden muss. Das Schlüssel-Array braucht pro Eintrag $\log_2$ von der Anzahl der Schlüssel Bits. Das Hash-Index-Array braucht pro Eintrag $\log_2$ von der Anzahl der Hashfunktionen Bits. Das Startindex-Array braucht schließlich $\log_2$ vom Produkt der Hashfunktionsanzahl und der Schlüssel Anzahl Bits.
- Ein Bucket-Kosten Array, das speichert, wie groß die Kosten sind, um einen Schlüssel über einen Bucket zum nächstgelegenen freien Bucket zu verschieben. Dieses Array braucht so viele Einträge, wie es Buckets gibt. Die benötigte Größe der Einträge ist nicht wirklich vorhersehbar und hängt von der Tabelle und der Länge der sich bildenden Pfade ab. Für unsere Implementierung haben wir uns an die im Paper[33] empfohlenen 16 Bits für einen großen Puffer und Bitshift-Freie Zugriffs-Operationen entschieden.
- Ein Vorgänger-Element Array, das speichert, welches Element einem Bucket auf einem Pfad vorausgeht. Dementsprechend werden so viele Einträge wie Buckets benötigt. Jeder Eintrag nimmt $\log_2$ von der Zahl der Schlüssel plus 1 Bits ein.

- Ein Vorgänger-Bucket Array, das speichert, welcher Bucket einem Bucket auf einem Pfad vorausgeht. Dementsprechend werden so viele Einträge wie Buckets benötigt. Jeder Eintrag nimmt $\log_2$ von der Zahl der Hashfunktionen plus 1 Bits ein, da Buckets über die Hashfunktion und das Element ermittelt werden können.
- Ein Flag-Array mit je einem Bit pro Eintrag. Dieses wird verwendet, um sich in der Suchphase zu merken, welche Buckets noch aktiv sind, und in der Bewegungsphase, um sich zu merken, welche Elemente bewegt wurden. Dementsprechend muss es, je nachdem welche Zahl größer ist, entweder so viele Einträge haben, wie es Buckets oder Schlüssel gibt.

Nachdem die notwendigen Datenstrukturen angefordert wurden, wird das Assignment-Array mit ungültigen Assignments befüllt und der Bucket-Füllstand auf 0 gesetzt. Anschließend werden die Arrays für die ausgehenden Kanten initialisiert. Da jeder Bucket theoretisch 0 oder alle Schlüssel auf sich abgebildet haben könnte und die Kanten nur ermittelt werden können, indem von den Schlüsseln aus berechnet wird, auf welche Buckets sie abgebildet werden können, muss zunächst gezählt werden, wie viele Schlüssel-Hash-Kombinationen auf welchen Bucket abgebildet werden. Dazu wird über alle Schlüssel iteriert und für jede Hashfunktion für diesen der Bucket ermittelt und dann für diesen Bucket die Anzahl um 1 erhöht. Anschließend werden auf Basis der Anzahlen die Startindizes ermittelt. Dann wird der Durchlauf über alle Schlüssel und Hashfunktionen wiederholt und die Schlüssel und Hashfunktionen in die entsprechenden Arrays für ausgehende Kanten eingetragen. Es wird ein separates Hilfsarray verwendet, um zu zählen, wie viele Schlüssel-Hash-Daten bereits für alle Buckets eingetragen wurden, damit stets an eine freie Stelle geschrieben wird. Die vier verbleibenden Arrays werden lediglich für die Standarddurchläufe benötigt und dort näher erläutert.

Initialisierungsdurchläufe

Für die Initialisierungsdurchläufe wird über alle Schlüssel iteriert und für sie der Bucket für die erste Hashfunktion berechnet. Um zu überprüfen, ob der Bucket bereits voll ist, wird bei jeder erfolgreichen Zuweisung der Füllstand im entsprechenden Array um 1 erhöht und vorher verglichen, ob der Bucket noch nicht voll ist. Ist dies nicht der Fall, so wird 1, bzw. im zweiten Durchlauf 2, für die entsprechenden Schlüssel im Assignment-Array eingetragen. Außerdem wird gezählt, wie viele Schlüssel erfolgreich eine Zuweisung erhalten haben. Falls alle Schlüssel am Ende der Initialisierungsdurchläufe ein Assignment erhalten haben, sind Standarddurchläufe nicht vonnöten und die Schlüssel werden entsprechend der Assignments eingefügt.

Standarddurchläufe

Die Standarddurchläufe bestehen aus der Such- und der Bewegungsphase, welche wiederholt werden, bis alle Schlüssel ein Assignment haben.

Als Vorbereitung auf die Suchphase werden erst alle Bucket-Kosten auf den Maximalwert gesetzt, außer sie sind offene Buckets, also solche, die noch weniger Assignments als Plätze haben. Außerdem werden alle Buckets, die offen sind, als aktiv eingestuft und die restlichen Buckets als inaktiv. Die Pfade werden darüber hinaus auch zurückgesetzt, indem alle Vorgänger Schlüssel und Buckets auf ungültige Elemente gesetzt werden. Damit beginnt die Suchphase. In dieser werden so oft Iterationen über alle Buckets ausgeführt, bis keine neuen aktiven Buckets gefunden werden. Für jeden Bucket wird dabei geprüft, ob er aktiv ist und falls ja, werden für alle ausgehenden Kanten die Buckets ermittelt, in denen die Schlüssel evtl. gespeichert werden. Dort wird überprüft, ob der Pfad vom aktuellen Bucket günstiger ist und falls ja, wird der neue Bucket aktiv geschaltet, seine Kosten

aktualisiert und der aktuelle Bucket zu seinem Vorgänger. Am Ende der Suchphase sind dann alle Pfade bekannt.

Als Vorbereitung auf die Bewegungsphase werden alle Schlüssel als unbewegt markiert. Dann wird über alle Schlüssel ohne Assignment iteriert und überprüft, ob es für sie einen Pfad zu einem offenen Bucket gibt, also ob sie einen Vorgänger haben. Falls nicht, existiert kein Assignment, welches alle Schlüssel in der Tabelle unterbringen kann. Anschließend werden die Pfade bis zum offenen Bucket zurückverfolgt und geprüft, ob keine Elemente auf dem Weg schon bewegt wurden. Falls schon, wird der Schlüssel diesen Durchlauf nicht eingefügt. Sonst wird er entlang des Pfades verschoben.

Einfügen auf Basis der Assignments

Sobald ein gültiges Assignment ermittelt wurde, also entweder nach den Initialiserungsdurchläufen, falls dann alle Schlüssel ein Assignment haben, oder nachdem alle Schlüssel ein Assignment über die Standarddurchläufe erhalten haben, werden die Schlüssel an die entsprechenden Stellen in der Tabelle geschrieben. Dabei wird über alle Schlüssel iteriert und ihr Assignment aus dem Assignment-Array ausgelesen, dann die entsprechende Hashfunktion für den Schlüssel aufgerufen und anschließend der Schlüssel mitsamt des Wertes an die erste freie Stelle im berechneten Bucket eingefügt.

Speicherbedarf

Der Speicherbedarf der kostenoptimalen Einfügestrategie lässt sich mit einer Formel, die sich aus dem Paper[33] herleitet, berechnen. Der Bedarf hängt dabei von vier Parametern ab:

- der Anzahl der Hashfunktionen h
- der Anzahl an Slots in jedem Bucket b
- der Anzahl der Buckets p
- der Anzahl der einzufügenden Schlüssel n

Der Speicherbedarf resultiert aus den zusätzlichen Datenstrukturen, welche für die Berechnung der Pfade, entlang denen die Belegungen optimiert werden, benötigt werden. Auf diese wird konkret im Abschnitt 6.5.2 eingegangen. Es ergibt sich folgende Formel für die Anzahl der Bits, die in Anspruch genommen wird, wobei der Speicherverbrauch jedes individuellen verwendeten Arrays auf ein Vielfaches von 64 Bit aufgerundet wird.

$$
\begin{aligned}
&p \cdot (\lceil \log_2(b+1) \rceil + \lceil \log_2(n) \cdot \log_2(h) \rceil + \lceil \log_2(n+1) \rceil + \lceil \log_2(h+1) \rceil + \lceil 16 \rceil) + \\
&n \cdot (\lceil \log_2(h+1) \rceil) + \\
&n \cdot h \cdot (\lceil \log_2(n) \rceil + \lceil \log_2(h) \rceil) + \\
&\max\{n, p\}
\end{aligned}
$$

Der Speicherverbrauch wächst also hauptsächlich linear mit der Bucketzahl und Schlüsselzahl. Da die Zahl der Schlüssel kein Konfigurationsparameter der Hashtabelle ist, sollen die verbleibenden Parameter im Folgenden diskutiert werden. Da der Bedarf nur logarithmisch mit der Zahl der Slots pro Bucket wächst, führen größere Buckets bei gleicher Tabellengröße zu einem geringeren Speicherbedarf. Auch zu berücksichtigen ist, dass jede eingesetzte Hashfunktion als linearer Faktor auf die Zahl der Schlüssel multipliziert wird und so recht signifikant den Speicherverbrauch erhöhen kann.

6.5.3 Suchen und Löschen

Neben dem Einfügen von Werten in die Hashtabelle werden ebenfalls die Funktionalitäten zum Suchen und Löschen von Werten bereitgestellt. Beim Suchen von Werten in der Hashtabelle erfolgt die Abfrage über den Schlüssel. Die Hashtabelle liefert dann den gespeicherten Wert für diesen Schlüssel zurück, falls einer existiert. Hierbei hat das Value Module der Tabelle noch die Möglichkeit, den tatsächlich in der Tabelle gespeicherten Wert auf einen anderen Wert abzubilden. Bei der Suche von einem Schlüssel in der Hashtabelle werden nacheinander die verschiedenen Buckets, in denen sich der Schlüssel befinden kann, betrachtet. Hierzu werden die Hashfunktionen der Tabelle aufgerufen, um den Bucket zu finden, und dann wird für alle Einträge im Bucket überprüft, ob diese den gesuchten Schlüssel enthalten. Falls dies der Fall ist, kann der gespeicherte Wert direkt zurückgegeben werden und die Suche ist beendet. Andernfalls müssen die anderen Buckets weiter durchsucht werden.

Während der Suche in der Hashtabelle muss beachtet werden, dass die betrachteten Werte nicht Teil einer Verschiebung sind. Falls ein betrachteter Eintrag in der Hashtabelle zurzeit verschoben wird, muss die Hashtabelle die Suchoperation für den Schlüssel wiederholen. Es besteht die Möglichkeit, dass der gesuchte Schlüssel nicht gefunden wurde, weil er gerade verschoben wurde. Nur wenn alle möglichen Buckets und alle Einträge in den Buckets nicht den Schlüssel enthalten und keiner der Einträge verschoben wird, enthält die Tabelle den gesuchten Schlüssel nicht. Weil diese Überprüfung allerdings nicht atomar erfolgt kann es zu einer Wettlaufsituation kommen, in welcher der gesamte Vorgang von Markierung bis zur Verschiebung ausgeführt wird während die Suchoperation grade an anderer Stelle sucht und der Eintrag so überlesen wird. Aufgrund der langen Laufzeit der Verschiebung ist dies allerdings unwahrscheinlich. Trotzdem sollten Suchen für wichtige Analysen erst nach fertiger Füllung der Tabelle erfolgen.

Das Löschen verläuft äquivalent zum Suchen in der Tabelle. Der einzige Unterschied besteht darin, dass wenn der Schlüssel gefunden wird, dieser ebenfalls aus der Tabelle entfernt wird. Auch beim Löschen wird das Verschieben von Einträgen in der Tabelle beachtet und analog zu der Suche behandelt.

6.6 Serialisierung

Eine wichtige Funktionalität der Hashtabelle besteht darin, dass diese dauerhaft gespeichert werden kann. So kann die Hashtabelle später als Index verwendet werden und muss nicht erneut generiert werden. Bei der Speicherung ist es wichtig, dass nicht nur die Daten in der Tabelle gespeichert werden, sondern auch die Zugriffsfunktionen der Hashtabelle.

Zum Speichern der Tabelle werden mehrere Arrays mit der *NumPy* Funktion `savez` oder `savez_compressed` gespeichert. Neben den Daten der Tabelle, welche schon in einem Array gespeichert werden, wird zudem das `LLConfig`-Objekt gespeichert. Dieses wird dafür in ein *JSON*-Objekt umgewandelt und daraufhin wird der *JSON*-String in einem Bytearray gespeichert.

Beim Speichern der Tabelle ist es möglich, dass zusätzliche Daten abgespeichert werden. Diese werden in einem `Dict[str, Any]` optional übergeben und müssen mit `json` in einen String umwandelbar sein. Die Daten werden beim Laden der Tabelle wiederhergestellt und stehen dann als Attribut der Klasse `HashtableUtilities` zur Verfügung.

Das Speichern der Hashtabelle darf nicht zeitgleich mit dem Einfügen von Werten geschehen, da sonst eventuell Locks gespeichert werden und diese nach dem Laden nicht aufgelöst werden.

6.6.1 Reproduzierbare Hashtabelle

Damit die Hashtabelle vollständig wieder in den Hauptspeicher geladen werden kann, muss nur das `LLConfig`-Objekt aus dem gespeicherten Array wiederhergestellt werden. Hiermit können dann alle Hashfunktionen wieder generiert werden, da in dem `LLConfig`-Objekt nicht nur Name und Parameter der Funktion gespeichert werden, sondern auch die Zufallsdaten zum Generieren der Funktion. Alle weiteren Einstellungen der Hashtabelle, die sonst über die Config Datei übergeben werden, sind ebenfalls in dem `LLConfig`-Objekt gespeichert. Die von der Hashtabelle gespeicherten Daten wie der `key_count` werden sowieso bereits im Datenarray der Hashtabelle gespeichert. Somit muss bei der erneuten Generierung kein neues leeres Datenarray erstellt werden, sondern es kann direkt das geladene Datenarray zugegriffen werden. So erhalten wir die gleiche Hashtabelle wie die, die zuvor gespeichert wurde.

6.6.2 Speicherbedarf

Der Speicherbedarf der serialisierten Tabelle wird durch das Array der Tabelle dominiert, welche wiederum abhängig von dem verwendeten Speichermodus ist. In Abschnitt 6.7.2 werden die Speichermodi und die Größen der resultierenden Tabellen verglichen.

6.7 Benchmarks

Für das Bestimmen der Hashtabellen-Performance sollen Benchmarks ausgeführt werden. Dazu werden zunächst Hashtabellen generiert und anschließend in mehreren Runden eine definierte Menge von Operation ausgeführt. Anhand der dafür benötigten Zeit lassen sich verschiedene Konfigurationen miteinander vergleichen. Daneben wird auch der grundsätzliche Durchsatz, der mit der Hashtabelle möglich ist, erkennbar. Geprüft werden sollen das Einfügen in die Hashtabelle und das anschließende Suchen nach Schlüsseln. Das Einfügen lässt sich jeweils einmal mit der Random Walk und der kostenoptimalen Einfügestratgeie ausführen. Das Löschen von Schlüsseln wird von der Hashtabelle unterstützt. Die Implementierung ist algorithmisch jedoch ähnlich zum Suchen nach Schlüsseln und wird deshalb nicht explizit in den Benchmarks ausgeführt. Daneben ist das Löschen vermutlich auch kein häufiger Anwendungsfall für die implementierte Hashtabelle.

6.7.1 Streuung

Neben der direkten Performance-Messung der Hashtabelle über das Ausführen von großen Batches an Operationen ist auch die Verteilung der eingefügten Schlüssel auf die Tabelle von Interesse. Dies ist vor allem im Hinblick auf die Eignung der Hashfunktionen für Schlüsselmengen von Interesse, für welche eine möglichst hohe Streuung erwünscht ist, um hohe Füllraten bei gleichzeitiger Verwendung von nur wenigen Hashfunktionen zu erzielen.

Um Aussagen über die Streuung machen zu können, wurde eine Funktion implementiert, welche die Hashtabelle in eine parametrisch vorgebbare Zahl an Segmenten einteilt und dann für jedes Segment zwei Eigenschaften berechnet:

- Die Verteilung der Hashfunktion-Einsätze auf das Segment
- Die Verteilung auf die Slot-Indizes auf das Segment

Die Implementierung nutzt einen speziellen Iterator, welcher den in der Tabelle für das Quotienting gespeicherte Hash-Index sowie den Slot jedes Schlüssels und den Bucket-Index zusätzlich zu dem Schlüssel und Wert zurückgibt. Auf diese Art und Weise wird über die Tabelle iteriert, für jeden Schlüssel das Segment ermittelt und dann in einem *NumPy* Array

gezählt, wie viele Einträge in Summe wo einzuordnen sind. Als Ergebnis werden die *NumPy* Arrays zurückgegeben, welche anschließend entweder über beliebigen Code ausgewertet werden können oder grafisch dargestellt werden können. Zu diesem Zweck stellt das Paket Funktionen zur Visualisierung der Ergebnisse über matplotlib zur Verfügung, welche die beiden Verteilungen als gestapelte Balkendiagramme rendern.

6.7.2 Speichermodi

In der Konfiguration der Tabelle ist die Auswahl des Speichermodus großteils unabhängig vom Einfügealgorithmus und allen anderen Parametern und wird daher gesondert betrachtet. Wie bereits in Abschnitt 6.4 erwähnt, unterstützt die Tabelle drei verschiedene Modi zur Speicherung der Daten. Abbildung 6.1 zeigt einen Laufzeit-Vergleich der drei Modi unter ansonsten gleichen Bedingungen. Tabelle 6.1 vergleicht die Speichergröße der Tabellen für eine einheitliche Größe. Darin bedeutet *Concurrency Support*, dass entweder direkt atomare Prozessorinstruktionen zum Tausch der Werte genutzt wurden oder dass *locking* genutzt wurde, um gleichzeitige Zugriffe auf denselben Eintrag zu verhindern. In der Grafik ist eine Anomalie zu sehen, in welcher der `Buckets` Modus ohne atomare Operationen langsamer ist als mit ihnen und in der Tabelle ist zu sehen, dass der Speicherverbrauch um 200 MB gestiegen ist. Der Grund dafür ist, dass Einträge unter 128 Bit im `Buckets`-Modus mit *Concurrency Support* grade so aufgefüllt werden, dass sie innerhalb eines adressierbaren 128 Bit Blocks liegen. Daher brauchen sie etwas mehr Platz als ohne *Concurrency Support*, aber profitieren ähnlich wie der `Fast`-Modus von einer vereinfachten Berechnung der einzelnen Felder. Im Modus `Fast` werden alle Einträge aufgerundet, um atomare Prozessoroperationen zu nutzen, welche 64 oder 128 Bit Speicherinhalt vergleichen und tauschen, wobei fast keine Laufzeitunterschiede zu normalen Speicherzugriffen feststellbar waren. Der Modus `Packed` fällt erwartungsgemäß zurück, da in diesem der Versatz der einzelnen Datenfelder zur Laufzeit berechnet werden muss und *locking* gegenüber atomaren Instruktionen ein Mehraufwand ist. Durch den Vergleich zeigt sich insgesamt, dass der Modus `Buckets` unter Verwendung von atomaren Instruktionen fast die gleiche Performance wie `Fast` bietet, ohne die Länge aller Einträge aufzurunden und so erheblich Speicher spart. Daher wird der Modus `Buckets` für die weiteren Vergleiche genutzt.

Modus	ohne Con.	mit Con.
Packed	500	512
Buckets	600	800
Fast	1600	1600

Tabelle 6.1: Hauptspeicher-Größe in MB für 10^8 Schlüssel (62 Bit) ohne Werte, mit und ohne *Concurrency Support*

6.7.3 Konfigurationen

Die Hashtabelle bietet mehrere Optionen zur Konfiguration. Nicht jede Option schränkt den Wertebereich explizit ein, stattdessen sind Begrenzungen eher technisch gegeben. Somit gibt es theoretisch keine obere Grenze für die Bucketgröße oder die Anzahl der Hashfunktionen. Für jene wird eine Menge von Werten herangezogen, die für eine realistische Anwendung sinnvoll erscheint. Tabelle 6.2 zeigt die gewählten Optionen. Als Speichermodus der Tabelle wird bei allen Konfigurationen `Buckets` verwendet, um von atomaren Prozessorinstruktionen Gebrauch machen zu können und den Speicherbedarf der Tabelle zu senken.

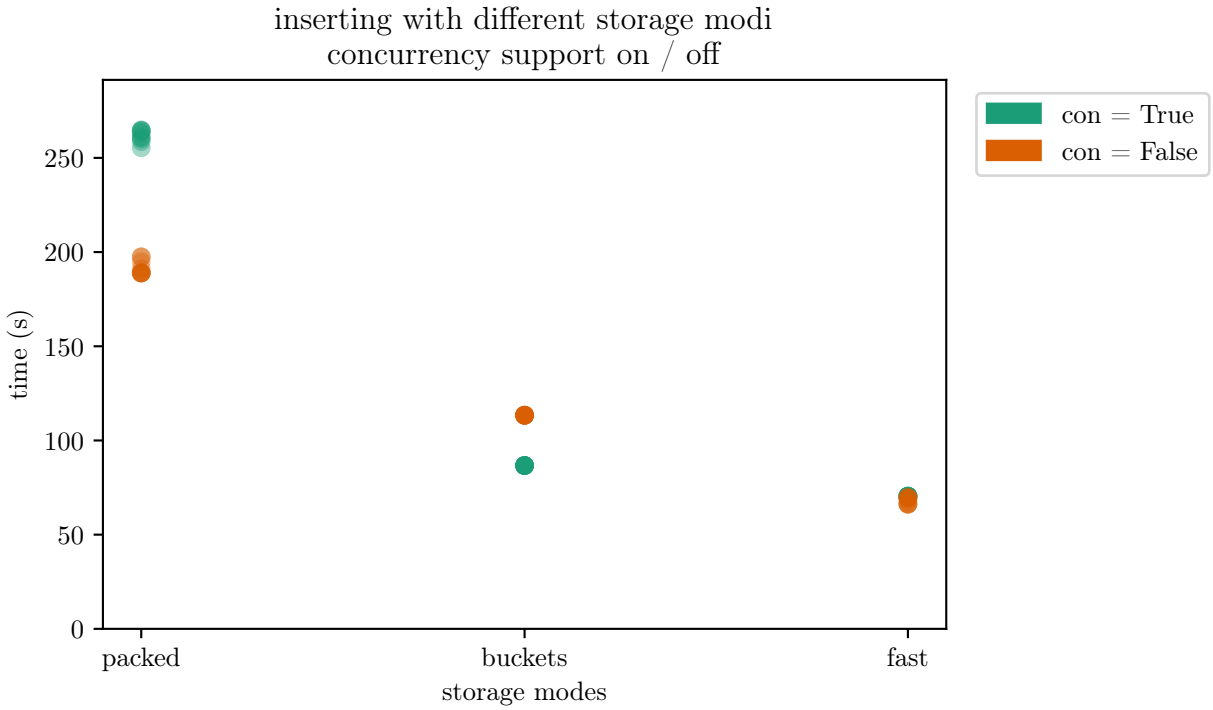


Abbildung 6.1: Vergleich der Speichermodi beim Einfügen

Option	Werte
Bucketgröße	$\{2, 4, 5, 6, 10\}$
Anzahl Hashfunktionen	$\{2, 3, 4\}$
k-mer Größe	31
Füllgrad	$\{0.75, 0.95, 0.97\}$
Tabellengröße	10^8

Tabelle 6.2: Konfigurationswerte für Hashtabellen

Aus den genannten Konfigurationswerten lassen sich schließlich eine Reihe Konfigurationen ableiten. Dabei wird prinzipiell ein Produkt mit allen Optionen gebildet, sodass jede Konfigurationskombination, die sich in der definierten Wertemenge befindet, überprüft wird. Hierbei ergeben sich 45 unterschiedliche Konfigurationen.

6.7.4 Ausführung

Die erforderlichen Berechnungen werden auf dem Linux-HPC-Cluster der Technischen Universität Dortmund (*LiDO3*) durchgeführt. Dabei werden mittels *Snakemake* pro Konfiguration jeweils zwei Benchmarks in die Warteschlange des Clusters eingefügt. Auf diese Weise können mehrere Benchmarks gleichzeitig auf verschiedenen Knoten des Clusters laufen, ohne sich gegenseitig zu beeinflussen.

Der erste Benchmark fügt 10^8 zufällig gewählte 31-mere in die Hashtabelle ein. Dies wird zunächst mit nur einem Thread, dann mit 4, 8 und 16 Threads durchgeführt. Auf diese Weise soll der Durchsatz bei der Nutzung verschieden vieler Threads untersucht werden. Nach dem Einfügen der 31-mere wird das Suchen in der Hashtabelle betrachtet. Dabei werden zwei verschiedene Szenarien geprüft. Als Erstes werden 10^8 31-mere generiert, die nicht bereits in die Hashtabelle eingefügt wurden. Dadurch lässt sich das Verhalten der

Hashtabelle bei der Abfrage von *nicht vorhandenen* Schlüsseln prüfen. Als Nächstes werden die 10^8 ursprünglich eingefügten 31-mere herangezogen und aus dieser Menge wiederum 10^8 zufällige Einträge entnommen. Auf diese Weise lässt sich das Verhalten der Hashtabelle bei der Abfrage von *vorhandenen* Schlüsseln prüfen. Das Suchen der Schlüssel wird wieder auf verschiedene Threads verteilt. Dabei werden erneut ein einzelner Thread, 4, 8 und schließlich 16 Threads gewählt.

Der zweite Benchmark verwendet die kostenoptimale Einfügestrategie zur Initialisierung der Hashtabelle. Dazu werden zur Vergleichbarkeit die gleichen 10^8 zufälligen 31-mere des ersten Benchmarks herangezogen. Die kostenoptimale Einfügestrategie lässt sich in der vorliegenden Implementierung nicht auf mehrere Threads aufteilen. Aus diesem Grund wird diese immer auf nur einem Thread ausgeführt. Nach dem Einfügen wird das Suchen auf identische Weise geprüft, wie es zuvor beschrieben ist.

LiDO3 stellt verschiedene Knoten zur Berechnung zur Verfügung. Diese weisen jedoch unterschiedliche Hardware, wie etwa Prozessor oder Arbeitsspeicher, auf. Um eine Vergleichbarkeit der Ergebnisse, die auf verschiedenen Knoten entstanden sind, zu gewährleisten, werden die zur Ausführung gewählten Knoten eingegrenzt. Angefordert werden Knoten mit dem gleichen Prozessor und 32 GB Arbeitsspeicher. Der Prozessor ist so immer ein Intel Xeon E5-2640 v4 mit 10 Kernen. Dieser ist in zwei Sockeln verbaut, sodass insgesamt 20 Kerne zur Verfügung stehen. Hyper-Threading ist für die Prozessoren deaktiviert.

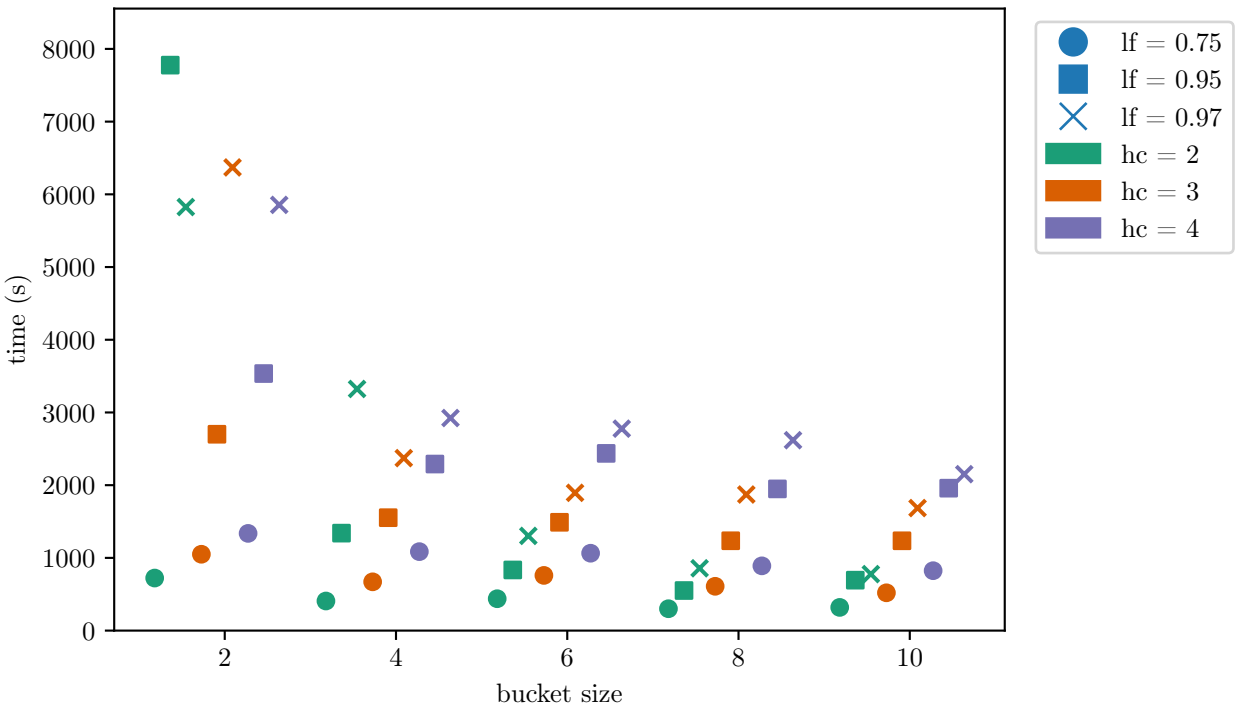
6.7.5 Ergebnisse

In Abbildung 6.2 sind die Laufzeiten des Einfügens in die Hashtabelle mit einem Thread und verschiedenen Konfigurationen des Füllgrades, der Anzahl der Hashfunktionen und der Bucketgröße. Bei den Laufzeiten ist zu sehen, dass die kostenoptimale Strategie die zehnfache Zeit benötigt. Die Bucketgrößen beeinflussen die Laufzeit des Einfügens nicht essenziell, dafür beeinflussen sowohl die Anzahl der Hashfunktionen und die Füllgrade die Laufzeit linear. Multithreading verbessert die Laufzeit ungefähr im Faktor 5 bis 10, dies kann man in den Abbildungen 6.3, 6.4 und 6.4 sehen. Dort wurden verschiedene Füllgrade mittels Multithreading und der Random Walk Einfügestrategie, sowie verschiedene Anzahlen der Hashfunktionen, Bucketgrößen und Anzahl von Threads mit dem Einfügen von 10^8 Werten untersucht. Die Streuung der Laufzeiten bei einem Füllgrad von 0.75 sind gering, wobei bei den Füllgraden von 0.95 und 0.97 die Laufzeiten eine größere Varianz haben. Im Allgemeinen kann die Laufzeit mittels Multithreading verbessert werden und die Laufzeiten werden von den Bucketgrößen nicht stark beeinflusst. Dabei zeigt sich allerdings, dass die Laufzeit bei kleineren Bucketgrößen sich mit einer höheren Anzahl an Hashfunktionen verbessert. Wenn die Bucketgröße größer als die Hashfunktionenanzahl ist, erhöht sich die Laufzeit der Random Walk Einfügestrategie.

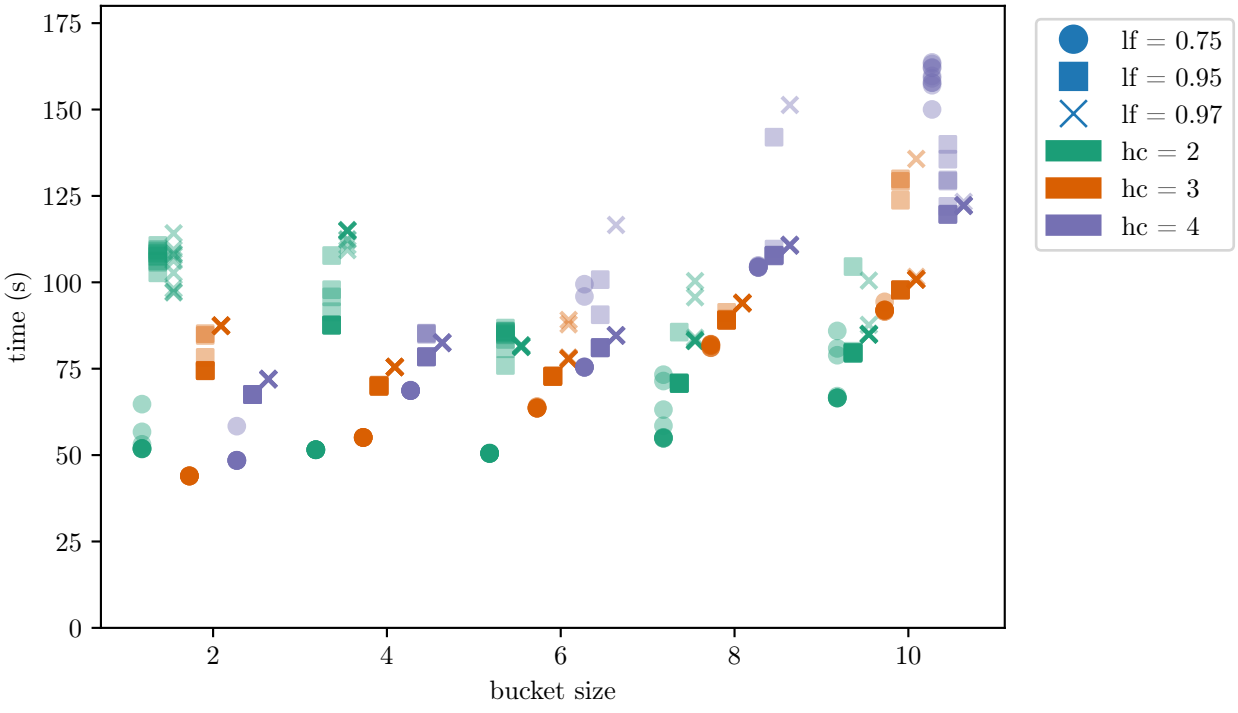
Die Laufzeiten des Einfügens mittels Random Walk und Multithreading mit unterschiedlichen Konfigurationen und dem Füllgrad von 0.75 ist in der Abbildung 6.3 dargestellt. Durch das Multithreading kann man einen Laufzeitgewinn im Faktor von 2 bis 10 erhalten, je nach Anzahl der Threads und Konfiguration. Die Laufzeit erhöht sich sowohl mit wachsender Bucketgröße als auch mit wachsender Anzahl der Hashfunktionen. Je größer die Bucketgrößen sind, desto höher sind mittels Multithreading mögliche Laufzeitgewinne.

In Abbildung 6.4 sind die Laufzeiten des Einfügens mittels Random Walk in unterschiedlichen Konfigurationen mit einem Füllgrad von 0.95 gezeigt. Die Besonderheit besteht darin, dass die Laufzeiten sich bei Erhöhung der Hashfunktionen bis zur Bucketgröße von 4 verbessern, danach aber wieder verschlechtern. Zusätzlich ist eine größere Streuung der Laufzeiten zu erkennen, welche sich im Bereich von 20 ms befinden kann, was einer Verdopplung der Laufzeit entspricht.

Das Einfügen einer Hashtabelle mit einem Füllgrad von 0.97 mittels Random Walk und



(a) Kostenoptimal



(b) Random-Walk

Abbildung 6.2: Benchmarkergebnisse vom Einfügen von 10^8 Werte in die Hashtabelle mit einem Thread, mit der kostenoptimalen Strategie (6.2a) und dem Random-Walk (6.2b).

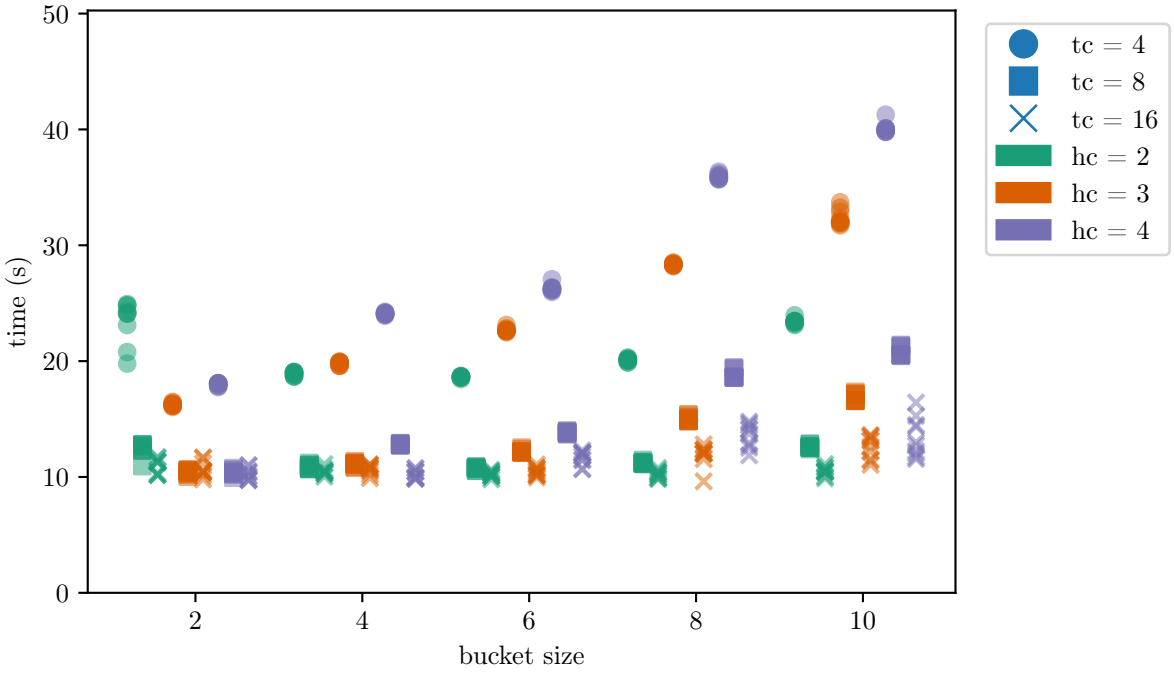


Abbildung 6.3: Benchmarkergebnisse vom Einfügen mit Multithreading von 10^8 Werte in einer Hashtabelle und einer Füllrate von 0.75, welche mit der Random Walk Einfügestrategie gefüllt wird.

verschiedenen Konfigurationen sowie Multithreading wird in Abbildung 6.5 gezeigt. Dort ist eine größere Varianz der Laufzeiten im Vergleich zu den anderen Füllgraden zu erkennen. Die Schwankungen befinden sich wie beim Füllgrad 0.97 im Bereich von 20 ms, sind hier aber häufiger vertreten. Die Laufzeit verbessert sich mit der Anzahl der Hashfunktionen bis zur Bucketgröße von 6 und wird danach wieder negativ beeinflusst.

Nun sind die Laufzeiten der Suche in verschiedenen Konfigurationen und Einfügestrategien untersucht worden, sowohl mit einem Thread, als auch mittels Multithreading. Für den Benchmark werden jeweils 10^8 Werte gesucht, wobei einmal alle Werte und einmal keiner der gesuchten Werte in der Hashtabelle enthalten ist. Damit wird ein Benchmark für den bestmöglichen und den schlechtesten Fall simuliert. Die Laufzeiten für die Suche mit einem Thread und dem Fall des kostenoptimalen Einfügens in die Hashtabelle wird in Abbildung 6.6 gezeigt. Die Laufzeiten sind bei der Suche von vorhanden Schlüsseln bei jeder Konfiguration ähnlich, die Laufzeit steigt leicht mit der Bucketgröße an. Ebenfalls führen mehr Hashfunktionen als auch ein höherer Füllgrad zur Erhöhung der Laufzeit. Im Fall der Suche nach Werten, die nicht vorhanden sind, steigt die Laufzeit mit der Anzahl der Hashfunktionen stark an, jedoch haben die Füllgrade keinen großen Einfluss auf die Laufzeit.

Im Vergleich ist in Abbildung 6.7 die Suche nach 10^8 Werten in eine per Random Walk initialisierte Hashtabelle mit einem Thread gezeigt. Diese zeigt bei der Suche vorhandener Werte eine größere Varianz an, trotz ähnlicher Laufzeit zur kostenoptimal initialisierten Hashtabelle. Die Suche nach nicht vorhanden Werten in der Tabelle führt zu größeren Varianzen bei der Suche, wobei der Trend der Laufzeiten sich wie bei der kostenoptimalen Hashtabelle verhält.

Nun werden die Laufzeiten der Suchoperationen mittels Multithreading für unterschiedliche Füllgrade der Hashtabelle sowie die verschiedenen Einfügestrategien in mehreren Kon-

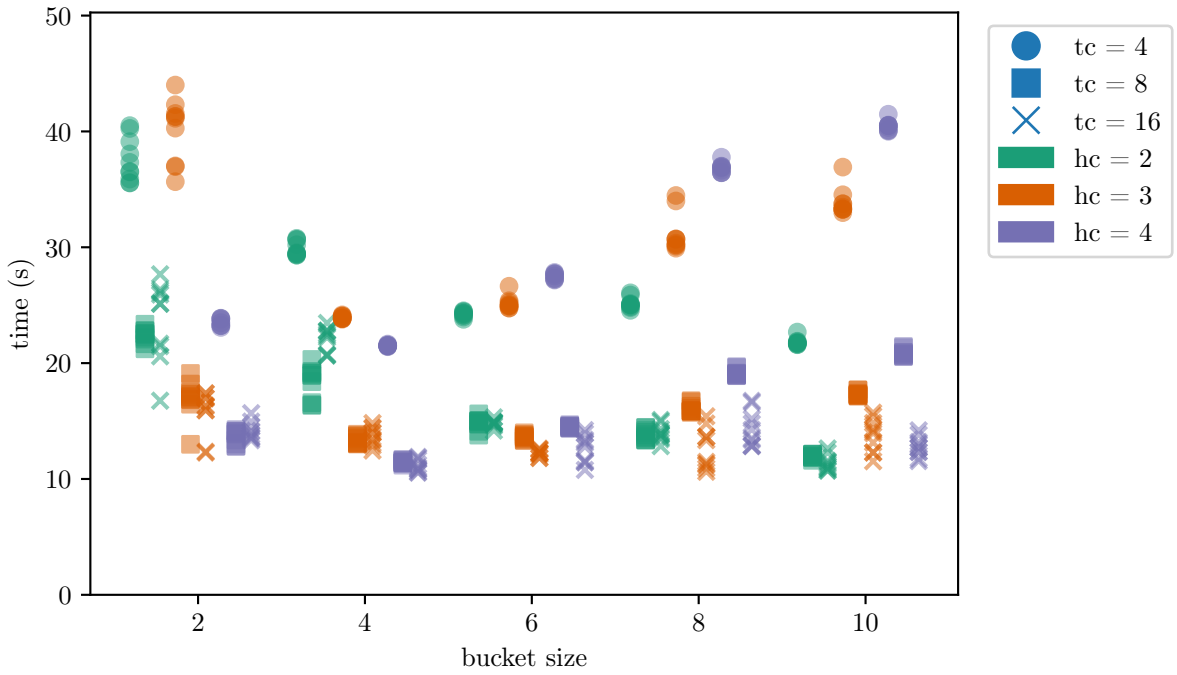


Abbildung 6.4: Benchmarkergebnisse vom Einfügen mit Multithreading von 10^8 Werte in einer Hashtabelle und einer Füllrate von 0.95, welche mit der Random Walk Einfügestrategie gefüllt wird.

figurationen evaluiert.

In Abbildung 6.8 und 6.9 sind die Laufzeiten einer kostenoptimal initialisierten Hashtabelle mit einem Füllgrad von 0.75 gezeigt. Die Laufzeiten mittels Multithreading sind im Vergleich zu einem Thread, wie in Abbildung 6.6, mindestens um den Faktor 4 schneller. Wie schon zuvor hat die Bucketgröße bei erfolgreichen Suchen keinen Einfluss auf die Laufzeit, bei nicht erfolgreichen Suchen erhöht sich die Laufzeit. Die Anzahl der Hashfunktionen erhöht ebenfalls die Laufzeit der nicht erfolgreichen Suchen. Die Laufzeiten besitzen eine geringe Varianz.

Die Laufzeiten bei einem Füllgrad von 0.75 und der Random Walk Einfügestrategie sind ähnlich der kostenoptimalen Einfügestrategie, wobei die Laufzeiten in Abbildung 6.10 für erfolgreiche Suchen zu sehen ist. Jedoch ist eine etwas größere Varianz bei der Suche nicht vorhandener Werte in Abbildung 6.11 zu sehen. Die Laufzeit wird wie bei der kostenoptimalen Strategie ebenfalls von der Bucketgröße und der Anzahl der Hashfunktionen beeinflusst. Diese ist im Vergleich zu einem Thread um den Faktor 2 bis 4 schneller.

Die Laufzeiten der Suchoperationen der Hashtabellen mit der kostenoptimalen Einfügestrategie in unterschiedlichen Konfigurationen und verschiedenen Threadanzahlen, sowie einem Füllgrad von 0.95 sind in Abbildung 6.12 und 6.13 zu sehen. Dort erhöht sich die Laufzeit mit Anzahl der Hashfunktionen, als auch mit größeren Bucketgrößen bei fehlgeschlagenen Suchen. Die Laufzeit der erfolgreichen Suchen ist mittels Multithreading um den Faktor 4 schneller, bei nicht erfolgreichen Suchen um den Faktor 4 bis 20.

Bei den Random Walk initialisierten Hashtabellen mit Füllgrad 0.95 sind die Laufzeiten in Abbildungen 6.14 und 6.15 dargestellt. Die Laufzeiten sind ähnlich der Laufzeiten der kostenoptimalen Variante, wobei bei der Konfiguration mit zwei Hashfunktionen und der Bucketgröße von 2, die Laufzeit um den Faktor 2 schlechter ist. Zusätzlich ist eine größere Varianz in den verschiedenen Durchläufen gegeben.

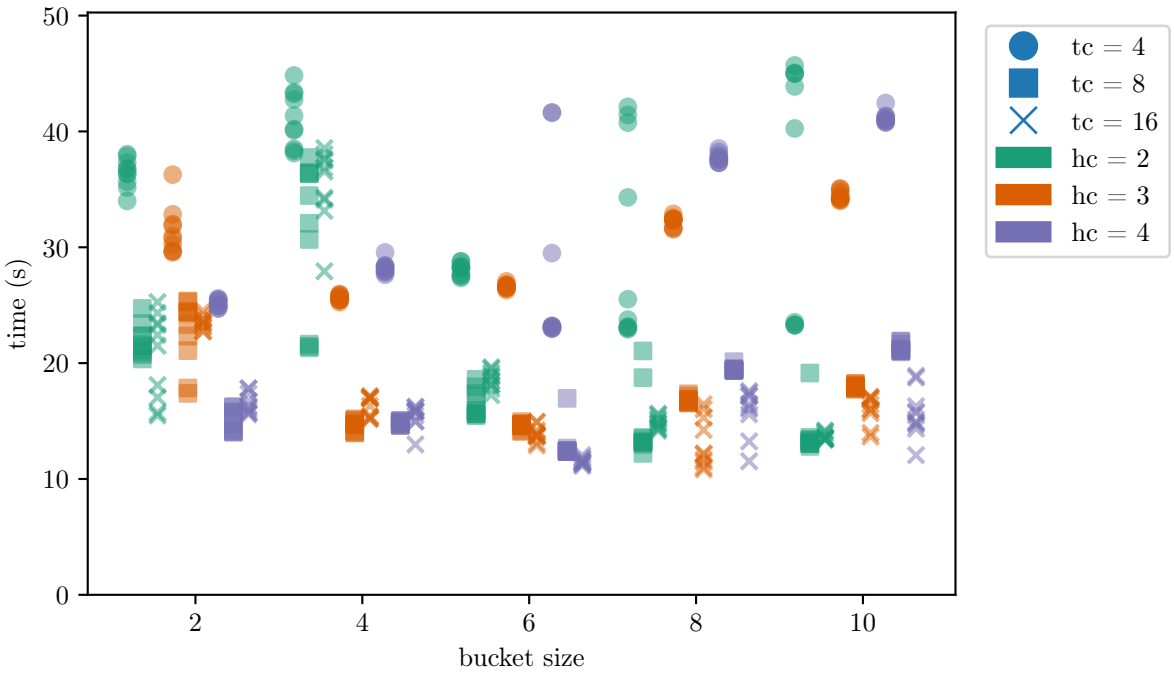


Abbildung 6.5: Benchmarkergebnisse vom Einfügen mit Multithreading von 10^8 Werte in einer Hashtabelle und einer Füllrate von 0.97, welche mit der Random Walk Einfügestrategie gefüllt wird.

Die Laufzeiten der Suchoperationen der Hashtabellen mit der kostenoptimalen Einfügestrategie in unterschiedlichen Konfigurationen und verschiedenen Threadanzahlen, sowie einem Füllgrad von 0.97 sind in Abbildung 6.16 und 6.17 zu sehen. Dort erhöht sich die Laufzeit mit der Anzahl der Hashfunktionen als auch mit größeren Bucketgrößen bei fehlgeschlagenen Suchen. Die Laufzeit der erfolgreichen Suchen ist mittels Multithreading um den Faktor 4 schneller, bei nicht erfolgreichen Suchen um den Faktor 4 bis 8.

Für die per Random Walk initialisierten Hashtabellen mit Füllgrad 0.97 sind die Laufzeiten in den Abbildungen 6.18 und 6.19 dargestellt. Die Laufzeiten sind ähnlich der Laufzeiten der kostenoptimalen Variante, wobei bei der Konfiguration mit zwei Hashfunktionen und der Bucketgröße von 2 die Laufzeit um den Faktor 2 schlechter ist. Zusätzlich ist eine größere Varianz in den verschiedenen Durchläufen gegeben.

Im Allgemeinen ähneln sich die durchschnittlichen Laufzeiten der Suchen bei der kostenoptimalen und der Random Walk Einfügestrategie. Ebenso kann mittels Multithreading die Laufzeit der Suche um den Faktor 2 bis 20, je nach Konfiguration, Füllstand und Anzahl an Threads, verbessert werden.

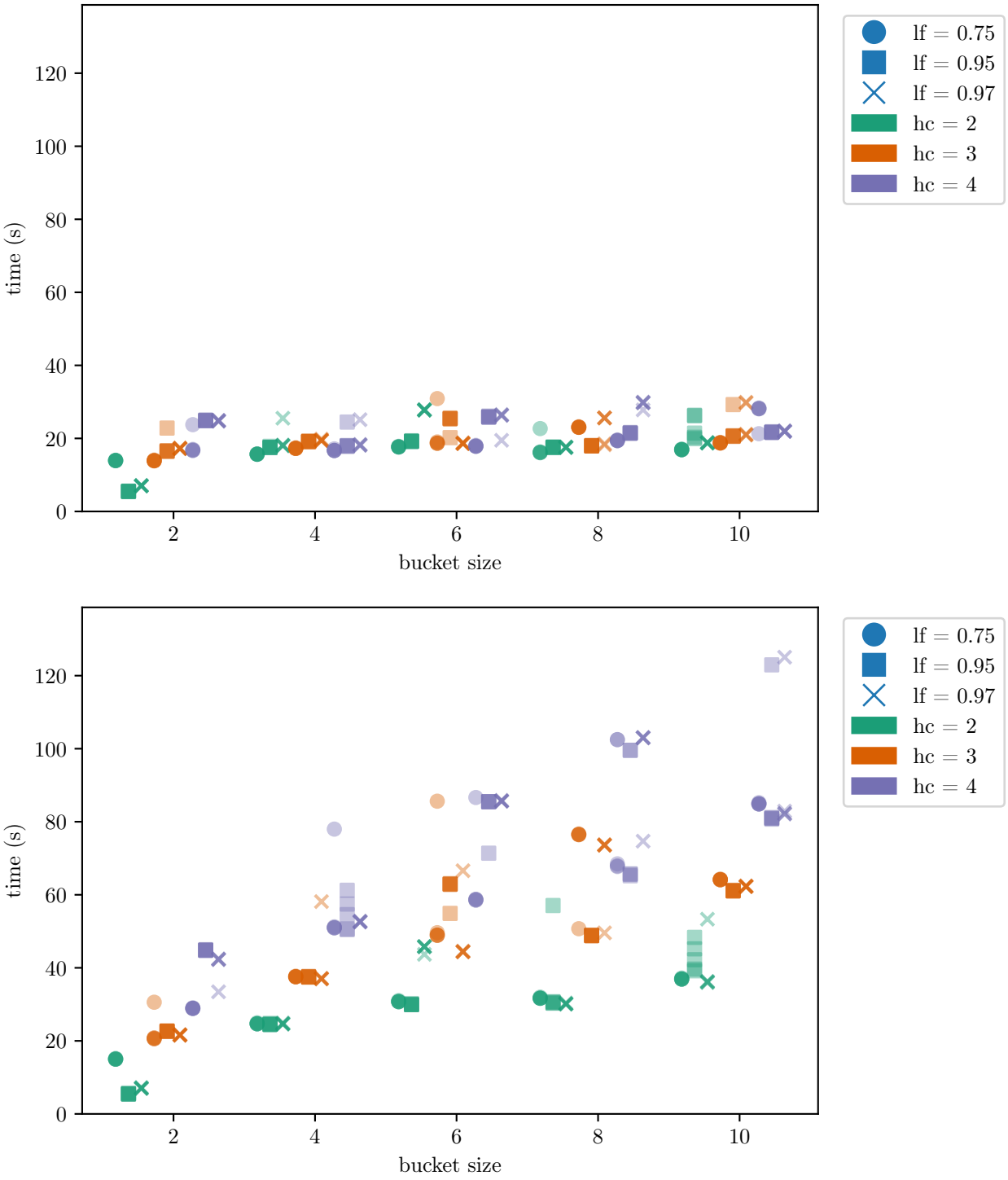


Abbildung 6.6: Benchmarkergebnisse vom Suchen mit einem Thread von 10^8 Werte in einer Hashtabelle, welche mit der kostenoptimalen Einfügestrategie gefüllt worden ist.

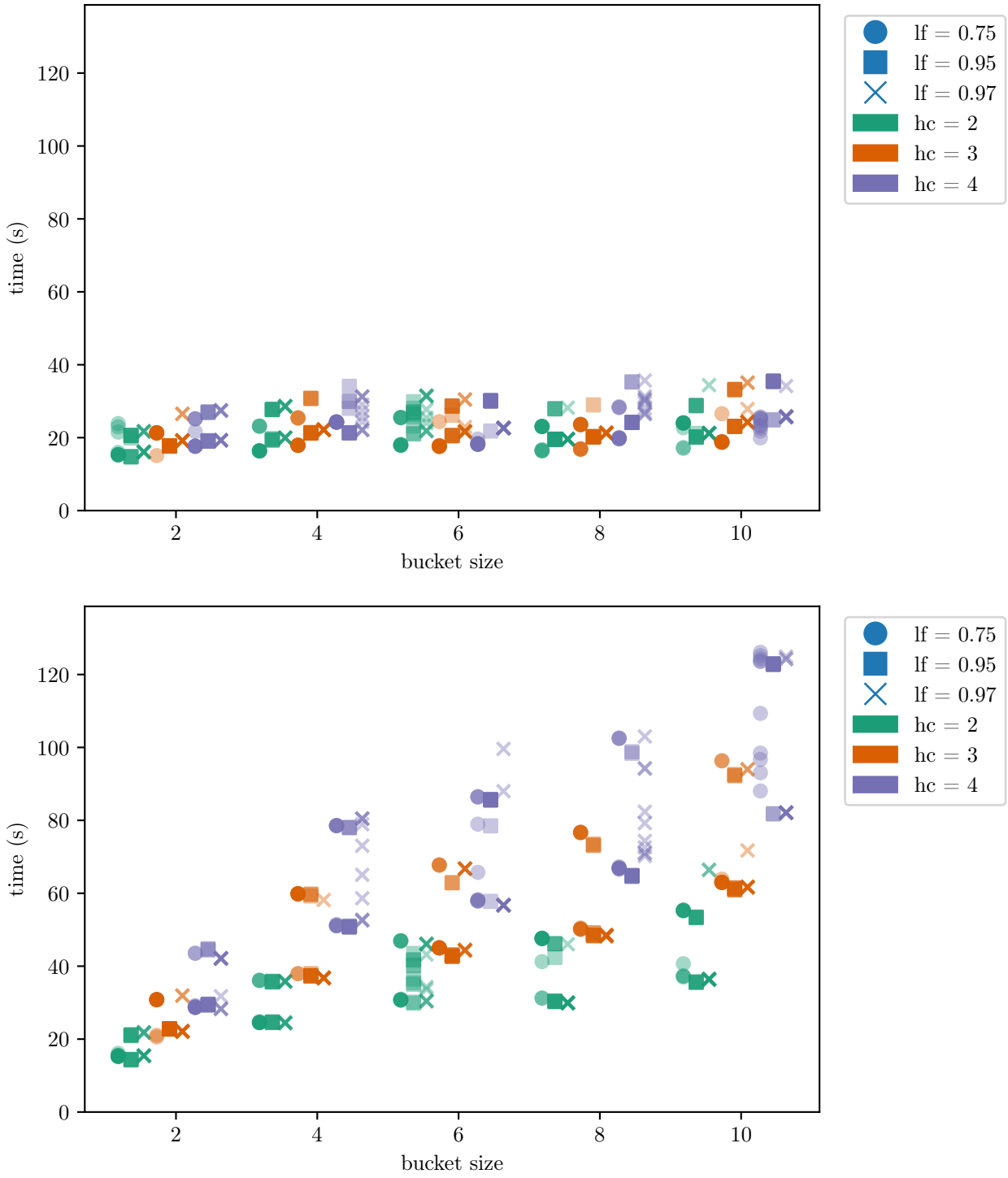


Abbildung 6.7: Benchmarkergebnisse vom Suchen mit einem Thread von 10^8 Werte in einer Hashtabelle, welche mit der Random Walk Einfügestrategie gefüllt worden ist.

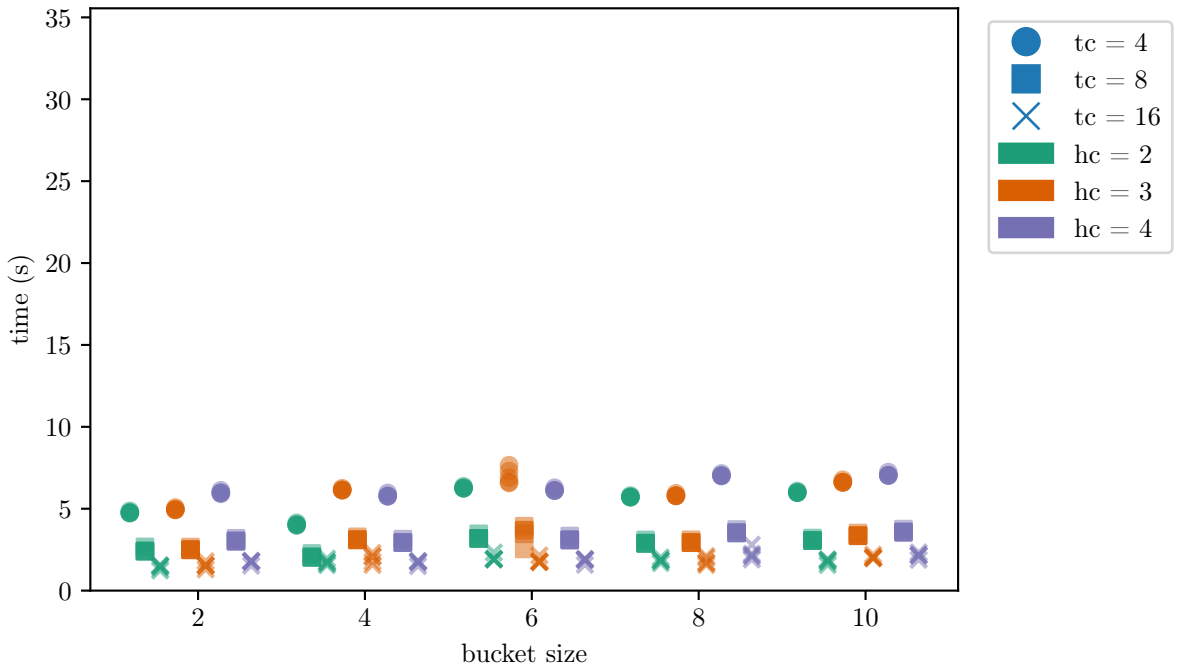


Abbildung 6.8: Benchmarkergebnisse vom Suchen mit Multithreading von 10^8 Werte in einer Hashtabelle, welche mit einer kostenoptimalen Einfügestrategie gefüllt worden ist und einer Füllrate von 0.75. 100% der gesuchten Elemente sind enthalten.

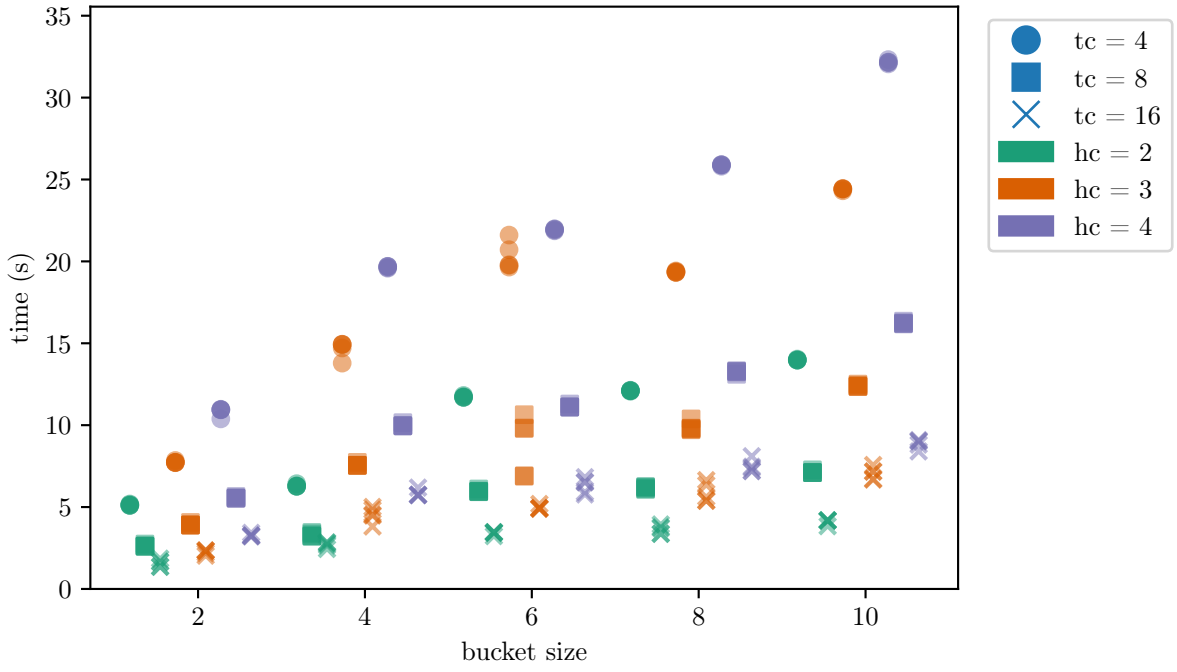


Abbildung 6.9: Benchmarkergebnisse vom Suchen mit Multithreading von 10^8 Werte in einer Hashtabelle, welche mit einer kostenoptimalen Einfügestrategie gefüllt worden ist und einer Füllrate von 0.75. 100% der gesuchten Elemente sind nicht enthalten.

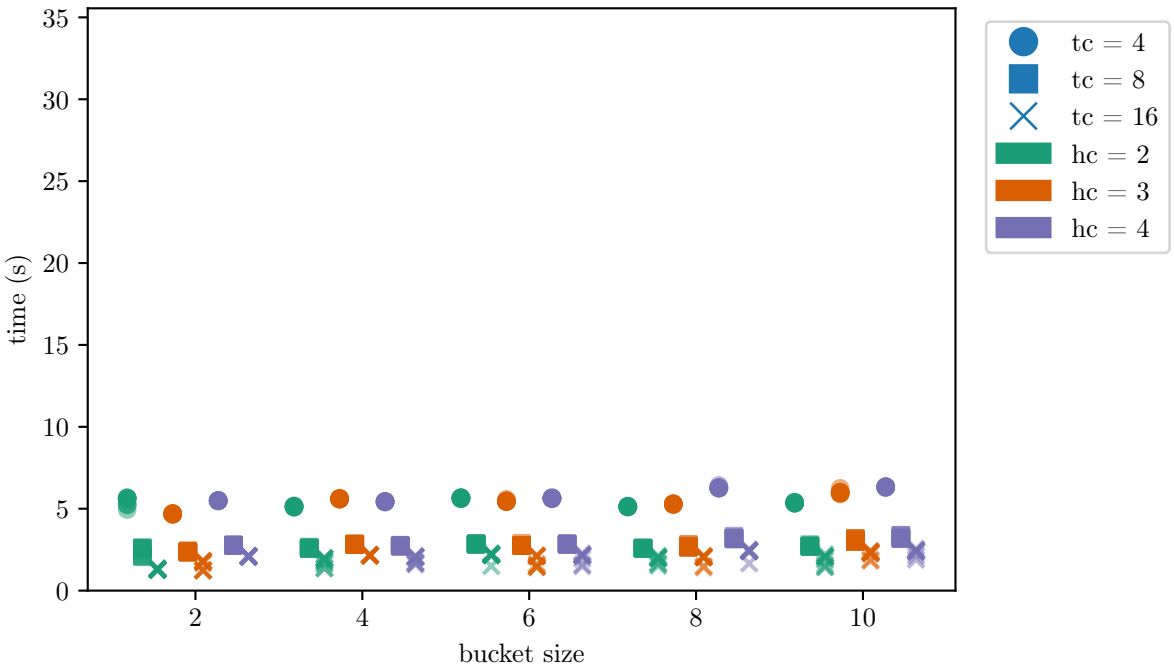


Abbildung 6.10: Benchmarkergebnisse vom Suchen mit Multithreading von 10^8 Werte in einer Hashtabelle, welche mit einer Random Walk Einfügestrategie gefüllt worden ist und einer Füllrate von 0.75. 100% der gesuchten Elemente sind enthalten.

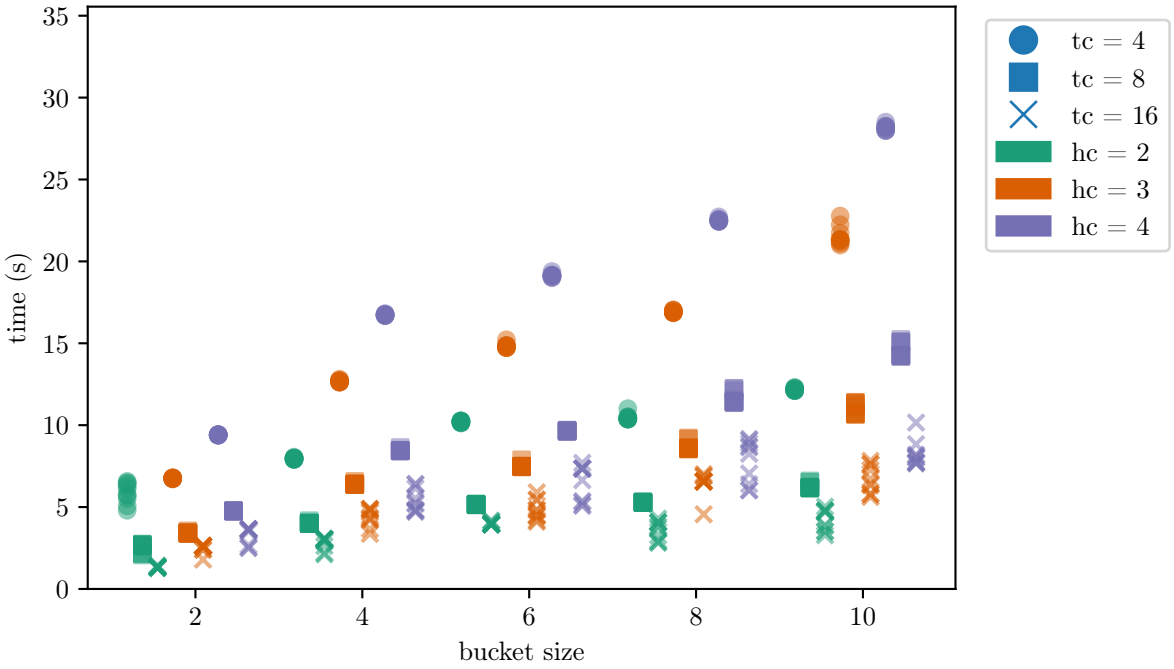


Abbildung 6.11: Benchmarkergebnisse vom Suchen mit Multithreading von 10^8 Werte in einer Hashtabelle, welche mit einer Random Walk Einfügestrategie gefüllt worden ist und einer Füllrate von 0.75. 100% der gesuchten Elemente sind nicht enthalten.

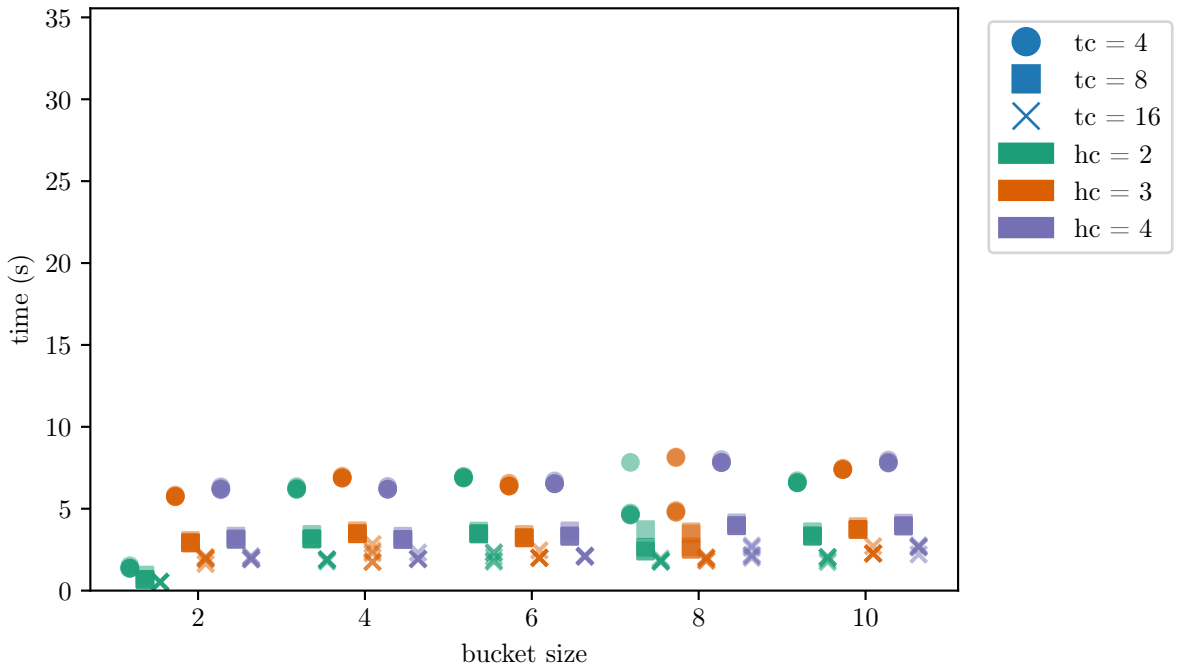


Abbildung 6.12: Benchmarkergebnisse vom Suchen mit Multithreading von 10^8 Werte in einer Hashtabelle, welche mit einer kostenoptimalen Einfügestrategie gefüllt worden ist und einer Füllrate von 0.95. 100% der gesuchten Elemente sind enthalten.

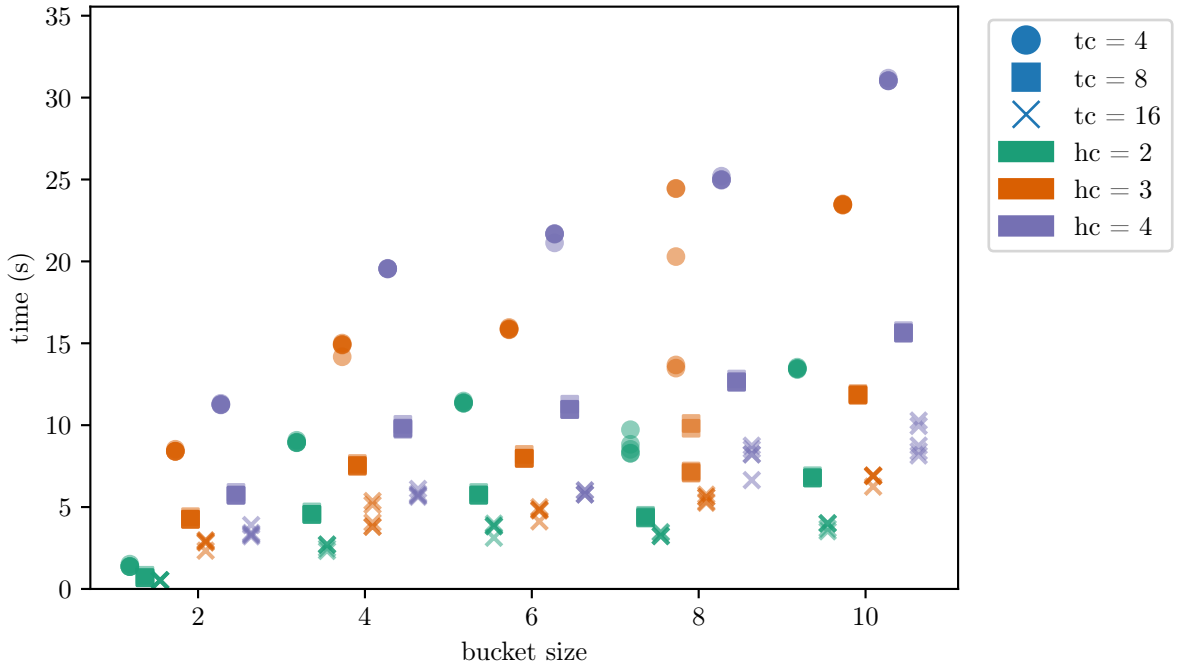


Abbildung 6.13: Benchmarkergebnisse vom Suchen mit Multithreading von 10^8 Werte in einer Hashtabelle, welche mit einer kostenoptimalen Einfügestrategie gefüllt worden ist und einer Füllrate von 0.95. 100% der gesuchten Elemente sind nicht enthalten.

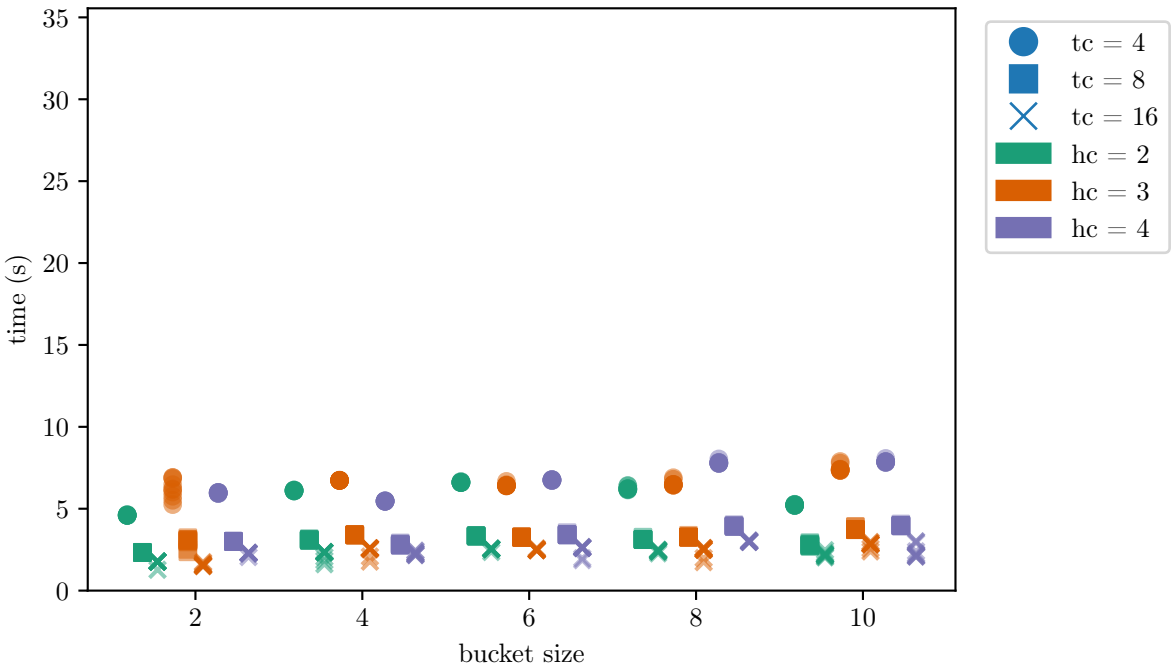


Abbildung 6.14: Benchmarkergebnisse vom Suchen mit Multithreading von 10^8 Werte in einer Hashtabelle, welche mit einer Random Walk Einfügestrategie gefüllt worden ist und einer Füllrate von 0.95. 100% der gesuchten Elemente sind enthalten.

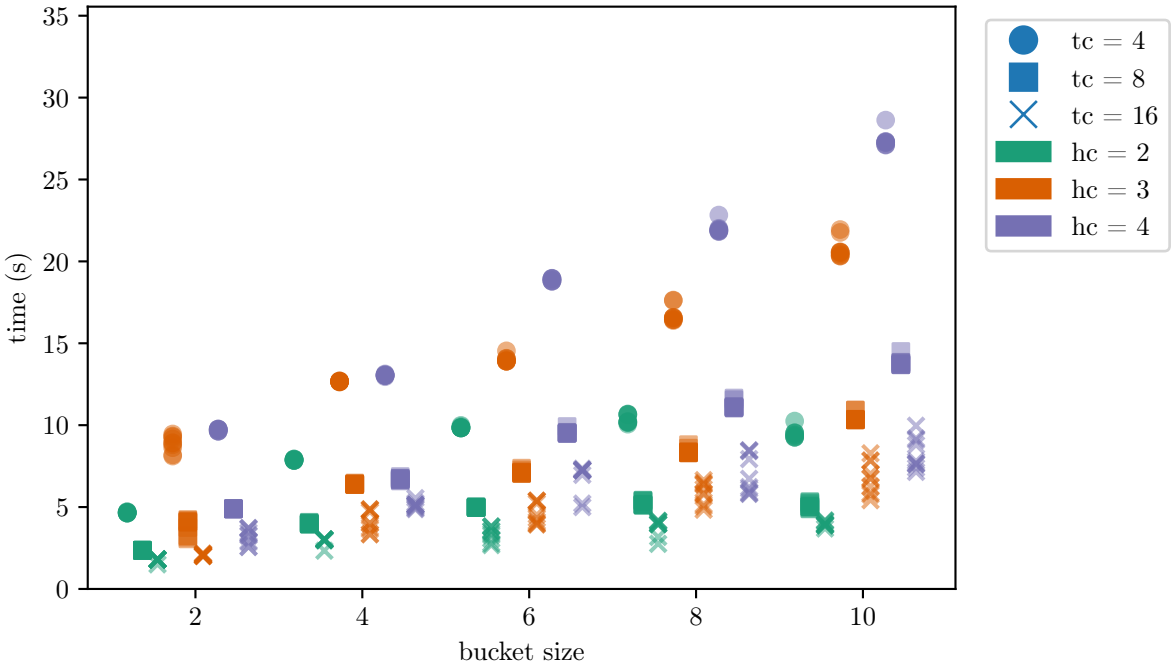


Abbildung 6.15: Benchmarkergebnisse vom Suchen mit Multithreading von 10^8 Werte in einer Hashtabelle, welche mit einer Random Walk Einfügestrategie gefüllt worden ist und einer Füllrate von 0.95. 100% der gesuchten Elemente sind nicht enthalten.

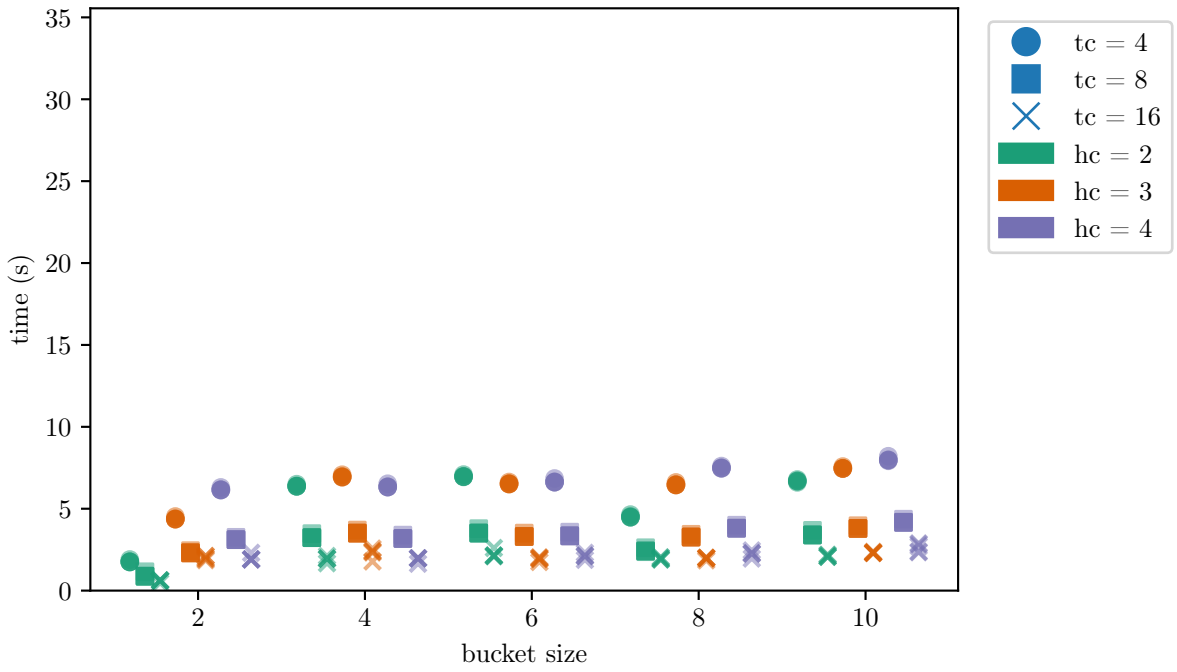


Abbildung 6.16: Benchmarkergebnisse vom Suchen mit Multithreading von 10^8 Werte in einer Hashtabelle, welche mit einer kostenoptimalen Einfügestrategie gefüllt worden ist und einer Füllrate von 0.97. 100% der gesuchten Elemente sind enthalten.

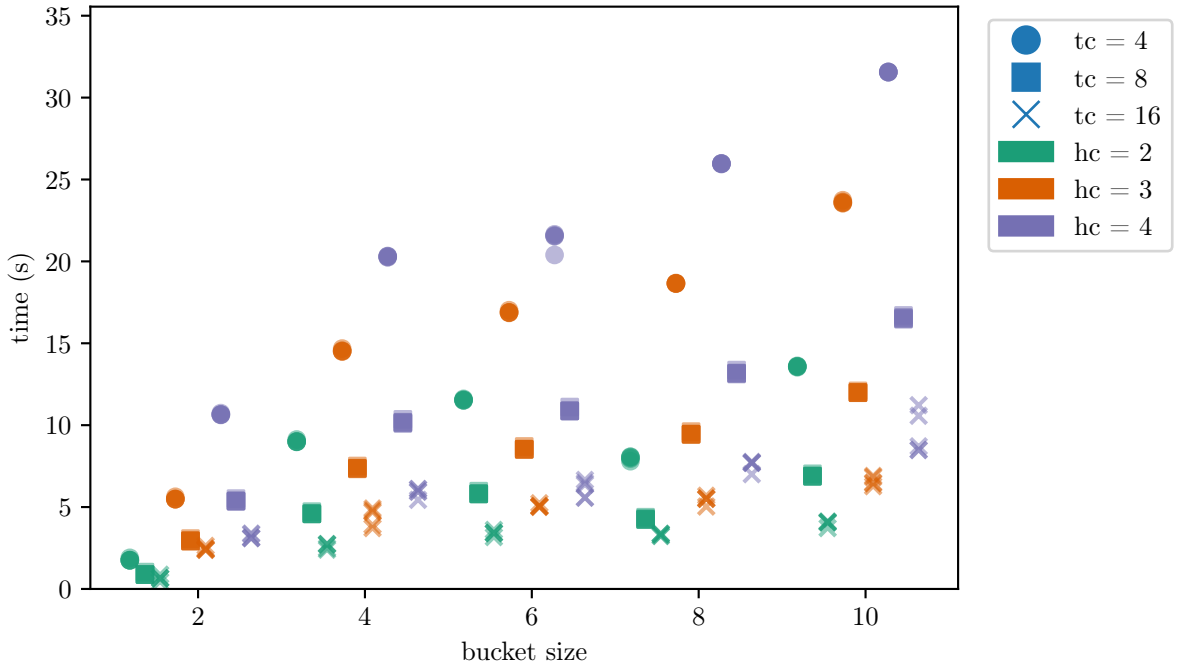


Abbildung 6.17: Benchmarkergebnisse vom Suchen mit Multithreading von 10^8 Werte in einer Hashtabelle, welche mit einer kostenoptimalen Einfügestrategie gefüllt worden ist und einer Füllrate von 0.97. 100% der gesuchten Elemente sind nicht enthalten.

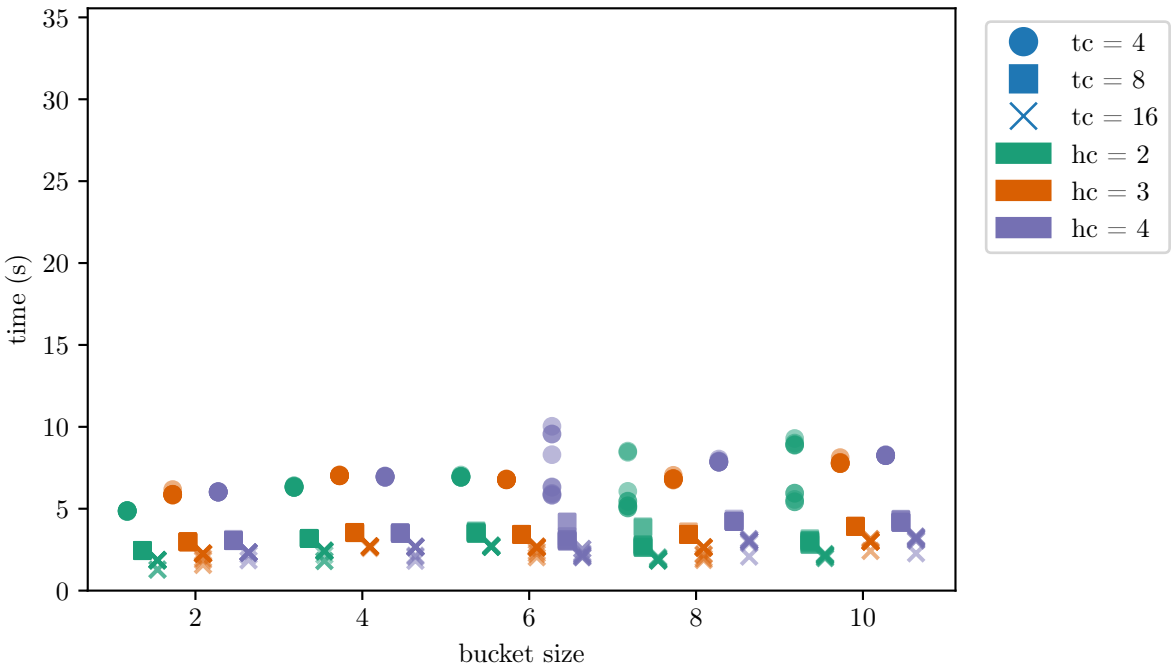


Abbildung 6.18: Benchmarkergebnisse vom Suchen mit Multithreading von 10^8 Werte in einer Hashtabelle, welche mit einer Random Walk Einfügestrategie gefüllt worden ist und einer Füllrate von 0.97. 100% der gesuchten Elemente sind enthalten.

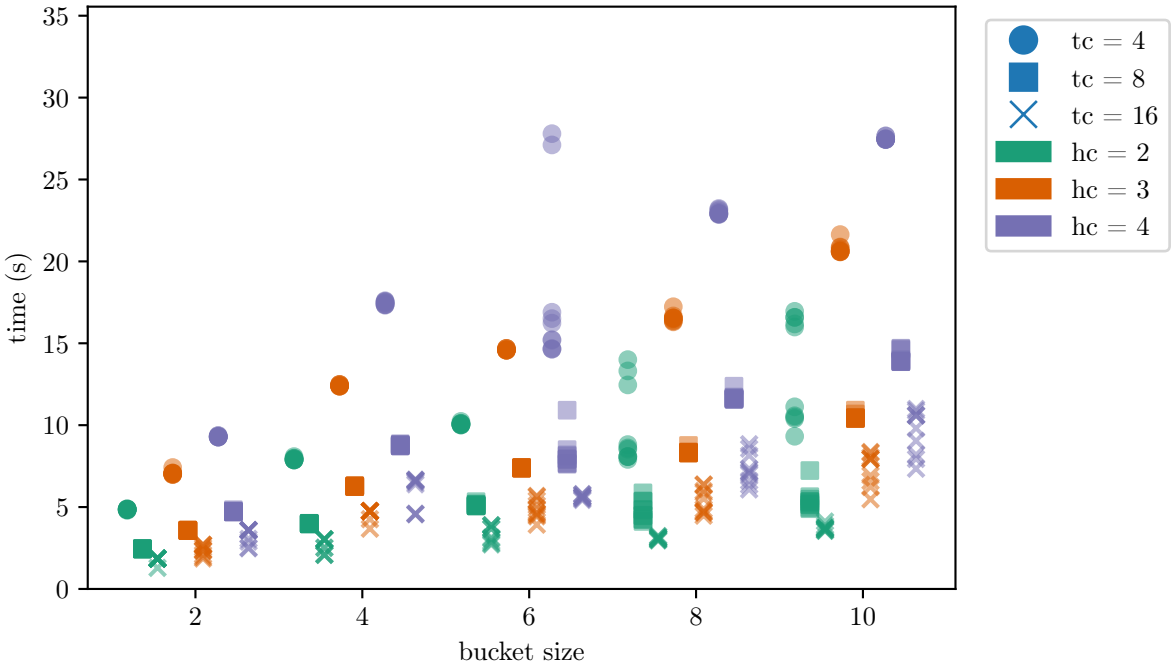


Abbildung 6.19: Benchmarkergebnisse vom Suchen mit Multithreading von 10^8 Werte in einer Hashtabelle, welche mit einer Random Walk Einfügestrategie gefüllt worden ist und einer Füllrate von 0.97. 100% der gesuchten Elemente sind nicht enthalten.

Kapitel 7 — Yet Another FASTQ Quality Control

Nachdem die Umsetzung der in Kapitel 5 vorgestellten Konzepte im vorherigen Kapitel 6 erfolgt ist, soll dieses Kapitel von einem Anwendungsszenario der Hashtabelle handeln. In Abstimmung mit den Betreuern wurde in einem Gruppentreffen der Projektgruppe gemeinsam entschieden, ein Tool zur Qualitätssicherung von *FASTQ*-Dateien zu erstellen. Das Grundgerüst dafür bietet der in Kapitel 4 vorgestellte Low Complexity Filter. Das Ziel bestand darin, ein Tool zu erstellen, welches *FASTQ*-Dateien hinsichtlich mehrerer Qualitätsmerkmale prüft und eine Kontaminationsprüfung durchführt. Die Funktionalität der Hashtabelle ist dabei hauptsächlich für die Kontaminationsprüfung nützlich.

In diesem Kapitel soll zuerst auf vergleichbare Tools eingegangen werden, welche als Inspiration dienen. Im Anschluss wird die Implementierung des Tools vorgestellt und Entscheidungen während der Entwicklung erläutert. Daraufhin wird auf die Ergebnisse des Benchmarking eingegangen, um die Performance des Tools einordnen zu können. Hierzu wird schließlich auch ein Vergleich zu den existierenden Tools gezogen.

7.1 Bestehende Tools

Um einen Eindruck von möglichen Merkmalen, die später implementiert werden sollen, zu erlangen, wurden bestehende Tools betrachtet. Diese sollen im Folgenden kurz vorgestellt und auf eventuelle Verbesserungsmöglichkeiten eingegangen werden.

7.1.1 FastQC

FastQC[1] analysiert gegebene Sequenz-Dateien bezüglich verschiedener Qualitätsmerkmale. Pro Merkmal wird anhand eines festen Schwellwerts eine Beurteilung vorgenommen: Okay, Warnung oder Fehler. Diese Beurteilungen sind Hinweise, welche Merkmale betrachtet und im Kontext der weiteren Verarbeitung beachtet werden sollten. Es werden folgende Dateiformate unterstützt: *FASTQ*, BAM und SAM. *FastQC* stellt eine Grafikoberfläche bereit, in der die Ergebnisse der Module in Diagrammen dargestellt werden. In der Grafikoberfläche lassen sich die Ergebnisse mehrerer Dateien parallel als Tabs anzeigen. Darüber hinaus können die Ergebnisse als *HTML*-Datei exportiert werden. Zu den behandelten Merkmalen zählen die Länge der Sequenzen, das Auftreten der einzelnen Basen pro Position im *read*, die durchschnittliche Qualität pro Position im *read*, das Auftreten durchschnittlicher Qualitätswerte pro *read* oder auch der GC-Gehalt. Hinzu kommen Merkmale, die sich mit Duplikaten von *reads* oder auch übermäßig vorhandenen k-meren befassen.

Ein Nachteil besteht allerdings darin, dass *FastQC* für einige Merkmale nur ungefähr 2% der Eingabedatei betrachtet. Zusätzlich sind die Diagramme teilweise missverständlich gewählt oder die Achsenbeschriftungen unklar. Es bestehen also bereits Verbesserungsmöglichkeiten für die Umsetzung dieser Merkmale in unserem Tool.

7.1.2 FastQ Screen

FastQ Screen[2] ist in der Lage, eine Sequenz-Datei mit verschiedenen Referenzgenomen zu vergleichen und so den Ursprung der Sequenzen zu ermitteln. Diese Funktion ist wichtig

[1] `https://www.bioinformatics.babraham.ac.uk/projects/fastqc/`
[2] `https://www.bioinformatics.babraham.ac.uk/projects/fastq_screen`

um festzustellen, ob die Zusammensetzung der Sequenzierten Daten den Erwartungen entspricht oder ob Verunreinigungen vorliegen. Das Tool arbeitet im Hintergrund mit Sequenzalignment, welches von bestehenden Tools wie *bowtie*[3], *bowtie2*[4] oder *BWA*[5] bereitgestellt wird. Durch den Einsatz von Alignment kann es präzise unterscheiden, ob ein *read* einmal oder mehrfach in einem Referenzgenom vorkommt, oder ob er einmal oder mehrfach in mehreren Referenzgenomen vorkommt. Die Informationen über die Anzahl der auftretenden *reads* pro Referenzgenom und die Art des Vorkommens (einfach oder mehrfach, exklusiv oder in mehreren Genomen) werden von dem Tool auf Wunsch des Nutzers als gefärbte Säulendiagramme ausgegeben.

Da das Tool darauf basiert, die *reads* abzugleichen, hat es einen inhärenten Laufzeitnachteil gegenüber einer Zuordnung, welche auf Hashing basiert. Im Gegenzug bietet es die Möglichkeit festzustellen, ob ein *read* mehrfach in einem Referenzgenom vorkommt, was mittels Hashing von k-meren schwierig bis unmöglich zu reproduzieren ist. Ein weiterer Nachteil des Tools für die Laufzeit ist, dass ein Vergleich mit verschiedenen Referenzgenomen erfolgt, indem ein *read* nacheinander gegen alle Referenzgenom geprüft wird, während eine Hashtabelle die gleichzeitige Zuordnung zu beliebig vielen Referenzgenomen ermöglicht, so lange diese in den Speicher passen.

Die Referenzindizes für das Tool müssen dafür in einem passenden Format des jeweiligen Aligners vorliegen oder vom Nutzer erst mit dem Aligner aus *FASTA* Dateien erstellt werden. Dies ist ein Bruch im Arbeitsablauf, weil erst ein anderes Tool genutzt werden muss. Dadurch bietet es aber auch den Vorteil, dass die erstellten Indizes mit anderen Nutzern geteilt und wiederverwendet werden können.

Insgesamt sollte im Bereich Laufzeit die größte Stärke einer Hashing-Methode liegen, während nur wenige Informationen gegenüber einem Alignmentverfahren verloren gehen.

7.1.3 FastQ Scan

Das Tool *fastq-scan*[6] berechnet verschiedene einfache Statistiken zu einer gegebenen *FASTQ*-Datei. Die Eingabe erfolgt dabei über STDIN und die Ausgabe wird in einer *JSON*-Datei bereitgestellt. Die berechneten Statistiken beziehen sich ausschließlich auf *read*-Längen und Qualitätswerte, wobei für die Qualitätswerte das arithmetische Mittel über den *read* gebildet wird. Als Ausgabe erfolgt dabei jeweils das arithmetische Mittel, die Standardabweichung, die Quartile, sowie das Minimum und das Maximum für Längen und den Qualitäten. Außerdem wird noch das arithmetische Mittel der Qualitätswerte pro Base und die Häufigkeit der jeweiligen Längen ausgegeben.

Der größte Nachteil von *fastq-scan* ist, dass das Tool nur die einfachsten Statistiken aus einer *FASTQ*-Datei berechnet. Viele Statistiken, wie der GC-Gehalt oder eine Kontaminationsprüfung, sind mit *fastq-scan* nicht verfügbar.

[3] `http://bowtie-bio.sourceforge.net/index.shtml`
[4] `http://bowtie-bio.sourceforge.net/bowtie2/index.shtml`
[5] `http://bio-bwa.sourceforge.net`
[6] `https://github.com/rpetit3/fastq-scan`

7.2 Implementierung

Im Folgenden soll die Implementierung von *YAFQC* näher erläutert werden und während der Entwicklung getroffene Entscheidungen begründet werden.

7.2.1 Qualitätsmerkmale

Neben der Kontaminationsprüfung bietet *YAFQC* eine Reihe an Qualitätsmerkmalen zur Beurteilung von *FASTQ*-Dateien an. Diese sind in Anlehnung an das Tool *FastQC* (siehe Abschnitt 7.1.1) ausgewählt und implementiert worden. Die ausgewählten Merkmale werden hier genannt und kurz erläutert.

k-mer Verteilung

Die Häufigkeiten der k-mere können sortiert in einem Histogramm betrachtet werden. Zur Berechnung dieser Verteilung werden während des *processings* alle k-mere der Sequenzen in einer Hashtabelle gesammelt und gezählt.

Qualitätswerte an Positionen eines *reads*

Die Häufigkeiten der verschiedenen Phred-Score Werte der Basen werden pro Position gezählt. Dafür wird ein zweidimensionales *Array* verwendet. Im Report wird dann eine Visualisierung mit Boxplots für jede einzelne *read*-Position erstellt. Außerdem markiert eine rote Linie den Verlauf der mittleren Phred-Scores über die Positionen.

Qualitätswerte pro *read*

Für die Qualitätswerte pro *read* wird für jeden *read* der mittlere Phred-Score aller Basen berechnet. Die Häufigkeiten dieser Mittelwerte werden gezählt und in einem Kurvendiagramm dargestellt.

Verteilung der Nukleinbasen

Um die Verteilung der Nukleinbasen darzustellen, werden die Häufigkeiten der Basen `A`, `C`, `G` und `T` pro Position im *read* gezählt sowie die Häufigkeit von auftretenden `N` (in der Sequenzierung konnte die Base nicht eindeutig identifiziert werden). Anschließend werden ihre prozentualen Anteile berechnet und in einem Kurvendiagramm abgebildet.

GC-Gehalt

Der prozentuale Gehalt der Basen `G` und `C` zusammen wird pro *read* berechnet. In der Grafik werden dann die Häufigkeiten der einzelnen Prozentwerte aufgerundet auf ganze Prozentpunkte dargestellt. Zusätzlich wird eine passende ideale Normalverteilung abgebildet, um die Abweichung der Daten von der Normalverteilung beurteilen zu können.

Länge der *reads*

Die Längen der *reads* werden ebenfalls gezählt und in einem Histogramm dargestellt.

7.2.2 Command Line Interface

Für *YAFQC* wurde, wie bereits für den *lcfilter*, ein nutzerfreundliches *command line interface* erstellt. Hierfür wurde wieder auf das Paket argparse [7] zurückgegriffen, sodass der

[7] `https://docs.python.org/3/library/argparse.html`

Nutzer an eine automatisch generierte Hilfe anknüpfen kann und beim Aufruf über gegebenenfalls fehlende Argumente informiert wird. Es sind drei *subcommands* für *YAFQC* verfügbar: `contamination`, `process` und `report`.

Das *subcommand* `contamination` dient dazu, eine Hashtabelle aus Referenzgenomen aufzubauen, die für die Kontaminationsprüfung nötig ist. Es erhält als Eingabe eine oder mehrere *FASTA*-Dateien, die jeweils als ein Referenzgenom interpretiert werden. So kann eine gebündelte Tabelle für eine Sammlung von verschiedenen Referenzgenomen aufgebaut werden, zwischen denen während der Kontaminationsprüfung unterschieden wird. Nach Abschluss des Einlesens und dem Erstellen der Tabelle wird diese serialisiert und als Datei abgespeichert.

Mittels des *subcommands* `process` kann eine *FASTQ*-Datei auf verschiedene Qualitätsmerkmale hin untersucht werden. Die verschiedenen verfügbaren Merkmale sind in Abschnitt 7.2.1 gelistet. Außerdem kann eine Referenztabelle angegeben werden, die mit `contamination` generiert wurde. In diesem Fall werden zusätzlich pro *read* zwei verschiedene Distanzmaße zu den Referenzgenomen der Tabelle berechnet. Sämtliche Ergebnisse werden im *JSON*-Format in die Standardausgabe geschrieben und können beispielsweise in eine Datei oder direkt in den `report` Befehl weitergeleitet werden.

Das *subcommand* `report` erhält per Standardeingabe eine Datei im *JSON*-Format und generiert daraus einen *HTML*-Bericht, der die Ergebnisse visualisiert und ebenfalls per Standardausgabe ausgibt.

Die *subcommands* `process` und `contamination` besitzen jeweils ein positionales und eine Reihe an optionalen Argumenten, die in Tabelle 7.1 aufgelistet sind. Darüber können bestimmte Parameter verändert und optionale Eingaben getätigt werden. Optionale Argumente sind durch `--` markiert.

Option	Beschreibung
`fastq`	legt die Eingabe *FASTQ*-Datei fest (nur für `process`)
`fasta`	legt die Eingabe *FASTA*-Datei fest (nur für `contamination`)
`--reference`	Hashtabelle mit Referenzgenomen (nur für `process`)
`--threadcount`	gibt an wie viele Threads genutzt werden sollen
`--kmer`	bestimmt die Größe der k-mere
`--canonical`	deaktiviert die Verwendung von kanonischen k-meren
`--config`	Config-Datei für die Hashtabelle
`--nohash`	nur für interne Tests, deaktiviert die Nutzung der Hashtabelle (nur für `process`)
`--output`	legt die Ausgabe Datei fest (nur für `contamination`)
`--size`	legt die Größe der Hashtabelle fest

Tabelle 7.1: Liste der Kommandozeilenargumente

7.2.3 Processing

Der Hauptschritt bei der Verarbeitung kann mittels des *subcommand* `yafqc process` ausgeführt werden. Hierbei wird eine angegebene *FASTQ*-Datei eingelesen und jeder *read* einzeln verarbeitet. Für das Einlesen der Datei wurden die bereits für den *lcfilter* 4 angepassten Funktionen für das Einlesen einer einzelnen *FASTQ*-Datei aus dem Paket *Xengsort*

wiederverwendet. Hierbei wurden die *linemarks*, die Start- und End-Indizes der Sequenzen und *reads* enthalten, um einen Eintrag für den Start-Index der Qualitätswerte ergänzt.

Der k-mer *processor* aus den Miniprojekten wurde für *YAFQC* ebenfalls wiederverwendet. Dieser wurde so angepasst, dass die als *processor* verwendete Funktion keinen Rückgabewert mehr besitzt, sondern lediglich das jeweilige k-mer sowie ein Tupel weiterer Argumente erhält. Diese *processor*-Funktion wird genutzt, um die k-mere jedes *reads* in eine Hashtabelle zur späteren Verarbeitung einzutragen. Für die Generierung der Gecocoo-Hashtabelle wird eine Standard-Konfiguration verwendet, alternativ kann aber auch eine *YAML*-Konfigurationsdatei angegeben werden. Die erwartete Anzahl an Einträgen für die Hashtabelle wird abgeschätzt auf Basis der Größe der Eingabedatei und der Wahl des ks für die zu speichernden k-mere.

Im Weiteren werden die nötigen Daten für die Qualitätsmerkmale berechnet, wie in Abschnitt 7.2.1 beschrieben, und in einem jeweiligen *Array* eingetragen. Sofern eine Hashtabelle mit Referenzgenomen angegeben wird, wird ebenfalls eine Kontaminationsprüfung für die *FASTQ*-Datei ausgeführt, wie in Abschnitt 7.2.4 beschrieben.

Als Ausgabe erzeugt das *processing* eine *JSON*-Datei, die per Standardausgabe ausgegeben wird. Dies ermöglicht größtmögliche Flexibilität zur Nutzung von *YAFQC* im Rahmen eines Workflows.

7.2.4 Kontaminationsprüfung

Die Kontaminationsprüfung läuft in zwei Schritten ab. Im ersten Schritt wird mit dem *subcommand* `yafqc contamination` eine Referenzhashtabelle konstruiert anhand der angegebenen Referenzgenome. Zur Generierung der Referenzhashtabelle können bis zu 64 verschiedene *FASTA*-Dateien als Referenzgenome angegeben werden. Dabei wird die Standard-Konfiguration verwendet, alternativ kann diese aber auch durch Eingabe optionaler Parameter oder einer *YAML*-Konfigurationsdatei angepasst werden.

Es wird empfohlen, den Parameter `--size` für die Größe der Referenzhashtabelle anzugeben, da ansonsten nur eine grobe Approximation der Anzahl k-mere erfolgt. Das Tool *ntCard* [16] kann beispielsweise dafür genutzt werden, die Anzahl der möglichen k-mere in den Referenzgenomen abzuschätzen. Das Ergebnis kann dann *YAFQC* als Parameter übergeben werden.

Zu Beginn werden die *FASTA*-Dateien in der angegebenen Reihenfolge eingelesen und mithilfe eines angepassten k-mer *processors* aus den Miniprojekten in die Referenzhashtabelle eingespeichert. Außerdem wird ein neues *value module* zum Abspeichern der k-mere mithilfe von *Bitflags* benutzt. Dabei wird für jede angegebene *FASTA*-Dateien ein *Bitflag* gesetzt, sodass rekonstruiert werden kann, aus welchen Dateien das k-mer stammt. Nach Abschluss der Konstruktion der Referenzhashtabelle wird diese serialisiert und als Datei abgespeichert.

Im zweiten Schritt mit dem *subcommand* `yafqc process` wird die angegebene *FASTQ*-Datei, wie in Abschnitt 7.2.3 erläutert, prozessiert. Um die Kontaminationsprüfungs durchzuführen, muss mit dem Parameter `--reference` eine Referenzhashtabelle angegeben werden. Wird ein k als Parameter übergeben, wird für die Kontaminationsprüfungs automatisch das k priorisiert, mit der die Referenzhashtabelle konstruiert wurde.

Die *reads* werden anhand von zwei Kriterien den verschiedenen Referenzgenomen zugeordnet. Das erste Kriterium basiert darauf, wie viele k-mere, die aus einem *read* aus den *FASTQ*-Dateien stammen, auch in einem Referenzgenom vorkommen. Das zweite Kriterium zählt, wie viele Basen innerhalb des *reads* durch die k-mere aus einem Referenzgenom abgedeckt werden. Diese Informationen werden, wie in Abschnitt 7.2.3 angegeben, als zusätzliche Qualitätsmerkmale abgespeichert und in der *JSON*-Datei mit ausgegeben.

7.2.5 Report

Um die Ergebnisse der vorherigen Berechnungen der Qualitätsmerkmale, sowie der Kontaminationsprüfungs nutzerfreundlich aufzubereiten, besteht die Möglichkeit, mit einem weiteren *subcommand* `yafqc report` einen *HTML*-Bericht zu erzeugen. In diesem werden die Daten kompakt in unterschiedlichen, an das jeweilige Qualitätsmerkmale angepassten Diagrammen dargestellt. Zur Erstellung der Visualisierungen wird das *JavaScript*-Framework *Vega-Lite*[22] genutzt. *Vega-Lite* bietet eine kurze und prägnante Grammatik, basierend auf *D3*[3], im *JSON*-Format mit der sich auch kompliziertere Visualisierungen mit wenigen Zeilen Code realisieren lassen. Abbildung 7.1 zeigt beispielhaft eine Visualisierung eines Qualitätsmerkmals mit *Vega-Lite*. Mithilfe der *Template Engine jinja2* [8] werden daraufhin die Visualisierungen in Form von *JSON*-Objekten in eine *HTML*-Datei gerendert. *HTML*-Reports sind leicht zugänglich, da lediglich ein funktionierender Web-Browser benötigt wird und können somit einfach mit Anderen geteilt werden. Durch die Trennung von Visualisierung und der Berechnung der Daten ist dem Nutzer selbst überlassen, ob und wann die Visualisierung erfolgen soll, wodurch *YAFQC* flexibel eingesetzt werden kann und somit für die Nutzung in einem Workflow optimiert ist. Im Vergleich zu einer statischen PDF-Datei bietet der *HTML*-Report mit optionalen *Tooltips* oder *Highlighting* in den Visualisierungen zusätzliche Interaktionsmöglichkeiten.

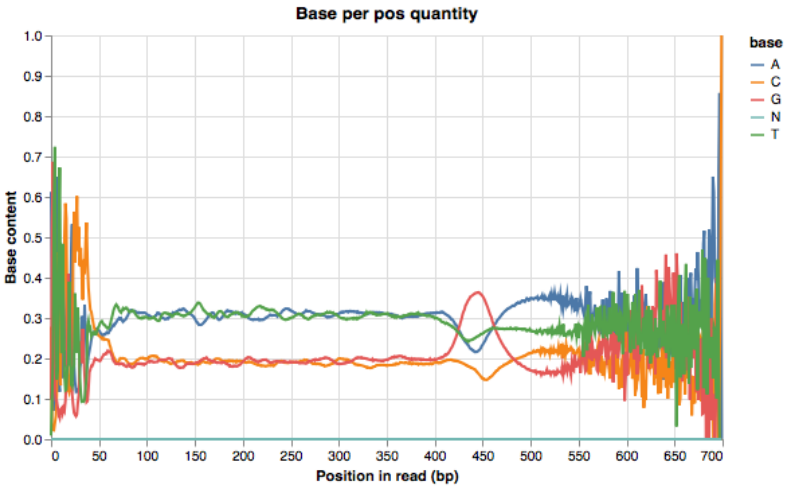


Abbildung 7.1: Visualisierung der Verteilung der Nukleinbasen

[8] Jinja Dokumentation

7.3 Benchmarks

In diesem Abschnitt wird die Performanz von *YAFQC* gemessen und interpretiert. Dafür wurde ein Benchmark-Plan erstellt, der vorsieht, zur Evaluation des Tools verschiedene Konfigurationen der Hashtabelle miteinander zu vergleichen. Dabei wird zum Einen die für die Konstruktion der Referenz-Hashtabelle benötigte Zeit gemessen als auch die Zeit zur Berechnung der Qualitätsmerkmale, inklusive der Kontaminationsprüfung mit der zuvor generierten Referenz-Hashtabelle.

7.3.1 Testdaten

Für die Benchmarks wurde die Referenztabelle aus einer Reihe von *FASTA*-Dateien erstellt. Zum Einen wurde der *primary assembly* eines Humangenoms eingelesen, im Folgenden abgekürzt als *Human*[9]. Zum Anderen wurden Sammlungen von verschiedenen Bakterien[10], Viren[11] und Pilzen[12] jeweils als Referenzgenome eingelesen. Hierbei ist zu beachten, dass es sich um zusammengefasste Versionen der jeweiligen Genome handelt. Aus den Zusammenfassungen wurden jeweils nur die eindeutigen Genome gewählt, die den Versionsstatus *latest* und das *assembly level Complete Genome* besitzen. Für die Bakterien wurde zusätzlich die Auswahl der Genome mit $taxid \leq 100000$ beschränkt. Die einzelnen Bakterien-, Viren- und Pilzgenome wurden zu jeweils einer Datei zusammengefügt.

Zusätzlich wird noch die Phage *PhiX*[13] als Referenzgenom hinzugefügt. Diese dient als Qualitätskontrolle für die Kontaminationsprüfung. Der Nachweis von *PhiX* deutet auf einen Fehler in dem Tool oder eine fehlerhafte Kontaminationsanalyse hin.

Aus aktuellem Anlass wurde ein *SARS-CoV-2 sample*[14] gewählt, für das die Qualitätsmerkmale berechnet und die Kontaminationsprüfung durchgeführt wird. Das *sample* wurde von einem Menschen extrahiert und liegt im *FASTQ*-Format vor.

Aufgrund der zur Verfügung stehenden Ressourcen, auf die in Kapitel 7.3.3 näher eingegangen wird, wurde, wie bereits beschrieben, die Auswahl der Genome eingeschränkt. Durch diese Einschränkungen ergeben sich folgende Dateigrößen für die Referenzgenome, aus denen im Anschluss die Referenztabelle konstruiert wird 7.2.

Datei	Größe in MiB
human.fasta	3006
bacteria.fasta	5018
viral.fasta	365
fungi.fasta	197
phix.fasta	1
corona.fastq	276

Tabelle 7.2: Dateigrößen der Referenzgenome

[9] `http://ftp.ensembl.org/pub/release-102/fasta/homo_sapiens/dna/Homo_sapiens.GRCh38.dna_sm.primary_assembly.fa.gz`

[10] `https://ftp.ncbi.nlm.nih.gov/genomes/refseq/bacteria/assembly_summary.txt`

[11] `https://ftp.ncbi.nlm.nih.gov/genomes/refseq/viral/assembly_summary.txt`

[12] `https://ftp.ncbi.nlm.nih.gov/genomes/refseq/fungi/assembly_summary.txt`

[13] `ftp://ftp.ncbi.nlm.nih.gov/genomes/Viruses/enterobacteria_phage_phix174_sensu_lato_uid14015/NC_001422.fna`

[14] `https://sra-download.ncbi.nlm.nih.gov/traces/sra65/SRR/012944/SRR13255543`

7.3.2 Konfigurationen

Da die Hashtabelle unterschiedliche Konfigurationen bietet, wurde die Konstruktion der Referenztabelle mit verschiedenen Kombinationen von Parametern ausgeführt. Die Tabelle 7.3 zeigt die dafür gewählten Optionen. Hierbei wurden Werte genommen, die für eine sinnvolle Anwendung realistisch erschienen. Die Benchmarks wurden mit 16 Threads ausgeführt. Die Referenztabellengröße hängt dabei von der Anzahl der verschiedenen k-mere aus den Referenzgenomen ab. Diese kann beispielsweise mit dem Tool *ntCard* [16] berechnet werden. Durch Kombination der verschiedenen Parameter ergeben sich 27 verschiedene Konfigurationen 7.3.

Option	Werte
Bucketgröße	$\{4, 8, 12\}$
Anzahl Hashfunktionen	$\{2, 3, 4\}$
k-mer Größe	$\{25, 27, 31\}$
Füllgrad	0.95
Verschiedene k-mere	$\{25 : 5.984.845.496,$ $27 : 6.031.233.027,$ $31 : 6.261.140.111\}$
Threads	16

Tabelle 7.3: Konfigurationswerte für Hashtabellen

7.3.3 Ausführung

Die Benchmarks werden auf dem dem Linux-HPC-Cluster der Technischen Universität Dortmund (*LiDO3*) durchgeführt. Mit Hilfe des Tools *Snakemake* wird pro generierter Konfiguration jeweils ein Knoten des Clusters angefragt, auf denen dann zwei Benchmarks hintereinander ausgeführt werden. Auf diese Weise können die verschiedenen Konfigurationen gleichzeitig auf unterschiedlichen Knoten des Clusters laufen, ohne sich gegenseitig zu beeinflussen.

Der erste Benchmark misst die Zeit zur Konstruktion der Hashtabelle aus den Referenzgenomen aus Abschnitt 7.3.1 mit einer der generierten Konfigurationen aus Abschnitt 7.3.2. Dabei werden zu Beginn die Daten (*i.e.* die Referenzgenome sowie das *sample*), die sich bereits heruntergeladen in einem Netzwerkverzeichnis befindet, auf den zugewiesenen Knoten kopiert, damit es zu keinen Performanzunterschieden durch eine mögliche Netzwerkauslastung kommt. Im Anschluss wird die Referenztabelle mit den Optionen aus der zugewiesenen Konfiguration erstellt. Für jede Konfiguration wird die Konstruktion der Referenztabelle 10 mal ausgeführt und jeweils die benötigte Zeit gemessen. Diese wird mit 16 Threads ausgeführt. Auf diese Weise soll die Konstruktionszeit der Referenztabelle mit verschiedenen Konfigurationen untersucht werden.

Der zweite Benchmark verwendet die zuvor konstruierten Referenztabellen für die Kontaminationsprüfung. Dabei wird automatisch das k verwendet, mit dem auch die Referenztabelle erzeugt worden ist. Zusätzlich werden die Qualitätsmerkmale für das *sample* berechnet und der Bericht erstellt. Hierbei wird der Benchmark auch 10 mal auf 16 Threads ausgeführt und jeweils die benötigte Zeit gemessen. Mit diesem Benchmark wird die Zeit zur Berechnung der Qualitätsmerkmale sowie der Kontaminationsprüfung und die Zeit zur Erstellung des Berichts gemessen.

Beide Benchmarks werden nacheinander auf demselben Knoten durchgeführt, sodass es zu keinen Performanzunterschiedenen innerhalb einer Konfiguration durch unterschiedliche Hardware kommt.

LiDO3 stellt verschiedene Knoten zur Berechnung zur Verfügung. Diese weisen jedoch unterschiedliche Hardware, wie etwa Prozessor oder Arbeitsspeicher, auf. Um eine Vergleichbarkeit der Ergebnisse, die auf verschiedenen Knoten entstanden sind, zu gewährleisten, werden die zur Ausführung gewählten Knoten eingegrenzt. Dabei werden Knoten mit dem gleichen Prozessor und 60 GB Arbeitsspeicher angefordert. Der Prozessor ist somit immer ein Intel Xeon E5-2640 v4 mit 10 Kernen. Dieser ist in zwei Sockeln verbaut, sodass insgesamt 20 Kerne zur Verfügung stehen. Hyper-Threading ist für die Prozessoren deaktiviert.

7.3.4 Ergebnisse

Die Benchmarkergebnisse der Ausführung aus Abschnitt 7.3.3 sind unterteilt in Laufzeiten und Speicherbedarf. Die Laufzeitergebnisse werden erläutert in Abschnitt 7.3.4 und die Ergebnisse bezüglich des Speicherbedarfs werden erläutert in Abschnitt 7.3.4.

Laufzeit

Die Abbildung 7.2 zeigt die Laufzeiten mit verschiedenen Konfigurationen, welche in Tabelle 7.3 beschrieben worden sind. In der Abbildung werden jeweils die Laufzeiten für jede Konfiguration und jeden Durchlauf des Experiments als einzelner Datenpunkt dargestellt. Die Anzahl der Hashfunktionen ist über verschiedene Symbole und die Anzahl der Buckets über unterschiedliche Farben codiert. Bei der k-mer Größe von 25 sind keine größeren Laufzeitunterschiede in den Konfigurationen zu sehen, diese betragen alle etwa 200 s. Bei einer Größe von 27 und 31 ist ein Trend zu erkennen, dass die Laufzeit jeweils mit der Anzahl der Buckets abnimmt. Die Anzahl der Hashfunktionen hat keinen größeren Einfluss auf die Laufzeiten, wobei eine leichte Steigung zu sehen ist in einigen Konfigurationen.

In der Abbildung 7.3 sind die Laufzeiten zur Generierung der Hashtabelle mit den Konfigurationen aus Tabelle 7.3 und den Referenzgenomen aus Tabelle 7.2 gezeigt. Die Anzahl der Hashfunktionen ist über verschiedene Symbole und die Anzahl der Buckets über unterschiedliche Farben codiert. Jeder Durchlauf des Experiments wird als einzelner Datenpunkt gezeigt. Es ist zu erkennen, dass ein größeres k die Laufzeit bei der Generierung erhöht. Im Allgemeinen ist ein wachsender Trend der Laufzeiten bei einer Erhöhung der Hashfunktionen bei gleicher Bucketgröße zu erkennen. Die Änderung der Bucketgrößen können je nach Anzahl der Hashfunktionen unterschiedliche Trends zeigen, wobei diese in den meisten Fälle steigend sind. Im Fall von $k = 25$ und $k = 31$ ist eine Verbesserung der Laufzeit bei zwei Hashfunktionen und der Erhöhung der Bucketgröße zu sehen. Die Bucketgröße von 8 hat in den meisten Konfigurationen die beste Laufzeit. Die Konstruktionszeiten für die Referenztabelle liegen zwischen ca. 800 s und 1200 s.

Es zeigt sich, dass die Laufzeiten bei der Generierung der Referenztabelle stark von der Anzahl der Hashfunktionen abhängen. Bei der Kontaminationsprüfung als auch bei der Berechnung der Qualitätsmerkmale ist die Laufzeit stark von der Bucketgröße abhängig.

Speicherplatz

Zusätzlich zur Laufzeit wird der geforderte Speicherplatz der generierten Referenztabelle, welche mit den Daten aus Tabelle 7.2 konstruiert worden ist, evaluiert. Abbildung 7.4 zeigt die berechneten Größen der Referenztabellen unter den verschiedenen Konfigurationen aus Tabelle 7.3. Ebenfalls haben die Bucketgrößen bis auf den Fall $k = 27$ und die Bucketgröße von 4 keinen weiteren Einfluss auf die Größe der Hashtabelle. Dieser Fall braucht mehr Speicherplatz als die Konfigurationen mit $k = 25$, aber benötigt um einiges weniger Speicherplatz als die restlichen Fälle mit $k = 27$. Bei Konfigurationen mit $k = 25$ ist

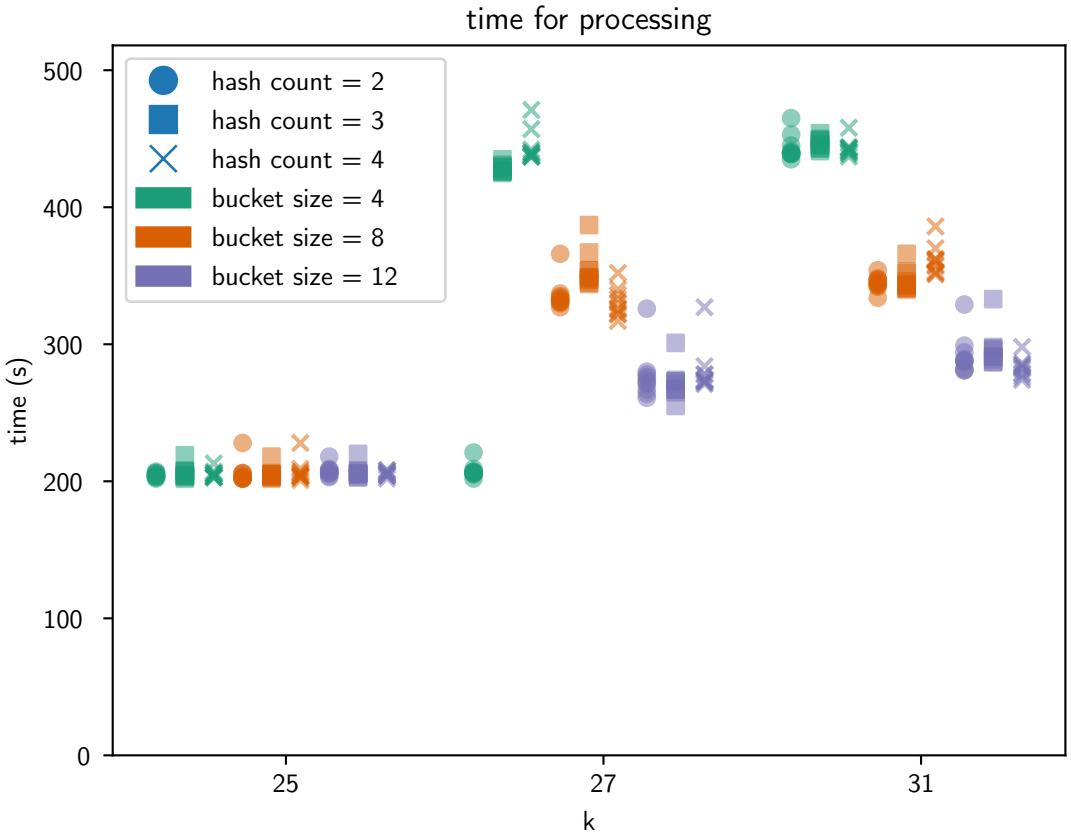


Abbildung 7.2: Benchmarkergebnisse der Laufzeit des Aufrufs `yafqc process` mit verschiedenen Konfigurationen auf dem *SARS-CoV-2 sample*.

der Speicherbedarf in jeder Konfiguration gleich groß, nämlich 24 031,97 MiB. Die Konfigurationen mit $k = 31$ und einer Bucketgröße von 4 zeigen die größten Referenztabellen mit 50 282,85 MiB. Durch die Abbildung wird deutlich, dass der benötigte Speicherplatz sich mit der Erhöhung der Bucketgrößen verringert und mit einem größeren k wächst. Die Größen der Referenztabelle können ebenfalls die Laufzeiten aus Abbildung 7.2 erklären, da die Laufzeit bei der Suche während der Kontaminationsprüfung stark von der Größe der Referenztabelle abhängig ist.

7.3.5 Kontaminationsprüfung

Um die Qualität der Kontaminationsprüfung zu untersuchen, wurde eine *FASTQ*-Datei eines Menschen genommen, der mit einem unbehandelten Epstein-Barr-Virus (EBV) infiziert ist[15]. Dabei wurden die Referenzgenome aus Abschnitt 7.3.1 verwendet. Zu beachten ist, dass das Referenzgenom *viral* das EBV-Genom enthält. Für die Kontaminationsprüfung wurde $k = 31$ gewählt. Die Abbildungen 7.5 und 7.6 sind die Visualisierung der Kontaminationsprüfung aus dem generierten Bericht. In Abbildung 7.5 wird deutlich, dass der Großteil der *reads* dem Referenzgenom *viral* und danach *human* zugeordnet werden. Die meisten *reads* werden zu 99% bis 100% durch k-mere abgedeckt. Ein weiter großer Ausschlag befindet sich bei 81%. Zusätzlich gibt es einen kleinen Ausschlag für *phix* im

[15] `https://sra-downloadb.be-md.ncbi.nlm.nih.gov/sos1/sra-pub-run-5/SRR1501064/SRR1501064.1`

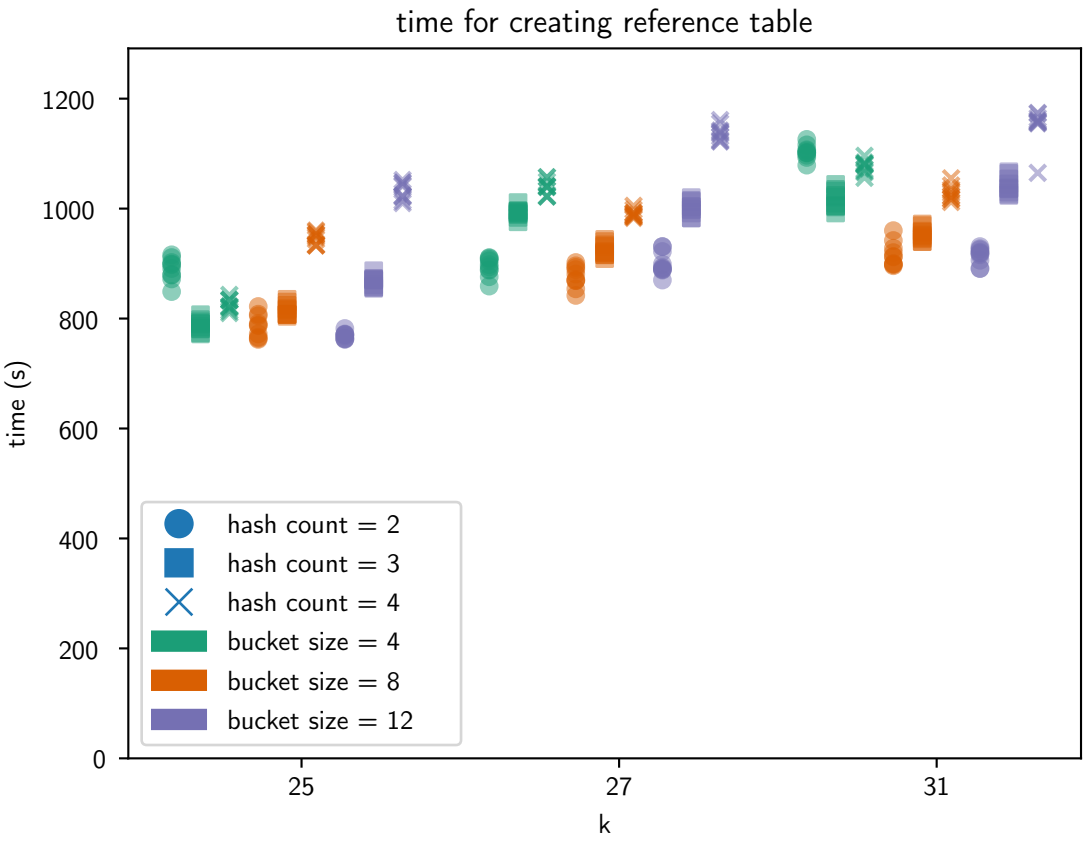


Abbildung 7.3: Benchmarkergebnisse des Konstruktionszeit der Referenztabelle mit den in Tabelle 7.2 angegebenen Referenzgenomen.

Bereich von 90% Überdeckung der k-mere, dies ist ein geringer Teil der *reads* und kann auf die Ähnlichkeit zu viralen Genomen zurückzuführen sein. Insbesondere existieren keine Ausschläge für die Referenzgenome *fungi* und *bacteria*.

Eine genauere Untersuchung bezüglich Kontaminationen kann durchgeführt werden, indem zusätzlich die Abbildung 7.6 betrachtet wird. Dort wird die prozentuale Überdeckung der Basen eines *reads* dargestellt. Für das Referenzgenom *viral* steigt die Basenabdeckung eines *reads* ab 50% stetig. Der Großteil der *reads* ist zu $99\% - 100\%$ abgedeckt. Des Weiteren gibt es außerdem einen kleinen Anteil an Überdeckungen aus dem Referenzgenom *human*, wobei *reads* zu 100% überdeckt werden. Das zeigt, dass einige *reads* in dem *EBV-sample* menschlichen Ursprungs sind. Dies ist logisch, da die Probe einem Menschen entnommen worden ist. Zusätzlich existieren keine Überdeckungen bezüglich der Referenzgenome *fungi* und *bacteria*, wodurch sich zeigt, dass es keine Kontamination durch diese gibt. Außerdem gibt es für das Referenzgenom *phix* einen kleinen Ausschlag bei 99%. Dies kann auf eine große Ähnlichkeit zu dem *viral* Referenzgenom zurückzuführen sein.

Abschließend kann daraus geschlossen werden, dass die Mehrheit der *reads* aus dem *EBV-sample* dem Referenzgenom *viral* zuzuordnen ist, gefolgt von *human*.

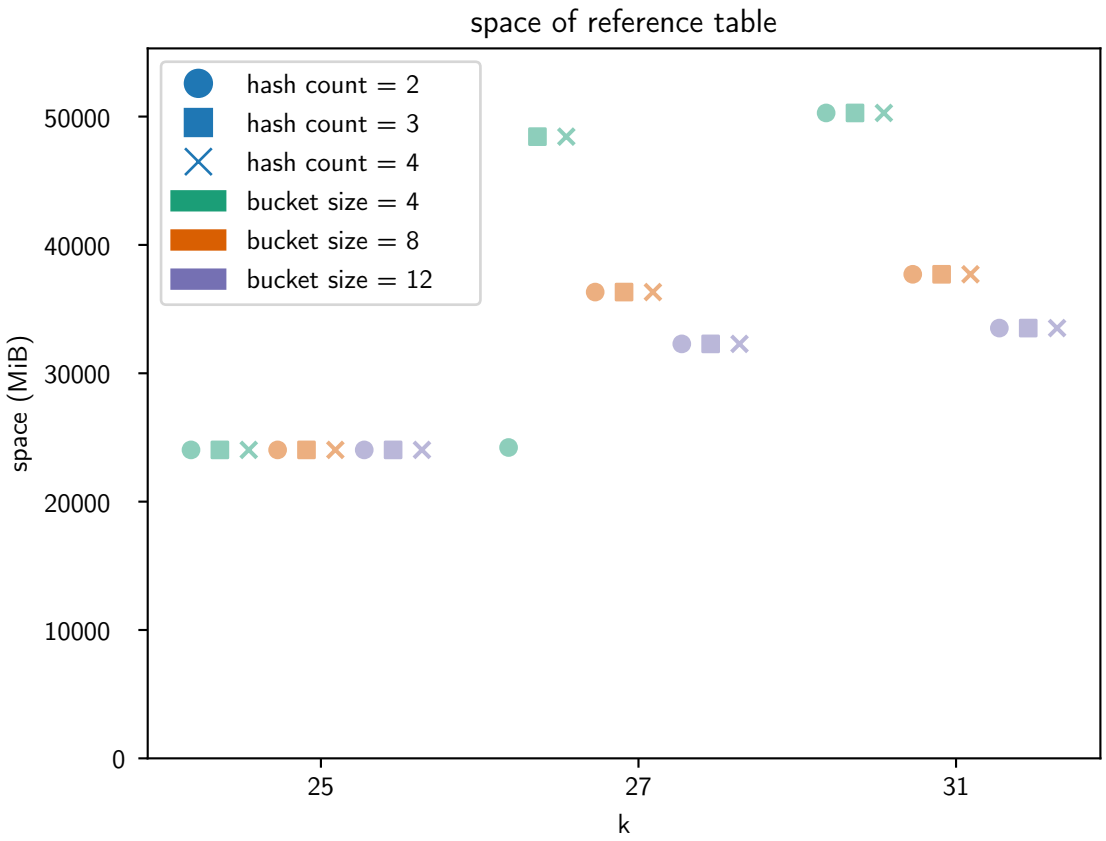


Abbildung 7.4: Benchmarkergebnisse der Speichergröße der Referenztabelle mit den in 7.2 Referenzgenomen.

7.4 Vergleich mit den anderen Tools

In Abschnitt 7.1 wurden drei Tools aus dem Bereich der *FASTQ*-Qualitätskontrolle vorgestellt, die als Inspiration für die Entwicklung von *YAFQC* dienen sollten. In diesem Kapitel wird nun thematisiert, inwiefern sich *YAFQC* von diesen drei Tools unterscheidet. Zusätzlich wurde mit dem Tool *Snakemake* ein *Workflow* erstellt, das die Laufzeit von *YAFQC*, *FastQC* und *fastq-scan* vergleicht. Dazu wurden die drei Tools mit einer 7,3 GB großen Datei jeweils zehn mal ausgeführt. Der für den Vergleich verwendete Server ist mit 16 GB RAM und einem Intel i3-9100 Prozessor ausgestattet. Die Ergebnisse sind in Abbildung 7.7 zu sehen.

7.4.1 FastQC

Wie bereits in Abschnitt 7.1.1 erläutert, berechnet *FastQC* diverse Qualitätsmerkmale für Sequenzdaten und stellt die Ergebnisse graphisch dar. *FastQC* akzeptiert die Dateiformate *FASTQ*, BAM und SAM. Von *YAFQC* wird lediglich das Dateiformat *FASTQ* akzeptiert.

Sämtliche Qualitätsmerkmale, die in *YAFQC* implementiert sind, enthält *FastQC* ebenfalls mit Ausnahme der Kontaminationsprüfung. Merkmale, die von *YAFQC* nicht übernommen wurden, sind: *sequence duplication levels*, *overrepresented sequences*, *adapter content* und *per tile sequence quality*. Der Grund hierfür ist, dass der Hauptfokus von *YAFQC* auf der Verwendung der Gecocoo-Hashtabelle liegen sollte, die allerdings nur für *k-mer content* und die Kontaminationsprüfung sinnvoll zu verwenden ist. Außerdem können mit

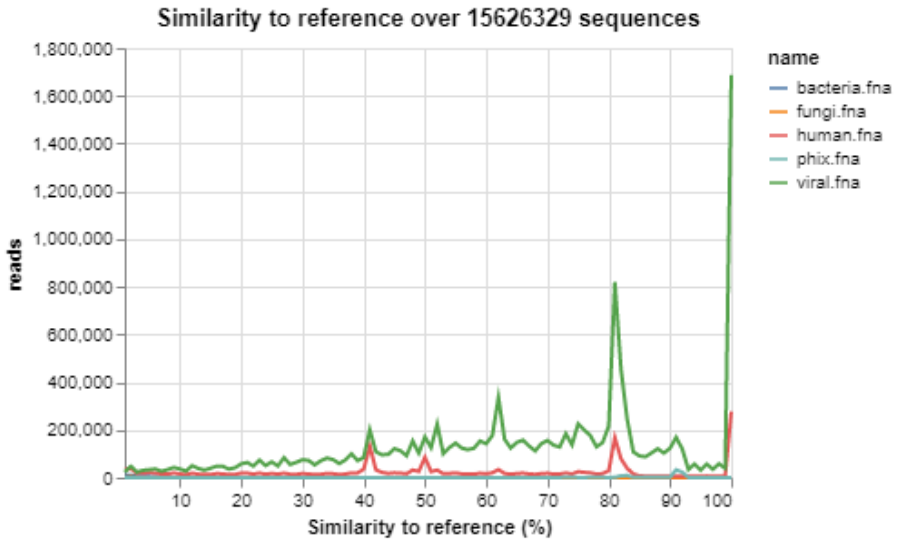


Abbildung 7.5: Anzahl der k-mere aus einem *read*

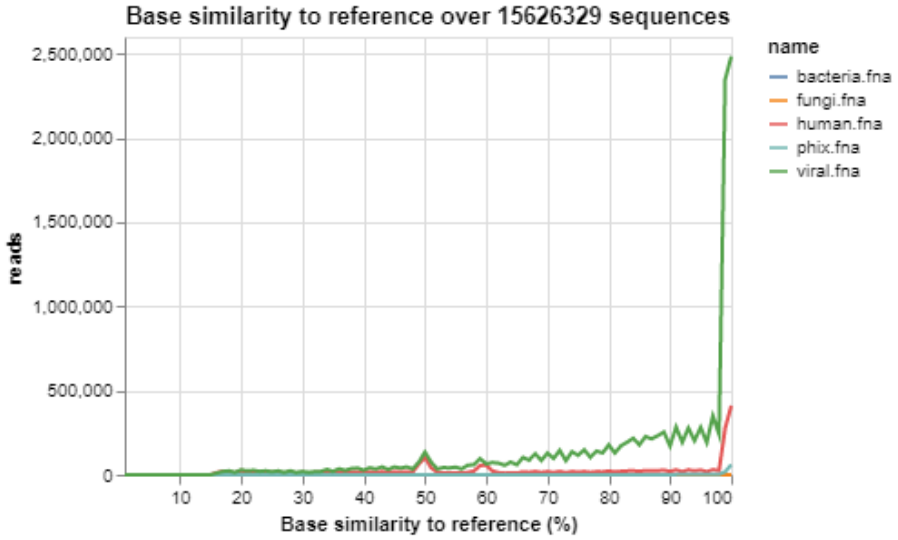


Abbildung 7.6: Basenabdeckung pro *read*

der Kontaminationsprüfung Adaptersequenzen ebenfalls gefunden werden, wie bei dem Merkmal *adapter content* das Ziel ist. Ein anzumerkender Vorteil von *YAFQC* gegenüber *FastQC* ist, dass bei den k-mer basierten Techniken sämtliche k-mere der Eingabedatei verarbeitet werden. *FastQC* verarbeitet im Gegensatz dazu nur 2% der k-mere.

Für einen groben Vergleich der Laufzeiten wurde *FastQC* auf einem lokalen Rechner ausgeführt. Für eine knapp 16GB große *FASTQ*-Datei[16] betrug die Laufzeit 3 Minuten 20 Sekunden. Hochgerechnet auf 100% der k-mere ergäbe sich eine Laufzeit von 166 Minuten 40 Sekunden. Die Analyse der selben Datei mittels *YAFQC* (ohne Kontaminationsprüfung) dauerte etwa 20 Minuten. Hier zeigt sich eine deutlich höhere Verarbeitungsgeschwindigkeit bei *YAFQC*, die die Analyse sämtlicher k-mere erlaubt (s. Abbildung 7.7).

Die Ergebnisse von *FastQC* und *YAFQC* können beide als *HTML*-Reports exportiert werden. Hier zeigt *YAFQC* den Vorteil der Hervorhebung einzelner Linien zur besseren Sichtbarkeit in bestimmten Grafiken. Außerdem ermöglicht *FastQC* ein Exportieren der Ergebnisse im *TSV*-Format, während *YAFQC* einen Export im *JSON*-Format ermöglicht.

[16] SRR9130495_1.fastq, siehe https://trace.ncbi.nlm.nih.gov/Traces/sra/?run=SRR9130495

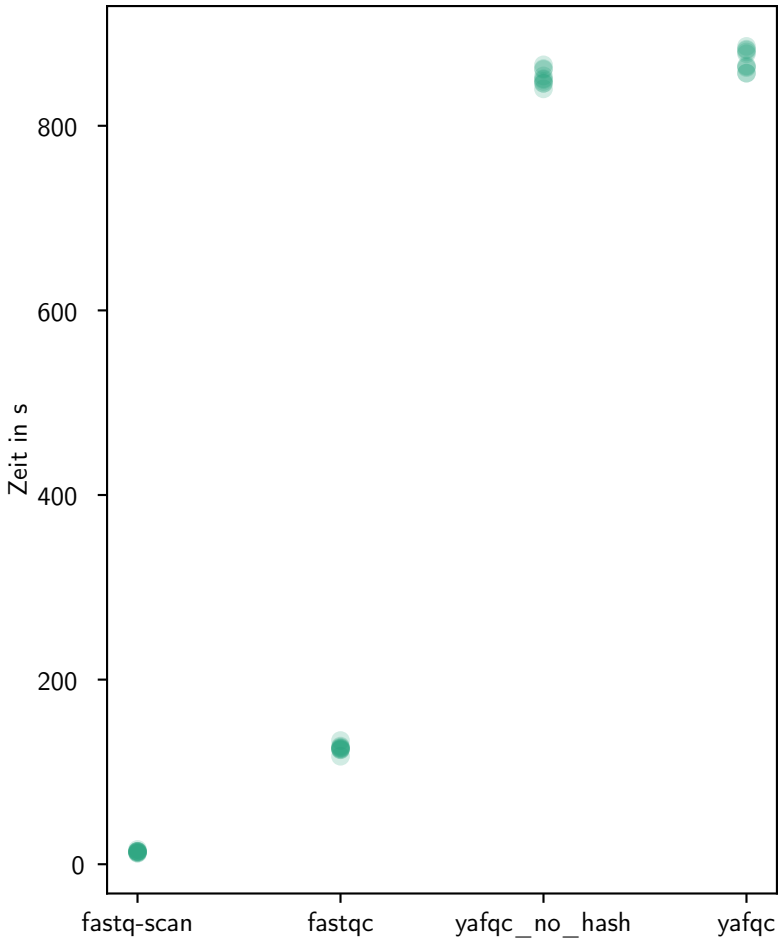


Abbildung 7.7: Vergleich der Laufzeiten von *YAFQC*, *FastQC* und *fastq-scan*

Dadurch lassen sich die Ergebnisse in Anwendungen wie beispielsweise *MultiQC* verwenden. *FastQC* bietet außerdem Warnungen oder Fehlermeldungen an, wenn Qualitätsmerkmale bestimmte Grenzwerte überschreiten.

7.4.2 FastQ Screen

FastQ Screen kann wie in Abschnitt 7.1.2 erwähnt eingesetzt werden, um durch Vergleich mit Referenzgenomen mögliche Kontaminationen in Sequenzdaten zu erkennen. Dazu nutzt es einen Aligner, um die Übereinstimmung zu ermitteln. *YAFQC* hingegen nutzt Hashing der k-mere, um wie in Anschnitt 7.2.4 beschrieben, die Anteile der übereinstimmenden k-mere (genannt *similarity*) und deren Überdeckung der Basenpaare des *reads* (genant *base similarity*) festzustellen. Diese Informationen werden aber nicht zu einer Klassifizierung des *reads* genutzt. Stattdessen wird über alle *reads* aggregiert, wie hoch die Übereinstimmungen mit jedem Referenzgenom war und diese Verteilung dem Nutzer präsentiert. Das gehäufte Vorkommen einer unerwarteten Klasse mit hoher Übereinstimmung ist dann ein Zeichen für eine Kontamination. Damit soll es möglich sein zu erkennen, ob eine Kontamination

vorliegt.

Vergleich der Ausgaben

Beide Tools erzeugen Ausgaben in Textform und als *HTML*-Dokument. In Abbildung 7.8 sind Beispiel für die Diagramme in den *HTML*-Dokumenten der beiden Tools zu sehen. Die verwendete Probe besteht aus konzentrierten Sequenzen (mittels *target enrichment*) des Eppstein-Barr-Virus[17] (Human gammaherpesvirus 4) mit 2,4 GigaBasenPaaren (GBP). Die verwendeten Referenzgenome sind dieselben wie die in Kapitel 7.3.1. Wie in Abbildung (b) zu sehen ist, klassifiziert *FastQ Screen* den Großteil der *reads* als viral und einen kleineren Teil als menschlich, was den Erwartungen an den Datensatz entspricht. *YAFQC* hingegen erzeugt Abbildung (a), welche auf der x-Achse die Ähnlichkeit eines *reads* zu einem bestimmten Referenzgenom zeigt und auf der y-Achse die Häufigkeit dieser Ähnlichkeit. Diese Ähnlichkeit wird anhand der anteiligen Überdeckung der Basenpaare durch die gefundenen k-mere bestimmt. Der Ursprung der *reads* zeigt sich hier durch die Häufungen der Klassen *viral* und *human* mit höher Übereinstimmung. Mangels einer abschließenden Klassifizierung jedes *reads* sind die Ausgaben von *YAFQC* erheblich schwieriger zu deuten.

Außerdem ist *FastQ Screen* in der Lage, eine *FASTQ*-Datei zu filtern, um nur *reads* zu erhalten, welche mit bestimmten Referenzgenomen übereinstimmen oder eine Datei mit den ermittelten Anmerkungen zu erstellen. Diese Funktion wird von *YAFQC* nicht implementiert, da eine Klassifizierung der *reads* nicht vorgenommen wird.

Vergleich der Laufzeit

Der bedeutende Vorteil von *YAFQC* gegenüber *FastQ Screen* ist die Laufzeit. Für den Vergleich wurden die vorher erwähnten Sequenzen mit 2,4GBP und die besagten Referenzgenome genutzt. Als externer Aligner für *FastQ Screen* wurde *bowtie2* genutzt. Die Erstellung des Indexes durch *bowtie2* ist vergleichbar mit dem Aufbau der Referenztabelle, welches *YAFQC* mittels eines Aufrufs von `yafqc contamination` selbst tun kann. In Tabelle 7.9a ist zu sehen, dass *YAFQC* die Tabelle in etwa der Hälfte der Zeit erstellt, die *bowtie2* für den Index benötigt. Außerdem hat die Nutzung einer Hashtabelle einen Laufzeitvorteil gegenüber dem mehrfachen Aufruf eines Aligners, was sich anhand der Verarbeitung der Sequenzen in Tabelle 7.9b zeigt.

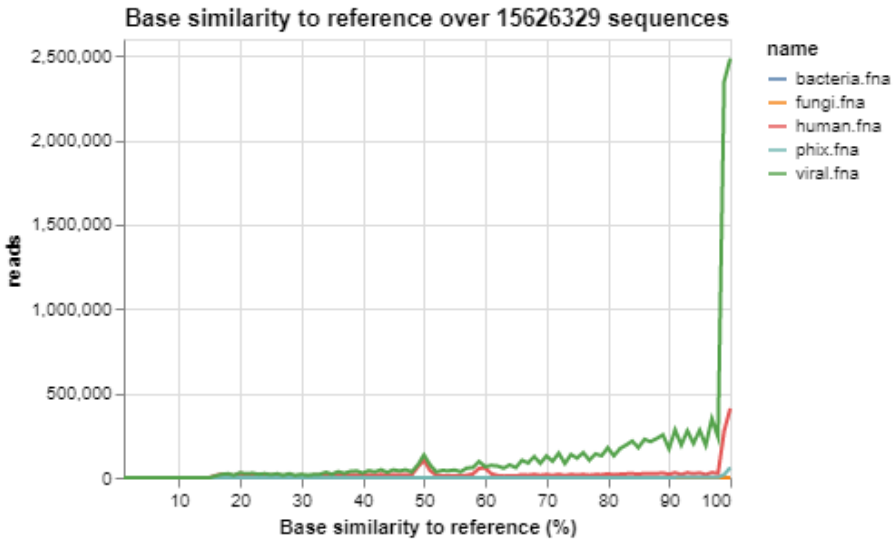
7.4.3 fastq-scan

Der größte Nachteil an *fastq-scan* ist, wie bereits in Abschnitt 7.1.3 angesprochen, dass nur die einfachsten Statistiken zu der Eingabe berechnet werden. Durch *YAFQC* lassen sich alle von *fastq-scan* berechneten Werte ebenfalls gewinnen. Zusätzlich werden noch weitere Statistiken wie der GC-Gehalt, die Verteilung von k-meren und die Häufigkeit der möglichen Basen berechnet. Ebenso ist mit *YAFQC* eine Kontaminationsprüfung mit gegebenen Referenzgenomen möglich, um Ähnlichkeiten zu typischen DNS-Sequenzen zu erkennen.

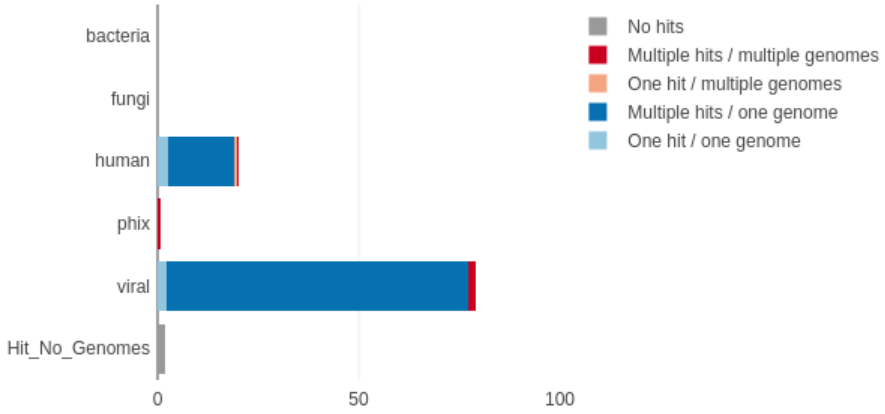
Wie die Benchmark-Ergebnisse in Abbildung 7.7 zeigen, ist *fastq-scan* deutlich schneller als *YAFQC*. Dies lässt sich einerseits dadurch erklären, dass *fastq-scan* weniger Statistiken berechnet. Andererseits benutzt der Autor nach eigenen Angaben einen Algorithmus, der darauf optimiert ist, große *FASTQ*-Dateien einzulesen. In diesem Punkt könnte *YAFQC* noch verbessert werden.

Beide Tools lesen die Eingabe von *STDIN* und geben die Ausgabe auf *STDOUT* im *JSON*-Format aus und können daher sehr gut in einen *Workflow* eingebunden werden.

[17] `https://www.ncbi.nlm.nih.gov/sra/SRX625465[accn]`



(a) *YAFQC* zeigt wie viele *reads* zu welchem Teil den Referenzgenomen ähneln



(b) *FastQ Screen* klassifiziert *reads*

Abbildung 7.8: Vergleich der Ausgaben von *YAFQC* und *FastQ Screen*

Allerdings kann *YAFQC* mit einer zuvor erzeugten *JSON*-Datei noch einen *HTML*-Report generieren, der die Ergebnisse graphisch darstellt.

Tool	Realzeit	CPU-Zeit
YAFQC	7:30	18:30
bowtie2	44:42	2:25:19

(a) Vergleich der Zeit zum Aufbau des Index/Tabelle

Tool	Realzeit	CPU-Zeit
YAFQC	14:37	41:27
FastQ Screen	28:25	1:42:24

(b) Vergleich der Laufzeit der Verarbeitung

Abbildung 7.9: Vergleich der Laufzeiten von *YAFQC* und *FastQ Screen*, ausgeführt mit 4 Threads, Format h:mm:ss

Kapitel 8 — Fazit

Ziel der Projektgruppe war es, eine Hashtabelle umzusetzen, die es Nutzern ermöglicht, Genomdaten in Form von k-meren effizient zu speichern und einen schnellen Zugriff auf diese zu ermöglichen. Im Folgenden sollen dazu zum Einen in der Zusammenfassung das Vorgehen, die Entwicklung und die Ergebnisse, die über zwei Semester von der Projektgruppe erarbeitet wurden, abschließend dargelegt werden und zum Anderen ein Ausblick über mögliche Optimierungen und Erweiterungen, die auf der Arbeit aufbauen könnten, besprochen werden.

8.1 Zusammenfassung

Zum Erreichen des Ziels wurden Lösungsansätze zu Teilproblemen erarbeitet. So wurden Miniprojekte initiiert, um grundlegende Herausforderungen bei der Verarbeitung von Genomdaten zu erörtern. Darauf folgend wurde der Grundstein der Implementierung in Form einer ersten exemplarischen Anwendung, dem Low Complexity Filter, gelegt und ein Konzept für die Hashtabelle erarbeitet. Abschließend wurde die Entwicklung der Hashtabelle umgesetzt, sowie die Anwendung dieser in einem Werkzeug zur Untersuchung von Genomdaten auf Qualitätsmerkmale realisiert.

8.1.1 Miniprojekte

Für die Bearbeitung der Projekte wurden drei Kleingruppen gebildet (siehe Kapitel 3).

Das Projekt „Codegenerierung für *FASTQ*-Dateiverarbeitung“ (siehe Abschnitt 3.1) betrachtete verschiedene Möglichkeiten zur Verarbeitung von *FASTQ*-Dateien. *FASTQ*-Dateien enthalten die DNA-Sequenzen, die sich mithilfe der zu entwickelnden Hashtabelle analysieren lassen sollen. Deshalb ist eine schnelle Verarbeitung solcher Dateien essenziell. Hierfür wurden Benchmarks zur Überprüfung der Effizienz zwischen generiertem und nicht-generiertem Code durchgeführt. Wobei nutzerdefinierter Code beliebige Verarbeitungsschritte ermöglicht. Die angestellten Untersuchungen konnten keine großen Differenzen der Ausführungszeiten feststellen.

Hierfür untersuchte das Projekt „JIT-kompilierte Funktionen und Klassen“ (siehe Abschnitt 3.2) die Unterschiede im Bezug auf Ausführungsgeschwindigkeit zwischen durch *Numba* kompilierte Klassen und Funktionen. Es bildet eine wichtige Grundlage für die Generierung von Hashtabellen, da generierter Code JIT-kompiliert werden soll und dieser eine möglichst schnelle Verarbeitung von großen Datenmengen ermöglichen soll. Hierbei war die Performance der JIT-kompilierten Funktionen etwas besser und zusätzlich bieten die Klassen noch nicht alle Funktionalitäten, die *Numba* bei Funktionen aufweist, an. Dadurch entschied sich die Projektgruppe bevorzugt JIT-kompilierte Funktionen statt Klassen für den weiteren Verlauf des Projekts zu nutzen.

Zuletzt wurde im Projekt „k-mer Processing“ (siehe Abschnitt 3.3) eine Lösung zur Unterteilung von DNA-Sequenzen in k-mere geschaffen. Die Analyse einer DNA-Sequenz mithilfe der zu entwickelnden Hashtabelle erfordert eine solche Unterteilung, da die einzelnen k-mere als Schlüssel innerhalb der Tabelle verwendet werden. Mit den Ergebnissen kann für jedes k-mer Code ausgeführt werden, der eine Verarbeitung vornimmt. Wichtig ist hierbei zu beachten, dass k ein Parameter ist und so vom Nutzer definiert werden kann.

8.1.2 Low Complexity Filter und Konzept der Hashtabelle

Auf der Grundlage der Erkenntnisse aus den Miniprojekten wurden zwei weiterführende Projekte realisiert. Das Projekt „Low Complexity Filter“ (siehe Kapitel 4) diente als erste reale Anwendung zur Verarbeitung von *FASTQ* Dateien. Die Anwendung ermöglicht das Unterscheiden von DNA-Sequenzen in Komplexitätsbereichen, welche eine erste Anwendung von Hashtabellen beschreibt. Parallel wurden von einer zweiten Gruppe Konzepte für die Hashtabelle entwickelt (siehe Kapitel 5).

Bei dem Projekt „Low Complexity Filter“ wurden vorherige Erkenntnisse aus den Miniprojekten „Codegenerierung für *FASTQ*-Dateiverarbeitung“ und „k-mer *Processing*“ genutzt, um eine Anwendung zur Trennung von DNA-Sequenzen nach deren Komplexität zu ermöglichen. Dazu wurde es ermöglicht, dass der k-mer *Processor* in der Dateiverarbeitung aufgerufen wird, um die k-mere in jedem Read zu zählen. Auf Basis der Anzahl der einzelnen sowie verschiedenen k-mere werden Schwellwerte berechnet, um die Reads zu klassifizieren. Diese werden in unterschiedliche Ausgabedateien geschrieben. Ein Standardgrenzwert wurde mittels manuell konstruierter Grenzfälle ausgerichtet. Zur Evaluierung wurde die Anwendung mit Filtermethoden vergleichbarer Programme verglichen. Dabei hat sich herausgestellt, dass die DNA-Sequenzen ähnlich getrennt wurden, allerdings von dem Projekt eine kürzere Laufzeit erzielt wurde.

Die Konzepte für die Hashtabelle dienen als Vorüberlegung für die Implementierung einer Hashtabelle. Dazu wurden einige Anforderungen gesammelt und diese zu mehreren Bestandteilen gruppiert. Dabei erfolgte einerseits die Definition einer Schnittstelle für die Nutzer der Hashtabelle. Daneben wurden die verschiedenen Konfigurationsmöglichkeiten festgestellt. Eine flexible Konfiguration ist für den effizienten Einsatz der Hashtabelle erforderlich. Ein erstellter Prototyp, mit dem Teile des Konzepts umgesetzt werden konnten, lieferte Aufschluss über unterschiedliche Implementierungsaspekte. Der Prototyp konnte somit Anhaltspunkte für die folgende Implementierung bieten.

8.1.3 Implementierung der Hashtabelle

Die großen Datenmengen, die bei der Verarbeitung von Genomdaten anfallen, erfordern eine schnelle Verarbeitung und eine effiziente Speicherung der Daten. Hierzu wurde eine konfigurierbare, speichereffiziente Hashtabelle implementiert, die vom Nutzer generiert werden kann. Die Konfiguration der Hashtabelle kann dabei programmatisch oder über eine Konfigurationsdatei erfolgen, wobei insbesondere auf den benutzerfreundlichen Umgang Wert gelegt wurde. Die gewählten Konfigurationsparameter werden bei der Generierung der Hashtabelle herangezogen.

Für die Zuordnung der Schlüssel zu einem Index bzw. einer Position in der Hashtabelle kam das Cuckoo-Hashing zum Einsatz, das mehrere Hashfunktionen zur Auflösung von Hash-Kollisionen nutzt. Es stehen hierbei vordefinierte Hashfunktionen zur Verfügung, zusätzlich besteht die Möglichkeit, in der Konfiguration eigene Hashfunktionen zu definieren und zu verwenden. Mit der Hashtabelle kann mittels der generierten Schnittstelle interagiert werden. Diese weist insbesondere Funktionen zum Einfügen, Suchen und Entfernen von Schlüssel-Werte-Paaren auf, wobei es sich bei den Funktionen selbst und den dahinter liegenden Implementierungen um von *Numba* generierte Funktionen handelt. Je nach Konfiguration durch den Nutzer wird so eine entsprechende Implementierung generiert. Die den Schlüsseln zugewiesenen Werte werden mittels Value Modulen berechnet. Analog zu den Hashfunktionen stehen hierbei vordefinierte Value Module zur Verfügung und es können benutzerdefinierte Value Module in der Konfiguration definiert werden. Bei der Implementierung musste stets zwischen den Kriterien der Zugriffsgeschwindigkeit und dem Speicherbedarf abgewogen werden. Da diese Kriterien anwendungsspezifisch unterschiedlich gewichtet werden können, sollten diese konfigurierbar gestaltet werden. Hierzu wurden

von der Hashtabelle drei verschiedene Speichermodi unterstützt, die dem Nutzer zur Wahl stehen. Um eine dauerhafte Speicherung der Hashtabelle zu gewährleisten, wurde zudem die Deserialisierung sowie Serialisierung dieser umgesetzt.

In der Untersuchung der Speichereffizienz und der Geschwindigkeit des Einfügens zwischen den verschiedenen Speichermodi war der `Buckets`-Modus der speichereffizienteste, aber zugleich langsamste. Der `Fast`-Modus war erwartungsgemäß der schnellste, aber benötigte den meisten Speicher. Vergleichend benötigte der `Fast`-Modus etwa dreimal mehr Speicher als der `Buckets`-Modus, aber war vier- bis fünfmal schneller. Bei dem Vergleich der kostenoptimalen und der Random Walk Einfügestrategie besitzt die kostenoptimale Einfügestrategie eine mindestens achtfache Laufzeit. Die Laufzeit des Einfügens verbessert sich bei der kostenoptimalen mit einer größeren Bucketgröße, wobei sich die Laufzeit beim Random Walk dadurch verschlechtert. Die Laufzeiten der kostenoptimalen Einfügestrategie sind ohne größere Varianzen, wobei der Random Walk größere Varianzen bei hohen Füllgraden besitzt. Das Suchen in den verschiedenen konfigurierten Hashtabellen ist sowohl beim Random Walk, als auch beim kostenoptimalen Einfügen sehr ähnlich, wobei wie zuvor der Random Walk höhere Varianzen besitzt. Das Multithreading führt je nach Konfiguration und Threadanzahl zu einer Verbesserung der Laufzeit um den Faktor 2 bis 20.

8.1.4 YAFQC

YAFQC stellt ein Tool dar, das die entstandene Hashtabelle nutzt, um eine Qualitätssicherung auf *FASTQ*-Dateien durchzuführen. Hierbei wurde der entstandene „Low Complexity Filter“ genutzt und um Funktionen erweitert. Zunächst wurden bestehende *Tools* analysiert und erörtert, welche Funktionen umgesetzt werden sollen, um einen Mehrwert zu generieren. So wurden Analysen mehrerer Qualitätsmerkmale und eine Kontaminationsprüfung umgesetzt. Dies umfasst die Untersuchung der DNA-Sequenzen hinsichtlich ihrer Länge, ihrer durchschnittlichen Qualität, ihrem GC-Gehalt und dem Auftreten von k-meren in den DNA-Sequenzen. Ebenso werden die Qualität und das Auftreten der Nukleinbasen pro Position der DNA-Sequenzen betrachtet.

Des Weiteren beinhaltet *YAFQC* eine Umsetzung einer Kontaminationsprüfung, welche auf zwei Merkmale aufgeteilt ist. Zur Kontaminationsprüfung müssen Referenzgenome als *FASTA*-Dateien übergeben werden. Die Prüfung läuft dabei in zwei Schritten ab. Im ersten Schritt wird die Hashtabelle genutzt, um eine Referenztabelle aufzubauen. Im zweiten Schritt werden *reads* der zu untersuchenden *FASTQ*-Datei sowohl anhand der Anzahl der vorkommenden k-mere als auch der Anzahl der Basen im *read* den verschiedenen Referenzgenomen zugeordnet.

Zudem wurde ein nutzerfreundliches *command line interface* erstellt. Das *subcommand* `contamination` dient dabei dem Erstellen der Referenztabelle, `process` ermittelt die Qualitätsmerkmale inklusive der Kontaminationsprüfung und `report` erstellt aus den zuvor erstellten Resultaten einen Bericht im *HTML*-Format. Der Bericht dient als kompakte Visualisierung der Daten. Hier werden die ermittelten Qualitätsmerkmale in Diagrammen veranschaulicht. Aufgrund des *HTML*-Formats kann der Bericht über den Browser betrachtet und somit für Dritte nutzerfreundlich zugänglich gemacht werden. Ebenso wurden *Tooltips* und *Highlighting* umgesetzt, sodass der Bericht mit zusätzlichen Informationen angereichert wurde, die interaktiv vom Betrachter eingesehen werden können.

Da die Laufzeit der Kontaminationsprüfung stark von der Größe der Referenztabelle abhängig ist, verhält sich die Laufzeit ähnlich dem Speicherverbrauch der Referenztabelle. Insbesondere erhöht die Anzahl der Hashfunktionen die Größe der Referenztabelle nicht, ein größeres k für die k-mere hingegen schon, da so mehr Kombinationen möglich sind. Die Bucketgröße verringert die Größe der Hashtabelle, was dadurch die Laufzeit der Kon-

taminationsprüfung mit der Berechnung der Qualitätsmerkmale beschleunigt. Eine höhere Anzahl an Hashfunktionen führt zu einer längeren Laufzeit bei der Generierung der Referenztabelle, aber zu keiner Laufzeitverbesserung bei der Kontaminationsprüfung.

YAFQC bietet im Vergleich zu schnelleren Tools wie *fastq-scan*, mehr statistische Informationen, sowie die Kontaminationsprüfung. Im Vergleich zu *FastQC*, welches eine bessere Laufzeit besitzt, werden bei *YAFQC* die gesamte Datei betrachtet und ähnliche Funktionen angeboten. *FastQ Screen* nutzt für die Kontaminationsprüfung einen *Aligner*, sodass zwei Tools genutzt werden müssen. Außerdem zeigt *FastQ Screen* eine schlechtere Laufzeit als *YAFQC*. Zusätzlich bietet *YAFQC* die Möglichkeit eines interaktiven *HTML*-Reports an, wodurch die Ergebnisse gut dargestellt werden und darauf aufbauend weitere Analysen geplant werden können.

8.2 Ausblick

Für die Verwendung der Hashtabelle ist, je nach Anforderungen der jeweiligen Anwendung, die Performance dieser entscheidend. Daher sind Optimierungen der Geschwindigkeit wünschenswert. *Cache-Misses*, die bei der Verwendung der Hashtabelle auftreten, verringern die Geschwindigkeit von Zugriffen auf die Hashtabelle. Zur Vermeidung von *Cache-Misses* könnte *Prefetching* zum Einsatz kommen. Hierbei werden Ressourcen, die zukünftig benötigt werden, bereits vor dem Verarbeitungsschritt aus dem ursprünglichen Speicher in den Cache geladen. Dazu muss bekannt sein, wann eine Ressource benötigt wird und entsprechende Befehle zum *Prefetching* ausgeführt werden. Beim *Cuckoo-Hashing* müssen zwangsläufig bei Hashkollisionen Elemente in der Hashtabelle verschoben werden. In der aktuellen Implementierung wird zur Suche einer freien Position randomisiert eine Position ausgewählt und überprüft, ob diese frei ist. Hier besteht die Möglichkeit, eine Breitensuche durchzuführen, um den kürzesten Weg zu einem freien Platz zu finden.

Für das Tool *YAFQC* wären insbesondere Erweiterungen, die mehr Erkenntnisse in Bezug auf die Qualitätssicherung liefern, von Interesse. *YAFQC* besitzt bereits eine Kontaminationsprüfung mittels Referenzgenomen. Hier könnte eine Erweiterung in Form einer Klassifizierung durchgeführt werden, die die einzelnen Sequenzen einer Klasse zuweist. Diese Klassen könnten auf den Referenzgenomen basieren, gegebenenfalls mit einer zusätzlichen „Unklar“-Klasse. Ebenso besteht die Möglichkeit, weitere Qualitätsmerkmale umzusetzen. Einige mögliche Merkmale sind diejenigen von *FastQC*, die bisher nicht von *YAFQC* umgesetzt wurden, wie in Abschnitt 7.4 beschrieben.

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Literatur

[1] Stephen F Altschul u. a. „Basic local alignment search tool“. In: *Journal of molecular biology* 215.3 (1990), S. 403–410. DOI: 10.1016/s0022-2836(05)80360-2.

[2] Kent Beck. *Test Driven Development. By Example (Addison-Wesley Signature)*. Addison-Wesley Longman, Amsterdam, 2002. ISBN: 0321146530.

[3] M. Bostock, V. Ogievetsky und J. Heer. „D^3 Data-Driven Documents“. In: *IEEE Transactions on Visualization and Computer Graphics* 17.12 (2011), S. 2301–2309. DOI: 10.1109/TVCG.2011.185.

[4] Peter J. A. Cock u. a. „Biopython: freely available Python tools for computational molecular biology and bioinformatics“. In: *Bioinformatics* 25.11 (März 2009), S. 1422–1423. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btp163.

[5] Thomas H. Cormen u. a. *Introduction to Algorithms*. 2nd. The MIT Press, 2001. ISBN: 0262032937.

[6] Thomas Dandekar und Meik Kunz. *Bioinformatik - Ein einführendes Lehrbuch*. Berlin Heidelberg New York: Springer-Verlag, 2017. ISBN: 978-3-662-54698-7.

[7] John Ellson u. a. „Graphviz and dynagraph – static and dynamic graph drawing tools“. In: *GRAPH DRAWING SOFTWARE*. Springer-Verlag, 2003, S. 127–148.

[8] *European '1+ Million Genomes' Initiative*. European Commission. 2020. URL: https://ec.europa.eu/digital-single-market/en/european-1-million-genomes-initiative (besucht am 15. 07. 2020).

[9] Python Software Foundation. *Python*. 2020. URL: https://docs.python.org/3/l.

[10] Johannes Köster und Sven Rahmann. „Snakemake — a scalable bioinformatics workflow engine“. In: *Bioinformatics* 28.19 (Aug. 2012), S. 2520–2522. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bts480. eprint: https://academic.oup.com/bioinformatics/article-pdf/28/19/2520/819790/bts480.pdf. URL: https://doi.org/10.1093/bioinformatics/bts480.

[11] Holger Krekel u. a. *pytest*. 2020. URL: https://docs.pytest.org/en/stable/.

[12] Siu Kwan Lam, Antoine Pitrou und Stanley Seibert. „Numba: A LLVM-Based Python JIT Compiler“. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450340052. DOI: 10.1145/2833157.2833162. URL: https://doi.org/10.1145/2833157.2833162.

[13] Chris Lattner und Vikram Adve. „LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation“. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, S. 75. ISBN: 0769521029.

[14] Chris Lattner und Vikram Adve. „LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation“. In: San Jose, CA, USA, März 2004, S. 75–88.

[15] *LLVM for Grad Students*. Adrian Sampson. 2015. URL: https://www.cs.cornell.edu/~asampson/blog/llvm.html (besucht am 26. 10. 2020).

[16] Hamid Mohamadi, Hamza Khan und Inanc Birol. „ntCard: a streaming algorithm for cardinality estimation in genomics data“. In: *Bioinformatics* 33.9 (Jan. 2017), S. 1324–1330. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btw832. eprint: https://academic.oup.com/bioinformatics/article-pdf/33/9/1324/25151243/btw832.pdf. URL: https://doi.org/10.1093/bioinformatics/btw832.

[17] Hamid Mohamadi u. a. „ntHash: recursive nucleotide hashing“. In: *Bioinformatics* 32.22 (), S. 3492–3494. ISSN: 1460-2059. DOI: 10.1093/bioinformatics/btw397.

[18] Aleksandr Morgulis u. a. „A fast and symmetric DUST implementation to mask low-complexity DNA sequences“. In: *Journal of Computational Biology* 13.5 (2006), S. 1028–1040. DOI: 10.1089/cmb.2006.13.1028.

[19] N. Nguyen und P. Tsigas. „Lock-Free Cuckoo Hashing“. In: *2014 IEEE 34th International Conference on Distributed Computing Systems*. 2014, S. 627–636. DOI: 10.1109/ICDCS.2014.70.

[20] *Numba tutorial for GTC 2017 conference*. GTC 2017. 2017. URL: https://github.com/ContinuumIO/gtc2017-numba (besucht am 09. 07. 2020).

[21] Calvin Giroud Ross Coker. *Lock-free Cuckoo Hash*. eourcs. 2021. URL: https://eourcs.github.io/LockFreeCuckooHash/finalreport (besucht am 04. 03. 2021).

[22] Arvind Satyanarayan u. a. „Vega-Lite: A Grammar of Interactive Graphics“. In: *IEEE Transactions on Visualization & Computer Graphics (Proc. InfoVis)* (2017). DOI: 10.1109/tvcg.2016.2599030. URL: http://idl.cs.washington.edu/papers/vega-lite.

[23] Robert Schmieder und Robert Edwards. „Quality control and preprocessing of metagenomic datasets“. In: *Bioinformatics* 27.6 (2011), S. 863–864. DOI: 10.1093/bioinformatics/btr026.

[24] Paul M. Selzer, Richard J. Marhöfer und Oliver Koch. *Angewandte Bioinformatik - Eine Einführung*. Berlin Heidelberg New York: Springer-Verlag, 2018. ISBN: 978-3-662-54135-7.

[25] *Sphinx - Python Documentation Generator*. Georg Brandl und the Sphinx team. 2020. URL: https://www.sphinx-doc.org/en/master/ (besucht am 23. 09. 2020).

[26] R Tatusov und DJ Lipman. *Dust*.

[27] Seq Team. *Seq*. 2020. URL: https://seq-lang.org/l.

[28] The Rust Release Team. *Rust*. 2020. URL: https://blog.rust-lang.org/2020/06/18/Rust.1.44.1.html.

[29] *The 100,000 Genomes Project*. Genomics England. 2018. URL: https://www.england.nhs.uk/healthcare-science/personalisedmedicine/ (besucht am 15. 07. 2020).

[30] Stéfan van der Walt, S Chris Colbert und Gael Varoquaux. „The NumPy array: a structure for efficient numerical computation“. In: *Computing in science & engineering* 13.2 (2011), S. 22–30.

[31] Stefan Walzer. *Load Thresholds for Cuckoo Hashing with Overlapping Blocks*. 2019. eprint: https://arxiv.org/pdf/1707.06855v3.pdf (cs.DS). URL: https://arxiv.org/abs/1707.06855v3.

[32] Jens Zentgraf und Sven Rahmann. „Fast lightweight accurate xenograft sorting“. In: *bioRxiv* (2020). DOI: 10.1101/2020.05.14.095604. eprint: https://www.biorxiv.org/content/early/2020/05/19/2020.05.14.095604.full.pdf. URL: https://www.biorxiv.org/content/early/2020/05/19/2020.05.14.095604.

[33] Jens Zentgraf, Henning Timm und Sven Rahmann. „Cost-optimal assignment of elements in genome-scale multi-way bucketed Cuckoo hash tables“. In: *2020 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, S. 186–198. DOI: 10.1137/1.9781611976007.15. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611976007.15. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611976007.15.