technische universität
dortmund

Abschlussbericht der Projektgruppe 583

VATRAM
VAriant Tolerant ReAd Mapper

Benjamin Kramer, Jens Quedenfeld
Sven Schrinner, Marcel Bargull
Kada Benadjemia, Jan Stricker
David Losch
30. März 2015

Betreuer:
Sven Rahmann
Johannes Fischer
Dominik Kopczynski
Dominik Köppl
Henning Timm

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

Inhaltsverzeichnis

Kapitel 1

Einleitung

Der Readmapper VATRAM (*VAriant Tolerant ReAdMapper*) ist das Arbeitsergebnis der Projektgruppe 583 unter dem Arbeitstitel „Algorithmen zur Entdeckung krebsauslösender Genvarianten“.

Alle Erbinformationen von Lebewesen befinden sich in kodierter Form in der Desoxyribonukleinsäure (*englisch: deoxyribonucleic acid*, DNA), dem Träger der Erbinformationen. Aus ihr leitet sich der Bauplan von Proteinen ab. Daher sind Unterschiede in ihr dafür verantwortlich, dass Lebewesen sich unterscheiden und verschiedene spezifische Eigenschaften besitzen. Die Bestandteile der DNA, die sogenannten Nukleotide, können sich punktweise oder großflächig unterschieden, was als Genvariante bezeichnet wird. Jedoch sind nicht alle diese Varianten gutartig. Tritt eine Veränderung an einer falschen Position auf, kann sie Auslöser für verschiedene Krankheiten wie beispielsweise Krebs sein.

Krebs ist eine der häufigsten Todesursachen in Deutschland. Alleine im Jahr 2010 wurden bei 477.303 Menschen ein Tumor neu diagnostiziert. Dem gegenüber stehen über 200.000 Personen im selben Jahr, bei denen ein Tumor die direkte Todesursache darstellt (Kaatsch et al., 2013). Dabei ist das Risiko, an Krebs zu erkranken, genetisch veranlagt. Diese Informationen befinden sich in den Erbinformationen der jeweiligen Personen.

Um diese untersuchen zu können, muss die Information der DNA in eine lesbare Form gebracht werden. Durch die sogenannte Sequenzierung wird die Abfolge der Nukleotiden bestimmt. Die ersten Sequenzierungsmethoden waren noch zeit- und kostenintensiv. Mit den *Sequenzierern der nächsten Generation* wurden die Verfahren effizienter. Gleichwohl ist es auch mit ihnen nicht möglich, die gesamte DNA an einem Stück zu erfassen. Die Ausgaben der Maschinen sind kurze, unsortierte Abschnitte, welche als *Reads* bezeichnet werden. Diese müssen erst mit großem Rechenaufwand in die *richtige Reihenfolge* gebracht werden, um das Genom rekonstruieren zu können. Wissenschaftliche Projekte wie das 1000 Genomes Project (2010) haben diese Arbeit in verschiedenen Referenzgenomen festgehalten. Dadurch ist es Wissenschaftlern möglich, Zusammenhänge zwischen verschiedenen vererbbaren Krankheiten zu finden, um Therapiemöglichkeiten zu verbessern.

Readmapper ordnen Reads Positionen im Referenzgenom zu und können sie dadurch in die richtige Reihenfolge bringen. Da Reads Genvarianten enthalten können,

ist ihre Zuordnung nicht immer eindeutig oder nur schwer zu finden. Aber auch durch Sequenzierfehler kann eine Alignierung an die richtige Position erschwert werden. Unsere Aufgabe bestand darin, einen Readmapper zu konstruieren, welcher Reads mit Varianten korrekt an ein Referenzgenom alignieren kann, für welches vermerkt wurde, an welchen Positionen bereits bekannte Varianten auftreten können. Bekannte Varianten werden dabei nicht fälschlicherweise als Abweichung von Referenzgenomen erkannt. Hierdurch werden unbekannte Varianten und Sequenzierfehler besser hervorgehoben, was deren Nachweis in nachfolgenden Verarbeitungsschritten erleichtert. Eine genauere Beschreibung der formalen Problemstellung findet sich in Abschnitt 2.3.

Unser Ansatz verfolgte ein zweistufiges Verfahren. Im ersten Schritt führen wir mittels eines Hashingverfahrens ein Mapping durch, welches für jeden Read eine Menge von passenden Positionen in der Referenz findet. Im nächsten Schritt werden die Reads durch einen auf dynamischer Programmierung basierenden Algorithmus an die Referenz aligniert. Durch das vorherige Mapping erfolgt die Alignierung nur an kurze Referenzabschnitte, wodurch der Alignierungsalgorithmus akzeptable Laufzeiten erreicht. Detaillierte Beschreibungen finden sich in den Kapiteln 4.1 und 4.2.

Des Weiteren unterliegt unser Bericht folgender Gliederung: Im Einführungskapitel 2 wird sowohl biologisches Basiswissen zum Aufbau und Sequenzierung von DNA vermittelt als auch technische Grundlagen wie die Datenstrukturen und Kodierungen für Ein- und Ausgabedaten.

In Kapitel 3 folgen statistische Auswertungen über das Vorkommen von bereits bekannten Varianten im Humangenom. Darauf folgen in Kapitel 4 die bereits angesprochenen Kapitel über die von uns verwendeten Algorithmen.

Der letzte Teil ab Kapitel 5 befasst sich mit der Evaluierung der Qualität unseres Programms, beginnend mit der Evaluation des Generators für synthetische Daten. Danach folgt die Qualitätsüberprüfung der beiden großen Bestandteile des Readmappers: Im Abschnitt 5.3 wird das *Locality-Sensitive Hashing*, im Abschnitt 5.4 der Aligner überprüft. In Abschnitt 5.5 folgt ein Vergleich zwischen VATRAM und zwei *state-of-the-art*-Readmappern. Zuletzt wird in Kapitel 5.6 noch ein externes Tool zur Qualitätsüberprüfung eingesetzt und dessen Ergebnisse diskutiert.

Zum Abschluss des Berichts wird ein Ausblick gegeben, welche Anpassungen und Erweiterungen an die Software gestellt werden können, um dessen Umfang und Qualität zu vergrößern bzw. verbessern. Abschließend wird ein Fazit gezogen, in welchem auch auf die Entstehung der Software im Kontext einer Projektgruppe eingegangen wird.

Kapitel 2

Grundlagen

2.1 Biologische Grundlagen

In diesem Kapitel sollen die für unsere Projektgruppe relevanten biologischen Grundlagen erläutert werden. Die Informationen wurden (wenn nicht anders angegeben) dem Buch *Molekulare Genetik* von Rolf Knippers (2006) entnommen. Dieses Kapitel behandelt dabei die folgenden Themen: Im ersten Abschnitt (2.1.1) geht es um den Aufbau der DNA (dem Träger der Erbinformation). Anschließend wird erklärt, was Chromosomen sind (Abschnitt 2.1.2) und wie sich Pro- und Eukaryoten unterscheiden (Abschnitt 2.1.3). In Abschnitt 2.1.4 geht es darum, wie sich die DNA bei der Zellteilung verdoppelt und wie sie auf die beiden Tochterzellen aufgeteilt wird. Wir kommen anschließend (in Abschnitt 2.1.6) zur Protein-Biosynthese, also dem Vorgang, bei dem die auf der DNA gespeicherte Information in Proteine übersetzt wird. Abschnitt 2.1.7 behandelt die Vererbung und soll einige Grundbegriffe der Vererbungslehre erklären. Für unsere Projektgruppe sind vor allem Mutationen des Genoms bzw. der DNA interessant. Welche Arten von Mutationen es gibt, wird in Abschnitt 2.1.8 näher erläutert.

2.1.1 DNA

Die Desoxyribonukleinsäure, kurz DNA[1] (deoxyribonucleic acid), kommt in allen Lebewesen vor (zum Beispiel auch Bakterien) und ist der Träger der Erbinformation. Die DNA enthält somit die Informationen, die die Nachkommen von ihren Eltern geerbt haben und die für den Bau von Proteinen benötigt werden. Proteine können die verschiedensten Aufgaben in unserem Körper übernehmen. In Abschnitt 2.1.6.2 wird der Aufbau von Proteinen noch ausführlich behandelt.

DNA-Moleküle bestehen aus einen Doppelstrang von *Nukleotiden*, die miteinander verbunden sind (siehe Abbildung 2.1). Diese Nukleotide setzen sich wiederum aus drei Bausteinen zusammen: Einem Zucker-Molekül (der Desoxyribose), das mit einem Phosphatrest sowie einer von vier möglichen Nukleobasen chemisch verbunden ist. Die Nukleotide können zu einem Strang zusammengesetzt werden, in dem das Zuckermolekül des einen Nukleotids mit dem Phosphatrest des Nächsten verbunden

[1] Die deutsche Abkürzung DNS wird nur noch sehr selten verwendet und ist laut Duden veraltet.

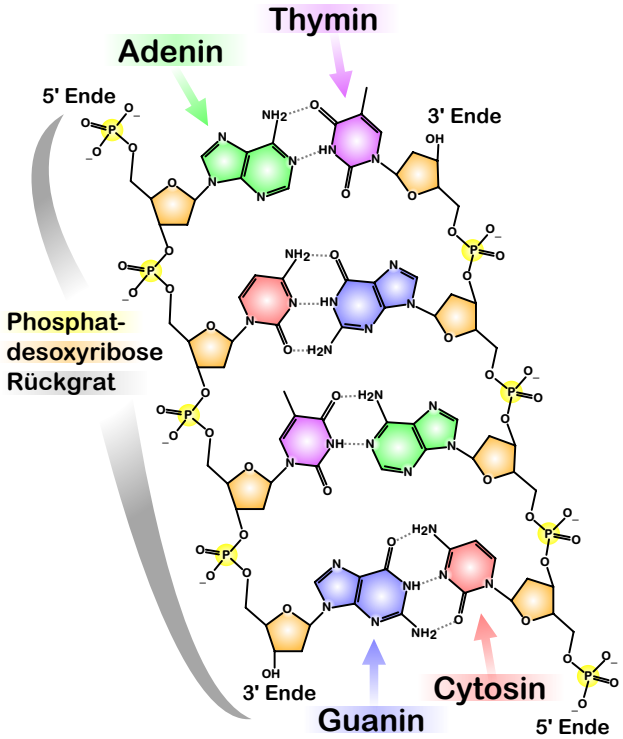


Abbildung 2.1: Schematischer Aufbau der DNA. ((aus http://commons.wikimedia.org/wiki/File:Chromosome.svg))

wird. Die Zucker-Phosphat-Ketten eines DNA-Moleküls befinden sich im äußeren Bereich des Doppelstrangs und bilden das Rückgrat der DNA. Zwischen den beiden Zucker-Phosphat-Ketten sind jeweils zwei Nukleobasen über Wasserstoffbrückenbindungen miteinander verbunden[2].

Die Reihenfolge in der diese Nukleobasen (oder einfach kurz Basen) vorkommen, stellen die auf der DNA gespeicherte Information dar. Wie bereits erwähnt gibt es vier verschiedene Basen, nämlich Adenin, Cytosin, Guanin und Thymin, die jeweils mit ihrem Anfangsbuchstaben (d.h. A, C, G, T) abgekürzt werden. Eine wichtige Eigenschaft des DNA-Doppelstranges ist die Komplementarität: Adenin bindet stets an Thymin und Guanin immer an Cytosin. Kennt man also einen Einzelstrang eines DNA-Moleküls, so kann man den fehlenden DNA-Strang rekonstruieren. Adenin und Thymin sind über zwei Wasserstoffbrückenbindungen miteinander verbunden, Cytosin und Guanin mit dreien. Aufgrund ihres chemischen Aufbaus werden Adenin und Guanin als *Purin*-Basen bezeichnet: Sie bestehen aus zwei Kohlenstoff-Stickstoff-Ringen und sind damit ein wenig länger als die *Pyrimidin*-Basen Cytosin und Thymin, die nur aus einem solchen Ring bestehen.

Die *Desoxyribose*, die der DNA ihren Namen gibt, ist ein Zuckermolekül, das aus fünf Kohlenstoff-Atomen besteht. Diese C-Atome werden wie in Abbildung 2.2 gezeigt von 1' bis 5' durchnummeriert. Das erste Kohlenstoff-Atom (1') ist mit der Nukleobase

[2]Streng genommen besteht ein DNA-Doppelstrang somit aus zwei Molekülen, da nur zwischen Zucker und Phosphatrest sowie zwischen Zucker und Nukleobase chemische Bindungen vorliegen, nicht aber zwischen den Nukleobasen.

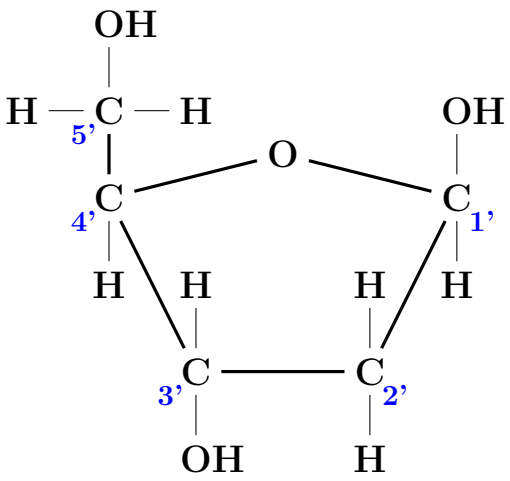


Abbildung 2.2: Chemischer Aufbau der Desoxyribose.

(A, C, G oder T) verbunden, wobei beim Verbinden Wasser (H_2O) abgespalten wird. Beim zweiten C-Atom fällt auf, dass hier die Hydroxy-Gruppe fehlt (-H statt -OH); daher kommt die Vorsilbe „Desoxy“ im Namen des Zuckermolekül. Am 5'-Ende des Zuckermoleküls hängt die Phosphatgruppe des Nukleotids. Die Phosphatgruppe eines anderen Nukleotids kann mit dem dritten C-Atom des Zucker-Moleküls verbunden werden, sodass sich Ketten aus Phosphatresten und Zucker-Molekülen bilden. Hier zeigt sich eine weitere wichtige Eigenschaft der DNA: Die beiden Stränge besitzen jeweils eine Richtung. Man kann die Reihenfolge der Nukleobasen in 3'-5'-Richtung lesen oder in 5'-3'-Richtung. Analysiert man die Struktur der DNA genauer, stellt man fest, dass die beiden DNA-Stränge eines DNA-Moleküls antiparallel zueinander stehen. Ein Strang liegt in 3'-5'-Richtung, der andere in 5'-3'-Richtung.

DNA-Moleküle liegen als rechtsläufige Doppelhelix vor. Dabei besteht jede Windung aus etwa 10 Basenpaaren und ist 3,4 nm lang. Die Doppelhelix ist nicht exakt symmetrisch, sondern besteht aus einer großen und einer kleinen Furche. Der Durchmesser eines DNA-Doppelstrangs beträgt ca. 2 nm.

2.1.2 Chromosomen

DNA-Moleküle können sehr lang werden, beispielsweise besteht das größte DNA-Molekül des Menschen aus etwa 263 Millionen Basenpaaren. Aus diesem Grund wird die DNA-Doppelhelix um bestimmte Proteine, sogenannte Nukleosomen, herum gewickelt. Durch zusätzliche Proteine wird die DNA weiter zusammengefaltet, wie diese Faltungen jedoch genau aussehen und welche Proteine daran beteiligt sind, ist noch Gegenstand der Forschung (Hansen, 2012).

Der Komplex aus DNA und Proteinen wird als Chromosom bezeichnet. Chromosomen können in drei verschiedenen Formen vorliegen. Kurz vor einer Zellteilung besteht jedes Chromosom aus zwei Chromatinfäden (d.h. zwei DNA-Doppelsträngen), die jeweils dieselbe Information tragen und am *Centromer* miteinander verbunden sind. In schematischen Darstellungen von Chromosomen (wie zum Beispiel Abbildung 2.3) wird fast immer diese Form gezeigt. Während der Zellteilung werden die

beiden Chromatinfäden eines Chromosoms auf die beiden Tochterzellen aufgeteilt, sodass das Chromosom nach einer Zellteilung nur noch aus einem Chromatinfaden besteht. Die Chromatinfäden sind während der Zellteilung stark komprimiert und von daher unter einem Lichtmikroskop sichtbar. Zwischen den Zellteilungen liegen die Chromosomen als freies Chromatin vor. Nur in diesem Zustand kann die auf der DNA gespeicherte Information gelesen oder kopiert werden. Unter einem Lichtmikroskop sind die freien, unkomprimierten Chromatinfäden nicht sichtbar.

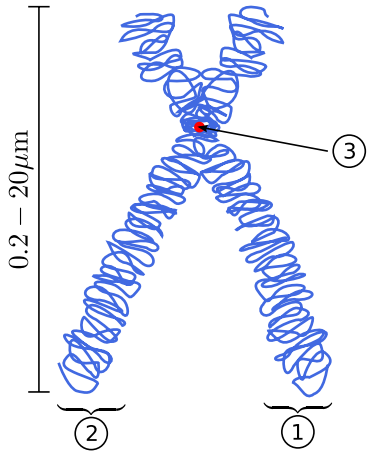


Abbildung 2.3: Schematische Abbildung eines Chromosoms mit zwei Chromatiden. (1) und (2) die beiden Chromatiden, (3) Centromer. (aus http://commons.wikimedia.org/wiki/File:Chromosome.svg).

Menschliche Zellen enthalten bis auf wenige Ausnahmen[3] stets 46 Chromosomen, bzw. 23 Chromosomenpaare. Von jedem Paar haben wir ein Chromosom vom Vater geerbt und eines von der Mutter. Die Informationen, die auf den Chromosomen eines Paares stehen, sind ähnlich, aber nicht zwingend gleich. Das 23. Chromosomenpaar bestimmt das Geschlecht eines Menschen: Frauen haben zwei X-Chromosomen, Männer ein X- und ein Y-Chromosom. Dementsprechend werden diese beiden Chromosomen auch als Geschlechtschromosomen oder kurz Gonosomen bezeichnet.

Insgesamt besteht das menschliche Genom (also die Vereinigung aller Chromosomen einer Zelle) aus drei Milliarden Basenpaaren oder kürzer aufgeschrieben 3 Gbp. Die Vorsilbe G steht dabei für Giga also 10^9; bp ist die Abkürzung für Basenpaar. Dementsprechend gibt es auch die Einheiten Mbp und kbp für Millionen bzw. Tausend Basenpaare. Da es für jedes Basenpaar vier Möglichkeiten (A, C, G und T) gibt, beträgt der Informationsgehalt des menschlichen Genoms sechs Milliarden Bit bzw. 750 MB. An dieser Stelle soll nochmal erwähnt werden, dass jede Zelle eines Lebewesens denselben DNA-Code enthält, d.h. jede menschliche Zelle enthält die erwähnten drei Milliarden Basenpaare.

[3] Ausnahmen bilden zum einen die Geschlechtszellen, d.h. Spermien bzw. Eizellen (siehe Abschnitt 2.1.7), sowie Zellen, bei denen sich die Chromosomenzahl durch Krankheiten verändert hat (siehe Abschnitt 2.1.8).

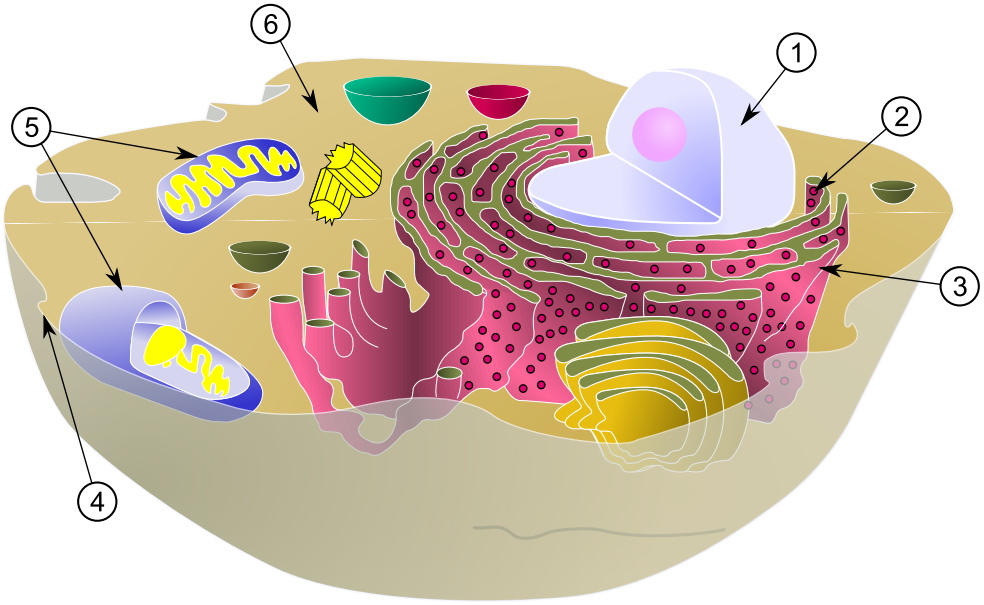


Abbildung 2.4: Typische eukaryotische Tierzelle. (1) Zellkern, (2) Ribosom, (3) Endoplasmatisches Retikulum, (4) Zellmembran, (5) Mitochondrien, (6) Zytoplasma. (aus http://commons.wikimedia.org/wiki/File:Biological_cell.svg)

2.1.3 Prokaryoten und Eukaryoten

An einigen Stellen ist es wichtig zwischen Pro- und Eukaryoten zu unterscheiden. Eukaryoten sind alle höheren Lebensformen, das heißt Menschen, Tiere, Pflanzen und Pilze. Zu Prokaryoten gehören Bakterien und die Archaebakterien, die auch als Urbakterien bezeichnet werden. Eukaryotische Zellen besitzen die im vorherigen Abschnitt erwähnten Chromosomen, die sich im Zellkern befinden. Prokaryotische Zellen besitzen dagegen keinen Zellkern, ihre DNA ist somit nicht durch eine Membran von den restlichen Zellorganellen abgegrenzt. Des Weiteren enthalten Prokaryoten in der Regel nur ein DNA-Molekül, das ringförmig, also in sich geschlossen ist.

Da wir uns vor allem mit dem menschlichen Genom beschäftigen, sind für uns hauptsächlich Eukaryoten von Interesse. Trotzdem wird an einigen Stellen erklärt, welche Unterschiede zu Prokaryoten bestehen.

Aufbau einer eukaryotischen Zelle Jede Zelle ist von einer Zellmembran umgeben, die die Zelle von der Umgebung abgrenzt. Im Innern der Zellen befinden sich verschiedene, sogenannte Organellen, die bestimmte Funktionen in der Zelle übernehmen. Zu diesen gehört unter anderen der Zellkern, in dem sich die DNA befindet. Das endoplasmatische Retikulum ist mit seinen Ribosomen an der Protein-Biosynthese beteiligt, also an der Übersetzung eines DNA-Abschnitts in ein Protein. Dieser Vorgang wird noch in Abschnitt 2.1.6 genauer erklärt. Der Raum, der zwischen der Zellmembran und den einzelnen Organellen liegt, wird als Zytoplasma bezeichnet. Zellen enthalten noch weitere Organellen, eine Beschreibung dieser würde hier aber zu weit führen. Die Mitochondrien sollen trotzdem kurz erwähnt werden:

Mitochondrien Mitochondrien sind im Prinzip die *Energiekraftwerke* unserer Zellen. Sie stellen Adenosintriphosphat (kurz ATP) her, das bei sehr vielen Prozessen in unserem Körper als Energieträger genutzt wird. Wird eine Phosphatgruppe von ATP abgespalten, entsteht Adenosindiphosphat (kurz ADP) und Energie wird frei. Die freiwerdende Energie wird beispielsweise in Muskelzellen in mechanische Arbeit umgesetzt. Auch die für die Herstellung organischer Moleküle benötigte Energie wird durch ATP bereitgestellt. In den Mitochondrien werden ADP und der Phosphatrest wieder zu ATP verbunden. Die dafür nötige Energie liefert zum Beispiel Glucose (Traubenzucker), die direkt oder auch indirekt über die Nahrung aufgenommen wird.

Das in diesem Bericht die Mitochondrien erwähnt werden, hat aber einen anderen Grund: Bisher wurde immer behauptet, dass sich die gesamte DNA einer Zelle im Zellkern befindet. Es gibt jedoch eine Ausnahme: Mitochondrien enthalten ebenfalls einen eigenen, kurzen DNA-Ring. Beim Menschen hat dieser DNA-Ring eine Länge von 16.569 Basenpaaren[4] und enthält zahlreiche überlebenswichtige Gene.

2.1.4 Zellteilung

Wie bereits erwähnt, enthalten alle Zellen eines Lebewesens (wenn man von Mutationen absieht) dieselbe DNA-Information. Teilt sich eine Zelle, so muss ihre DNA zunächst verdoppelt werden. Dieser Vorgang wird DNA-Replikation genannt. Anschließend wird bei der Mitose die replizierte DNA auf beide Tochterzellen gleichmäßig aufgeteilt, sodass sich die Zelle teilen kann. Wir schauen uns zunächst die DNA-Replikation an.

2.1.5 DNA-Replikation

Bei der DNA-Replikation, also der Verdopplung der DNA, sind eine ganze Reihe an Proteinen beteiligt. Zunächst entwindet die *Topoisomerase* die DNA-Doppelhelix, sodass sie durch die *Helicase* in die zwei Einzelstränge gespalten werden kann. Die eigentliche Replikation wird nun von der *DNA-Polymerase* durchgeführt. Diese liest den DNA-Strang in 3'-5'-Richtung. Dementsprechend wird der neu gebildete DNA-Strang stets am 3'-Ende verlängert. Als Ausgangsmaterial verwendet die DNA-Polymerase *Desoxyribonukleosidtriphosphate* (dNTPs). Diese bestehen aus einem Zuckermolekül (der Desoxyribose), einer der vier Nukleobasen (also Adenin, Cytosin, Guanin oder Thymin) sowie einer Triphosphat-Gruppe. Je nach dem welche Base in dem Molekül verbaut ist, wird dNTP auch als dATP, dCTP, dGTP bzw. dTTP bezeichnet. Bei der DNA-Synthese werden die (zum Original-Strang komplementären) dNTPs unter Abspaltung von Diphosphat miteinander verbunden, sodass sich wieder ein DNA-Doppelstrang mit komplementären Basenpaaren bildet.

Aufgrund der Antiparallelität der DNA (die beiden Stränge eines DNA-Moleküls laufen in entgegengesetzte Richtungen) findet nur beim 3'-5'-Strang eine kontinuierliche Synthese statt. Die Helicase spaltet den DNA-Doppelstrang immer weiter auf, sodass die DNA-Polymerase beliebig lange weiterarbeiten kann. Der Strang, bei dem die Synthese kontinuierlich stattfindet, wird als Leitstrang bezeichnet. Im Gegensatz

[4] http://www.mitomap.org/MITOMAP/HumanMitoSeq

dazu findet beim Folgestrang, der in 5'-3'-Richtung verläuft, eine diskontinuierliche Synthese statt. Die DNA-Polymerase arbeitet hier ebenfalls in 3'-5'-Richtung und entfernt sich somit immer weiter von der Helicase. Irgendwann trifft die DNA-Polymerase auf ein bereits repliziertes DNA-Stück, sodass sich die DNA-Polymerase von der DNA löst. Der Bereich zwischen Helicase und bereits verdoppelter DNA wird nun von einer weiteren DNA-Polymerase repliziert, wobei sich diese wiederum von der Helicase entfernt. Es entstehen sogenannte *Okazaki-Fragmente*. Diese werden durch ein weiteres Protein, der *Ligase*, miteinander verbunden, sodass wieder ein ununterbrochener DNA-Doppelstrang vorliegt.

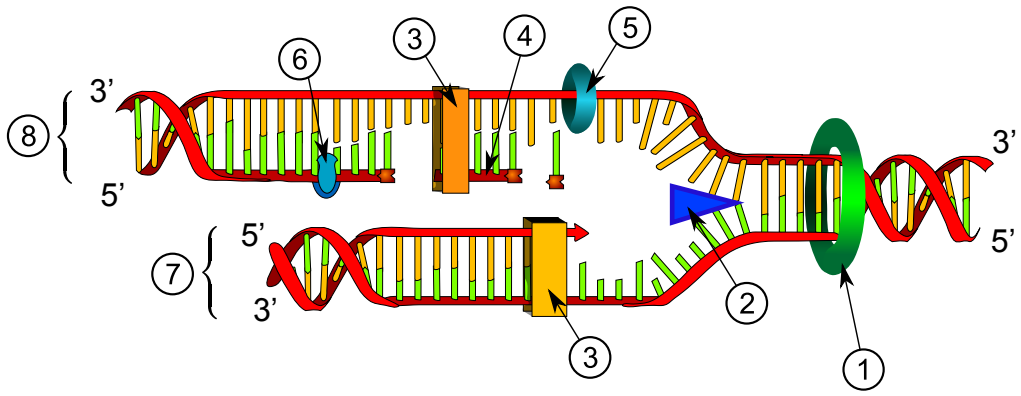


Abbildung 2.5: DNA-Replikation. (1) Topoisomerase, (2) Helicase, (3) Polymerase, (4) Okazaki-Fragment, (5) Primase, (6) Ligase, (7) Leitstrang (kontinuierliche Synthese), (8) Folgestrang. (aus http://de.wikipedia.org/wiki/Datei:DNA_replication_de.svg)

Aufgrund der Komplementarität der DNA sind die beiden hergestellten Doppelstränge exakte Kopien voneinander. Die neu synthetisierte DNA in 5'-3'-Richtung am Leitstrang entspricht dem originalen 5'-3'-Strang des Folgestranges. Analog dazu gleichen sich auch der alte und neue Strang in 3'-5'-Richtung.

Nach der DNA-Replikation bestehen die Chromosomen aus zwei Chromatiden, die am Centromer miteinander verbunden sind. Die DNA-Doppelstränge in den beiden Chromatiden sind exakte Kopien voneinander. Da die Chromosomen eines Chromosomenpaares ähnliche Informationen enthalten, kann es sein, dass direkt nach der DNA-Replikation bestimmte Code-Sequenzen in der Zelle in vierfacher Ausführung vorkommen.

2.1.5.1 Mitose

Nach der DNA-Replikation findet die eigentliche Zellteilung statt. Hierbei muss die DNA gleichmäßig auf beide Tochterzellen aufgeteilt werden. Dieser Vorgang wird Kernteilung oder in Fachsprache Mitose genannt und besteht aus mehreren Phasen (siehe auch Abbildung 2.6).

In der *Prophase* kondensieren die Chromosomen, sodass sie unter einem Lichtmikroskop sichtbar werden. Sie haben jetzt die kompakte, charakteristische Form mit zwei Chromatiden. Während der *Prometaphase* löst sich die Kernhülle auf. Außerdem wird von den gegenüberliegenden Seiten der Zelle ein Spindelapparat ausgebaut. Die Spindeln heften sich in der *Metaphase* an die Centromere der Chromosomen. In der *Anaphase* werden die beiden Chromatiden der Chromosomen durch den Spindelapparat auseinander gezogen, sodass sich auf beiden Seiten der Zelle 46 Chromosomen mit jeweils einem Chromatid befinden. In der *Telophase* bilden sich dann zwei neue Kernhüllen — auf jeder Seite der Zelle eine. Gleichzeitig dekondensieren die Chromosomen, d.h. sie breiten sich wieder aus und werden damit unter dem Lichtmikroskop unsichtbar. Die Teilung des Zellkerns ist damit abgeschlossen.

In der *Interphase*, die in der Regel nicht mehr zur Mitose gezählt wird, teilt sich die Zelle, sodass jede Tochterzelle genau einen Zellkern mit dem vollen Chromosomensatz enthält. Während der Interphase findet auch die Proteinbiosynthese (siehe Abschnitt 2.1.6) statt, bei der die auf der DNA gespeicherten Informationen genutzt werden, um Proteine herzustellen. Die Interphase ist also quasi der *normale* Zustand einer Zelle. Vor einer erneuten Zellteilung wird die DNA repliziert. Auch dieser Vorgang gehört zur Interphase.

2.1.5.2 Künstliche DNA-Replikation - PCR

Die DNA-Replikation, die vor jeder Zellteilung stattfindet, kann auch künstlich durchgeführt werden. Mit Hilfe der Polymerase-Kettenreaktion (PCR, engl. Polymerase Chain Reaction) können bereits geringste DNA-Mengen vervielfältigt werden. Zum Erstellen eines genetischen Fingerabdrucks (zum Beispiel bei Mordfällen) werden hinreichend große DNA-Mengen benötigt. Findet man am Tatort beispielsweise eine Hautzelle des Täters, kann die darin enthaltende DNA mit Hilfe der PCR vervielfacht werden. Ohne dieses Verfahren wäre es nicht ohne weiteres möglich, von einer so geringen DNA-Menge einen genetischen Fingerabdruck zu erstellen. Weitere Anwendungen sind Vaterschaftstests oder der Nachweis von Erbkrankheiten. Auch für die Sequenzierung (siehe Abschnitt 2.2), also das Bestimmen der Nukleotid-Abfolge eines DNA-Moleküls, werden hinreichend große DNA-Mengen benötigt. Die PCR ist somit die Grundlage für zahlreiche Anwendungsfälle, bei denen man DNA in hinreichend großer Menge benötigt.

Die PCR funktioniert dabei folgendermaßen: Zunächst wird die DNA auf 94°C erhitzt, wodurch der DNA-Doppelstrang in zwei Einzelstränge gespalten wird (Denaturierung). Nach Abkühlung auf 70°C können sich Primer an die DNA lagern. Primer sind kurze DNA-Sequenzen und werden von der DNA-Polymerase als Startpunkte benötigt. Die Wahl guter Primer ist wichtig, aber auch sehr schwierig: Optimal wäre es, wenn sich die Primer nur an die 3'-Enden der DNA heften, sodass die Polymerase

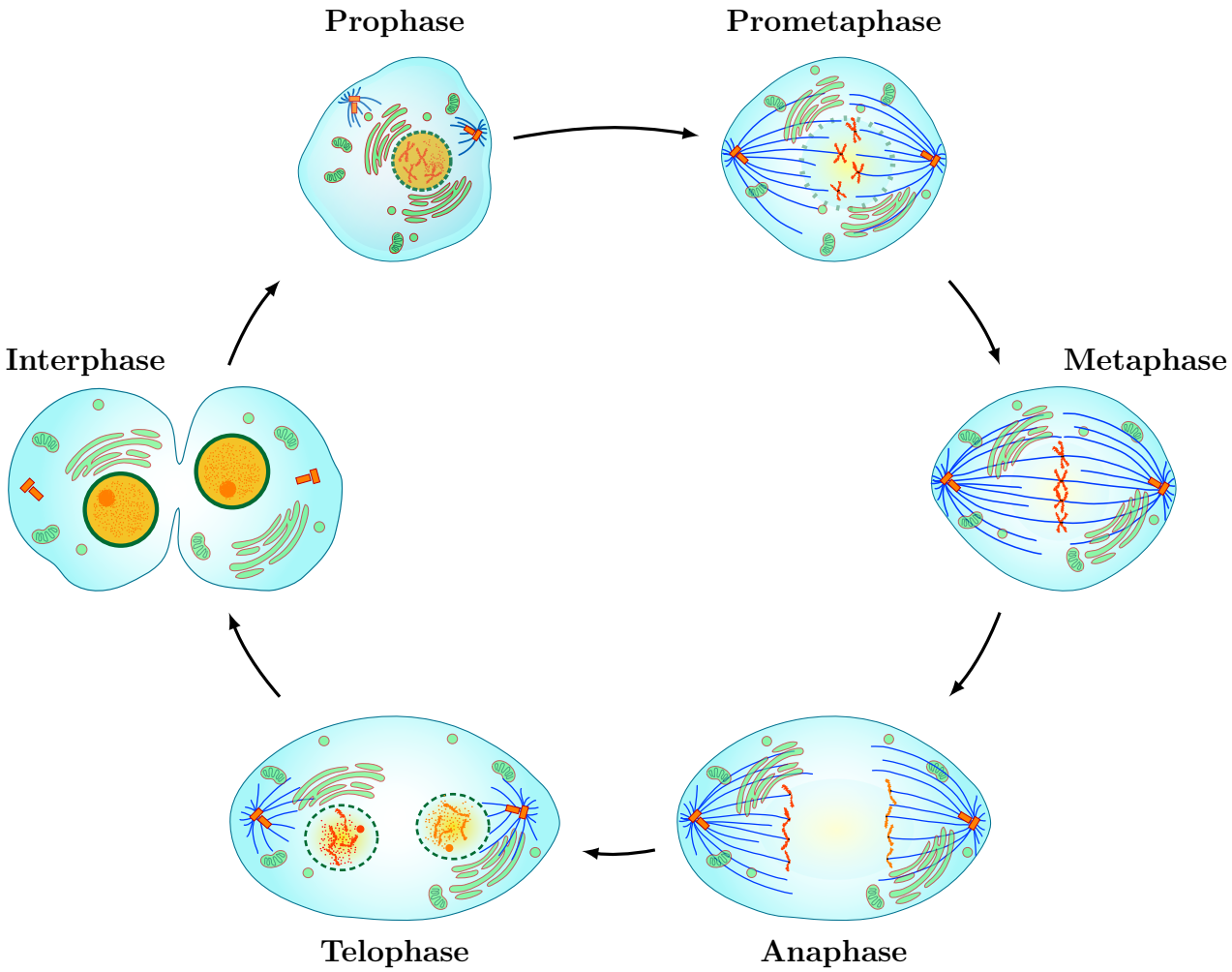


Abbildung 2.6: Mitose. (aus http://de.wikipedia.org/wiki/Mitose)

die gesamte DNA vervielfältigen kann. Würden sich die Primer an die Mitte oder gar an das 5'-Ende der DNA lagern, wird ein großer Teil der DNA nicht vermehrt, da die DNA-Polymerase nur in 3'-5'-Richtung arbeitet. Zudem sollte sich nur ein Primer an jeden DNA-Strang lagern, da sonst bei der Replikation nicht miteinander verbundene Fragmente entstehen.

Nachdem sich die Primer an die DNA gelagert haben, replizieren DNA-Polymerasen die DNA-Stränge. Aus dem ursprünglichen DNA-Doppelstrang entstehen so zwei Kopien. Der beschriebene Zyklus kann beliebig häufig wiederholt werden, wobei natürlich hinreichend viele freie Nukleotide als Ausgangsstoff für die Polymerase vorhanden sein müssen.

Das Erhitzen und Abkühlen übernehmen spezielle Geräte (sogenannte Thermocycler). Sie wiederholen den beschriebenen Zyklus der PCR beliebig oft. Da sich bei jeden Schritt die DNA-Menge verdoppelt, steigt die Anzahl der DNA-Kopien exponentiell an: Bereits nach 20 Zyklen liegen theoretisch eine Millionen Kopien vor.

Ein Problem existiert allerdings noch: Bei 94° wird nicht nur die DNA denaturiert, sondern es zersetzen sich auch gewöhnliche DNA-Polymerasen irreversibel. Früher (als die PCR-Methode 1986 eingeführt wurde) wurden nach jeden Zyklus neue Polymerasen zugesetzt. Eine entscheidende Verbesserung lieferte die Taq-Polymerase des Bakteriums *Thermus aquaticus*. Das Bakterium gehört zu den thermophilen, also zu den wärmeliebenden Bakterien und lebt beispielsweise in heißen Quellen oder Geysiren. Seine DNA-Polymerase ist auch noch bei Temperaturen über 94° stabil.

2.1.6 Protein-Biosynthese

Unter Protein-Biosynthese versteht man die Übersetzung eines DNA-Abschnitts (eines sogenannten Gens) in ein Protein. Dieser Vorgang unterteilt sich in zwei Abschnitte:

Zunächst wird bei der *Transkription* ein Gen der DNA abgelesen und eine RNA-Kopie erstellt. Die Ribonukleinsäure, kurz RNA, ist ähnlich wie DNA aufgebaut, es gibt jedoch drei Unterschiede: Im Gegensatz zur DNA besteht RNA nur aus einem Einzelstrang. Des Weiteren enthält sie als Zuckermolekül — wie ihr Name bereits vermuten lässt — Ribose anstelle von Desoxyribose. Statt der Base Thymin wird in der RNA Uracil verbaut. Chemisch gesehen unterscheidet sich Uracil durch eine fehlende Methyl-Gruppe von Thymin, die möglichen Basenpaarungen sind aber dieselben, d.h. Uracil verbindet sich stets mit Adenin und umgekehrt. Die transkribierte RNA wird auch als messenger-RNA oder kurz mRNA bezeichnet.

Im zweiten Schritt der Proteinbiosynthese wird die mRNA in eine Aminosäure-Sequenz übersetzt. Dieser Vorgang wird als *Translation* bezeichnet. Aminosäure-Sequenzen sind die Vorstufen von Proteinen.

2.1.6.1 Transkription

Bei der Transkription erstellt ein bestimmter Proteinkomplex, die sogenannte RNA-Polymerase, eine mRNA-Kopie eines DNA-Abschnitts. Dieser Vorgang kann in drei Phasen unterteilt werden:

Bei der *Initiation* setzt sich eine RNA-Polymerase an den Promotor eines Gens. Der Promotor ist eine spezielle DNA-Sequenz, die Informationen darüber enthält, wann und in welchen Zelltyp ein Gen transkribiert werden soll. Der Promotor codiert somit selbst kein Protein, sondern reguliert die Genexpression. Diese Regulation ist sehr wichtig, da jede Zelle dieselbe DNA-Information enthält. Eine Magenzelle muss beispielsweise ganz andere Proteine herstellen als eine Nervenzelle im Gehirn.

In der zweiten Phase, der *Elongation*, wird die DNA von der RNA-Polymerase in 3'-5'-Richtung abgelesen und eine komplementäre mRNA-Kopie erstellt. Die Synthese der RNA erfolgt somit in 5'-3'-Richtung.

Nachdem das Gen abgelesen wurde, löst sich die RNA-Polymerase vom DNA-Strang. Dieser Vorgang wird das *Termination* bezeichnet, es ist allerdings zumindest bei Eukaryoten noch teilweise ungeklärt, wann die Termination genau stattfindet bzw. wodurch sie eingeleitet wird.

2.1.6.2 Proteine

Bevor wir uns der Translation widmen, bei der die mRNA in ein Protein übersetzt wird, wollen wir zunächst klären, was überhaupt Proteine sind: Proteine bestehen aus Aminosäuren, von denen 20 verschiedene in natürlichen Proteinen vorkommen (siehe Abbildung 2.7). Aminosäuren können miteinander verbunden werden und bilden dann Aminosäure-Sequenzen, die auch als Polypeptide bezeichnet werden. Proteine können aus mehreren solcher Polypeptide bestehen, wobei die räumliche Faltung der Aminosäuren-Sequenzen für die Funktion entscheidend ist.

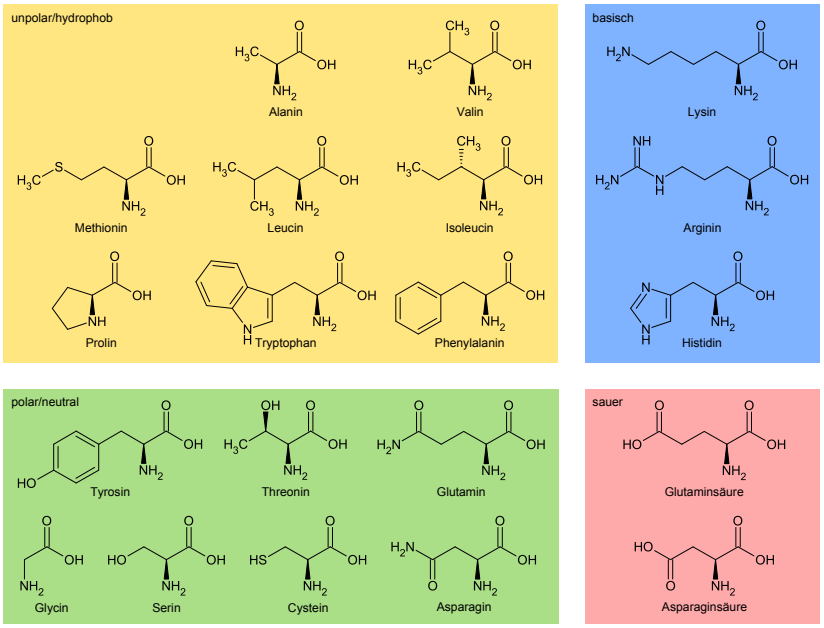


Abbildung 2.7: Übersicht der 20 natürlich vorkommenden Aminosäuren. (aus http://commons.wikimedia.org/wiki/File:Overview_proteinogenic_amino_acids-DE.svg)

Proteine können die unterschiedlichsten Aufgaben im Organismus übernehmen: Das Protein Hämoglobin ist beispielsweise in der Lage Sauerstoff zu binden, damit dieser durch die roten Blutkörperchen zu den Zellen transportiert werden kann. Proteine, die als Katalysator für chemische Reaktionen dienen, werden auch Enzyme genannt. Als Beispiel könnte man Lactase nennen, ein Enzym, dass Milchzucker (Lactose) in Galactose und Glucose (Traubenzucker) spaltet. Es gibt noch viele weitere Funktionen, die Proteine erfüllen können, beispielsweise als Ionen-Kanal in der Zellmembran.

Die Aminosäuren aus denen Proteine bestehen werden häufig mit drei Buchstaben abgekürzt (z.B. Gly für Glycin, Ala für Alanin, usw.). Darüber hinaus gibt es einen Ein-Buchstaben-Code, die verwendeten Buchstaben überschneiden sich aber mit den Abkürzungen für die Nukleobasen in DNA und RNA. So wird beispielsweise die Aminosäure Glycin ebenso mit G abgekürzt wie auch die Base Guanin.

2.1.6.3 Translation

Unter Translation versteht man die Übersetzung der mRNA in ein Protein. Die Translation der Eukaryoten unterscheidet sich von der Translation der Prokaryoten. Bei Prokaryoten (d.h. Bakterien und Archaebakterien) findet die Translation noch während der Transkription statt. Bei Eukaryoten ist die Translation räumlich von der Transkription getrennt. Außerdem findet bei Eukaryoten eine Nachverarbeitung der mRNA statt. Dieser Vorgang wird Prozessierung genannt und in Abschnitt 2.1.6.4 genauer erklärt. Nach der Prozessierung verlässt die reife mRNA durch eine Kernpore den Zellkern. Die eigentliche Translation läuft dann ähnlich wie bei Prokaryoten ab und soll nun genauer erklärt werden.

Wie bereits erwähnt, codiert ein mRNA-Strang eine Aminosäure-Sequenz. Es gibt jedoch 20 verschiedene Aminosäuren, aber nur vier verschiedene Basen (A, C, G und U) in der mRNA. Aus diesem Grund bilden immer drei Basen ein sogenanntes Codon und codieren eine Aminosäure. In der Code-Sonne (siehe Abbildung 2.8) können wir ablesen, welche Basentripletts welche Aminosäure codieren.

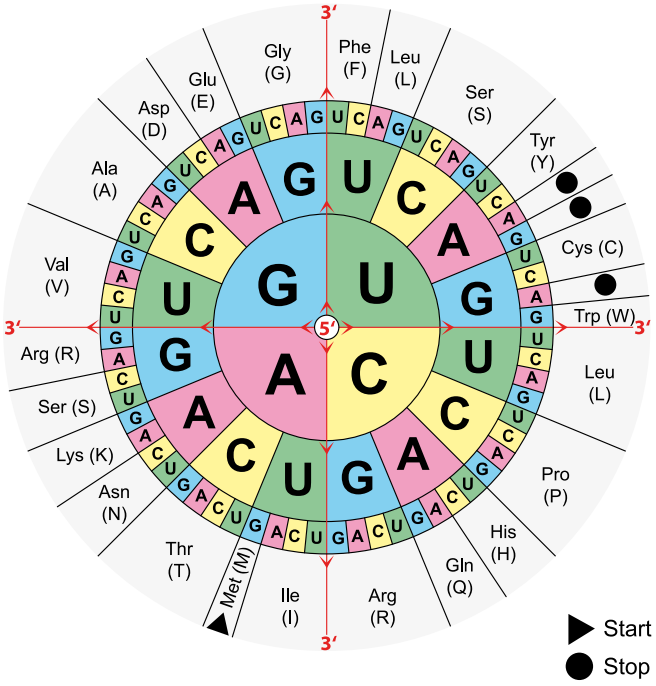


Abbildung 2.8: Genetischer Code. Die Basentripletts der mRNA in 5'-3'-Richtung werden (von innen nach außen gelesen) in die gezeigten Aminosäuren übersetzt. (aus http://commons.wikimedia.org/wiki/File:Aminoacids_table.svg)

Die Translation findet in 5'-3'-Richtung statt, dementsprechend wird die Code-Sonne von innen nach außen gelesen. Die in der Code-Sonne gezeigte Übersetzung von Codons in Aminosäuren wird auch als genetischer Code bezeichnet.

Die Translation übernehmen die Ribosomen einer Zelle. Sie verbinden sich mit dem mRNA-Strang und lesen ihn solange, bis sie auf das Startcodon AUG treffen. Erst

dann wird mit der eigentlichen Translation begonnen. Transfer-RNA-Moleküle (kurz tRNA) transportieren je eine Aminosäure zum Ribosom. Dazu besitzt die tRNA ein zum entsprechenden Codon komplementäres Anticodon, wobei für jede Aminosäure mindestens eine tRNA existiert. Die tRNA ist eine besondere Form der RNA: Sie besitzt teilweise Doppelstrangstrukturen und Schleifen, sowie spezielle modifizierte Basen. Im Ribosom verbindet sich das Codon der mRNA mit dem Anticodon einer passenden tRNA. Das Ribosom fügt die von der tRNA transportierte Aminosäure an die aktuelle Aminosäure-Sequenz an. Dieser Vorgang wird so lange durchgeführt, bis das Ribosom auf eines der drei möglichen Stopp-Codons trifft (UAA, UAG und UGA). Hier endet die Translation und das fertige Polypeptid löst sich vom Ribosom. Typischerweise bestehen Proteine aus 100 bis 800 Aminosäuren, es gibt aber auch weitaus größere Proteine.

2.1.6.4 Prozessierung

Bei Eukaryoten findet zwischen Transkription und Translation eine Nachverarbeitung der mRNA statt. Bei diesem Vorgang, den man auch als Prozessierung bezeichnet, wird die sogenannte *prä-mRNA* in *reife mRNA* umgewandelt. Dabei passiert folgendes:

Im Gegensatz zu Prokaryoten (bei denen die Translation noch während der Transkription stattfindet) muss die mRNA bei Eukaryoten zwischen Transkription und Translation einen weiten Weg zurücklegen. Zellen enthalten Enzyme, die versuchen jegliche mRNA abzubauen. Um die prä-mRNA davor zu schützen wird ein zusätzliches, modifiziertes Guanin-Nukleotid am 5'-Ende angebracht. Dieser Vorgang wird als *Capping* bezeichnet. Auch die *Polyadenylierung*, bei der das 3'-Ende der mRNA mit Adenin-Nukleotiden verlängert wird, dient der Verhinderung des vorzeitigen Abbaus. Für uns sind aber vor allem das Splicing und RNA-Editing interessant, bei der die codierenden Bereiche der prä-mRNA nachträglich verändert werden.

RNA-Editing Beim RNA-Editing werden einzelne Basen der mRNA verändert, sodass zum Beispiel andere Aminosäuren im Protein verbaut werden. Dadurch unterscheidet sich die reife mRNA von der komplementären DNA-Vorlage. Als Beispiel für RNA-Editing sei das *Apolipoprotein B* genannt. Dieses kommt in zwei verschiedenen Formen in unserem Körper vor: In Leberzellen in der langen Form mit 4536 Aminosäuren, in Dünndarmzellen in der kurzen Form mit 2153 Aminosäuren. Beide Formen entstehen aus derselben prä-mRNA und damit aus demselben DNA-Abschnitt. Die Ursache für die zwei verschiedenen Formen ist das erwähnte RNA-Editing. In den Dünndarmzellen wird an einer bestimmten Stelle in der prä-mRNA Cytosin in Uracil umgewandelt. Dadurch entsteht ein Stopp-Codon, sodass eine entsprechend kürzere Aminosäure-Sequenz gebildet wird.

RNA-Editing kommt bei Eukaryoten sehr häufig vor. Teilweise werden hierbei auch spezielle Basen (wie zum Beispiel Inosin) verbaut. Solche speziellen Basen kommen zum Beispiel an vielen Stellen der tRNA vor.

Splicing Unter Splicing bzw. Spleißen versteht man das Herausschneiden bestimmter Bereiche der prä-mRNA. Die prä-mRNA besteht aus Exons und Introns. Die

Introns werden beim Spleißen herausgeschnitten. Die übrigen Abschnitte codieren Proteine und werden Exons genannt.

Beim *alternativen Spleißen* kann eine prä-mRNA auf verschiedene Weisen gespleißt werden. Beispielsweise kann es vorkommen, dass in manchen Zellen bestimmte Exons übersprungen oder andere Exons eingebaut werden. Eine weitere Möglichkeit sind alternative Spleißstellen. Hierbei wird nur ein Teil eines Exons übersprungen. Abbildung 2.9 veranschaulicht die verschiedenen Möglichkeiten des alternativen Spleißens. Ein Gen (also ein DNA-Abschnitt) kann somit wie beim RNA-Editing verschiedene Proteine codieren.

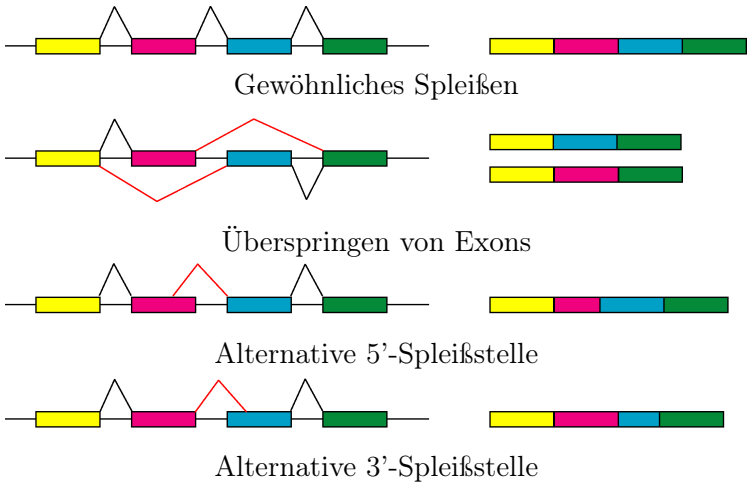


Abbildung 2.9: Verschiedene Möglichkeiten beim alternativen Spleißen. (aus http://commons.wikimedia.org/wiki/File:Alternative_splicing.jpg)

2.1.7 Vererbung

Die auf der DNA enthaltene Information wird an die Nachkommen weitervererbt. Dabei ergibt sich jedoch folgendes Problem: Würden Vater und Mutter jeweils ihre 46 Chromosomen weitervererben, so hätte das Kind 92 Chromosomen. Mit jeder weiteren Generation würde sich die Chromosomenzahl erneut verdoppeln. Aus diesem Grund besitzen Eizellen und Spermien nur einen haploiden Chromosomensatz, der aus 23 Chromosomen besteht, und zwar von jedem Chromosomenpaar genau eines. Alle anderen Zellen unseres Körper sind diploid und besitzen somit 46 Chromosomen. Die Halbierung der Chromosomenzahl findet bei der Meiose statt und soll hier nur kurz erläutert werden soll:

2.1.7.1 Meiose

Zu Beginn der Meiose findet eine Replikation der DNA statt, sodass die Chromosomen der Zelle aus zwei Chromatiden bestehen. Während der ersten Reifeteilung baut sich ähnlich wie bei der Mitose ein Spindelapparat aus. Im Unterschied zur Mitose werden aber nicht die Schwester-Chromatiden voneinander getrennt, sondern die homologen Chromosomen (d.h. die beiden Chromosomen eines Chromosomenpaares).

Nachdem sich die Zelle geteilt hat, liegen zwei Zellen mit haploiden Chromosomensatz vor, deren Chromosomen aber immer noch aus zwei Chromatiden bestehen. Bei der zweiten Reifeteilung werden nun analog zur Mitose die Chromatiden voneinander getrennt, sodass letztendlich vier haploide Zellen mit 1-Chromatid-Chromosomen vorliegen.

Während der ersten Reifeteilung kann ein Crossing-Over stattfinden. Dabei werden DNA-Sequenzen zwischen den homologen Chromosomen ausgetauscht. Die Chromosomen eines Kindes sind somit keine exakten Kopien der Chromosomen der Großeltern. Durch das Crossing-Over erhöht sich die genetische Vielfalt, was für das Überleben einer Art sehr vorteilhaft sein kann.

2.1.7.2 Grundbegriffe der Vererbungslehre

Der Begriff Gen wurde bisher schon mehrfach benutzt, allerdings noch nicht genau definiert. Eine eindeutige Definition ist schwierig zu formulieren, fest steht aber, dass es sich bei einem Gen um einen Abschnitt auf DNA handelt. Im Laufe der Geschichte gab es viele Versuche, den Begriff Gen festzulegen. Nach der Entdeckung der DNA-Struktur (1953) wurde ein Gen als ein Abschnitt auf der DNA gesehen, der die Information zur Herstellung eines Proteins trägt. Jedoch insbesondere bei Eukaryoten codiert ein und derselbe DNA-Abschnitt häufig verschiedene Proteine. Wir haben dies bereits beim RNA-Editing und beim alternativen Spleißen kennen gelernt (siehe Abschnitt 2.1.6.4). Darüber hinaus gibt es DNA-Abschnitte, die zur Herstellung der Transfer-RNA oder anderer besonderer RNA dienen (siehe Abschnitt 2.1.6.3). Aus diesem Grund, wird heutzutage ein Gen als ein DNA-Abschnitt definiert, der die Information zur Herstellung einer biologisch aktiven RNA enthält. Dabei kann es sich um mRNA handeln, die später in ein Protein übersetzt wird, aber auch um andere RNA-Typen, wie zum Beispiel die erwähnte tRNA.

Gene können in zwei oder mehr unterschiedlichen Ausbildungsformen vorliegen, die als *Allele* bezeichnet werden. Unter *Genotyp* verstehen wir die Gesamtheit der Gene eines Individuums, unter *Phänotyp* sein äußeres Erscheinungsbild. *Dominante* Allele wirken bei der Ausbildung des Phänotyps bestimmend und unterdrücken *rezessive* Allele in ihrer Wirkung.

Ein Beispiel soll die gerade genannten Begriffe verdeutlichen. Beim Menschen gibt es vier verschiedene Blutgruppen, die sich anhand der gebildeten Blutgruppensubstanz unterschieden. Bei Blutgruppe A wird die Substanz A gebildet, bei Blutgruppe B die Substanz B, bei AB beide Substanzen, bei 0 keine von beiden. Ursache für die verschiedenen Blutgruppen sind drei Allele eines Gens. Das Allel i^A codiert die Blutgruppensubstanz A, i^B die Substanz B. Beim Allel i wird keine Blutgruppensubstanz gebildet. Die Allele i^A und i^B wirken dominant. Wenn sie vorliegen, wird immer die jeweilige Substanz gebildet. Das rezessive Allel i kommt nur zur Wirkung, wenn kein dominantes Allel vorhanden ist. Da jedes Gen in zweifacher Ausführung vorkommt (nämlich auf den beiden homologen Chromosomen eines Paares), besitzen wir immer zwei Allele. Eines haben wir von der Mutter geerbt, das andere vom Vater. Nun können wir den verschiedenen Phänotypen (also den Blutgruppen) die möglichen Genotypen zuordnen. Bei Blutgruppe 0 müssen beide Allele vom Typ i sein. Der Genotyp ist also ii. Bei Blutgruppe AB werden beide Blutgruppensubstanzen gebildet,

als Genotyp kommt also nur $i^A i^B$ in Frage. Bei Blutgruppe A und B gibt es jeweils zwei Möglichkeiten, nämlich $i^A i$ und $i^A i^A$ (bzw. $i^B i$ und $i^B i^B$).

Liegen auf den homologen Chromosomen dieselben Allele vor, so nennen wir dies reinerbig oder *homozygot*. Sind die Allele unterschiedlich, so liegt das Gen mischerbig oder *heterozygot* vor.

2.1.8 Mutationen

Mutationen sind eine dauerhafte Veränderung des Erbgutes. Man unterscheidet drei verschiedene Arten von Mutationen:

Bei einer **Genom-Mutation** liegt eine Veränderung der Chromosomenzahl vor. Menschen mit einer Genom-Mutationen haben also mehr oder auch weniger als 46 Chromosomen. Eine mögliche Ursache sind Fehler bei der Meiose (oder auch bei Mitose, wenn nur einzelne Zellen des Organismus betroffen sind). Ein bekanntes Beispiel ist die Trisomie 21, besser bekannt als Down-Syndrom. Das 21. Chromosomenpaar liegt hier dreifach vor, was sich bei den Betroffenen unter anderem in einer geistigen Behinderung äußert.

Unter **Chromosomen-Mutationen** versteht man die strukturelle Veränderung eines Chromosoms. Beispielsweise können durch ungleiches Crossing-Over bei der Meiose Teile von Chromosomen verloren gehen. Als Beispiel könnte man das Katzenschrei-Syndrom nennen, bei dem ein kleiner Teil des 5. Chromosoms fehlt.

Für unsere Projektgruppe sind aber vor allem die **Gen-Mutationen** relevant, die im folgenden Abschnitt genauer beschrieben werden.

2.1.8.1 Gen-Mutationen

Eine Gen-Mutation bezeichnet eine Veränderung einer Basenpaarsequenz innerhalb eines Gens. Unterschieden wird zwischen Punktmutationen[5], an denen sich ein einzelnes Nukleotid verändert, und Rasterverschiebungen, welche durch sogenannte **Indels** ausgelöst werden. Dieses Wort vereint die Veränderungen von Basenpaaren durch Einfügen (**In**sertion) oder Entfernen (**Del**etion) (Knippers, 2006). Schwerwiegend werden diese Änderungen, wenn durch die veränderte Sequenz andere Proteine kodiert werden. Bei der bereits genannten **Leseraster-Mutation** entsteht durch Einfügen oder Löschen von 1 oder 2 Basenpaaren[6]. Es verändert sich die Kodierung aller weiteren Aminosäuren, sodass es zu schwerwiegenden Folgen kommen kann.

Ursachen von Mutationen können exogen, also durch Umwelteinflüsse wie Strahlungen oder chemischen Substanzen, oder endogen sein. Darunter fallen beispielsweise auch durch fehlerhafte Replikation verursachte Schäden an der DNA. Diese sind sehr häufig und werden auf bis zu 100.000 pro Zelle und Tag geschätzt (Jiricny, 2013). Durch Reparaturmechanismen kann diese auf eine akzeptable Anzahl von überdauernden Mutationen gesenkt werden (Knippers, 2006). Diese lassen sich in drei Arten

[5] Eine Punktmutation wird auch als *Substitution* oder *Single Nucleotide Polymophism* bezeichnet. Zu letzterem wird auf Abschnitt 2.1.8.2 verwiesen.

[6] Da eine Aminosäure durch jeweils drei Nukleotide kodiert wird, kommt es auch bei Einfügen und Löschen von 4 und 5 (7 oder 8 usw.) Basenpaaren zu Verschiebungen.

unterschreiben, die unter Umständen verschiedene Funktionsstörungen mit sich bringen können.

Stille/Neutrale Mutation Diese Art bezeichnet den Austausch eines Basenpaars, welches nicht zu einer Kodierung einer anderen Aminosäure führt. Bereits in Abbildung 2.8 ist zu sehen, dass für viele Aminosäuren nicht nur eine mögliche Kodierung existiert. Besonders bei einer Veränderung des letzten Basenpaars eines Tripletts stehen die Chancen gut, dass keine andere Aminosäure kodiert wird.

Missense-Mutation Der englischen Bezeichnung entsprechend führt diese Art der Mutation zu einer Sinnveränderung. Durch eine Punktmutation erfolgt die Kodierung einer anderen Aminosäure. Nach Rump (2009) ist zwischen zwei Arten zu unterscheiden. Bei dem *konservativen Aminosäureaustausch* wird eine chemisch ähnliche Aminosäure kodiert, wodurch es nicht zwangsläufig zu Einschränkungen der Proteinfunktion kommt. Jedoch kann der *nicht-konservative Austausch* zu einem Funktionseinschränkung oder gar einem Funktionsverlust führen. Beispiel für eine Erbkrankheit, die durch diese Mutation ausgelöst wird, ist die *Sichelzellanämie* (Rump, 2009).

Nonsense-Mutationen führen zur Erzeugung eines *Stop-Codons* und somit zum Abbruch der Synthese. Schwerwiegende Folgen sind oftmals der Funktionsverlust des Proteins und Erbkrankheiten wie beispielsweise der *Muskeldystrophie* (Rump, 2009).

Die Zelle, in der die Mutation auftritt, ist entscheidend für die Folgen für den Organismus. Liegt eine Mutation in einer Keimzelle vor, hat dies oft keine direkten Konsequenzen für den betroffenen Organismus. Die Veränderungen werden dann erst bei Nachkommen sichtbar, können dort aber das Krebsrisiko erheblich steigern, da alle Zellen des Nachkommen die Mutation in sich tragen (Kassen und Hofmockel, 2000). Tritt eine Mutation jedoch in einer Körperzelle auf, kann dies wie bereits beschrieben zu Funktionsverlusten und im schlimmsten Fall zum Tod der Zelle führen.

2.1.8.2 Einzelnukleotidpolymorphismus

Als Einzelnukleotidpolymorphismus (nach dem englischen Begriff *Single Nucleotide Polymophism*) oder SNP (ausgesprochen: *snip*) bezeichnet man die Variation einzelner Basenpaare in einer DNA, wie in Abbildung 2.10 dargestellt. Sie sind dabei die häufigste Art der Genvarianten und treten durchschnittlich an jedem 1000. Basenpaar auf (Knippers, 2006). Dabei existieren *Hotspots*, Regionen, an denen SNPs häufiger auftreten. Im Mai 2014 waren in der *dbSNP*, einer Datenbank des amerikanischen *National Center for Biotechnologie Information* (NCBI)[7] 62.387.983 SNPs verzeichnet[8]. Auf Grund dieser Vielzahl sind diese Varianten Ursache für die Unterschiede zwischen verschiedenen Menschen, bei beispielsweise Haut- und Haarfarbe oder Körpergröße und -form. Auch sind sie für die Empfänglichkeit von Krankheiten verantwortlich.

[7] http://www.ncbi.nlm.nih.gov/SNP/snp_summary.cgi/snp_summary.cgi?view+summary=view+summary&build_id=141/

[8] Im Jahr 1999 waren erst 7000 SNPs öffentlich bekannt (Brookes, 1999)

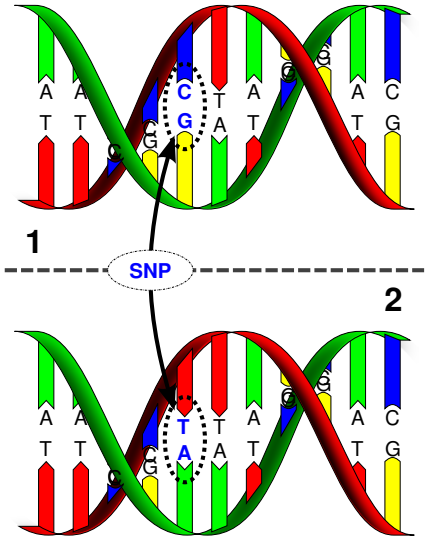


Abbildung 2.10: Visualisierung eines SNP. (aus http://commons.wikimedia.org/wiki/File:Dna-SNP.svg)

Da die Veränderungsrate bei 10^{-8} Änderungen pro Nukelotid und Generation liegt, sind einzelne Allele sehr stabil (Brookes, 1999) (Li et al., 1996).

Um die DNA eines Organismus überhaupt untersuchen zu können, muss diese sequenziert werden. Die Grundideen und verschiedene Arten der Sequenzierung werden im Abschnitt 2.2 beschrieben.

2.1.8.3 Tumore

Ein wichtiges Thema, welches bereits im Titel dieses Projektes angesprochen wird, ist Krebs. Umgangssprachlich werden damit bösartige Tumore, also schädliche Gewebeneubildungen, bezeichnet. Sie zeichnen sich durch unkontrolliertes Wachstum aus, dringen auch in benachbarte Zellen ein und bilden Metastasen in anderen Organen (Kassen und Hofmockel, 2000).

In einem gesunden Organismus besteht ein Gleichgewicht zwischen Vermehrung der Zellen durch Teilung und dem Zelltod (Kassen und Hofmockel, 2000). Es kann in beide Richtungen verschoben werden und die Entstehung von Tumoren vereinfachen. Diese entstehen durch eine Häufung von *unreparierten* Mutationen und kann verschiedene Arten von Genen betreffen (Kassen und Hofmockel, 2000):

- **Protoonkogene** steuern das Wachstum von Zellen. Durch eine Mutation in einem Allel entwickelt sich Onkogene, welche sich dem Zellzyklus entziehen und zum unkontrollierten Wachstum der Zelle führen können.

- **Tumorsuppressorogene** dagegen steuern den Zelltod, um das bereits beschriebene Gleichgewicht einhalten zu können. Hier kommt es zum Defekt, wenn Mutationen in beiden Allelen vorliegen. Das Wachstum der Zelle lässt sich nicht mehr kontrollieren.

- Mutationen in **Reparaturgenen** können dafür sorgen, dass die häufig auftretenden Schädigungen nicht mehr korrigiert werden können und somit die Mutationsrate steigt.

Diese Mutationen können spontan auftreten oder durch sogenannte Karzinogene ausgelöst werden, welche chemischen, physikalischen oder viralen Ursprungs sind. Durch rezessive Mutationen an bereits genannten Tumorsuppressorogenen kann bereits eine genetisch vererbte Veranlagung bestehen, einen Tumor zu entwickeln (Kassen und Hofmockel, 2000). Die Untersuchung des Erbguts kann also helfen, bestimmte Veranlagungen zu entdecken. Mittels der DNA-Sequenzierung, welche im folgenden Abschnitt erläutert wird, können die Informationen des Erbguts in eine lesbare Form gebracht werden.

2.2 Sequenzierung

Die Sequenzierung bezeichnet eine Methode zur Bestimmung der Nukleotid-Abfolge der untersuchten DNA. Seit 1977 wurden dabei verschiedene Methoden entwickelt, welche sich in Funktionsweise und Leistung deutlich unterscheiden. Die ersten entwickelten Verfahren waren dabei die chemische Methode von Maxam und Gilbert und die deutlich überlegene Kettenabbruchmethode von Sanger et al. (1977), welche nachfolgend vorgestellt wird. Alle weiteren Methoden, die zeitlich später entwickelt wurden, werden auch als **Next-Generation Sequencing** bezeichnet, von denen einige weitere im Verlauf dieses Kapitels vorgestellt werden.

Alle Verfahren haben gemeinsam, dass sie nicht in der Lage sind, die komplette DNA in einem einzigen Durchgang zu sequenzieren. Stattdessen bestehen die Ausgaben aus sehr vielen, sich überlappenden und möglicherweise redundanten DNA-Fragmenten, sogenannten Reads. Die jeweiligen Längen sind teilweise auf die chemischen Prozesse der Methoden zurückzuführen, aber auch auf die Tatsache, dass die Wahrscheinlichkeit von falsch erfassten Basen mit wachsender Readlänge steigt (Stanke, 2013). Dies wird als Sequenzierfehler bezeichnet.

2.2.1 Kettenabbruchmethode

Die Kettenabruchmethode (auch als *Dideoxy-Methode* oder *Sanger-Sequenzierung* bezeichnet) sequenziert eine eingebrachte DNA mittels Synthese durch eine Polymerase (Jansohn et al., 2011).

Zur Vorbereitung muss die zu untersuchende DNA in hoher Stückzahl verfügbar sein. Die Klonierung erfolgt beispielsweise mit der in Abschnitt 2.1.5.2 vorgestellten Technik *PCR*. Die Polymerase verlängert nun einen eingebrachten Primer, dessen Sequenz bekannt ist, so dass ein Komplement entsteht. In jedem Zyklus wird ein markiertes Nukleotid, ein sogenanntes *Didesoxyribonukleosid-Triphosphat* (ddNTP), eingebracht. Diese besitzen am 3' Ende keine Hydroxygruppe. Da sich dort normalerweise die Verbindung zum nächsten Nukleotid befindet, kommt es beim Einbau eines ddNTP zum Abbruch der Kette und die Synthese terminiert. Dadurch entsteht eine Vielzahl von DNA-Fragmenten mit unterschiedlichen Längen. Das Ablesen der

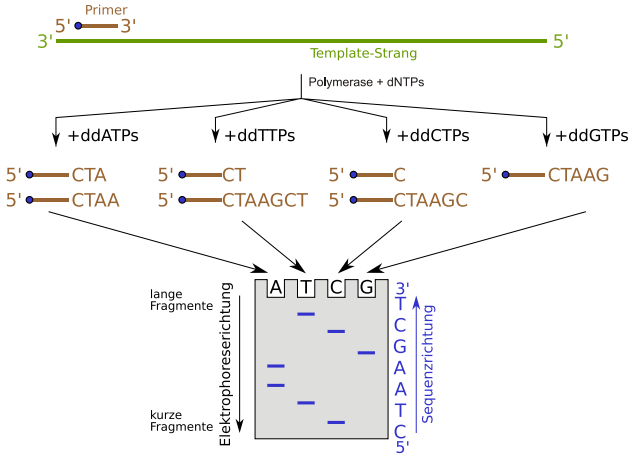


Abbildung 2.11: Visualisierung der Kettenabbruchmethode nach Sanger. (aus http://commons.wikimedia.org/wiki/File:Didesoxy-Methode.svg)

Kodierung erfolgt in einer *Elektrophorese*. Kurze Fragmente wandern, wie in Abbildung 2.11 illustriert, am weitesten. Durch die Markierungen des jeweils letzten Nukleotids jedes Fragments lässt sich der genetische Code schrittweise erweitern.

In der Vergangenheit nutzte man eine radioaktive Markierung der Nukleotide. Um das Gefahrenrisiko zu senken und dadurch die Nutzung zu vereinfachen, werden nun verschiedene fluoreszierende Markierungen verwendet. Die entstehenden Reads besitzen eine Länge von bis zu 1000 Basenpaaren, deren Sequenzierung Kosten in Höhe von \$0,50 pro Tausend Basenpaaren verursachten. Auch wurde eine hohe Genauigkeit von bis zu 99,9% erzielt (Shendure und Ji, 2008). Leider ist die Kettenabbruchmethode aufwendig und kostet viel Zeit. Daher wurden neue Sequenziertechniken entwickelt, welche nachfolgend vorgestellt werden.

2.2.2 Sequenzierung durch Synthese (SBS)

Die Sequenziermethode *SBS* (sequencing by synthesis) nutzt ebenfalls einen Abbruch der Kette zur Nukleotidbestimmung aus. Im Gegensatz zur Methode von Sanger ist dieser Abbruch aber reversibel. In jedem Zyklus wird versucht, genau ein farblich markiertes Nukleotid an den komplementären Strang der zu untersuchenden DNA zu binden. Dieser wirkt als *reversible Terminator* und unterbricht die Bindung von weiteren Molekülen. Nachdem die Fluoreszenz gemessen wurde und damit das gebundene Nukleotid identifiziert wurde, muss der fluoreszierende Terminator von der DNA gespalten werden, bevor der nächste Zyklus beginnt.

Wahlmöglichkeiten gibt es bei der Art der Markierung. Die erste Möglichkeit besteht darin, gleichzeitig alle vier verschiedenen Nukleotide hinzuzufügen, wobei jedes eine andere Fluorophore zur Identifikation erhält. Die andere Möglichkeit sieht für jedes Molekül denselben Farbstoff vor. In jedem Zyklus muss dabei darauf geachtet werden, dass ein anderes dNTP hinzugefügt wird, um die korrekte Sequenz ermitteln zu können.

Die entstehenden Reads hatten bei Verwendung der ersten Maschinen dieser Plattform eine Länge von ca. 30 Basenpaaren, die neueren Modelle wie der Sequenzierer *MiSeq* der Firma *Illumina*[9] erreichen Längen von bis zu 300 Basenpaaren.

2.2.3 Pyrosequenzierung

Das Grundvorgehen der Pyrosequenzierung entspricht der Beobachtung einer DNA-Replikation. Zur Vorbereitung werden die folgenden vier Enzyme zugegeben: DNA-Polymerase, ATP-Sulfurylase, Luciferase, Apyrase (Ahmadian et al., 2006).

Die eigentliche Sequenzierung läuft nun zyklenweise ab. Die Polymerase sucht ein freies Nukelotid, welches an den komplementären DNA-Strang andocken kann. Iterativ werden jeweils die Desoxyribonukleosidtriphosphate dATP, dCTP, dGTP, dTTP hinzugefügt. Kann ein Nukleotid andocken, wird durch die Polymerase *Pyrophosphat* freigesetzt, welches dann durch ein weiteres Enzym, der ATP-Sulfurylase, zu *Adenosintriphosphat (ATP)* umgewandelt wird. Dieser Vorgang ist in Abbildung 2.12 visualisiert. Die Luciferase, welche ursprünglich aus Glühwürmchen extrahiert wurde, katalysiert das ATP zu einem Lichtblitz (Ahmadian et al., 2006). Tritt ein Lichtblitz auf, konnte die Polymerase das aktuelle Nukleotid binden. Nach mehreren Zyklen und der Zugabe der verschiedenen dNTPs lässt sich der gesamte Read konstruieren. Nach jeder Zugabe eines dNTP wird ein Lichtblitz registriert und der jeweiligen Base zugeordnet (Shendure und Ji, 2008). Auch die Intensität wird gemessen, da diese proportional zur Anzahl der eingebauten Nukleotide steigt. Dieser Zusammenhang ist auch in Abbildung 2.13 dargestellt. Für die Sequenz bedeutet dies, falls ein Lichtblitz mit doppelter Intensität auftritt, kommt das entsprechende Nukleotid zweimal vor.

[9] http://systems.illumina.com/systems/sequencing.ilmn

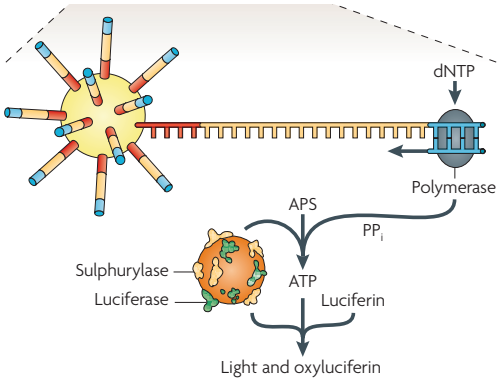


Abbildung 2.12: Darstellung der Pyrosequenzierung. (aus Metzker (2010))

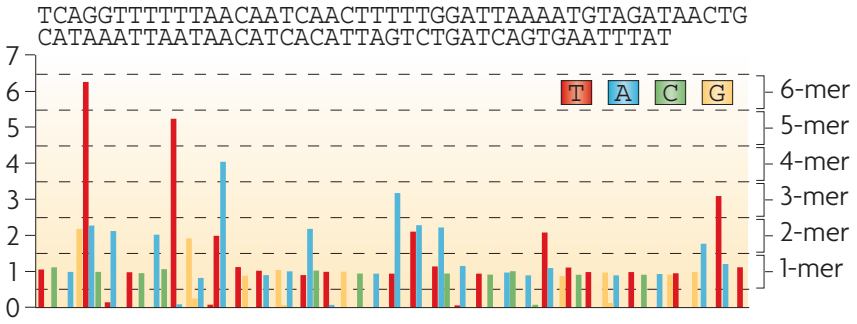


Abbildung 2.13: Intensität der Lichtblitze. (aus Metzker (2010))

Mit Hilfe der Pyrosequenzierung ist eine Generierung von hunderttausenden Reads mit einer Länge von bis zu 400 Basenpaaren möglich. Die neueren Modelle des Herstellers Roche[10] erreichen im Maximum auch Längen von bis zu 1000 Basenpaaren. Auch ist es möglich, dass verschiedene Proben gleichzeitig analysiert werden können, was sich kostensenkend auswirkt (Siqueira Jr et al., 2012). Durch die langen Reads ist eine zuverlässige Erkennung von SNPs möglich, jedoch ist die Fehlerrate bei der Erkennung von Indels hoch, da *Homopolymere*, also Wiederholungsfolgen derselben Base, nur anhand der verschiedenen Intensität der Lichtblitze erkannt werden können (Shendure und Ji, 2008). Durch Normalisierung der Messwerte ist der lineare Zusammenhang zwischen Lichtintensität und Anzahl der eingebauten Basen bis zu Wiederholungen von höchstens 8 Basenpaaren gewährleistet, wodurch bei längeren Reads die Genauigkeit sinkt (Margulies et al., 2005).

2.2.4 Echtzeit-Sequenzierung

Bei der Echtzeit-Sequenzierung kann die ermittelte DNA-Sequenz direkt durch Beobachtung der Polymerase abgelesen werden. Im Gegensatz zur Pyrosequenzierung muss bei diesem Ansatz die DNA-Synthese nicht gestoppt werden. Einzelne Polymerasen werden an der Oberfläche eines Detektors angebracht, der Nukleotide erkennen kann, die um ein fluoreszierendes Phosphat erweitert wurden (Eid et al., 2009), welche in hoher Anzahl hinzugegeben werden. In jedem Zyklus (dargestellt in Abbildung 2.14) wird versucht, eines der Nukleotide einzubinden [1]. Ist der Versuch erfolgreich, löst sich das farblich markierte Pyrophosphat und löst währenddessen einen messbaren Lichtpuls aus [2]. Der Detektor misst diesen und setzt daraus die Sequenz zusammen. Ist das Pyrophosphat komplett abgespalten, lässt der Lichtpuls nach [3]. Das Molekül wird durch die Polymerase weiterbewegt [4] und ein neuer Zyklus beginnt [5].

[10] http://454.com/products/gs-flx-system/index.asp

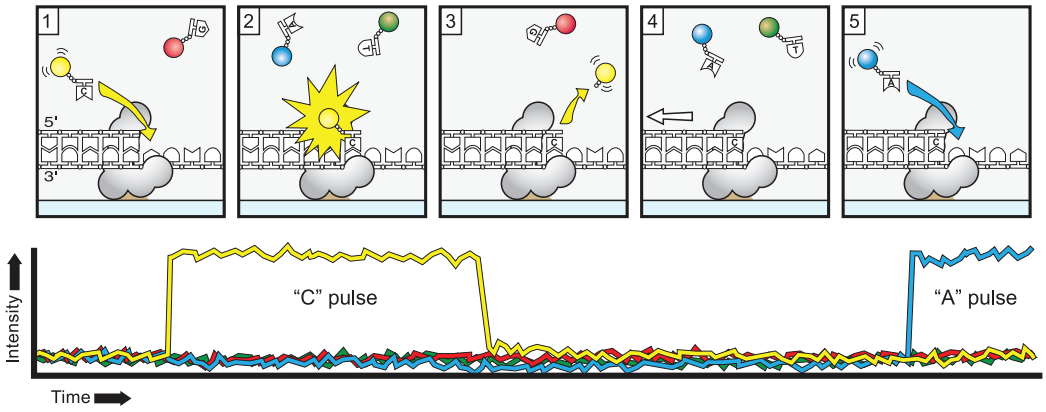


Abbildung 2.14: Darstellung der Echtzeitsequenzierung. (aus Eid et al. (2009))

Vorteilhaft wirken sich hohe Readlängen (durchschnittlich 4200 bis 8500 Basenpaaren[11] und die hohe Geschwindigkeit der Sequenzierung aus, welche nur von der Synthesegeschwindigkeit[12] der verwendeten Polymerase abhängig ist (Metzker, 2010). Die Genauigkeit eines einzelnen Sequenzierungsdurchgangs ist relativ ungenau ($\sim$ 83%). Durch mehrfache Wiederholung der Sequenzierung desselben Reads lässt sich diese Genauigkeit auf über 99% erhöhen (Eid et al., 2009).

2.3 Readmapping

In diesem Kapitel wird das eigentliche Problem erläutert, mit dem sich unsere Projektgruppe auseinandergesetzt hat. Es handelt sich um das sogenannte *Readmapping*, einen Prozess, bei dem für kurze Gensequenzen eines Individuums die passenden Stellen in einem Referenzgenom gesucht werden. Die kurzen Gensequenzen, auch als *Reads* bezeichnet, werden durch Sequenzierung eines individuellen Genoms gewonnen (vgl. Abschnitt 2.2). Das Referenzgenom entspricht vom Aufbau und der Länge einem vollständigen Humangenom. Allerdings stammt es nicht zwingend von einem einzelnen Individuum, sondern kann Genvarianten von vielen Individuen aufweisen, um Letztere möglichst gut zu repräsentieren.

Unsere Projektgruppe hat einen *Readmapper* entwickelt, der die Zuordnung zwischen Reads und ihren Positionen im Referenzgenom leistet. Im Folgenden wird zunächst eine algorithmische Problemstellung für diesen Prozess formuliert, sowie auf mögliche Schwierigkeiten eingegangen. Wie der Lösungsansatz der Projektgruppe aussieht und welche praktische Relevanz das Readmapping hat, wird anschließend erläutert.

[11] http://www.pacificbiosciences.com/products/smrt-technology/smrt-sequencing-advantage/
[12] Diese Geschwindigkeiten liegen zwischen 0.7 und 1.5 Basen pro Sekunde (Eid et al., 2009).

2.3.1 Formale Problemstellung

Das Readmapping-Problem lässt sich zu einer Mustersuche in einem Text abstrahieren. Die vier Basen Adenin (**A**), Cytosin (**C**), Guanin (**G**) und Thymin (**T**) stellen das Alphabet Σ dar. Das Referenzgenom kann also als ein Text über dem Alphabet Σ aufgefasst werden, während die Reads die Suchmuster für diesen Text darstellen. Readmapping ist allerdings deutlich mehr als eine simple Textsuche. Das sequenzierte Genom zur Gewinnung der Reads stimmt im Allgemeinen nicht vollständig mit dem Referenzgenom überein, sodass die Reads nicht zwingend exakt im Referenzgenom vorkommen müssen. Da aber beispielsweise menschliche Genome zu etwa 99% übereinstimmen (Reich et al., 2002), ist es dennoch sinnvoll von einer Position des Reads im Referenzgenom zu sprechen, auch wenn es lokal einige Unterschiede gibt. Beim Sequenzieren können zusätzlich noch Lesefehler auftreten, die die Reads leicht verfälschen. Solche Lesefehler sind meist Substitutionen, bei denen an einer festen Position eine falsche Base gelesen wurde. Etwas seltener treten Deletionen bzw. Insertionen auf, wodurch im resultierenden Read Basen fehlen bzw. zusätzliche Basen vorhanden sind, welche in der tatsächlichen DNA-Probe gar nicht vorlagen.

Für einen Readmapper bedeutet dies, dass er eine Fehlertoleranz aufweisen muss, um Reads mit Fehlern überhaupt korrekt zuordnen zu können. Durch die Fehlertoleranz ergibt sich wiederum die Gefahr, dass Reads auch irrtümlich einer falsche Position zugeordnet werden, da die Unterschiede innerhalb der vorgegebenen Toleranz liegen. Readmapping ist daher zwangsläufig ein Kompromiss zwischen einer hohen Zuordnungrate aller Reads und einer hohen Quote an korrekt zugeordneten Reads. Die Ausgabe eines Readmappers beschränkt sich zudem nicht nur auf eine einfache Position, an der ein Read im Referenzgenom vorkommt, sondern umfasst üblicherweise ein *Alignment* des Reads. Alignments enthalten neben der Position des Reads auch zeichenweise seine Unterschiede und Gemeinsamkeiten zwischen zum Ausschnitt des Referenzgenom und werden in Kapitel 4.2 näher erklärt.

Zusammengefasst muss ein Readmapper also folgendes Problem lösen:

Problem ReadMapping

Gegeben Referenzgenom $t \in \Sigma^*$, Menge von Reads $P := \{p_1, \ldots, p_l\}$ mit $p_i \in \Sigma^* \; \forall 1 \leq i \leq l$.

Gesucht Für jeden Read $p_i \in P$: Eine Menge J von Startposition in t, sodass für alle $j \in J$ die Anzahl an Fehlern zwischen p_i und dem Suffix $t_j \ldots t_{|t|}$ minimal ist bezüglich aller Startposition in t. Für jede Startposition $j \in J$ ist zusätzlich ein Alignment A zwischen p_i und $t_j \ldots t_{j+|p_i|}$ gesucht.

Ein *Alignment* zwischen zwei Strings enthält zeichenweise alle Differenzen zwischen den beiden Strings und erlaubt eine genaue Zuordnung, welches Zeichen des Reads auf welches Zeichen des Referenzgenoms abgebildet wird. Eine formale Definition von Alignments befindet sich in Definition 4.2.3 in Abschnitt 4.2.

Eine Besonderheit beim Readmapping sind die Größe des Referenzgenoms und die Anzahl der Reads. Das menschliche Genom entspricht einem Text mit drei Milliarden Zeichen und aktuelle Sequenziertechnologien erzeugen mehrere hundert Millionen

Reads mit einer Länge von etwa 100 Zeichen bzw. Basenpaaren. Dadurch sind klassische Textsuchalgorithmen, die den Text für jedes Muster sequenziell durchlaufen, für einen Readmapper nicht geeignet, da die Laufzeit in der Praxis viel zu hoch wäre. Viele bestehende Readmapper nutzen einen Index über dem Referenzgenom, sodass die Verarbeitung für jeden einzelnen Read nur von dessen Länge, nicht jedoch von der Länge des Gesamtgenoms abhängt.

2.3.2 Variantentolerantes Readmapping

Für das humane Referenzgenom gibt es zusätzlich eine Aufzählung aller häufigen und bekannten Genvarianten. Eine Anwendung des Readmappings stellt die Entdeckung neuer, bisher unbekannter Genvarianten dar. Dies geschieht dadurch, dass die Fehler in der Alignierung eines Reads genauer analysiert werden und von diesen Fehlern auf mögliche unbekannte Genvarianten geschlossen wird. Eine Differenz zwischen dem Read und dem Referenzgenom kann aber zunächst mehrere Ursachen haben:

1. Sequenzierfehler führen sehr wahrscheinlich dazu, dass der Read an dieser Stelle nicht mit dem Referenzgenom übereinstimmt.

2. Der Read enthält eine der bekannten Varianten und divergiert damit ebenfalls vom Referenzgenom.

3. Falsche Position bzw. falsches Alignment des Reads.

4. Es liegt tatsächlich eine unbekannte Genvariante vor.

Sequenzierfehler sind technologisch bedingt und für einen Readmapper ohne Weiteres nicht von neuen Genvarianten zu unterscheiden. Bekannte Genvarianten können dagegen berücksichtigt werden. Dazu vergleicht der Readmapper einen Read nicht nur mit dem Referenzgenom selbst, sondern zusätzlich mit allen bekannten Varianten des Genoms. Dazu werden alle möglichen *Varianten* des Referenzgenoms betrachtet, die durch Anwendung einer Kombination von bekannten Genvarianten entstehen können. Jeder Read wird beim Alignieren mit der Variante des Referenzgenoms verglichen, die zur geringsten Anzahl an Fehlern führt. Ein Readmapper, der auf diese Weise arbeitet, heißt *variantentolerant*.

Der Vorteil von variantentoleranten Readmappern besteht darin, dass Differenzen zum Referenzgenom, die auf bekannte Varianten zurückzuführen sind, gar nicht mehr als solche aufgeführt werden. Es bleiben also nur noch Sequenzierfehler übrig, die sich eventuell durch mehrfaches Sequenzieren verhindern lassen, und jene unbekannten Genvarianten, die letztlich von Interesse sind. Dadurch, dass insgesamt weniger Differenzen pro Read vorhanden sind, ergibt sich auch bei der Fehlertoleranz mehr Spielraum. Da bekannte Genvarianten nicht mehr als „Fehler“ gezählt werden, kann die Toleranz des Readmappers insgesamt gesenkt werden, um die Anzahl von *false positives*, d. h. Reads, die an eine falsche Position aligniert wurden, zu reduzieren. Ein solcher variantentoleranter Readmapper stellt das Ziel unserer Projektgruppe dar.

2.3.3 Mapping und Alignierung

Wir haben den Readmapping-Prozess aus Sicht der Reads in zwei Teilprobleme aufgeteilt:

1. Mapping: Der Read wird grob einer Stelle im Referenzgenom zugeordnet.
2. Alignierung: Ausgehend von der groben Zuordnung wird das exakte Alignment des Reads berechnet.

Dazu verwenden wir zwei verschiedene Algorithmen, die auf das jeweilige Teilproblem spezialisiert sind. Für das Mapping teilen wir das Referenzgenom in Abschnitte auf, die etwa der Länge eines Reads entsprechen. Anschließend wenden wir *Locality-Sensitive Hashing* an, um jedem Read mögliche Genomabschnitte zuzuordnen, die zu diesem Read passen. Im Alignierungsschritt wird der Read an jeden dieser Kandidaten aligniert, wobei das beste gefundene Alignment die Ausgabe des Reads darstellt. Die beiden Verfahren werden in Kapitel 4 näher vorgestellt.

2.4 Datenkodierungen und Dateiformate

Zur Speicherung der bei der DNA-Sequenzierung anfallenden Daten haben sich im Laufe der Jahre diverse De-facto-Standards (Cock et al., 2010) herausgebildet. Die rohen Sequenzen und die daraus aufbereiteten Daten liegen dabei meist in menschenlesbaren Textdateien vor.

In diesem Kapitel sollen die von uns verwendeten Dateiformate und die dort verwendeten Kodierungen beschrieben werden. Hierzu zählen das **FASTA**-Format in dem das Referenzgenom vorliegt sowie das **FASTQ**-Format, welches für die Reads verwendet wird. Weiterhin werden die zum Referenzgenom bekannten Varianten als **VCF**-Dateien bereitgestellt. Das **SAM**- bzw. **BAM**-Format wird letztlich zur Ausgabe der Alignierungen der Reads verwendet.

2.4.1 IUPAC-Alphabet

Bevor die eigentlichen Dateiformate beschrieben werden, soll hier auf eine allgemein verwendete textuelle Darstellung der DNA eingegangen werden.

Die Kodierung der Nukleinbasensequenzen erfolgt grundsätzlich als String über dem Alphabet $\Sigma = \{A, C, T, G\}$. Da jedoch die DNA-Sequenzierung fehlerbehaftet ist und mehr als ein einziges Referenzgenom existiert, gibt es das Bedürfnis nach einem erweitertem Alphabet, welches auch nicht eindeutig bestimmte Basen berücksichtigen kann. Die *International Union of Pure and Applied Chemistry* (IUPAC) empfahl die Verwendung ihres Alphabets, welches gemeinhin als **IUPAC-Alphabet**[13] bezeichnet wird. Zusätzlich zu den eindeutigen Basenzuordnungen werden, wie in Tabelle 2.1 zu sehen, eine Reihe weiterer Zeichen eingeführt. Mit Hilfe der zusätzlichen Zeichen kann ein IUPAC-String mehrere DNA-Strings identischer Länge, welche sich nur

[13] http://www.chem.qmul.ac.uk/iubmb/misc/naseq.html

IUPAC-Symbol	Zusammensetzung	**G**	**C**	**A**	**T**
U	**Uracil**	0	0	0	0
T	**Thymin**	0	0	0	1
A	**Adenin**	0	0	1	0
W	A, T	0	0	1	1
C	**Cytosin**	0	1	0	0
Y	C, T	0	1	0	1
M	A, C	0	1	1	0
H	C, A, T	0	1	1	1
G	**Guanin**	1	0	0	0
K	G, T	1	0	0	1
R	A, G	1	0	1	0
D	A, G, T	1	0	1	1
S	C, G	1	1	0	0
B	C, G, T	1	1	0	1
V	A, C, G	1	1	1	0
N	A, C, G, T	1	1	1	1

Tabelle 2.1: IUPAC-Alphabet und Binärkodierung der Kombinationen von Nukleinbasen.

in einzelnen Basen unterscheiden, repräsentieren. Ein einzelnes IUPAC-Zeichen kodiert hierbei für eine Position der repräsentierten DNA-Strings die Kombination aller dort vorhandenen Basen. Durch diese Ein-Zeichen-Kodierung kann einer Position im DNA-String genau ein Zeichen im IUPAC-String zugeordnet werden.

Das IUPAC-Alphabet wird in allen nachfolgend beschriebenen Dateiformaten verwendet. Neben den Zeichen A, C, G, T, welche die eindeutige Zuordnung der jeweiligen Basen aufzeigen, kommt dem Zeichen N noch eine besondere Bedeutung zugute. An einer Position im IUPAC-String, welche mit einem N kodiert ist, kann jede beliebige der vier Basen auftreten. Daher wird dieses Zeichen auch für nicht eindeutig sequenzierte Basen in Reads sowie als Platzhalter für noch unbestimmte Basen in den Lücken (engl.: gaps) des Referenzgenoms verwendet.

Durch die Verwendung einer Vier-Bit-Kodierung, die ebenfalls in Tabelle 2.1 dargestellt ist, kann das mögliche Vorhandensein einer Base an einer Position mittels einfacher Bitoperationen (bitweisem UND) überprüft werden. Die in der Tabelle dargestellten Bitmuster werden auch zur Behandlung der SNPs bei der Alignierung (siehe Abschnitt 4.2.2) benutzt.

2.4.2 FASTA

Reine DNA-Sequenzen, wie etwa das beim Readmapping verwendete Referenzgenom, werden üblicherweise im **FASTA**-Format gespeichert. Es stellt ein sehr einfach gehaltenes Textformat dar, welches lediglich aus zwei Komponenten besteht. Die erste Zeile eines FASTA-Eintrags wird mit dem ASCII-Zeichen > eingeleitet und dient zur Identifizierung der Sequenz. Alle weiteren Zeilen bilden aneinandergefügt die eigentliche Sequenz.

Obwohl das FASTA-Format als De-facto-Standard gilt, existiert kein explizites, den Standard beschreibendes Dokument (Cock et al., 2010). Ursprünglich wurde das Format als Eingabe für das gleichnamige Programm entwickelt, welches der Suche und dem Vergleich von DNA- und Aminosäuresequenzen dient (Pearson und Lipman, 1988).

Der Inhalt der ersten Zeile ist nach dem `>` frei wählbar. Es existieren jedoch verschiedene Richtlinien zu ihrer Gestaltung, die von den jeweiligen Datenanbietern vorgegeben werden. Die nachfolgende Sequenz kann entweder eine nach dem oben beschriebenen IUPAC-Alphabet kodierte Nukleotidsequenz sein, oder sie beschreibt eine Aminosäuresequenz. Da für unsere Zwecke nur die DNA-Sequenzen von Belang sind, wird hier nicht weiter auf die Kodierung der Aminosäuren eingegangen. Für Basensequenzen sind die in Tabelle 2.1 aufgeführten IUPAC-Zeichen sowohl in Groß- als auch Kleinschreibung gültige Zeichen. Die kleingeschriebenen Zeichen sind prinzipiell mit ihren Großbuchstabenpendants identisch, werden allerdings auch zur zusätzlichen Kennzeichnung von sich wiederholenden Abschnitten (engl.: repeats) im Humangenom verwendet. Zusätzlich zu den IUPAC-Zeichen ist noch der Bindestrich `-` als Zeichen erlaubt, welcher für eine Lücke unbestimmter Länge in der Sequenz steht.

In einer FASTA-Datei können zudem mehrere Sequenzen gespeichert sein, indem einfach die zuvor beschriebenen FASTA-Einträge aneinandergehängt werden, sodass jede Sequenz mit einer mit `>` beginnenden Zeile eingeleitet wird. Dadurch kann etwa das gesamte menschliche Genom in einer FASTA-Datei gespeichert sein, die aus mehreren, den Chromosomen (sowie dem Mitochondrium) entsprechenden Sequenzeinträgen besteht.

2.4.3 FASTQ

Reads werden häufig im **FASTQ**-Format gespeichert, welches das FASTA-Format um Qualitätsangaben für die korrekte Erfassung der jeweiligen Basen erweitert. Ebenso wie FASTA ist auch FASTQ ein De-facto-Standard ohne explizite Spezifikation. Aufgrund dessen haben Cock et al. (2010) eine ausführliche Beschreibung des Formates geliefert, welche als Spezifikationsersatz dienen soll.

Ein Eintrag in einer FASTQ-Datei besteht aus vier Komponenten. Die erste Zeile dient, entsprechend dem FASTA-Format, als Identifikation, wird allerdings mit dem Zeichen `@` anstatt `>` eingeleitet. Die zweite Komponente ist wie bei FASTA-Einträgen die Sequenz, welche sich über mehrere Zeilen erstrecken darf. Als Drittes folgt eine Zeile, welche mit einem `+` eingeleitet wird und optional eine Kopie der Identifikationsdaten aus der ersten Zeile enthält. Meist besteht diese Zeile nur aus einem `+`, da aus Speicherplatzgründen auf die redundante Kopie verzichtet wird. Die letzte Komponente besteht aus der Konkatenation der Qualitätswerte aller Sequenzpositionen, welche nach dem unten beschriebenen Verfahren als druckbare ASCII-Zeichen kodiert sind.

Für die Angabe der Sequenzierqualität der einzelnen Basen existieren verschiedene Metriken. Eine häufig verwendete Metrik, bezeichnet als **Sanger**- oder **Phred**-Qualitätswert, berechnet sich wie folgt (Ewing und Green, 1998):

$$Q = -10 \log_{10} P \quad \text{für } P \in (0, 1]$$

Dabei bezeichnet P die Wahrscheinlichkeit, dass die zugehörige Base falsch erfasst wurde. Ein Qualitätswert Q von 30 repräsentiert somit eine Fehlerrate von 0.1%. Theoretisch bewegt sich Q im Wertebereich von $\mathbb{R}_{\geq 0}$, jedoch liegen die tatsächlichen Werte üblicherweise weit unter 100 und werden zudem ganzzahlig gerundet. Die FASTQ-Variante 1.3+ der Firma Illumina hat beispielsweise einen maximalen Wert von 62 (Cock et al., 2010). Dies würde einer Wahrscheinlichkeit für die korrekte Erfassung der Base von $\approx 0.999999\%$ entsprechen.

Die Umrechnung der Fehlerwahrscheinlichkeiten in diese Qualitätswerte wird aus Platzgründen vollzogen. Die Qualitätswerte werden durch die druckbaren ASCII-Zeichen, welche von ASCII-Wert 33 (Zeichen `!`) bis Wert 127 (Zeichen `~`) reichen, kodiert. Je Hersteller und Softwareversion werden allerdings verschiedene Offsets bei der Umrechnung von Qualitätswert nach ASCII-Zeichen verwendet. Wird der Wert 33 als Offset gewählt, entspricht etwa das Zeichen `A` (ASCII-Wert 65) dem Qualitätswert $Q = 65 - 33 = 32$. Für das zuvor genannte FASTQ 1.3+ von Illumina beträgt der Offset hingegen 64 (Zeichen `@`), wodurch ein `A` den Qualitätswert $Q = 65 - 64 = 1$ kodiert. Die kodierte Qualitätsangabe wird für jede sequenzierte Base in derselben Reihenfolge in der Datei festgehalten, so dass eine zeichengenaue Zuordnung möglich ist.

Anzumerken ist noch, dass auch die Zeichen `@` und `+` bei der Kodierung der Qualitätswerte verwendet werden können. Daher kann nicht davon ausgegangen werden, dass jede Zeile, die mit einem der beiden Zeichen beginnt, eine der Identifikationszeilen ist. Aufgrund dessen wird angeraten die Sequenz- und Qualitätsstrings ohne Zeilenumbrüche zu speichern, um ein einfaches Parsing der FASTQ-Dateien zu ermöglichen.

2.4.4 VCF

Im „Variant Call Format“ (**VCF**) können genetische Variationen in Bezug zum Referenzgenom kompakt gespeichert werden (Danecek et al., 2011). Im Grunde besteht das Format aus einer tabulatorseparierten Textdatei, welche zeilenweise die Unterschiede der Varianten zum Referenzgenom unter Angabe der betreffenden Referenzpositionen beschreibt.

Die VCF-Dateien beginnen mit einem Dateikopf, der eine beliebige Anzahl von Zeilen umfasst, welche mit den Zeichen `##` eingeleitet werden und Metainformationen beinhalten. Weiterhin gehört noch eine mit einem einzigen `#` beginnende Zeile zum Dateikopf, die den Tabellenkopf für die nachfolgenden Varianteneinträge enthält.

Es gibt acht zwingend erforderliche Spalten im VC-Format. Die Spalte *CHROM* gibt das zugehörige Chromosom und *POS* die Position im Chromosom an, an welcher die Variante beginnt. In der Spalte *ID* kann der Variante eine eindeutige Identifikation zugewiesen werden. *REF* gibt einen Ausschnitt aus der Referenz, beginnend bei Position *POS*, an, welcher durch die Varianten ersetzt wird. In *ALT* wird eine kommaseparierte Liste von alternativen Allelen angegeben, welche *REF* ersetzen. In den Spalten *QUAL* und *FILTER* können zum einen Wahrscheinlichkeiten bezüglich des Auftretens einer der Varianten in Phredskalierung (siehe Abschnitt 2.4.3) und zum anderen Informationen über angewandte Filter abgelegt werden. Die letzte

erforderliche Spalte *INFO* besteht aus einer Liste von semikolonseparierten Metainformationseinträgen, welche zuvor im Dateikopf beschrieben werden sollten.

In einer VCF-Datei können zudem noch Informationen von einzelnen Samples und den dort vorkommenden Genotypen vorhanden sein. Hierzu existiert noch eine neunte Spalte *FORMAT* und für jedes Sample je eine weitere Spalte. Die *FORMAT*-Spalte gibt hierbei die Formatierung der nachfolgenden Spalten an. Die von uns verwendeten VCF-Dateien beinhalten keine Samples, sodass dort nur die ersten acht Spalten vorhanden sind.

Nach dem Tabellenkopf, welcher den Dateikopf abschließt, folgen zeilenweise die variantenbeschreibenden Einträge der VCF-Datei. Jede Zeile hat hierbei ebenso viele, mit Tabulatoren getrennte Spalten wie der Tabellenkopf. In allen Spalten bis auf *CHROM*, *POS* und *REF* kann jedoch durch Angabe eines Punktes (`.`) ein Auslassen des entsprechenden Tabelleneintrags gekennzeichnet werden.

Die für uns wichtigsten Spalten sind die positionsangebenden *CHROM* und *POS* sowie *REF* und *ALT*, welche die Arten der Varianten beschreiben. Die Spalte *CHROM* beinhaltet üblicherweise eine Zeichenkette aus $\{1, 2, \ldots, 22, X, Y, M\}$, welche das jeweilige Chromosom angibt (M stehen hierbei für das Mitochondrium). *POS* gibt die Position der Variante im Chromosom an, wobei Position 1 die erste Base des Chromosoms anzeigt. Die Varianten müssen je Chromosom in der VCF-Datei aufsteigend nach der Position geordnet vorliegen. Mehrere Einträge mit derselben Positionsangabe sind zulässig. Für jedes Chromosom sollten zudem alle Varianten in einem zusammenhängendem Bereich vorkommen. Für genauere Einschränkung bezüglich zugelassener Zeichen in den Chromosomenbezeichnern oder Sonderfälle für die Positionsangaben sei hier auf die VCF-Spezifikation (SAM, 2013) verwiesen.

Die Varianten werden durch Angabe des in der Referenz vorkommenden Allels in *REF* sowie dazu alternativen Allele in Spalte *ALT* beschrieben. Falls mehrere alternative Allele angegeben werden, sind diese durch Kommata getrennt. Die Allele werden jeweils als Text über den IUPAC-Zeichen $\{A, C, G, T, N\}$ (ohne Beachtung der Groß- und Kleinschreibung) dargestellt, wobei der IUPAC-Code N das Vorhandensein einer unbekannten Base anzeigt und nicht etwa schon mehrere Varianten durch sich selbst beschreibt. Die Allele der *ALT*-Spalte stellen hierbei eine Ersetzung des *REF*-Strings im Referenzgenom dar. Abweichende Darstellungen sind nur für die von uns nicht verwendeten strukturellen Varianten vorhanden, welche ebenfalls vom VC-Format unterstützt werden. Die von uns genutzten kleineren Varianten sind wie folgt vorzufinden: Im einfachsten Fall, bei den SNPs, bestehen die Allele jeweils nur aus den einzelnen Basen. Im Falle der kurzen Indels stimmt jeweils das erste Zeichen aller Allele überein, bevor eine beliebige Anzahl weiterer Zeichen die Variante darstellt. Das übereinstimmende Zeichen befindet sich im Chromosom an der Position *POS* und führt dazu, dass die Allele nicht durch leere Zeichenketten repräsentiert werden. Einziger Sonderfall ist hierbei eine Variante an Position 1, der kein Zeichen vorangehen kann, sodass hierbei das erste nachfolgend übereinstimmende Zeichen angehangen wird.

In Abbildung 2.15 ist ein Beispiel einer VCF-Datei angegeben. Dort ist zu sehen, dass in dem Dateikopf eine Fülle von Informationen zum Ursprung (`fileformat`, `filedate`, `source`, `reference`, `contig`) und zur Formatierung (den Spalten entsprechend `ALT`, `INFO`, `FORMAT`) der Daten angegeben werden kann. Hierbei ist einzig der

```
##fileformat=VCFv4.1
##fileDate=20110413
##source=VCFtools
##reference=file:///refs/human_NCBI36.fasta
##contig=<ID=1,length=249250621,species="Homo Sapiens">
##contig=<ID=X,length=155270560,species="Homo Sapiens">
##INFO=<ID=AA,Number=1,Type=String,Description="Ancestral Allele">
##INFO=<ID=H2,Number=0,Type=Flag,Description="HapMap2 membership">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=GQ,NUmber=1,Type=Integer,Description="Genotype Quality">
##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Read Depth">
##ALT=<ID=DEL,Description="Deletion">
##INFO=<ID=SVTYPE,Number=1,Type=String,Description="Type of structural variant">
##INFO=<ID=END,Number=1,Type=Integer,Description="End position of the variant">
#CHROM POS ID     REF  ALT   QUAL FILTER INFO               FORMAT SAMPLE1 SAMPLE2
1        1  .      ACG  A,AT  40   PASS   .                  GT:DP  1/1:13  2/2:29
1        2  .      C    T,CT  .    PASS   H2;AA=T            GT     0|1     2/2
1        5  rs12   A    G     67   PASS   .                  GT:DP  1|0:16  2/2:20
X      100  .      T    <DEL> .    PASS   SVTYPE=DEL;END=299 GT:GQ  1:12:.  0/0:20
```

Abbildung 2.15: Beispiel einer VCF-Datei. (aus (Danecek et al., 2011))

Zeile	Allel	Typ	VCF-Eintrag (Ausschnitt)				Alignierung
		Referenz	Chromosom 1, Position 1 – 5				AC GTA
			POS	REF	ALT	INFO	
1	1	Deletion	1	ACG	A	.	A---TA
1	2	Ersetzung	1	ACG	AT	.	AT--TA
2	1	SNP	2	C	T	.	AT-GTA
2	2	Insertion	2	C	CT	.	ACTGTA
3	1	SNP	5	A	G	.	AC-GTG
		Referenz	Chromosom X, Position 99 – 301				GTAC[...]ACGT
			POS	REF	ALT	INFO	
4	1	Strukturell	100	T	<DEL>	SVTYPE=DEL;END=299	GT--[...]--GT

Tabelle 2.2: Aufschlüsselung der Variantentypen der Einträge aus Abbildung 2.15.

Eintrag `fileformat` nach Spezifikation zwingend erforderlich. Allerdings empfiehlt die Spezifikation jegliche später in den Spalten der Varianteneinträge verwendete Bezeichner auch im VCF-Kopf anzugeben.

Die im Beispiel beschriebenen Varianten sind in Tabelle 2.2 mit ihren entsprechenden Alignierungen am Referenzgenom aufgeschlüsselt. Unterschiede zur Referenz sind in der Alignierung farblich hervorgehoben. Im Beispiel finden sich also in der zweiten (`C` → `T`) und dritten (`A` → `G`) Zeile der VCF-Einträge SNPs. Eine Deletion (`ACG` → `A`) ist in der ersten Zeile, eine Insertion (`C` → `CT`) zusätzlich in der Zweiten gegeben. Abgesehen von einfachen SNPs oder Indels können auch noch, wie im zweiten Allel der ersten Zeile zu sehen, komplexere mehrbasige Ersetzungen gepaart mit Indels (`ACG` → `AT`) auftreten. Die letzte Zeile zeigt eine einfache strukturelle Variante (Deletion) und soll hier nur ein kleines Beispiel zu einer der vielseitigen Notationen für strukturelle Varianten (lange Deletionen oder Insertionen, Inversionen, Tandemrepeats, etc.) aufzeigen.

2.4.5 SAM / BAM

Das „Sequence Alignment/Mapping“-Format (**SAM**) wird zur Ausgabe der Alignierungen verwendet. Vergleichbar mit dem VCF werden auch hier Textdateien verwendet, welche nach einem Kopfbereich die einzelnen Einträge zeilenweise bereitstellen. Die Informationen pro Eintrag sind ebenso tabellenartig angeordnet und durch Tabulatoren getrennt.

Jede Zeile im Dateikopf wird mit einem `@` eingeleitet, auf das ein Zwei-Buchstaben-Identifikator und ein Tabulator folgt. Den Rest der Zeile bilden tabulatorgetrennte Einträge, welche aus Schlüssel:Werte-Paaren bestehen. Die Schlüssel bestehen aus zwei Buchstaben oder einem Buchstaben gefolgt von einer Ziffer und sind durch einen Doppelpunkt (`:`) vom dazugehörigen Wert getrennt.

Die Spezifikation (SAM, 2014) gibt folgende fünf Kopfeinträge vor:

- Die `@HD`-Zeile gibt die SAM-Formatversion sowie eventuelle Sortierreihenfolge der Einträge an.
- In einer `@SQ`-Zeile werden ein eindeutiger Name und die Länge einer Referenzsequenz angegeben. Zusätzlich können in `@SQ` auch noch Angaben zur Spezies und des zugehörigen Referenzgenoms (Assembly Identifier) sowie eine Prüfsumme der Sequenz und ein URI (HTTP, FTP oder lokales Dateisystem) eingetragen werden.
- `@RG` beinhaltet Informationen zu einer Gruppe von Reads. Neben einer eindeutigen Kennung können noch weitere Informationen zur Herkunft der Reads, wie Institutsname, Datum, verwendete Geräte und Programme, angegeben werden.
- In den `@PG`-Zeilen können verwendete Programme, die zur Erzeugung der Alignierungen benutzt wurden, beschrieben werden. Zusätzlich zu Programmname, -version, -parametern, -beschreibung und einer eindeutigen Kennung sind Verweise auf andere `@PG`-Zeilen möglich, sodass die Reihenfolge, in der die Programme benutzt wurden, festgehalten werden kann.
- Zuletzt können noch mittels `@CO` beliebige Kommentarzeilen eingeleitet werden, in denen die Angabe von Schlüssel:Werte-Paaren nicht nötig ist.

Jegliche Zeilen im Dateikopf sind optional, außer sie werden im Falle von `@SQ`, `@RG` oder `@PG` in den Alignierungseinträgen mit ihren eindeutigen Bezeichnern referenziert. Des Weiteren können außer den genannten fünf Typen noch beliebige, vom Benutzer definierte Einträge benutzt werden.

Die einzelnen Alignierungen der Reads werden nach dem Kopfbereich zeilenweise angegeben. Jeder Eintrag hat folgende elf notwendige Spalten:

1. `QNAME`: Name des betrachteten Reads.
2. `FLAG`: Bitfeld, welches Informationen über den Status der Alignierung kompakt beschreibt. Hierzu zählen, ob der Read überhaupt gemappt werden konnte, in welche Richtung (Vorwärtsstrang oder reverses Komplement) er aligniert wurde, und weitere Informationen über andere Alignierungseinträge, welche in Verbindung mit dem aktuellen Eintrag stehen.

3. `RNAME`: Name der Referenzsequenz, welche zuvor in Dateikopf in einer `@SQ`-Zeile angegeben wurde.

4. `POS`: Anfangsposition der Alignierung in der Referenzsequenz (erste Position der Referenz hat den Wert 1).

5. `MAPQ`: Phredskalierte Qualitätsangabe des Mappings.

6. `CIGAR`: Beschreit die eigentliche Alignierung als Folge von Editieroperationen, was in Abschnitt 2.4.6 genauer beschrieben wird.

7. `RNEXT`: Bei zueinander gehörenden Reads wird hier die Referenzsequenz des nächsten Reads angegeben, oder `=`, falls `RNEXT` identisch mit `RNAME` ist.

8. `PNEXT`: Analog zu `RNEXT` gibt `PNEXT` die zugehörige Position des nächsten Reads an.

9. `TLEN`: Länge des Referenzabschnittes, welcher durch alle mittels `RNEXT`, `PNEXT` angegebenen zugehörigen Reads abgedeckt wird, oder 0, falls verschiedene Referenzen angegeben wurden, bzw. keine anderen zugehörigen Reads existieren. Für den Read ganz rechts im Abschnitt wird der Wert negiert, beim Read ganz links ist der Wert positiv.

10. `SEQ`: Beinhaltet die Sequenz des Reads, dem FASTQ-Format entsprechend. Falls das reverse Komplement aligniert wurde, wird hier ebenso das reverse Komplement des Reads angegeben.

11. `QUAL`: Beinhaltet die Sequenzierqualitätswerte zu `SEQ`, dem FASTQ-Format entsprechend.

Auch wenn die Angabe aller dieser Felder notwendig ist, können sie je nach Feld entweder durch ein `*` oder eine `0` ersetzt werden, um ein Nichtvorhandensein des entsprechenden Wertes anzuzeigen. So können zum Beispiel auch Reads ohne Mapping abgespeichert werden, in dem `RNAME` auf `*` gesetzt wird.

Nach den elf vorgegebenen Spalten folgt eine beliebige Anzahl optionaler, auch durch den Benutzer festlegbarer, Felder. Diese werden ähnlich zu den Schlüssel:Werte-Paaren im Kopfbereich als Schlüssel:Typ:Wert angegeben. Ein Schlüssel darf hierbei nur einmal pro Alignierung verwendet werden. Im Gegensatz zum Dateikopf wird in den Feldern noch der Datentyp der Werte (Zeichenketten, Fließkomma- oder ganze Zahlen, Bit- oder Zahlenfelder, etc.) spezifiziert. Hier können somit etwa die verwendeten Programme mit `PG:Z:Programmbezeichner` oder die Readgruppen mit `RG:Z:Readgruppenbezeichner` angegeben werden. Das `Z` steht hierbei für den Datentyp eines Strings, welcher aus druckbaren Zeichen und dem Leerzeichen bestehen kann. Eine Fülle von vordefinierten optionalen Feldern finden sich in der Spezifikation des SAM-Formates (SAM, 2014). Die Spezifikation gibt dem Benutzer auch hier die Möglichkeit, eigene Felder für seine Zwecke zu definieren.

In Abbildung 2.16 ist eine Referenzsequenz mit sechs Alignierungen, die fünf Reads zugeordnet sind, zu sehen. Die Reads `r001/1` und `r001/2` bilden ein Readpaar. Der Read `r001/1` wurde mit je einer Insertion und einer Deletion, `r001/2` nach Bildung seines reversen Komplements fehlerfrei aligniert. Der Read `r002` wurde ab seinem

```
Coor     12345678901234  5678901234567890123456789012345
ref      AGCATGTTAGATAA**GATAGCTGTGCTAGTAGGCAGTCAGCGCCAT

+r001/1        TTAGATAAAGGATA*CTG
+r002         aaaAGATAA*GGATA
+r003       gcctaAGCTAA
+r004                       ATAGCT..............TCAGC
-r003                              ttagctTAGGC
-r001/2                                          CAGCGGCAT
```

Abbildung 2.16: Beispiel mehrerer Alignierungen an einer Referenz. (aus (SAM, 2014))

```
@HD VN:1.5 SO:coordinate
@SQ SN:ref LN:45
r001  163 ref  7 30 8M2I4M1D3M = 37  39 TTAGATAAAGGATACTG *
r002    0 ref  9 30 3S6M1P1I4M *  0   0 AAAAGATAAGGATA    *
r003    0 ref  9 30 5S6M       *  0   0 GCCTAAGCTAA       * SA:Z:ref,29,-,6H5M,17,0;
r004    0 ref 16 30 6M14N5M    *  0   0 ATAGCTTCAGC       *
r003 2064 ref 29 17 6H5M       *  0   0 TAGGC             * SA:Z:ref,9,+,5S6M,30,1;
r001   83 ref 37 30 9M         =  7 -39 CAGCGGCAT         * NM:i:1
```

Abbildung 2.17: SAM-Datei zu den Alignierungen aus Abbildung 2.16. (aus (SAM, 2014))

vierten Zeichen unter Berücksichtigung einer Insertion aligniert. Read `r003` wurde ab seiner sechsten Base an Referenzposition neun mit einem Sequenzmismatch aligniert. Zusätzlich existiert noch eine Alignierung des reversen Komplements von `r003` ab seinem sechsten Zeichen an der Refenzposition 29. Der Read `r004` wurde mit Auslassung von 14 Zeichen der Referenzsequenz aligniert. Eine zu den beschriebenen Alignierungen gehörige SAM-Datei ist in Abbildung 2.17 dargestellt. Die Bedeutung der CIGAR-Strings, welche in Spalte sechs zu sehen sind, wird im nachfolgenden Abschnitt 2.4.6 genauer beschrieben.

Neben SAM wurde auch ein auf diesem aufbauendes Format namens **BAM** entwickelt. Im Gegensatz zum textbasierten SAM verwendet BAM ein Binärformat und benutzt standardmäßig eine gzip-Komprimierung. Die Alignierungsdaten des SAM-Formats werden im BAM-Format strukturell ähnlich kodiert, können dort allerdings durch die Binärkodierung viel kompakter beschrieben werden. Das BAM-Format ist neben Kompaktheit weiterhin auch auf hohe Verarbeitungsgeschwindigkeiten ausgelegt, indem neben offensichtlichen Verbesserungen gegenüber dem SAM-Format, wie der Repräsentation von Daten wie Zahlenwerten in Binär- statt Textform, auch noch andere Änderungen, wie etwa eine feste Spaltenanzahl, vorgenommen wurden. Der wohl größte Unterschied liegt allerdings bei der verwendeten **BGZF**-Komprimierung. Das BGZF-Format ist gzip-kompatibel und nutzt konkatenierte gzip-Dateien, um ein blockbasiertes Archivformat zu erzeugen. Die BGZF-Blöcke können im BAM-Format dazu verwendet werden, eine schnelle Extraktion von Alignierungen für bestimmte Abschnitte des Referenzgenoms vorzunehmen, ohne dass die komplette BAM-Datei ausgelesen werden muss. Hierzu werden die Alignierungseinträge zunächst nach den Referenzpositionen sortiert, um danach eine externe Indexdatei erzeugen zu können, welche eine Zuordnung der Referenzabschnitte auf die BGZF-Blöcke ermöglicht.

Operation	Beschreibung
M	Alignierungsmatch (Match oder Mismatch der Sequenz)
=	Sequenzmatch
X	Sequenzmismatch
I	Insertion in Bezug zur Referenz
D	Deletion in Bezug zur Referenz
S	„Soft Clipping“: abgeschnittene Sequenz nur im Read vorhanden
H	„Hard Clipping“: abgeschnittene Sequenz nicht im Read vorhanden
N	Übersprungende Region der Referenz
P	„Padding“: Deletion in der „padded“ Referenz

Tabelle 2.3: CIGAR-Operationen gemäß der SAM-Spezifikation.

2.4.6 CIGAR-String

Die Alignierungen werden im SAM-Format als eine Reihe von Editieroperationen mittels sogenannter **CIGAR**-Strings („Compact Idiosyncratic Gapped Alignment Report“) angegeben. Eine CIGAR-Operation ist ein Ein-Zeichen-Code, welchem eine Zahl vorangestellt wird, die die Anzahl des Ausführungen der entsprechenden Operation angibt. Ein CIGAR-String ist eine Konkatenation von CIGAR-Operationen und zeigt an, welche Editieroperationen auf einer Referenz auszuführen sind, um einen Read an ihr alignieren zu können.

Die SAM-Spezifikation (SAM, 2014) definiert die in Tabelle 2.3 angegebenen Operationen. M, I, D und S sind dabei die meist benutzten Operationen. Auch wenn die Verwendung von = und X intuitiver erscheint, da durch diese genau Matches und Mismatches der Sequenzen beschrieben werden können, wird in der Regel die Operation M anstelle der beiden Anderen benutzt. Für die Alignierung an sich, also die Ausrichtung der beiden Sequenzen zueinander, ist eine Angabe der Operation M genauso aussagekräftig wie die Angaben der separaten Match- und Mismatchoperationen auf der Sequenz. Dadurch kann die Ausgabe der Alignierung durch einen kompakteren CIGAR-String dargestellt werden, da alle aufeinanderfolgenden X- und =-Operationen zu einer Operation M zusammengefasst werden.

Anstatt Sequenzmismatches am Anfang oder Ende eines Reads willkürlich als Insertionen oder Sequenzmismatches zu definieren, können diese als S-Operationen (Soft Clipping) angegeben werden. Sinngemäß beschreiben die nur am Anfang oder Ende eines Reads vorkommenden S-Operationen, dass eine Alignierung stattgefunden hat, bei der der Read um die angegebene Anzahl von Zeichen zuvor gestutzt wurde.

Die Operationen H, N und P werden seltener verwendet. Die Operation H, welche nur als erste und letzte Operation im CIGAR-String vorkommen kann, gibt ebenfalls an, dass der Read um die angegebene Anzahl von Basen gestutzt wurde. Im Gegensatz zur S-Operation werden die abgeschnitten Zeichen jedoch in der im SAM-Eintrag gespeicherten Sequenz gelöscht. Mittels der Operation N können bei Alignierungen von mRNA zu DNA die Auslassungen von Introns beschrieben werden. Operation P wird bei Alignierungen im SAM-Format mit „Padding“ verwendet. Das Padding findet für unsere Zwecke keine Verwendung und wird im SAM-Format in der Regel für De-novo-Assemblierungen benutzt, bei der die Referenz durch Padding aufgefüllt wird, um Insertion anzuzeigen.

Kapitel 3

Verteilung der Varianten im Humangenom

In diesem Kapitel werden einige Statistiken über die Verteilung der bekannten Varianten des Humangenoms vorgestellt. Der Grund bzw. die Motivation für das Erstellen dieser Statistiken soll der folgende Abschnitt klären.

3.1 Motivation

Die Aufgabe unserer Projektgruppe ist es, einen Readmapper zu implementieren, der nach Möglichkeit bekannte Varianten direkt unterstützt. Beispielsweise könnte an einer bestimmten Position im Referenzgenom die Base Adenin stehen. Es ist aber bekannt, dass an dieser Position auch häufig Thymin vorkommt. Ein Read, der zu diesem Bereich im Referenzgenom passt, enthält möglicherweise an der besagten Stelle diese Variante (also Thymin statt Adenin). Beim Readmapping sollte dieser Basenunterschied zwischen Referenzgenom und Read jedoch nicht als Fehler gewertet werden, da es sich hierbei um eine häufige Variante handelt. Diese Variantentoleranz wird von vielen Readmappern nicht unterstützt. Da die Anzahl der bekannten Varianten jedoch stetig zunimmt, wäre eine direkte Unterstützung der Varianten beim Readmapping sehr wünschenswert.

Als Grundlage für einen Readmapper können sogenannte q-Gramme verwendet werden. Unter q-Grammen versteht man Teilstrings eines Textes der Länge q. Um Reads im Referenzgenom zu finden, werden das Referenzgenom und die Reads in ihre q-Gramme zerlegt. Kommen in einem Abschnitt des Referenzgenoms viele q-Gramme vor, die auch in einem bestimmten Read vorhanden sind, so ist die Wahrscheinlichkeit hoch, dass dieser Read zu jener Stelle im Referenzgenom passt. Durch die Verwendung der q-Gramme erhalten wir eine gewisse Fehlertoleranz: Enthält ein Read einen Sequenzierfehler, so ist nur ein Teil seiner q-Gramme davon betroffen. Das heißt, wir können den Read immer noch im Referenzgenom wiederfinden.

An dieser Stelle ist es hilfreich, eine Vorstellung über die Größenordnungen zu bekommen: Das Humangenom besteht, wie bereits erwähnt, aus 3 Milliarden Basenpaaren. Reads von Next-Generation Sequenzierern haben in der Regel eine Länge von ca. 100 Basenpaaren. Als q-Gramm-Länge werden meist Werte zwischen 8 und 24 verwendet.

Hier müssen Experimente zeigen, bei welcher q-Gramm-Länge die besten Ergebnisse erzielt werden. Bei dem sehr schnellen Readmapper PEANUT (ParallEl AligNment UTility) (Köster und Rahmann, 2014) haben die q-Gramme beispielsweise eine feste Länge von 16 Basen. Bei sehr langen q-Grammen leidet die Fehlertoleranz, da von einem Fehler bereits sehr viele q-Gramme betroffen sind. Sind die q-Gramme zu kurz, wird der Read möglicherweise an eine falsche Stelle gemappt, da die q-Gramme den Read nicht ausreichend repräsentieren.

Damit die Suche hinreichend schnell ist, wird mit den q-Grammen des Textes ein Suchindex aufgebaut. Wie dieser genau funktioniert hängt von dem verwendeten Algorithmus ab. In Kapitel 4.1 erklären wir detailliert den in unserer Implementierung verwendeten Suchindex.

Ein auf q-Grammen basierter Suchindex kann Varianten direkt berücksichtigen. Die einfachste Möglichkeit dazu wäre, die Varianten hinten an das Referenzgenom anzuhängen. Bei SNPs (Veränderung einer Base, vgl. Abschnitt 2.1.8.1) kann aber geschickter vorgegangen werden: Liegt an einer Position in einem q-Gramm eine Variante vor, so fügt man beide Möglichkeiten zum Suchindex hinzu. Dieses Vorgehen ist jedoch problematisch, wenn viele Varianten in einem q-Gramm vorkommen, da man alle Kombinationen zum Suchindex hinzufügen müsste. Im Extremfall gäbe es an allen q Positionen des q-Gramms jeweils drei Varianten, sodass sich 4^q Kombinationen ergeben würden.

Hier stellt sich die Frage, wie die Varianten verteilt sind. Kommen die Varianten gleichverteilt vor? Oder gibt es Abschnitte mit sehr vielen Varianten? Um diese Fragen zu beantworten haben wir verschiedene Statistiken zur Verteilung der Varianten erstellt.

3.2 Durchführung

Als Datengrundlage dienen die von der NCBI (National Center for Biotechnology Information) veröffentlichten Varianten des Humangenoms[1]. Die zur Zeit bekannten Varianten sind das Ergebnis des 1000-Genomes-Projekts[2], bei dem die DNA von mehr als 1000 Menschen sequenziert wurde. Für unsere Statistiken haben wir auf zwei Varianten-Dateien zurückgegriffen. Die eine Datei („all.vcf“) enthält alle bekannten Varianten, die andere Datei („common_all.vcf“) nur die im Folgenden als *häufige Varianten* bezeichneten. Es handelt sich bei den häufigen Varianten um jene, die bei mindestens 1% der Individuen in einer der 14 Großpopulationen, zu denen vom 1000-Genomes-Projekt Variantendaten vorliegen[3], vorkommen. Die besagten 1% müssen hierbei durch mindestens zwei nicht verwandte Personen gebildet werden[4].

[1] ftp://ftp.ncbi.nih.gov/snp/organisms/human_9606_b141_GRCh37p13/VCF/
[2] http://www.1000genomes.org/
[3] http://www.1000genomes.org/category/frequently-asked-questions/population
[4] http://www.ncbi.nlm.nih.gov/variation/docs/human_variation_vcf/#common_all

3.3 Ergebnisse und Auswertung

Im Laufe unserer Projektgruppe haben wir viele verschiedene Statistiken erstellt. Die Wichtigsten und Aussagekräftigsten sollen im Folgenden vorgestellt und analysiert werden.

3.3.1 Allgemeine Statistiken

Zunächst ein paar allgemeine Informationen über die beiden Variantendateien: Bisher sind etwa 63 Millionen Varianten bekannt, 85% davon sind SNPs, die restlichen 15% sind Indels. Etwa die Hälfte dieser Varianten (28 Millionen) sind häufige Varianten. Bei den Häufigen handelt es sich hauptsächlich um SNPs (95%). Nur 5% der häufigen Varianten sind Insertionen oder Deletionen.

3.3.2 Anzahl der SNPs in einem q-Gramm

Im Abschnitt 3.1 kam bereits die Frage auf, wie die Varianten verteilt sind. Insbesondere ist interessant, ob es q-Gramme gibt, bei denen an vielen oder fast allen Stellen SNPs vorkommen. Das Diagramm in Abbildung 3.1 zeigt, wie häufig q-Gramme der Länge 16 mit einer bestimmten Anzahl von SNPs vorkommen. Die konstante Länge von 16 wurde gewählt, da diese beim PEANUT-Algorithmus verwendet wird.

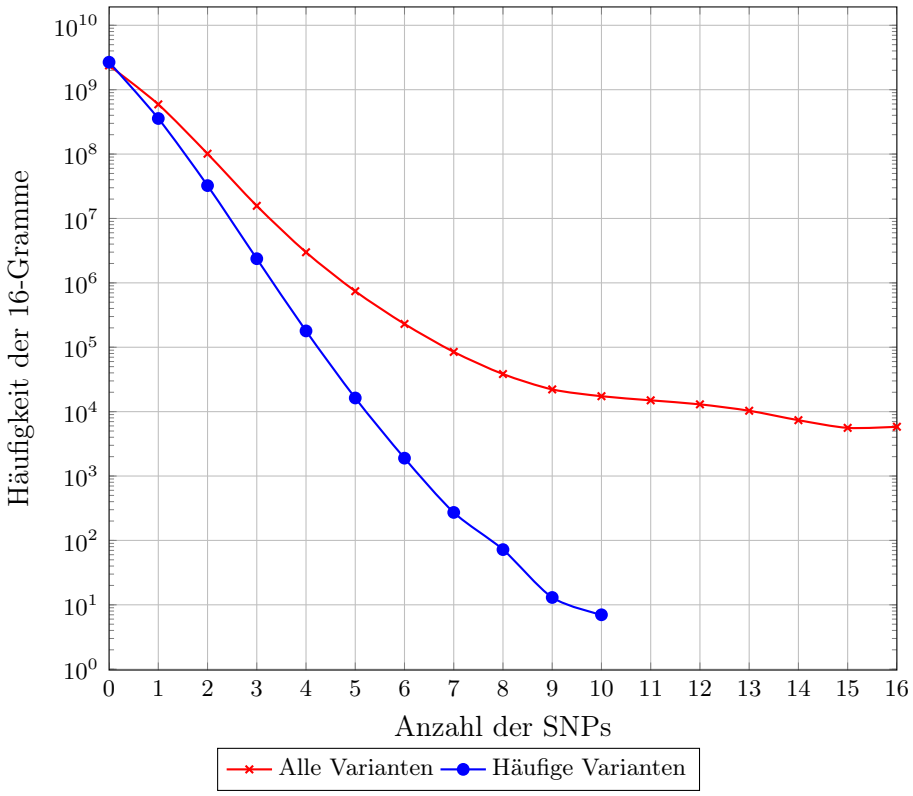


Abbildung 3.1: Häufigkeit von 16-Grammen mit entsprechender Anzahl von SNPs.

Berücksichtigt man alle Varianten (rote Kurve), so stellt man fest, dass es etwa 5000 16-Gramme gibt, bei denen an allen Stellen eine Variante vorkommt. Obwohl die Datei mit den häufigen Varianten nur halb so viele Varianten enthält, sind extreme Variantenhäufungen hier deutlich seltener (siehe blaue Kurve). 16-Gramme mit 11 oder mehr Varianten kommen beispielsweise gar nicht mehr vor.

Man kann also davon ausgehen, dass das Risiko der kombinatorischen Explosion bei ausschließlicher Verwendung der häufigen Varianten deutlich geringer ist als bei Verwendung aller Varianten.

Die Abbildung 3.1 zeigt jedoch nicht wie viele Varianten pro Position vorkommen. Dies ist aber durchaus relevant. Existiert an jeder Position eines 16-Gramms genau eine Variante, so gäbe es bei einem 16-Gramm mit 16 SNPs $2^{16} = 65536$ Kombinationen, die zum Suchindex hinzugefügt werden müssten. Das ist bereits eine große Menge von q-Grammen, aber angesichts der Tatsache, dass das Humangenom ohne Varianten circa 3 Milliarden q-Gramme enthält, könnte man mit diesen 65535 zusätzlichen q-Grammen noch zurechtkommen. Kann jedoch an jeder Position des 16-Gramms jede Base stehen (d.h. wir haben drei Varianten pro Basenpaar), dann gibt es $4^{16} \approx 4{,}3 \cdot 10^9$ Kombinationen. Damit würden sich die Menge der q-Gramme im Suchindex mehr als verdoppeln[5].

Um genauere Aussagen zu treffen, wäre es daher sinnvoll die tatsächliche Anzahl der Kombinationen zu zählen.

3.3.3 Kombinatorische Explosion

Um einen q-Gramm-basierten Suchindex variantentolerant zu machen, könnte man alle q-Gramme betrachten, die sich aus Kombinationen der verschiedenen bekannten SNPs ergeben. Dabei stellt sich die Frage, wie viele q-Gramme dann dem Suchindex hinzugefügt werden müssten. In Abbildung 3.2 ist genau dies dargestellt: Die y-Achse zeigt die Anzahl der q-Gramme in Abhängigkeit von der q-Gramm-Länge auf der x-Achse.

Ohne weitere Beschränkungen (rote Kurve) kommt es zu der bereits oben befürchteten kombinatorischen Explosion: Bei $q = 10$ müssten etwa 5 Milliarden q-Gramme hinzugefügt werden, also 66% mehr als ohne Variantenberücksichtigung. Bei $q = 11$ hat sich die q-Gramm-Anzahl bereits verdreifacht. Bei $q = 16$ lägen wir bei 2,17 Billionen q-Grammen, also circa Faktor 700 mehr. Diese extrem hohen Zahlen werden dadurch verursacht, dass an einigen Stellen sehr viele Varianten vorkommen. Es gibt 16-Gramme im Referenzgenom bei denen an allen Stellen mindestens zwei SNPs vorkommen. Damit erzeugt solch ein 16-Gramm über $3^{16} \approx 43 \cdot 10^6$ Varianten. An solch eine Stelle könnte man äußerst viele Reads mappen, indem einfach die passenden Varianten gewählt werden. Es ergibt somit keinen Sinn, jede dieser Kombinationen zum Suchindex hinzuzufügen. Darüber hinaus ist es höchst unwahrscheinlich, dass jede dieser Kombinationen in mindestens einem Menschen vorkommt. Vermutlich existiert nur ein kleiner Anteil der möglichen Varianten-Kombinationen im menschlichen Genpool. Da aber die Variantendatei keine Information darüber enthält in welchem sequenzierten Genom eine bestimmte Variante vorkommt, ist unklar, welche Varianten-Kombinationen tatsächlich vorkommen und welche nicht.

[5] Dieser Fall ist etwas künstlich, da man an solch einer Position jeden Read mappen könnte.

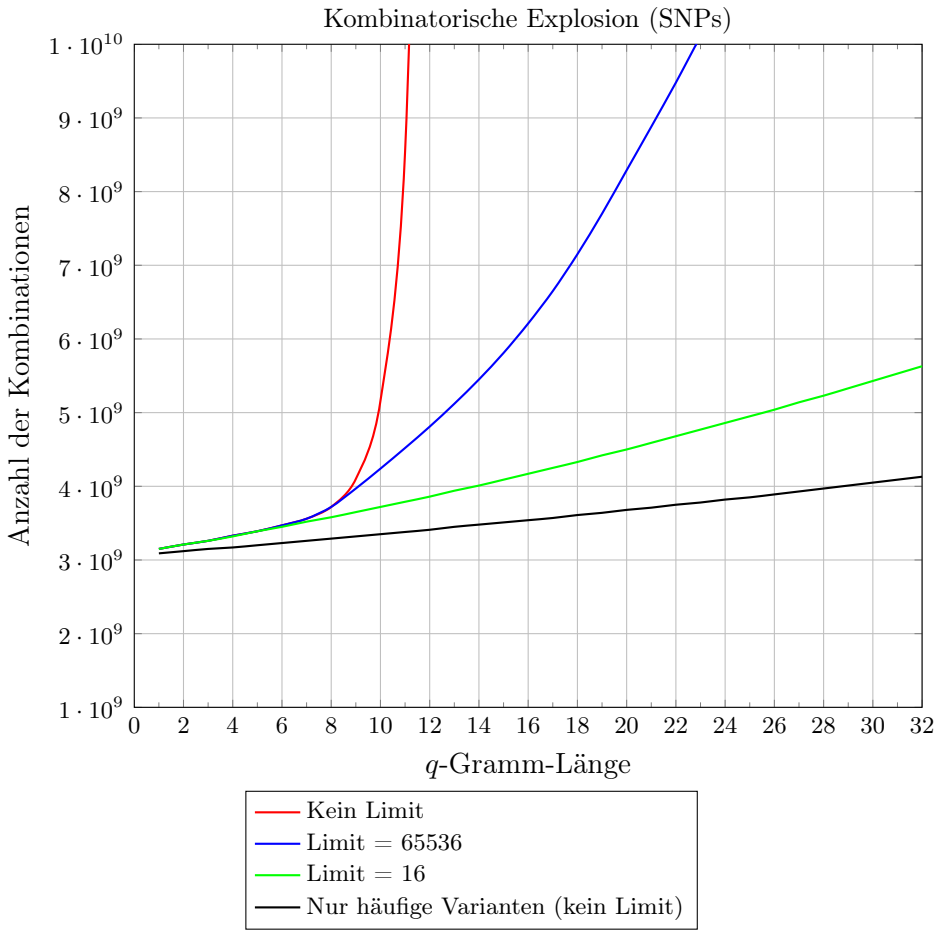


Abbildung 3.2: Anzahl der q-Gramme, die zum Suchindex hinzugefügt werden müssen, wenn alle SNP-Kombinationen berücksichtigt werden.

Aus diesem Grund kamen wir auf die Idee, die Anzahl der Kombinationen pro q-Gramm im Genom zu beschränken. Die blaue Kurve zeigt das Ergebnis bei einer Beschränkung (Limit) von 65536 Kombinationen. Gibt es q-Gramme mit mehr Kombinationen, würde man nur 65536 von diesen zum Suchindex hinzufügen. Obwohl dieser Wert relativ groß ist, kann dadurch die kombinatorische Explosion vermieden werden. Bei $q = 16$ erhalten wir etwa 6 Milliarden q-Gramme, also doppelt so viel wie ohne Berücksichtigung der Varianten. Dieser Wert ist gut beherrschbar. Reduziert man das Limit weiter auf 16, ergeben sich etwa 4 Milliarden q-Gramme, also nochmal deutlich weniger.

Besonders interessant ist die schwarze Kurve, bei der nur die häufigen Varianten berücksichtigt wurden. Bei 16-Grammen erhalten wir gerade einmal 3,5 Milliarden Kombinationen, obwohl es keine Beschränkung der Anzahl der Kombinationen in einem q-Gramm gibt. Wenn also nur die häufigen Varianten berücksichtigt werden, kann auf die Implementierung eines Limit verzichtet werden, da sich die Gesamtanzahl der q-Gramme im Suchindex nicht wesentlich erhöht. Trotzdem könnte eine Beschränkung der Kombinationsanzahl sinnvoll sein: Treten viele Varianten in einem q-Gramm auf, so können deutlich mehr Reads an diese Position gemappt werden als

an Stellen des Humangenoms an denen weniger Varianten vorkommen. Das heißt, variantenreiche Stellen werden beim Readmapping bevorzugt, wenn starke Variantenhäufungen vorkommen.

3.3.4 Länge von Sequenzen mit vielen Varianten

Bisher haben wir uns nur mit der Verteilung von SNPs beschäftigt. Im Folgenden betrachten wir die Verteilung aller Varianten, also SNPs und Indels. Das Diagramm in Abbildung 3.3 zeigt die Häufigkeit von Sequenzen bestimmter Länge, bei denen an allen Positionen Varianten vorkommen. Auf der x-Achse ist die Länge der Variantensequenz aufgetragen und auf der y-Achse die Häufigkeit in der jeweiligen Variantendatei. Bei der Berechnung des Diagramms wurden ab einer Länge von 12 Basenpaaren mehrere Werte auf der x-Achse zusammengefasst, um eine Glättung der Kurve zu erzielen. Dementsprechend können auch Häufigkeiten kleiner als 1 auftreten. Beispielsweise hat die rote Kurve im Bereich von 90 bis 100 Basenpaaren einen y-Wert von etwa 0,5. Somit gibt es $0{,}5 \cdot (100 - 90) = 5$ Sequenzen mit einer Länge zwischen 90 und 100 Basenpaaren, bei denen an allen Positionen Varianten vorkommen können. Bei den beiden roten Kurven wurden als Datengrundlage alle Varianten verwendet, bei den beiden blauen Kurven nur die häufigen Varianten. Bei den gestrichelten Kurven darf zwischen den Varianten eine Lücke von maximal vier variantenlosen Positionen des Referenzgenoms vorkommen, ohne dass dies als neue Variantensequenz gezählt wird. Die x-Achse zeigt dabei nicht die Länge der Variantensequenz, sondern die Anzahl der Positionen, an denen Varianten vorkommen können.

Wie auch im vorherigen Abschnitt fällt auf, dass starke Variantenhäufungen nur bei Verwendung aller bekannten Varianten vorkommen. Unter den häufigen Varianten besteht die längste lückenlose Variantensequenz gerade mal aus acht Basenpaaren. Im Gegensatz dazu gibt es bei Berücksichtigung aller Varianten Fälle mit 100 und mehr aufeinander folgenden Positionen, bei denen überall bekannte Varianten existieren. Selbst wenn man eine Lücke von bis zu vier variantenlosen Positionen zulässt, erhöht sich die Variantenanzahl der längsten Sequenz unter den häufigen Varianten gerade einmal auf 15.

Das Diagramm zeigt nur Variantensequenzen mit einer Länge von maximal 300 Positionen. Ein Blick auf die Rohdaten zeigt aber, dass es sogar eine lückenlose Variantensequenz mit einer Länge zwischen 334 und 368 Positionen im Referenzgenom gibt. Lässt man eine Lücke von vier variantenlosen Positionen zu, ergibt sich sogar noch ein Eintrag bei über 5000 Varianten.

Wir können also festhalten, dass es Stellen im Humangenom gibt, die bei den einzelnen Individuen sehr unterschiedlich aussehen. Unter den häufigen Varianten kommen solche starken Variantenhäufungen nicht vor. Wir hatten bereits erwähnt, dass beim Zuordnen eines Reads zu einer ungefähren Position im Humangenom die Verwendung der häufigen Varianten sinnvoll ist.

Auch bei der Berechnung des exakten Alignments (siehe Kapitel 4.2) eines Reads an das Humangenom macht es Sinn, nur die häufigen Varianten zu verwenden. Denn an einer sehr variantenreichen Stelle würde das Alignment sonst vermutlich Variantenkombinationen verwenden, die bei keinem Individuum vorkommen. Alternativ könnte

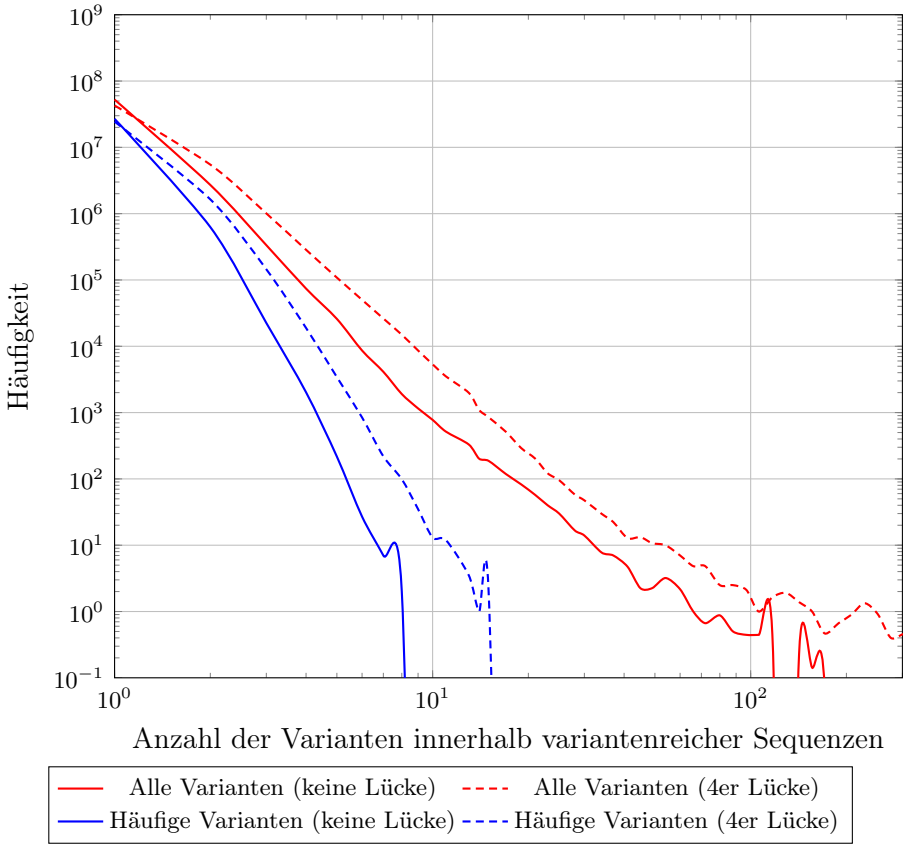


Abbildung 3.3: Häufigkeit von variantenreichen Sequenzen. Die durchgezogenen Linien repräsentieren Sequenzen mit Varianten an allen Positionen. Dementsprechend ist die Länge der Sequenz gleich der Anzahl der darin enthaltenen Varianten. Bei den gestrichelten Kurven dürfen zwischen zwei Varianten bis zu vier variantenlose Positionen stehen, ohne dass dies als neue Sequenz gewertet wird. Die Länge dieser Sequenzen ist größer als die im Diagramm gezeigte Anzahl von Positionen, an denen bekannte Varianten existieren.

man auch dynamisch entscheiden, welche Variantendatei verwendet wird. Liegt an jener Stelle eine starke Variantenhäufung vor, so verwendet man nur die häufigen Varianten für das Alignment. Liegen nur wenige Varianten an jener Stelle vor, kann auch auf alle Varianten zurückgegriffen werden.

3.3.5 Variantenhäufungen

Tabelle 3.1 zeigt die Chromosomen und Positionen von Stellen im Humangenom, bei denen viele Varianten vorkommen. „Viele“ wurde dabei so definiert, dass in einem Fenster mit einer Breite von 2000 Basenpaaren mindestens 1000 Varianten vorkommen müssen. Die innerhalb der Projektgruppe geäußerte Vermutung, dass die Variantenhäufungen möglicherweise nur am Anfang oder am Ende eines Chromosoms vorkommen, kann die Tabelle eindeutig widerlegen.

Das Balkendiagramm in Abbildung 3.4 zeigt ein Beispiel für einen Bereich mit besonders vielen Varianten im sechsten Chromosom. An jeder Position des 201 Basenpaare

Chromosom	Position	Längen	Varianten
1	43 297 968	3 383	1 560
1	121 483 203	2 854	1 154
1	207 511 977	22 815	17 703
1	207 669 174	11 066	7 732
1	207 792 710	3 570	1 678
1	207 804 914	4 556	2 392
2	127 413 243	30 798	21 454
2	127 443 578	2 000	1 000
2	127 443 580	5 055	2 767
6	10 532 526	54 184	37 545
6	32 628 205	2 816	1 159
6	32 630 860	4 510	2 493
6	58 775 341	4 660	2 434
7	61 967 275	4 090	2 088
16	46 388 873	5 233	3 089
16	46 393 109	2 001	1 000
16	46 393 113	3 580	1 718
16	46 403 046	4 835	2 506
19	569 895	2 873	1 183
X	2 707 159	6 025	3 452
X	2 714 790	12 269	7 971
Y	2 600 930	5 929	3 362

Tabelle 3.1: Sequenzen im Humangenom mit vielen Varianten (mindestens 1000 Varianten in einem Fenster mit einer Breite von 2000 Basenpaaren).

langen Fensters können Varianten vorkommen. Dabei sind SNPs in blau und Indels in rot dargestellt. Es fällt auf, dass an vielen Positionen drei SNP-Varianten vorkommen, d.h. hier kann jede der vier Basen (A, C, G und T) stehen. Stellen, an denen es nur eine SNP-Variante gibt, sind klar in der Minderheit. Der gezeigte Bereich verdeutlicht nochmals, dass die Verwendung aller bekannten Varianten beim Readmapping an manchen Stellen im Humangenom nicht sinnvoll ist.

3.3.6 Länge von Insertionen und Deletionen

Bisher haben wir uns nur damit beschäftigt, wie die Varianten verteilt sind. Im Folgenden wollen wir einen genaueren Blick speziell auf Indels werfen. Abbildung 3.5 zeigt die Länge von Insertionen und Deletionen. Auf der x-Achse ist die Anzahl der hinzugefügten Basenpaare (positive Werte) bzw. der gelöschten Basenpaare (negative Werte) aufgetragen. Die y-Achse zeigt, wie häufig Indels genau dieser Länge in der jeweiligen Variantendatei existieren. Bei der roten Kurve wurden alle Varianten berücksichtigt, bei der blauen nur die häufigen.

Unter den bekannten Varianten gibt es insgesamt 4,8 Millionen Deletionen und 4,1 Millionen Insertionen. Bei den häufigen Indels ergibt sich ein Verhältnis von 0,87 Millionen Deletionen zu 0,65 Millionen Insertionen. Das heißt bei beiden Varianten-

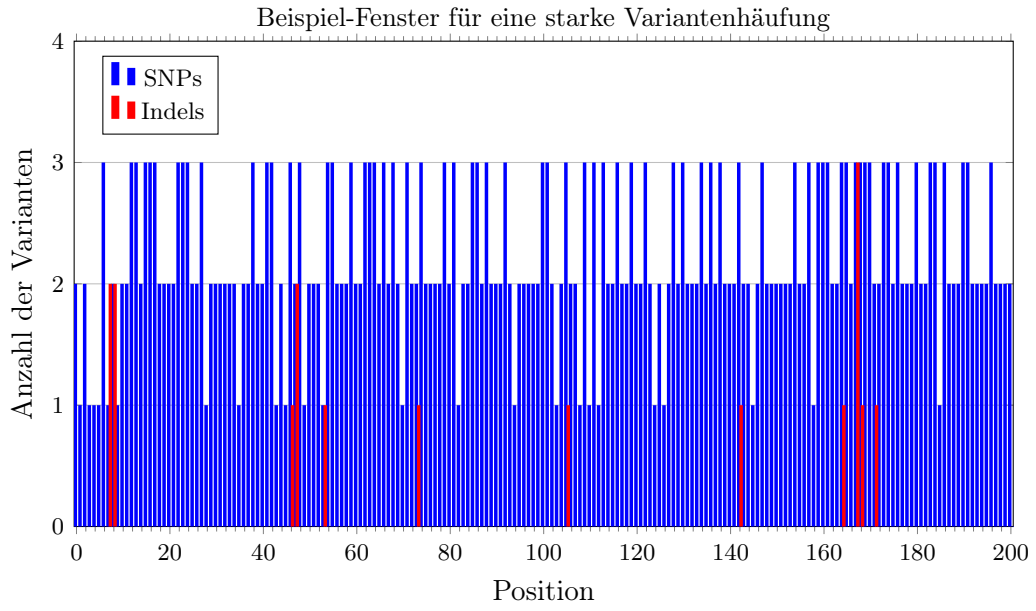


Abbildung 3.4: Beispiel für eine sehr starke Variantenhäufung (6. Chromosom, ab Position 32.631.689).

Dateien sind Deletionen geringfügig häufiger als Insertionen. Bei circa jeder zweiten Indel-Variante wird nur ein Basenpaar hinzugefügt oder gelöscht; die meisten Indels sind also sehr kurz. Das Diagramm zeigt nur die Deletionen und Insertionen, bei der höchstens 250 Basenpaare gelöscht bzw. eingefügt werden. Die längsten Insertionen sind mit maximal 252 Basenpaaren tatsächlich kaum länger. Es gibt allerdings erheblich längere Deletionen von bis zu 998 Basenpaaren. Insgesamt liegen etwa 2000 Deletionen außerhalb des sichtbaren Bereichs der x-Achse. Dieser Bereich wurde im Diagramm nicht dargestellt, damit der interessante mittlere Teil deutlicher zu erkennen ist. Darüber hinaus ist der Bereich von -1000 bis -250 auch stark verrauscht, sodass man nicht viel erkennen würde.

Da es Insertionen gibt, bei denen mehr als 100 Basenpaare hinzugefügt werden, kann es vorkommen, dass ein Read ausschließlich in einer Insertion zu finden ist. Allerdings ist dies recht selten: Unter allen bekannten Indels gibt es circa 1000 solcher Fälle, unter den häufigen Varianten sogar nur sechs. Vergleicht man dies mit den bekannten Deletionen, stellt man fest, dass lange Deletionen etwa doppelt so häufig sind wie lange Insertionen. Dies gilt aber nur unter Berücksichtigung aller Varianten. Unter den häufigen Varianten werden maximal 44 Basenpaare gelöscht, aber bis zu 167 Basenpaare hinzugefügt. Im Regelfall sind aber die Insertionen der häufigen Varianten kürzer als 50 Basenpaare – es gibt lediglich 47 Ausnahmen.

3.4 Fazit

Primäres Ziel der Statistiken war es, mehr über die Verteilung der Varianten herauszufinden. Wir haben festgestellt, dass teilweise extreme Variantenhäufungen auftreten. Der in Abschnitt 3.3.5 gezeigte, 200 Basenpaare lange Ausschnitt des Human-

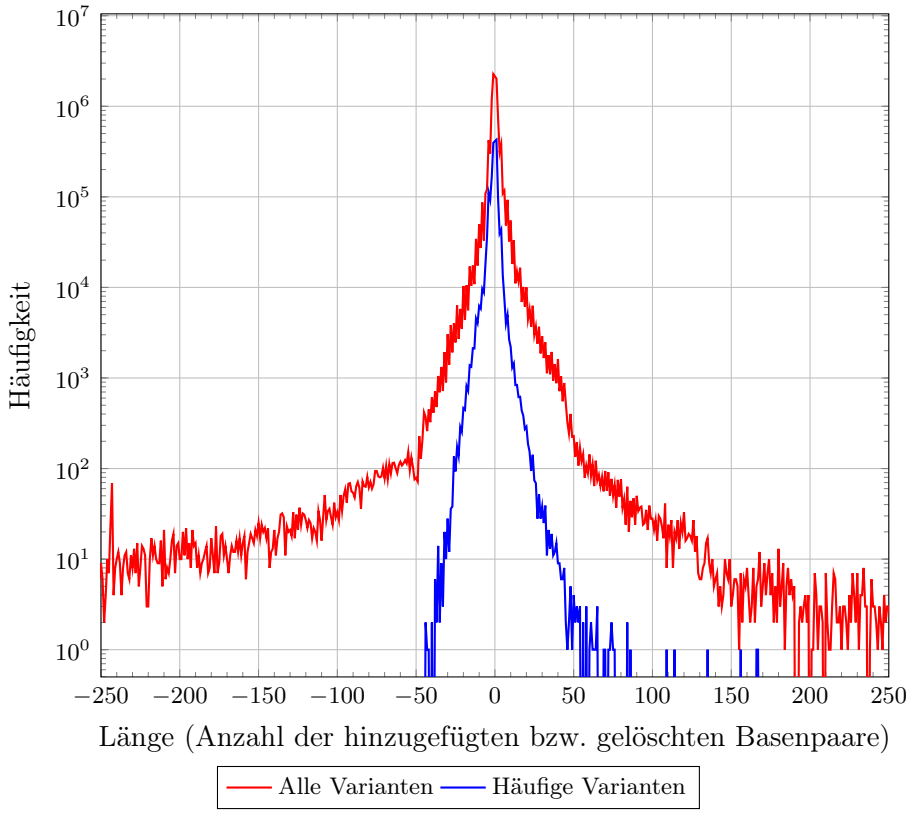


Abbildung 3.5: Länge von Insertionen und Deletionen.

genoms hatte an jeder Position mindestens eine Variante, meist aber sogar zwei bis drei. Alle SNP-Kombinationen innerhalb eines q-Gramms zum Suchindex hinzuzufügen, ist daher nicht praktikabel. Eine Lösungsmöglichkeit wäre, nur eine beschränkte Anzahl von Kombinationen eines q-Gramms in den Index einzufügen. Wählt man dieses Limit hinreichend klein, steigt die Gesamtzahl der q-Gramme im Suchindex nur geringfügig an. Deutlich besser ist aber, nur die häufigen Varianten zu berücksichtigen, die bei mindestens 1% der Individuen einer Großpopulation vorkommen. Obwohl diese Datei immer noch viele Varianten enthält, treten keine extremen Variantenhäufungen mehr auf. Eine Beschränkung der Kombinationsanzahl ist hier nicht notwendig.

In Abschnitt 3.3.6 haben wir die Länge von Indels genauer analysiert. Es kann vorkommen, dass ein Read ausschließlich in einer Insertion zu finden ist. Berücksichtigt man aber nur die häufigen Varianten ist dieser Fall äußerst selten. Unter allen bekannten Varianten gibt es sehr lange Deletionen von fast 1000 Basenpaaren. Unter den häufigen Indels werden bei Deletionen stets weniger als 50 Basenpaare gelöscht.

Zusammenfassend kann man sagen, dass wir durch Erstellung der Statistiken einen genaueren Einblick in die Struktur der Varianten bekommen haben. Dadurch konnten wir entscheiden, welche Algorithmen sich für eine direkte Variantenberücksichtigung eignen und wie sie am besten implementiert werden.

Kapitel 4

Readmapper

Der von der Projektgruppe entwickelte Readmapper löst das in Kapitel 2.3 beschriebene Readmapping-Problem mittels eines zweistufigen Verfahrens.

Die erste Stufe ist das sogenannte *Mapping*, welches jedem Read eine Menge von Abschnitten aus dem Referenzgenoms zuordnet. Diese Abschnitte sollen hierbei die Bereiche des Referenzgenoms abdecken, welche die stärksten Ähnlichkeiten zum Read aufweisen.

Die zweite Stufe, das *Alignment*, vergleicht einen Read mit je einem der Referenzgenomabschnitte, welche ihm durch das Mapping zugeordnet wurden. Im Gegensatz zum Mapping stellt das Alignment nicht nur eine Überprüfung auf grobe Ähnlichkeit dar. Stattdessen wird durch zeichenweisen Vergleich von Read und Genomabschnitt der Bereich innerhalb des gemappeten Abschnitts ermittelt, welcher die geringsten Unterschiede zum Read aufweist. Zusätzlich zur genauen Position dieses Bereichs beschreibt das Alignment auch die Art der Unterschiede von Read und Referenzgenom.

Die ersten beiden Teile dieses Kapitels behandeln die Verfahren und Implementierungen dieser beiden Stufen. In Abschnitt 4.1 wird das für VATRAM entwickelte Mappingverfahren beschrieben, welches auf dem *Locality-Sensitive Hashing* beruht und neben dem Referenzgenom auch SNPs berücksichtigt. Abschnitt 4.2 befasst sich mit der Alignmentphase und der Umsetzung des variantentoleranten *semiglobalen Aligners*. Abschließend werden in Abschnitt 4.3 die Programmstruktur von VATRAM und einige Implementierungsdetails erläutert.

4.1 Mapping von Reads mittels LSH

Das Readmapping wird durch ein Hashing-Verfahren realisiert, da ein direkter Brute-Force-Ansatz zur Alignierung der Reads an die Referenzgenome unpraktikabel ist. Wir haben uns dazu entschlossen, das sogenannte *Locality-Sensitive Hashing* einzusetzen. Dieses zeichnet sich dadurch aus, dass es Ähnlichkeit von Daten approximativ bestimmen kann. Eine geeignete Hashfunktion sorgt dafür, dass die Wahrscheinlichkeit einer Kollision umso größer ist, je ähnlicher die Daten sind (Gionis et al., 1999). Durch diesen Schritt erhalten wir eine Menge von Positionen, an denen sich Reads mit hoher Wahrscheinlichkeit an die Referenz alignieren lassen.

In diesem Abschnitt wird zunächst die Grundidee des LSH erläutert und einige wichtige Begriffe eingeführt. Im weiteren Verlauf wird genauer auf die Einzelheiten unseres Algorithmus eingegangen. Abschließend erfolgt die Präsentation des Algorithmus.

4.1.1 Grundidee

Hashing ist im Allgemeinen die Abbildung einer beliebig großen Eingabe auf eine Ausgabe fester Größe. Unter anderem können mit einer Hashfunktion auch Mengen von Werten auf eine feste Ausgabe abgebildet werden, was in VATRAM ausgenutzt wird. Während kryptographische Hashfunktionen ähnliche Werte auf sehr unterschiedliche Hashwerte abbilden sollen, zeichnen sich Locality-Sensitive-Hashfunktionen durch genau das Gegenteil aus. Durch den Einsatz von solchen Funktionen, deren genaue Erläuterung im Abschnitt 4.1.3 erfolgt, ist es möglich, Ähnlichkeiten anhand der Ähnlichkeit der Hashwerte zu erkennen. *Locality-Sensitive Hashing* (im folgenden *LSH*) wird häufig zur Duplikatauffindung in großen Datensätzen benutzt. Insbesondere beim Webcrawling können sehr ähnliche *Dokumente* bzw. exakte Duplikate mit diesem Verfahren schnell gefunden werden. Um dieses Konzept auch im Readmapping anwenden zu können, teilen wir das Referenzgenom in Abschnitte auf, deren Länge einer vorher spezifizierten Fensterbreite entsprechen. Diese sollte auf die Länge der Reads abgestimmt sein. Sowohl jeder Abschnitt des Referenzgenoms als auch jeder Read wird als *Dokument* aufgefasst. Nun erfolgen zwei Schritte:

- Im Min-Hashing wird eine Signaturmatrix berechnet. Diese repräsentiert spaltenweise die Menge aller Abschnitte (*Dokumente*) im Referenzgenom. Pro Spalte existieren mehrere Zeilen von Hashwerten, die ein Dokument repräsentieren. Diese Menge von Hashwerten nennt sich *Signatur* des Abschnitts.
- Der LSH-Algorithmus erhält diese Signaturmatrix als Eingabe und findet für gegebene Reads ähnliche Abschnitte im Referenzgenom.

Nur die Hashwerte der Referenzgenomabschnitte werden in Hashbuckets der Tabelle gespeichert, während Hashwerte der Reads *on the fly* erzeugt werden. Anders ausgedrückt wird ein Stream von eingehenden Reads mit einer festen Menge von Referenzgenomabschnitten vereinigt (*Join).* Ein Read wird einem oder mehreren Referenzgenomabschnitten zugeordnet, falls die Signaturen des Reads und des Abschnitts gleiche Hashwerte aufweisen. Somit sind die erzeugten Paare mit hoher Wahrscheinlichkeit sehr ähnlich zueinander, da die Hashwerte der Signaturen *locality-sensitive* sind. Daher sind die Ausgabepaare immer nur eine Kandidatenmenge, die durch einen weiteren Algorithmus an das Referenzgenom aligniert werden muss, wie in Kapitel 4.2 gezeigt wird. Ein großer Vorteil ist aber, dass somit auch Varianten erkannt werden können, da die Kollisionswahrscheinlichkeit für eine geringe Modifikation an der Eingabe immer noch groß genug ist.

4.1.2 Wichtige Begriffe und Definitionen

In diesem Abschnitt werden verschiedene Begrifflichkeiten geklärt, die für das Verständnis der Vorgehensweise unerlässlich sind.

Anwendung von q-Grammen

Ein wichtiger Aspekt zur Vergleichbarkeit der Ähnlichkeit von Textfragmenten liefert die folgende Abstraktion: Ein beliebiger String s über dem Alphabet Σ der Länge n lässt sich durch Teilstrings der Länge $q < n$ darstellen. Jeder Teilstring wird als **q-Gramm**[1] bezeichnet. Zur Veranschaulichung sei folgende DNA-Sequenz gegeben:

AGAGTC

Wählt man $q = 2$, lässt sich der String in folgende *2-Gramme* aufteilen:

AG, GA, (AG), GT, TC

Ein beliebiger String in einem Datensatz lässt sich durch seine q-Gramme auffinden. Dazu benötigt man einen **q-Gramm-Index**, welcher zu jedem q-Gramm seine Startpositionen der Vorkommnisse im Gesamtstring angibt. Sei zur Veranschaulichung wieder der oben eingeführte String mit seinen 2-Grammen gegeben. Für jedes 2-Gramm gibt man nun die Indizes an, an welchen das 2-Gramm im Ausgangsstring vorkommt:

AG:	1,3
GA:	2
GT:	4
TC:	5

Aus diesen Angaben lässt sich der Ausgangsstring jederzeit rekonstruieren - er kann also durch einen q-Gramm-Index repräsentiert werden.

Gapped q-Gramme

Gapped q-Gramme sind eine Variation der normalen q-Gramme, die den Text mit einer Art lückenhaften Schablone indiziert. Als Schablone verwendet man Strings über dem Alphabet $\{\#, -\}$, wobei $-$ eine Lücke darstellt. Eine mögliche Gapped 3-Gramm-Schablone wäre also #-#, die von drei aufeinanderfolgenden Zeichen immer nur das vordere und hintere berücksichtigt. Beispiel:

AGAGTC

Schablone: #-# Gapped q-Gramme:

AA, GG, AT, GC

Insbesondere sind normale q-Gramme ein Spezialfall der Gapped q-Gramme ohne Lücken. Gapped q-Gramme dürfen außerdem an erster und letzter Stelle keine Lücke enthalten. In der Praxis können sich die Lücken als vorteilhaft bei einem q-Gramm-Index erweisen. Durch die Lücken können Fehler oder Varianten besser toleriert werden, wenn sie genau in die Lücke fallen.

[1] In der Literatur finden sich weitere Bezeichnungen wie *n-Gramm*, in der Genomik auch *k-mer*.

Hamming-Distanz

Die Hamming-Distanz kann als Ähnlichkeitsmaß für zwei gleich lange Zeichenketten benutzt werden. Die Distanz entspricht der Anzahl der Stellen, an denen sich die Strings unterscheiden. Seien beispielsweise folgende Sequenzen gegeben:

$$\begin{array}{c} \text{A}\underline{\text{CC}}\text{AT} \\ \text{A}\underline{\text{GG}}\text{AT} \end{array}$$

An den unterstrichenen Positionen 2 und 3 unterscheiden sich beide Sequenzen. Dementsprechend ist die Hamming-Distanz $d = 2$.

Edit-Distanz

Die Edit-Distanz (oder auch Levenshtein-Distanz) erweitert die Hamming-Distanz dahingehend, dass auch Strings unterschiedlicher Länge vergleichen werden können. Ausgehend vom Ursprungsstring wird die minimale Anzahl an Änderungen berechnet, mit welcher dieser in den Zieltext transformiert werden kann. Als Änderung zählen dabei Einfüge- und Löschoperationen, sowie Ersetzen eines Zeichens. Sei beispielsweise der folgende String gegeben:

$$\begin{array}{c} \text{A}\underline{\text{C}}\text{C}\underline{\text{A}}\text{T} \\ \text{AGCT} \end{array}$$

Durch genau zwei Operationen an den unterstrichenen Stellen (Ersetzen von C durch G sowie Löschen von A) lässt sich der obere String in den unteren transformieren.

Jaccard-Ähnlichkeit

Eine allgemeinere Metrik, welche sich nicht nur auf Strings oder q-Gramme bezieht, ist die **Jaccard-Ähnlichkeit**. Sie wurde als Maß für die Ähnlichkeit zwischen Mengen von Jaccard (1901) wie folgt definiert (Leskovec et al., 2014):

$$J(A, B) = \frac{\|A \cap B\|}{\|A \cup B\|} \in [0, 1] \tag{4.1}$$

Dieses Ähnlichkeitsmaß hat eine besondere Bedeutung für das Min-Hashing, welches im folgenden Abschnitt behandelt wird.

4.1.3 Min-Hashing

Im typischen Anwendungszweck einer Hashfunktion soll sich der Ausgabehashwert für bereits kleine Unterschiede der Eingabe signifikant unterscheiden. Das *Locality-Sensitive Hashing* ist dazu in gewisser Weise komplementär:

Mittels Min-Hashing können Eingabedaten mit wenigen Unterschieden auf denselben Hashwert abgebildet werden.

Ein großes Problem ist dabei, welche Ähnlichkeitsmetrik verwendet wird. Für das gewünschte Verhalten von Min-Hashing-Funktionen lässt sich die in (4.1) eingeführte Jaccard-Ähnlichkeit einsetzen, welche nach Indyk und Motwani (1998) folgende Eigenschaften erfüllen soll:

$$h(A) = h(B), \text{ falls } J(A, B) \text{ nah an } 1$$
$$h(A) \neq h(B), \text{ falls } J(A, B) \text{ nah an } 0$$

Um auch die Randfälle abzudecken, könnte man dies auch als Wahrscheinlichkeit definieren. Die Wahrscheinlichkeit, dass die Hashwerte von A und B übereinstimmen soll mit dem Wert des Jaccard-Koeffizienten korrelieren. Also

$$P(h(A) = h(B)) = J(A, B)$$

Um das Min-Hashing anwenden zu können, müssen die zu untersuchenden Daten in Abschnitte aufgeteilt werden, die im Kontext des *Locality-Sensitive Hashings* auch häufig als *Dokumente* bezeichnet werden. Dies lässt sich einfach durch Bildung von sich überlappenden q-Grammen wie im folgenden Beispiel bewerkstelligen:

Dokument 1: TTACCG mit den q-Grammen: TT, TA, AC, CC, CG
Dokument 2: GCAT mit den q-Grammen: GC, CA, AT
Dokument 3: TTACC mit den q-Grammen: TT, TA, AC, CC
Dokument 4: CGCAT mit den q-Grammen: CG, GC, CA, AT

q-Gramme	Dokument 1	Dokument 2	Dokument 3	Dokument 4
AC	✓		✓	
CG	✓			✓
CA		✓		✓
AT		✓		✓
GC		✓		✓
CC	✓		✓	
TT	✓		✓	
TA	✓		✓	

Tabelle 4.1: Boolesche Matrix der q-Gramme der Dokumente

Alle Dokumente lassen sich auch mit Hilfe einer booleschen Matrix (*Signaturmatrix*) darstellen, wie in Tabelle 4.1 zu erkennen ist. Es wird in der Matrix nur die Existenz eines q-Gramms in einem Dokument aufgeführt, nicht jedoch die Anzahl der Vorkommen. In der Darstellung erhalten zwei ähnliche Dokumente eine ähnliche Signatur (Leskovec, 2014). Dadurch wird wie Auffindung von Dokumenten mit ähnlichen Sequenzen vereinfacht, welche jedoch nicht zwangsläufig in der gleichen Reihenfolge vorkommen müssen. Ursprünglich wurde dieser Zusammenhang für die Suche von Internetseiten mit ähnlichem Inhalt benutzt (Slaney und Casey, 2008).

In den nächsten Abschnitten wird auf die Konstruktion der Signaturmatrix eingegangen. Zunächst wird aber eine wichtige Eigenschaft eingeführt, welche für den Konstruktionsalgorithmus benötigt wird.

Min-Hash-Eigenschaft

Leskovec (2014) definiert eine Hashfunktion, welche auf der *Min-Hash-Eigenschaft* aufbaut und die von Indyk und Motwani (1998) geforderten Eigenschaften erfüllt. Diese sagt aus, dass die Ähnlichkeit von zwei Signaturen dem Anteil ihrer übereinstimmenden Hashfunktionen entspricht:

4.1.1 Theorem. *Seien D_1 und D_2 Dokumente, also Mengen von q-Grammen. Sei π eine zufällige Permutation aller existierenden q-Gramme aller Dokumente. Sei $h_\pi(D)$ eine Hashfunktion, welche als Wert den Index der ersten Zeile (bzgl. der Permutation π) hat, für die das dazugehörige q-Gramm im Dokument D existiert.*
Die Wahrscheinlichkeit, dass die Min-Hash-Werte zweier Dokumente D_1 und D_2 gleich sind, entspricht der Jaccard-Ähnlichkeit beider Dokumente.

$$P(h_\pi(D_1) = h_\pi(D_2)) = J(D_1, D_2)$$

Beweis. Seien D_1 und D_2 Dokumente, also Mengen von q-Grammen und π eine zufällige Permutation aller q-Gramme in $D_1 \cup D_2$. Sei $d \in D$ ein einzelnes q-Gramm. Dann gilt:

$$P(\pi(d) = \min(\pi(D))) = \frac{1}{|D|}$$

$\pi(D)$ bezeichnet dabei den Index des ersten q-Gramms gemäß π, welches im Dokument D enthalten ist. Durch die zufällige Permutation π ist es für jedes q-Gramm gleich wahrscheinlich, dass dieses das erste existierende q-Gramm gemäß π ist.

Sei d nun so gewählt, dass $\pi(d) = \min(\pi(D_1 \cup D_2))$, also dass das q-Gramm d in der Mengenvereinigung als minimales Element auftritt. Dann gilt:

$$\pi(d) = \begin{cases} \min(\pi(D_1)), & \text{falls } d \in D_1 \\ \min(\pi(D_2)), & \text{falls } d \in D_2 \end{cases}$$

Es können auch beide Fälle eintreten, und zwar genau dann wenn $x \in C_1 \cap C_2$. Führt man beides zusammen, gilt

$$P(\min(\pi(C_1)) = \min(\pi(C_2))) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|} = J(C_1, C_2) \qquad \square$$

Generierung der Signaturmatrix

Mit Hilfe dieser Eigenschaft kann nun die Signaturmatrix erstellt werden. Da es zu aufwendig ist, viele verschiedene zufällige Permutationen zu berechnen, bedient sich Leskovec (2014) eines Tricks und verwendet zufällige Hashfunktionen. Die Signaturmatrix wird nun nach Algorithmus 4.1 wie folgt aufgestellt:

Eingabe: Boolsche Matrix B
Ausgabe: Signaturmatrix M

```
1: Initialisiere die Signaturmatrix M mit ∞
2: for jede Zeile r der boolschen Matrix B do
3:     for all Spalte c aus B do
4:         if Zeile r in Spalte c in B hat ✓ then
5:             for all Hashfunktion h_i ∈ H do
6:                 if h_i(r) < M[i] then
7:                     M[i] ← h_i(r)
8:                 end if
9:             end for
10:        end if
11:    end for
12: end for
13: return M
```

Algorithmus 4.1: Hashfunktion für die Permutation

Die Signaturmatrix wird zunächst leer initialisiert. Nun wird jede Zeile der boolschen Matrix durchlaufen. Steht in Zeile r *false*, dann wird diese Zeile keinen Einfluss auf die Signaturmatrix haben können, da nach Definition die Zeile gespeichert wird, in der gemäß der jeweiligen Permutation zuerst der Wert *true* steht. Falls in Zeile r *true* steht, dann werden alle Hashwerte für jede Funktion berechnet und geprüft, ob der berechnete Wert kleiner ist als der aktuell in der Signaturmatrix gespeicherte Wert. Dies wird für jede Zelle der Ausgangsmatrix ausgeführt, sodass in der Signaturmatrix die minimalen Werte der Permutation stehen. Tabelle 4.2 zeigt die Signaturmatrix, welche nach Anwendung des Algorithmus auf die boolesche Matrix aus Tabelle 4.1 entsteht.

4.1.4 Locality-Sensitive Hashing

Durch Erstellung der Signaturmatrix konnte eine Reduktion der q-Gramm-Mengen auf eine konstante Anzahl von Signaturwerten durchgeführt werden. Da die Signaturwerte durch Min-Hashing erzeugt wurden, gehen dabei aber nicht viele Informationen verloren: Ähnliche Dokumente, also ähnliche q-Gramm-Mengen, werden auf weiterhin ähnliche Signaturen abgebildet. Mit Hilfe der Signaturmatrix könnte jetzt schon ein Ähnlichkeitsvergleich zwischen allen Dokumenten in $\mathcal{O}(n^2)$ Zeit durchgeführt werden, wobei n die Anzahl der Dokumente kennzeichnet. Der naive Ähnlichkeitsvergleich könnte zum Beispiel jedes Dokument i mit allen anderen Dokumenten $j \neq i$ vergleichen, indem die Anzahl der gemeinsamen Signaturwerte berechnet wird. Um aber noch mehr Laufzeit zu gewinnen, verwenden wir an dieser Stelle ein weiteres Hashing-Verfahren namens *Locality-Sensitive Hashing*.

Zu diesem Zweck wird die Signaturmatrix in Bänder aufgeteilt, d.h. es werden pro Dokument mehrere Signaturwerte zu einem Band zusammengefasst (wie in Tabelle 4.3 dargestellt).

Die Idee ist nun, Ähnlichkeit zwischen zwei Dokumenten dadurch zu charakterisieren, dass sie in einem Band die exakt gleiche Menge von Signaturwerten haben.

Permutation π				q-Gramme	Dokumente			
π_1	π_2	π_3	π_4		D_1	D_2	D_3	D_4
2	4	3	1	AC	✓		✓	
3	2	4	8	CG	✓			✓
7	1	7	3	CA		✓		✓
6	8	2	7	AT		✓		✓
1	6	8	2	GC		✓		✓
5	7	1	5	CC	✓		✓	
4	5	5	6	TT	✓		✓	
8	3	6	4	TA	✓		✓	

Diese Eingabematrix M führt zu folgender Signaturmatrix M:

	D_1	D_2	D_3	D_4
π_1	2	1	2	1
π_2	2	1	3	1
π_3	1	2	1	2
π_4	1	2	1	2

Tabelle 4.2: Erstellung der Signaturmatrix M nach Algorithmus 4.1

Permutation	Dokumente				
	D_1	D_2	D_3	D_4	
π_1	2	1	2	1	Band 1
π_2	2	1	3	1	
π_3	1	2	1	2	Band 2
π_4	1	2	1	2	

Tabelle 4.3: Aufteilung der Signaturmatrix in Bänder

Falls zwei Dokumente ähnlich sind, aber im ersten Band nicht die gleiche Menge von Signaturwerten haben, ist die Wahrscheinlichkeit trotzdem groß, dass sie in einem anderen Band die gleiche Menge haben. Der große Vorteil durch diese Vereinfachung ist, dass die Suche von gleichen Teildaten einer Join-Operation in Datenbanken entspricht. Join-Operationen lassen sich effizient durch Hashing lösen: Anstatt paarweise Vergleiche durchzuführen, kann für jedes Band pro Dokument ein Hashwert berechnet werden. Offensichtlich werden dadurch gleiche Werte auf gleiche Hash-Buckets abgebildet. Dadurch folgt eine Reduktion der Laufzeit von $\mathcal{O}(n^2)$ auf $\mathcal{O}(n)$, denn es muss nur einmal über alle Dokumente iteriert werden. Pro Dokument wird für jedes Band ein Hashwert berechnet. Anschließend wird das aktuelle Dokument in den entsprechenden Bucket in der Hashtabelle eingefügt. Waren vorher schon andere Dokumente in dem Hashbucket, dann tritt eine *Kollision* auf und diese Dokumente werden zu dem aktuellen Dokument als *ähnlich* klassifiziert. Mit anderen Worten: Wurde für zwei Dokumente in irgendeinem Band der gleiche Hashwert berechnet,

Band 1		Band 2	
Hashwert	Dokumente	Hashwert	Dokumente
22	D_1	22	D_2, D_4
11	D_2, D_4	11	D_1, D_3
23	D_3		

Tabelle 4.4: Resultierende Hashtabellen für die Bänder aus Abbildung 4.3

dann werden die Dokumente direkt als ähnlich klassifiziert. Tabelle 4.4 zeigt die resultierenden Hashtabellen für die beiden Bänder. Durch Anpassen der Bandgröße kann die Ähnlichkeitstoleranz variiert werden. In der Praxis zeigte sich allerdings, dass eine Bandgröße von eins tatsächlich die besten Ergebnisse im Kontext des Readmappers erzielt.

4.1.5 Implementierung

Im folgenden Abschnitt wird die Implementierung vorgestellt.

Wie bereits erläutert, werden Abschnitte aus dem Referenzgenom und einzelne Reads als *Dokument* im Sinne des LSH aufgefasst. In der ersten Implementierung wurden Referenzteile und Reads simultan in die Hashtabelle eingetragen. Zwei Kriterien sprechen gegen eine solche Arbeitsweise:

- Wird ein Dokument an einen Hash-Bucket eingefügt, muss zusätzlich gespeichert werden, ob es sich bei dem Dokument um einen Referenzabschnitt oder einen Read handelt
- Reads können in dieser Form auch mit anderen Reads kollidieren

Letzteres kann unter Umständen in einer anderen Anwendung nützlich sein, bringt aber keinen Vorteil für das Readmapping, wie es im Rahmen dieser Projektgruppe betrieben werden soll. Um das Speichern des Dokumententyps zu umgehen, werden Referenzteile und Reads nun nacheinander bearbeitet. Im ersten Schritt werden die Abschnitte aus dem Referenzgenom in die Hashtabelle eingetragen. Ist das komplette Genom abgearbeitet, werden anschließend paketweise Reads behandelt. Anstatt Reads in der Hashtabelle zu speichern, wird nur eine Kollision mit Referenzteilen gemeldet. Falls ein Read in keinem Band mit einem Referenzabschnitt kollidiert, wird dieser als non-matching markiert und in einer separaten Liste gespeichert. Alle anderen Referenzabschnitt-Read-Kollisionspaare werden ebenfalls in einer Liste gespeichert.

4.1.5.1 Variantenberücksichtigung

Die Erzeugung einer q-Gramm-Menge aus einem String ist eine systematische und einfache Berechnung, die linear viele Operationen im Bezug zur Stringlänge benötigt. Im Spezialfall des Referenzgenoms sind die Strings allerdings nicht immer eindeutig,

denn es können innerhalb eines Fensters mehrere Varianten auftreten, die gesondert berücksichtigt werden müssen. Wie in Abschnitt 3.3.3 gezeigt wurde, ist es wegen der kombinatorischen Explosion nicht möglich, alle durch die Varianten entstehenden Kombinationen zur q-Gramm-Menge hinzuzufügen. Stattdessen stehen in der Implementierung zwei Parameter zur Verfügung, die einen Kompromiss ermöglichen. Der Parameter *limit* legt die größte Anzahl von möglichen Kombinationen fest, die alle als q-Gramm zur q-Gramm-Menge hinzugefügt werden sollen. Überschreitet die mögliche Kombinationsanzahl diesen Wert, aber ist kleiner als *limit_ skip*, dann wird eine zufällige Teilmenge von Kombinationen zur q-Gramm-Menge hinzugefügt. Falls mehr Kombinationen als *limit_ skip* möglich sind, dann wird keine Kombination zur q-Gramm-Menge hinzugefügt. Dadurch wird verhindert, dass besonders variantenreiche Abschnitte im Referenzgenom nicht übermäßig große q-Gramm-Mengen erzeugen, die durch ihre Größe schlechter mit dem LSH-Verfahren gemappt werden können. Das liegt daran, dass die Wahrscheinlichkeit, gemäß einer bestimmten Permutation das gleiche Minimum zu haben, für zwei sehr unterschiedlich große Mengen eher gering ist. Standardmäßig werden die Werte 3 für *limit* und 8 für *limit_ skip* verwendet.

4.1.5.2 Signaturberechnung

Während der Signaturberechnung eines Dokuments wird einerseits die Menge der im Dokument vorkommenden q-Gramme und andererseits das Min-Hashing berechnet. Zunächst sei zu erwähnen, dass nicht die eigentliche q-Gramm-Menge gespeichert wird, sondern nur direkt berechnete Hashwerte der q-Gramme. Das dient dazu, den Speicher für große q nicht zu sehr zu beanspruchen, denn die konkreten q-Gramme werden nach der Min-Hash-Berechnung nicht mehr benötigt.
Im vorigen Abschnitt wurde im Rahmen der Signaturberechnung erläutert, dass verschiedene Hashfunktionen benutzt werden, um die Permutationen zu implementieren. Tatsächlich wird allerdings nur eine einzige Hashfunktion benutzt, wodurch sie die Implementierung erheblich vereinfacht. Die Standard-Hash-Funktion aus der C++11-Standardbibliothek stellte sich als zu langsam heraus. Stattdessen wird ein einfacher Rolling-Hash mit der Primzahl 33767 verwendet.

```
std::hash<unsigned int> hashFunction;
```

Um beliebig viele andere Hashfunktionen zu simulieren, werden stattdessen Zufallszahlen bitweise mit dem von *hashFunction* berechneten Wert verarbeitet. Dazu wird vor dem Ausführen des Algorithmus ein Array *hashFunctions* mit Zufallszahlen initialisiert, das so groß ist wie die gewünschte Signaturlänge. Für jedes q-Gramm in einem Dokument wird nur einmal die Hashfunktion berechnet und als *qGramHash* zwischengespeichert. Anschließend wird über das Array *hashFunctions* iteriert und *qGramHash* wird per XOR mit der Zufallszahl an der Stelle i aus dem Array verknüpft:

```
unsigned int hashValue = qGramHash ^ hashFunctions[i];
if (hashValue < signature[i])
    signature[i] = hashValue;
```

Um zu vermeiden, dass sich "schlechte" Zufallszahlen negativ auf die Verteilung der Hashwerte auswirken, kann zusätzlich eine gleichmäßige Verteilung erzwungen wer-

den: Hierzu werden die Bits von hashFunctions[i] jeweils in zwei Hälften aufgeteilt. Die höherwertigeren Bits werden dann in gleichen Abständen angeordnet, während die unteren Bits weiterhin Zufallswerte erhalten. Somit ist sichergestellt, dass keine besonders schlechte Verteilung entsteht.
In der Implementierung wird versucht, die Performanz der Signaturerzeugung zu optimieren, da sich dieser Teil des LSH als am rechenintensivsten herausstellte. Dazu wurde die Signaturberechnung der Reads mittels OpenMP parallelisiert. Da die Reads untereinander keinen Zusammenhang haben, lässt sich die for-Schleife, die über die Reads iteriert, direkt mittels einer OpenMP-Präprozessordirektive parallelisieren. Dadurch entstehen nur disjunkte Datenzugriffe sodass dazu keine Synchronisationsmechanismen benötigt werden. Zudem wurden beim Berechnen des MinHashes unnötig komplizierte Branches vermieden, sodass moderne Compiler automatische Vektorisierung einsetzen können. Der generierte Assemblercode nutzt dadurch SIMD-Instruktionen (*Single Instruction, Multiple Data SIMD*) und erhöht somit die Performanz.

4.1.5.3 Umsetzung der Bandhashes

Zum Speichern der Hashwerte und deren zugehöriger Fensterindizes kann die Datenstruktur std::unordered_map aus der Standardbibliothek verwendet werden. In jedem Hashbucket wird eine Liste von *HashBucketEntry*s gespeichert:

```
BandHashEntry {
        uint32_t key;
        uint32_t window;
}
std::unordered_map
    <unsigned int, std::vector<BandHashEntry>> hashTable;
```

Ein Dokument (Referenzabschnitt) wird mit einem *HashBucketEntry* durch die Dokument-ID innerhalb des Chromosoms eindeutig zugeordnet[2].
Schnell wird klar, dass die Datenstruktur in dieser Form viel Mehraufwand betreibt und ineffizient ist, weil für jeden Eintrag in der Hashtabelle eine Instanz von std::vector erzeugt wird, die unnötig viel Speicher verschwendet. In der Praxis zeigte sich, dass viele Einträge der Hashtabelle nur weniger als fünf Window-Indizes speichern müssen. Daher stellt sich die Frage, ob Einträge dieser Art auch in kompakterer Form abgespeichert werden können. Beim Betrachten der praxisrelevanten Werte von windowID fällt auf, dass nicht alle 32 Bits benutzt werden, um alle Windows in einem Chromosom adressieren zu können.. In der Praxis brauchen nur 26 Bits benutzt werden, um alle Fenster der Länge 100 in einem Chromosom speichern zu können. Diese Tatsache kann man ausnutzen, um Zusatzinformationen ohne weiteren Speicherverbrauch zu hinterlegen.

4.1.6 WindowManager

In der Implementierung steht dafür eine Klasse namens WindowManager bereit, die folgende Aufgabe übernimmt: Es werden zwei dynamische Arrays in Form von

[2]Es wird nur ein Chromosom pro Durchlauf betrachtet

std::vector gespeichert, die feste Arrays der Länge zwei und vier beinhalten. Außerdem steht ein dynamisches Array mit ebenfalls dynamischen std::vector zur Verfügung. Je nach Notwendigkeit können die zugehörigen Window-Indizes in einem dieser globalen Arrays gespeichert werden. Abbildung 4.1 zeigt einen BandHashEntry für

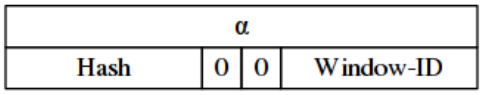


Abbildung 4.1: BandHashEntry ohne mit nur einem Window.

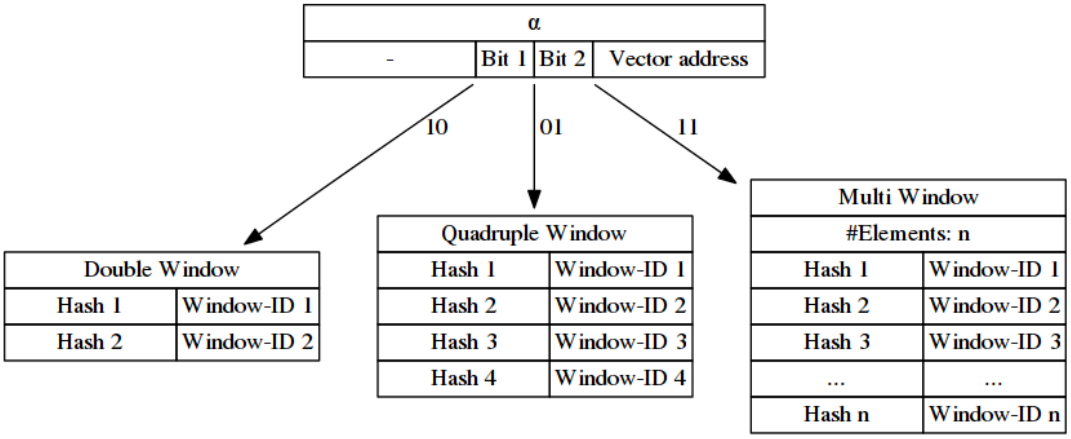


Abbildung 4.2: Struktur des Windowmanagers.

den Fall, dass nur eine einzige Window-ID unter diesem Hash gespeichert wird. Die beiden obersten höherwertigen Bits 00 signalisieren, dass es sich bei den anderen Bits um eine Window-ID handelt, die direkt ausgelesen werden kann. Abbildung 4.2 zeigt die anderen möglichen Konfigurationen der Bits, in denen der hintere Teil der Daten nicht mehr eine Window-ID repräsentiert, sondern als Adresse in einem externen zentral verwalteten Array interpretiert werden muss. Für die Konfigurationen 10, 01 und 11 der obersten beiden höherwertigen Bits wird jeweils signalisiert, dass es sich um eine Adresse auf das zentrale Array der Double-, Quadruple- bzw. Multi-Windows handelt. Um die eigentlichen Window-IDs zu erhalten, müssen also die externen Arrays an der entsprechenden Adresse ausgelesen werden. In dem BandHashEntry wird für diese Fälle kein Hash gespeichert, sodass Dieses Verfahren hat den Vorteil, dass für ein-, zwei- und vierfache Hashbucketeinträge keine Zusatzinformation über die Anzahl der Window-IDs gespeichert werden muss. Da dieser Fall relativ häufig auftritt, ist ein geringerer Speicherverbrauch in der Praxis deutlich bemerkbar.

4.1.7 Linear probing

Dadurch, dass std::unordered_map die Daten intern auch mittels Hashfunktion anordnet, können ungewollte Kollisionen zweier unterschiedlicher Schlüsselwerte auftre-

ten. Die Implementierung der Standardbibliothek erzeugt zur Auflösung der Kollisionen einen Speicheroverhead, der dadurch entsteht, dass im Falle einer Kollision eine Liste von Einträgen verwaltet werden muss. Um dieses Problem zu umgehen, wurde eine eigene Datenstruktur entwickelt, die diese Kollisionen ohne zusätzlichen Speicheraufwand behandeln kann. Hierzu wurde eine einfache Form des *Linear Probing* implementiert (in der Implementierung *CollisionFreeBandHashTable*). Im allgemeinen wird beim Linear Probing im Falle einer Kollision ein neuer Hashwert berechnet, solange bis ein freier Hashbucket gefunden wird. Der neue Hashwert berechnet sich hierbei einfach aus der Summe des Originalwertes mit einem Offset und einer anschließenden Modulooperation. In unserem Fall wurde ein Offset von eins verwendet, sodass bei einer Kollision einfach so lange der nächste benachbarte Hashbucket geprüft wird, bis ein leerer Bucket gefunden wurde. Durch Linear Probing konnte also der Einsatz von zusätzlichen Listen im Falle einer Kollision der Datenstruktur-Hashwerte vermieden werden.

4.1.8 SuperRank

Ein weiteres Problem der Datenstruktur, die zum Speichern der Hashwerte verwendet wird, ist die dünne Besetzung der Hashwerte im Zahlenbereich aller 32-Bit-Werte. Das ergibt sich vor allem dadurch, dass für jedes Band der MinHashes eine eigene Datenstrukturinstanz verwaltet wird. Um den Speicherbedarf für derartige Datenstrukturen zu verringern, verwendet man in der Regel Rank-Datenstrukturen. Eine Rank-Datenstruktur (in der Literatur häufig als *Succint data structure* bezeichnet) arbeitet nach folgendem Prinzip: Es soll eine Hashtabelle der Größe n gespeichert werden. Hierzu werden zunächst alle vorhandenen Schlüssel aufsteigend sortiert und

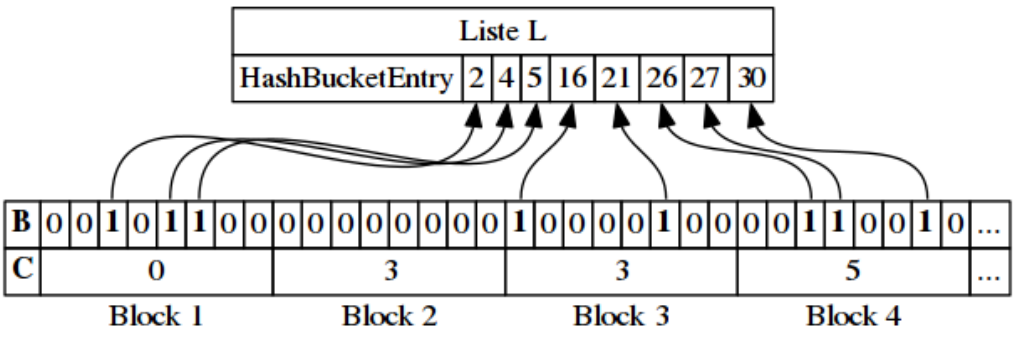


Abbildung 4.3: Struktur einer Rank-Datenstruktur.

anschließend in einer Liste L gespeichert, die exakt so groß ist wie notwendig. Zusätzlich wird ein Bitarray B der Länge n erzeugt, das an Stelle i den Wert 1 hat, falls der Hashwert i einen Schlüssel hat. Andernsfalls ist $B_i = 0$. Ein solches Bitarray wird mit möglichst großen Worten z.B. der Größe 64 blockweise abgespeichert. Für jeden Block wird in einem weiteren Array C die Summe aller Schlüssel (also die Summe aller Einsen) in allen vorangegangenen Blöcken gespeichert. Mit Hilfe dieser Summe und der Anzahl der Einsen vor der adressierten Stelle im adressierten Block kann direkt die Position des gesuchten Schlüssels in der Liste L berechnet werden. Abbildung 4.3 zeigt den prinzipiellen Aufbau der beschriebenen Arbeitsweise. Alle

HashBucketEntrys werden in der Implementierung auch in der Rank-Datenstruktur mit dem WindowManager gespeichert, sodass auch hier nur Listen mehrerer Windowindizes gespeichert werden, falls diese überhaupt notwendig sind. Die Tatsache, dass für jedes Chromosom und jedes Band eine eigene Datenstrukturinstanz erzeugt wird, führt dazu, dass das Bitarray B nur spärlich belegt ist. Beispielsweise ist in Abbildung 4.3 der dritte Block nur mit Nullen belegt. Daher wurde in der Implementierung eine zweite Hierarchiestufe verwendet, die das Bitarray erneut mit dem gleichen Prinzip komprimiert: Es wird ein zusätzliches Bitarray B_S (Super-Bitarray)

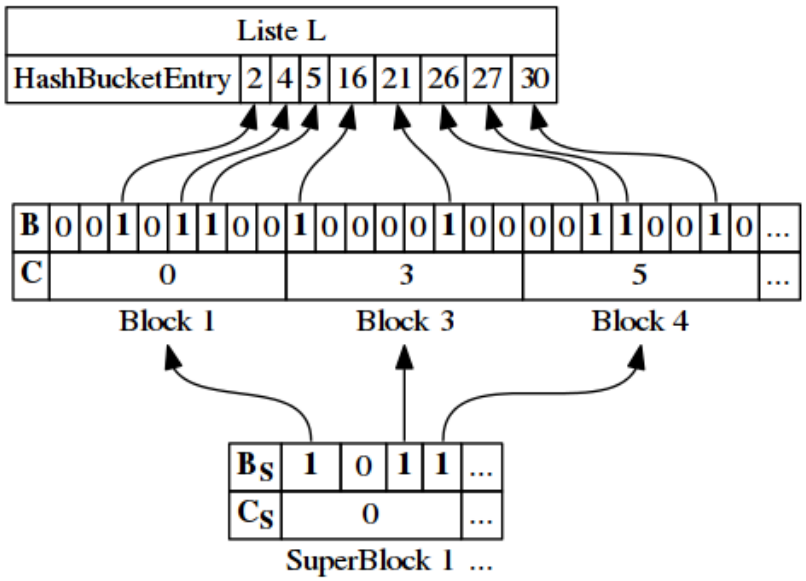


Abbildung 4.4: Zweistufige Rank-Datenstruktur.

verwaltet, dass für jeden Block in B ein Bit speichert. Wenn also im Block i des Bitarrays B eine Eins existiert, dann wird das i-te Bit in B_S auf Eins gesetzt. Somit brauchen nur nicht-leere Blöcke in B gespeichert werden, die mit einem weiteren zusätzlichen Indexarray C_S adressiert werden kann. Abbildung 4.4 zeigt den Aufbau der erweiterten Datenstruktur, der in der Implementierung auch als *SuperRank* bezeichnet wird. Um eine Anfrage zu verarbeiten, ob ein Hashwert h in der der Liste L vorhanden ist, muss also zunächst im entsprechenden Superblock geprüft werden, ob in dem Block, in dem das Existenzbit für h vorzufinden ist, überhaupt eine Eins steht. Anschließend wird B mittels C_S adressiert und ggf. danach L mittels C, wie bereits beschrieben. Die Zugriffszeit in $\mathcal{O}(1)$ ist in der Praxis vernachlässigbar klein. Der einzige Nachteil der Datenstruktur ist, dass sie nicht ohne großen Zusatzaufwand dynamisch benutzt werden kann. Da das Humangenom für die meisten Anwendungen als statisch betrachtet werden kann, ist die SuperRank-Datenstruktur ein guter Kompromiss zwischen Laufzeit und Speicherverbrauch. Falls in zukünftiger Arbeit auch dynamische Änderungen am Humangenom gemacht werden sollen, kann für diesen Zweck die CollisionFreeBandHashTable benutzt werden, die im vorigen Abschnitt diskutiert wurde.

4.1.9 Intervallbestimmung für den Aligner

Während bisher diskutiert wurde, wie ein Read grob auf einen Abschnitt(Window) im Referenzgenom gemappt werden kann, stellt sich noch die Frage, mit welchen Paramern der Aligner aufgerufen werden soll. Hierbei ist zu beachten, dass Reads im Regelfall nicht komplett im Window liegen, sondern eher an einer Seite überstehen. Für das LSH-Verfahren stellt das kein Problem dar, denn auch in diesem Fall können die Minima gemäß einer Permutation für das Window und den Read gleich sein. Der Aligner soll das genaue Mapping feststellen und braucht dafür natürlich einen hinreichend großen Abschnitt im Referenzgenom, der etwas größer ist als das Window. Gleichzeitig darf dieses Intervall aber auch nicht zu großzügig gewählt werden, weil damit auch die Laufzeit erheblich steigt.
In der Implementierung wird in der Methode *findInterval* deshalb folgendermaßen vorgegangen: Für einen Read werden im Mapping-Vorgang alle Fensterkandidaten mittels LSH-Verfahren bestimmt. Inbesondere kann ein Fenster mehrere Male durch Kollisionen in verschiedenen Bändern gefunden werden. Außerdem ist es auch möglich, dass mehrere Fenster gefunden werden, die sich entweder nebeneinander befinden oder auch z.B. wegen eines Mappingfehlers weit auseinander liegen. Um das bestmögliche Intervall zu bestimmen, wird zunächst für jedes Fenster gezählt, wie häufig es mit dem Read kollidiert ist. Der Parameter *maximumOfReturnedWindows* bestimmt hierbei, wie oft ein Fenster gefunden werden darf, um noch als Kandidat klassifiziert zu werden. Damit wird ausgeschlossen, dass derartige Fenster nicht die Laufzeit unnötig verlängern, ohne dass dabei ein merkbarer Unterschied in der Qualität entsteht. In der Implementierung ist hierzu die Methode *findReadOccurences* zuständig, die eine nach WindowIndex sortierte Menge von Windows zusammen mit deren Anzahl der Kollisionen zurückgibt. Anschließend wird eine Methode *compressOccurrences* aufgerufen, die aus der gefundenen Menge sogenannte WindowSequences extrahiert. Eine WindowSequence ist eine Menge unmittelbar aufeinanderfolgender Windows, die alle auf den gleichen Read gemappt wurden. Im häufigsten Fall hat eine WindowSequence die Länge zwei, wenn der Read genau in der Mitte der Windows liegt. Es können durchaus mehrere Sequences gefunden werden, wenn *false positives* durch LSH gefunden werden. Um die einzelnen WindowSequences nach Güte zu bewerten, werden die Summen der Kollisionen in den Windows der WindowSequences berechnet und anschließend auf die Länge der WindowSequence normalisiert (*countScore* im Quelltext). Die Normalisierung dient dazu, dass längere Sequences nicht zwangsweise besser bewertet werden, obwohl in einer anderen kürzeren Sequence beispielsweise mehr Kollisionen auftraten als in dem am häufigsten gefundenen Window der langen Sequence. Die WindowSequences werden letztendlich als Array nach ihrer Güte absteigend zurückgegeben. Im letzten Schritt werden in der Methode *selectWindows* die Sequences ausgewählt, die verarbeitet werden sollen. Der Parameter *maxSelect* beschränkt die maximale Anzahl und *nextFactor* die maximale Verschlechterung von einer Sequence zur nächsten (es wird nach Güte absteigend iteriert). Wurden beispielsweise zwei besonders gute Sequences gefunden, die eine Güte von 10 aufweisen, und danach noch einige weitere Sequences der Güte 3, dann würden diese nicht mehr bearbeitet werden, wenn *nextFactor* z.B. auf 0.5 gelegt wird.
Die Intervalle der ausgewählten WindowSequences sollen an den Aligner übergeben werden und werden dazu vorher noch mit einer Heuristik verarbeitet. Für eine Win-

dowSequence der Länge eins wird das Intervall nach links und rechts gleichmäßig gemäß 4.2 erweitert.

$$\text{extend} = \text{extendNumber} - \frac{\#\text{hits} \cdot \text{extendBandMult}}{\#\text{bands} \cdot \text{windowSize}} \tag{4.2}$$

extendNumber stellt hierbei die maximale mögliche Erweiterung in eine Richtung dar. Mit zunehmender Anzahl von Kollisionen (*#hits*) wird der Bereich mit dem Faktor *extendBandMult* verkleinert. Zusätzlich wird die Skalierung auf die Bandanzahl und die Windowgröße normalisiert.
Bei WindowSequences der Länge 2 wird ein Bereich ausgewählt, der in beiden Windows liegt. Abbildung 4.5 zeigt eine WindowSequence der Länge 2. Das Intervall

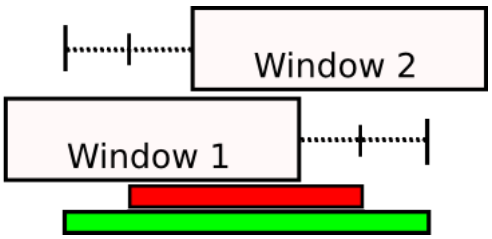


Abbildung 4.5: WindowSequence der Länge 2.

kann frühestens dort beginnen, wo ein Read beginnen würde, der exakt zu einer q-Gramm-Länge im Window 2 übersteht (Analog für das Ende des Intervalls). Der grüne Bereich stellt das größtmögliche Intervall dar, dass dem Aligner übergeben werden kann. Der rote Bereich ist das kleinstmögliche Intervall. Er ist durch die Mittelpunkte zwischen möglichen äußersten Reads und den Windows begrenzt. Der Start- und Endpunkt des Intervalls wird mittels der Heuristik 4.3 bestimmt.

$$\text{beginPos} = \text{window2BeginPos} - \text{extendNumber} * \#\text{hits1} * \text{extendMult} \tag{4.3}$$

$$\text{endPos} = \text{window1EndPos} - \text{extendNumber} * \#\text{hits2} * \text{extendMult} \tag{4.4}$$

Bei vielen Hits wird also das Intervall verkleinert.
Für die übrigen Fälle mit WindowSequences, die länger als 2 sind, wird das Intervall nicht gesondert behandelt, sondern einfach die der Anfang des ersten Windows und das Ende des letzten Windows verwendet.

4.1.10 Sim-Hashing

Bisher wurde Min-Hashing als Approximation an die Jaccard-Ähnlichkeit und eine Form von *Locality-Sensitive Hashing* erklärt und verwendet. Ebenfalls in der Praxis interessant ist ein anderes LSH-Verfahren namens *Simhash*Charikar (2002), das die Kosinus-Ähnlichkeit approximiert. Die Kosinus-Ähnlichkeit ist ein häufig verwendetes Ähnlichkeitsmaß für hochdimensionale Vektorräume. Stellt man das Skalarprodukt zweier Vektoren um, erhält man eine Rechenvorschrift für $cos(\theta)$:

$$a \cdot b = \|a\|\|b\| cos\theta \tag{4.5}$$

$$cos\theta = \frac{a \cdot b}{\|a\|\|b\|} \tag{4.6}$$

Reads und Abschnitte im Referenzgenom lassen sich als Vektor repräsentieren, wenn man für jedes q-Gramm eine eigene Dimension verwendet. Eine Vektorkomponente enthält als Wert die Anzahl der Vorkommnisse des entsprechenden q-Gramms, für das die Dimension steht. Die Kosinus-Ähnlichkeit kann dann als Ähnlichkeitsmaß verwendet werden.
SimHash ist ein effizientes Verfahren, dass die Kosinus-Ähnlichkeit approximiert. Es weist einem Dokument einen Hashwert s zu, wobei dessen Länge l vorher beliebig gewählt werden kann. Zur Erzeugung des Hashwerts wird ein Vektor $\vec{w}$*(weight)* benötigt, der so viele Dimensionen hat wie der Hashwert Bits. Für alle q-Gramme i des Dokuments werden Hashwerte h_i der Länge l mit einer normalen kryptographischen Hashfunktion berechnet. Hiermit wird der weight-Vektor nach der folgenden Vorschrift manipuliert:

$$\vec{w_i} = \begin{cases} \vec{w_i} + 1 & \text{falls } h_i > 0 \\ \vec{w_i} - 1 & \text{sonst} \end{cases} \tag{4.7}$$

Falls q-Gramme mehrfach im Dokument vorkommen, dann verändern sie auch w mehrfach. Nachdem alle q-Gramme abgearbeitet wurden, wird der letztendliche Hashwert s durch die Vorzeichen der Vektorkomponenten von $\vec{w}$ erzeugt.

$$s_i = \begin{cases} 1 & \text{falls } w_i > 0 \\ 0 & \text{sonst} \end{cases} \tag{4.8}$$

Dabei stellt s also eine Art gerundete Version von $\vec{w}$ dar. Bei hinreichend vielen q-Grammen kann s als gute Repräsentation des Dokuments verwendet werden. Im Gegensatz zum Min-Hashing werden ähnliche Dokumente aber nicht zwangsweise auf den exakt gleichen Wert abgebildet. Stattdessen können sie auch eine geringe Anzahl von unterschiedlichen Bits aufweisen. In der Praxis sucht man deswegen alle Dokumente, deren SimHash-Werte eine Hamming-Distanz von höchstens m zum SimHash-Wert des Anfragedokuments aufweisen. Ein häufig verwendeter Wert für m ist 3.
Um eine Suche nach SimHash-Werten mit geringer Hamming-Distanz effizient implementieren zu können, werden sortierte Listen aller SimHashes verwaltet. Die erste Liste enthält alle SimHash-Werte in ihrer Ursprungsform in sortierter Reihenfolge. Durch die Sortierung stehen Hashwerte mit ähnlichen höherwertigen Bits näher zueinander benachbart. Da die Sortierung diesen Effekt aber nur auf die höherwertigen Bits hat, werden zusätzlich weitere Listen erzeugt, in denen alle Hashwerte in zyklisch geshifteten Versionen stehen. Für einen 32-Bit-Hashwert könnten so also auch 32 verschiedene Versionen erzeugt werden. In jeder Liste können nun binäre Suchen ausgeführt werden. Die Hamming-Distanz des Elements, an dem die Suche beendet wurde, wird anschließend berechnet. Falls sie ausreichend gering ist, wird das dazugehörige Dokument als Kandidat markiert und anschließend wird die Liste aufwärts und abwärts weiter durchsucht, solange bis die Hammingdistanz der Hashwerte größer als m wird. Manku et al. (2007) zeigen ein auf dem Huffman-Code basierendes Verfahren, das die Listen komprimiert, um den Speicherplatz zu reduzieren. In Luo et al. (2013) wurde gezeigt, dass sich die Berechnung von SimHash auf FPGAs implementieren lässt und somit in Hardware um einen Faktor 362 schneller berechnen lässt. Die Dokumente werden in dieser Hardware byteweise gestreamt und daraus ein 64-Bit-Hashwert berechnet.
Im Readmapping-Kontext könnte sich SimHash vor allem als vorteilhaft für lange

Reads aufweisen. Das liegt daran, dass die SimHash-Werte bei größeren Dokumenten - also einer größeren Menge von q-Grammen - aussagekräftiger sind. Für lange Reads könnten die Referenzgenom-Windows auch möglichst groß gewählt werden und mittels SimHash dann Kandidatenpaare erzeugt werden.

4.2 Alignment von Reads

Der bisher beschriebene Algorithmus auf Basis des L*ocality-Sensitive Hashings* kann einen Read auf einen oder mehrere Abschnitte des Referenzgenoms abbilden. Allerdings hat dieses Verfahren aufgrund seiner Funktionsweise zwei Nachteile. Zum einen garantiert das Verfahren nicht, dass ein Read tatsächlich innerhalb einer vorgegebenen Fehlertoleranz zu den Referenzabschnitten passt, die ihm zugeordnet wurden. Zum anderen erfolgt die Zuordnung nur grob auf der Ebene von Referenzabschnitten, ein genaues Alignment findet jedoch nicht statt. Um ein solches Alignment zu berechnen, verwenden wir daher einen zweiten Algorithmus, der für jeden Read ein bestmögliches Alignment innerhalb seines Referenzabschnittes sucht. Dieser Algorithmus basiert auf dynamischer Programmierung und orientiert sich am Algorithmus für approximative Mustersuche von Rahmann et al. (2013), welcher wiederum auf dem Algorithmus aufbaut, der in der Publikation von Ukkonen (1985) beschrieben wird. Die Laufzeit des Algorithmus wächst linear mit der Länge des Referenzabschnittes, der untersucht wird. Somit ist die Vorverarbeitung durch das *Locality-Sensitive Hashing* essentiell, um praktikable Laufzeiten zu erreichen.

4.2.1 Globales und semiglobales Alignment

Ein Alignment ist informell eine zeichenweise Zuordnung eines Musters p (engl.: *pattern*) zu einem Text t. Dazu können sowohl in p als auch in t Lücken eingefügt werden, sodass ein Zeichen von p auch einer Lücke in t zugeordnet werden kann und umgekehrt. Einzig der Fall, dass zwei Lücken sich gegenseitig zugeordnet werden, ist nicht zulässig, weil solche Zuordnungen immer weggelassen werden können. Außerdem ist dadurch die Länge eines Alignments durch die kumulierte Länge von p und t begrenzt.

4.2.1 Beispiel. Zwei Alignments zwischen dem Text $t =$ `BARBARA` und dem Muster $p =$ `RABABA`. Die horizontalen Striche stehen für eingefügte Lücken.

```
R A B A - B A - -          R A B A B A - - - - - - -
- - B A R B A R A          - - - - - - B A R B A R A
```

Es folgt eine formale Definition von Alignments, für die zunächst der Begriff *String-Homomorphismus* benötigt wird.

4.2.2 Definition. Ein *String-Homomorphismus* π ist eine Abbildung $\pi : \Sigma_1 \to \Sigma_2$, die verträglich mit der Stringkonkatenation ist. Für alle Strings $s = s_1 \cdots s_l \in \Sigma_1^*$ gilt:

$$\pi(s_1 \cdots s_l) = \pi(s_1) \cdot \pi(s_2) \cdots \pi(s_l)$$

4.2.3 Definition (vgl. Rahmann et al. (2013), Definition 4.7). Seien $p, t \in \Sigma^*$ zwei Strings und sei ϵ das leere Wort. Ein Alignment $A = A_1 \cdots A_k$ zwischen p und t ist ein String über dem Alphabet $(\Sigma \cup \{-\})^2 \setminus \{(-,-)\}$ mit $\pi_1(A) = p$ und $\pi_2(A) = t$. Dabei ist π_1 ein String-Homomorphismus mit $\pi_1(A_i) = \pi_1((a,b)) := a$ und $\pi_1((-,b)) := \epsilon$ für alle $a, b \in \Sigma$ und für alle $i \in \{1, \ldots, k\}$. Analog ist π_2 für die zweite Komponente definiert.

Ein *String-Homomorphismus*

Ein solches Alignment wird auch als *globales* Alignment bezeichnet, da das gesamte Muster und der gesamte Text zueinander aligniert werden. Es fällt auf, dass es vier verschiedene Arten von Zeichen A_i in einem Alignment gibt:

1. Beide Komponenten sind gleich (entspricht einem *match* zwischen zwei Zeichen). Es gilt $A_i = (a, b)$ mit $a \in p, b \in t$ und $a = b$.
2. Beide Komponenten sind unterschiedlich und keine Lücken (entspricht einem *mismatch* zwischen zwei Zeichen). Es gilt $A_i = (a, b)$ mit $a \in p, b \in t$ und $a \neq b$.
3. Die erste Komponente ist ein Zeichen aus p, die andere eine Lücke (entspricht einer *Insertion*). Es gilt $A_i = (a, -)$ mit $a \in p$.
4. Die erste Komponente ist eine Lücke, die andere ein Zeichen aus t (entspricht einer *Deletion*) Es gilt $A_i = (-, b)$ mit $b \in t$.

Für Ähnlichkeitsbestimmungen sind Alignments interessant, bei denen möglichst viele Teile von p und t zusammenpassen. Der erste Ansatz ist die Minimierung der Fehlerzahl. Die Anzahl der *Fehler* entspricht dabei der Anzahl der Vorkommen von Zeichen des zweiten, dritten und vierten Falls. Diese Arten von Fehlern entsprechen gerade den üblichen Modifikationsoperationen der Edit-Distanz. Ein Alignment, welches die Anzahl der Fehler minimiert, gibt damit auch die Edit-Distanz zwischen p und t an.

Der zweite Ansatz ist die Verwendung von *scoring*, wobei eine hohe Punktzahl für eine hohe Ähnlichkeit zwischen zwei Strings steht. Jeder der vier Fälle bekommt eine bestimmte Punktzahl zugewiesen, die anschließend maximiert wird. Der erste Fall erhält dabei üblicherweise eine nicht-negative Punktzahl, weil Übereinstimmungen zwischen den Strings für eine Ähnlichkeit stehen, während die anderen Fälle entsprechend eine nicht-positive Punktzahl erhalten. Durch die feinere Gewichtung ist dieser Ansatz flexibler als der Erste. In unserer aktuellen Implementierung beschränken wir uns der Einfachheit halber dennoch auf den ersten Ansatz.

Die Suche eines optimalen globalen Alignments lässt sich mit Hilfe von dynamischer Programmierung lösen. Ursprünglich stammt der Ansatz zur Berechnung globaler Alignments von Needleman und Wunsch (1970). Die dort benutzte Rekursionsgleichung führt allerdings zu einer kubischen Laufzeit bezüglich Text- und Musterlänge, weshalb hier eine eingeschränkte Variante beschreiben wird, die aus einem Skript von Rahmann et al. (2013) entnommen wurde und mit quadratischer Laufzeit auskommt. Für ein Muster $p = p_1 \ldots p_m$ und einen Text $t = t_1 \ldots t_n$ wird eine Matrix F berechnet, wobei $F[i, j]$ die minimale Fehlerzahl eines Alignments zwischen $p_1 \ldots p_i$

und einem Suffix von $t_1 \dots t_j$ enthält. Die Matrix F besitzt $m+1$ Zeilen und $n+1$ Spalten, deren Indizierung bei 0 beginnt. Die erste Zeile und die erste Spalte werden wie folgt initialisiert:

$$\begin{aligned} F[i,0] &:= i \qquad \forall\, 0 \leq i \leq m \\ F[0,j] &:= j \qquad \forall\, 1 \leq j \leq n \end{aligned}$$

Wird ein String der Länge 0 (also der leere String) mit einem String der Länge i aligniert, so ergeben sich zwangsläufig i Insertionen bzw. Deletionen, sodass die Initialisierung korrekt ist. Die Rekursionsgleichung für die dynamische Programmierung besteht aus drei verschiedenen Fällen.

$$F[i,j] = \min \begin{cases} F[i-1,j-1] & + [\![p_i \neq t_j]\!] \\ F[i-1,j] & +1 \\ F[i,j-1] & +1 \end{cases}$$

Der obere Fall (diagonale Rekursionsrichtung) entspricht dabei einem *match* bzw. einem *mismatch*. Der mittlere Fall (vertikale Rekursionsrichtung) entspricht einer Insertion, d.h. das Muster enthält ein Zeichen, welches im Text nicht vorkommt. Analog dazu entspricht der dritte Fall (horizontale Rekursionsrichtung) einer Deletion, bei der das Alignment musterseitig eine Lücke enthält. Das Feld $F[m,n]$ enthält die Edit-Distanz zwischen p und t.

Um Reads und Referenzabschnitte zu alignieren ergibt sich jedoch noch ein Problem: Die vom LSH-Algorithmus zusammengeführten Paare von Reads und Referenzabschnitten passen nur grob zueinander. Der Anfang bzw. das Ende des Reads p muss also nicht notwendigerweise zum Anfang bzw. Ende des Referenzabschnitts t gehören. Folgendes Beispiel zeigt die Problematik:

4.2.4 Beispiel. Sei $p :=$ `ACGAT` und $t :=$ `CGACGATTA` und das optimale Alignment:

```
C G A C G A T T A
- - A C G A T - -
```

In diesem Fall ist p ein Substring von t, dennoch würde obiger Algorithmus hier insgesamt vier Fehler zählen. Da die Position eines Reads im Referenzgenom beliebig sein kann, würde uns bereits ein optimales Alignment zwischen p und einem Substring von t genügen. Ein solches Alignment wird auch als *semiglobales* Alignment von p an t bezeichnet (Rahmann et al., 2013; Ukkonen, 1985).

Der bestehende Algorithmus lässt sich leicht auf semiglobale Alignments erweitern. Dazu sind zwei Änderungen nötig:

1. Die erste Zeile der Matrix F wird mit Nullen initialisiert. Dadurch kann das eigentliche Alignment von p an jeder Stelle in t beginnen, ohne dass dies als Fehler gewertet wird.

2. Die minimale Fehlerzahl steht nicht mehr im Feld $F[m,n]$, sondern kann in einem beliebigen Feld der letzten Zeile stehen. Das Alignment von p darf also an beliebiger Stelle in t enden, ohne dass dies vom Algorithmus bestraft wird.

Abbildung 4.6 visualisiert die Berechnung der Edit-Distanz-Matrix F eines semiglobalen Alignments anhand des in Beispiel 4.2.4 gewählten Musters und Textes.

			t								
	F	-	**C**	**G**	**A**	**A**	**C**	**T**	**G**	**T**	**C**
	-	0	0	0	0	0	0	0	0	0	0
	A	1	1	1	0	0	1	1	1	1	1
	A	2	2	2	1	0	1	2	2	2	2
	C	3	2	3	2	1	0	1	2	3	2
p	**C**	4	3	3	3	2	1	1	2	3	3
	T	5	4	4	4	3	2	1	2	2	3
	T	6	5	5	5	4	3	2	2	2	3
	T	7	6	6	6	5	4	3	3	2	3

Abbildung 4.6: Beispielberechnung für semiglobales Alignment mit $p :=$ `AACCTTT` und $t :=$ `CGAACTGTC`. Die grünen Pfeile visualiseren, auf welche Werte bei der Berechnung eines Eintrags zugegriffen wird.

Anhand der Matrix F lässt sich mit Hilfe von Backtracing das tatsächliche Alignment von p zu t ableiten. Wie dies genau funktioniert, wird in Abschnitt 4.2.3 beschrieben.

4.2.2 Erweiterung auf Varianten

Wir haben den beschriebenen Algorithmus für semiglobale Alignments erweitert, sodass dieser auch variantentolerant arbeitet. Der Aligner verarbeitet SNPs, Insertionen, Deletionen und Substitutionen, wobei auf jede dieser Variantenarten noch explizit eingegangen wird. Variantentoleranz bedeutet in diesem Zusammenhang, dass der Algorithmus zusätzlich zum Referenzgenom auch alle Kombinationen der zusätzlich angegebenen Varianten berücksichtigt. Der Algorithmus berechnet also nicht nur ein optimales Alignment zwischen Referenzabschnitt und Read, sondern wählt zusätzlich alle Varianten aus, die das Alignment verbessern, falls sie auf den Referenzabschnitt angewandt werden.

Behandlung von SNPs

SNPs werden getrennt von Indels betrachtet, da sie deutlich einfacher zu verarbeiten sind. Die Idee hierbei ist, dass wir das Referenzgenom und alle Reads in der IUPAC-Kodierung (vgl. Abschnitt 2.4.1) verarbeiten. Diese verbraucht mit vier Bits pro Basenpaar doppelt so viel Speicher wie die einfache 2-Bit-Kodierung, bietet dafür jedoch die Möglichkeit SNPs zu behandeln ohne die Laufzeit des Algorithmus zu erhöhen.

Die Bitkodierung des IUPAC-Alphabets ist so gewählt, dass je eines der vier Bits für eine der vier DNA-Basen steht. SNPs können mit Hilfe dieser Kodierung direkt

in den betrachteten Referenzabschnitt eingepflegt werden, indem für jede Position und jede Base, die dort stehen können, das entsprechende Bit auf 1 gesetzt wird. Dadurch ändert sich die Semantik eines einzelnen Zeichens: Jedes Zeichen des Referenzabschnitts steht nicht mehr für eine konkrete Base, sondern für eine Menge von Basen, die an der entsprechenden Position stehen können. Die Rekursionsgleichung des Aligners muss also ebenfalls geändert werden, um der Auffassung des IUPAC-Zeichens als eine Menge von Basen zu entsprechen. Sei T_j die Menge aller Basen, die durch das IUPAC-Zeichen t_j repräsentiert werden, d.h. T_j enthält alle Basen, die an Position j im Referenzstring stehen können. Dann ist die neue Rekursionsgleichung wie folgt definiert:

$$F[i,j] = \min \begin{cases} F[i-1,j-1] & + [\![p_i \notin T_j]\!] \\ F[i-1,j] & +1 \\ F[i,j-1] & +1 \end{cases} \tag{4.9}$$

Beim ersten Fall wird die Edit-Distanz nur dann erhöht, wenn das Zeichen p_i des Reads mit **keinem** Zeichen aus T_j übereinstimmt. Für eine effiziente Berechnung des ersten Falles muss der zu verarbeitende Read ebenfalls in der IUPAC-Kodierung vorliegen, auch wenn jedes Zeichen dadurch immer genau ein gesetztes Bit enthält. Eine bitweise „und“-Operation zwischen Referenz- und Readzeichen liefert genau dann den Wert 0, falls die Readbase mit keiner der möglichen Referenzbasen übereinstimmt. Auf diese Weise können SNPs im Aligner ohne Geschwindigkeitsverlust berücksichtigt werden. Die Verarbeitung der VCF-Datei ist ein Vorverarbeitungsschritt und somit unabhängig von der Anzahl der zu alignierenden Reads.

Behandlung von Deletionen

Im Allgemeinen könnte man sich für Indelvarianten folgendes naives Vorgehen überlegen: Für jede Menge v von Varianten, die auf einen Text t anwendbar ist, berechnet der Aligner separat eine Edit-Distanz-Matrix F_v und wählt anschließend über alle Matrizen diejenige mit der kleinsten Edit-Distanz für p aus. Auf diese Weise wären zwar alle Varianten abgedeckt, allerdings führt dies bei vielen dicht zusammenliegenden Varianten zu einer kombinatorischen Explosion der auszuwählenden Teilmengen v. Im Folgenden wird daher ein Ansatz beschrieben, der Varianten lokal behandelt, ohne dafür jeweils eine komplette Edit-Distanz-Matrix neu zu berechnen.

Der einfachste Fall ist dabei die Deletionsvariante. Diese gibt an, dass an einer bestimmten Position im Referenzgenom genau k Zeichen entfernt werden für $k > 0$. Wir nehmen im Folgenden an, dass eine Deletionsvariante vorliegt, die die Zeichen $t_{j-k}, \ldots, t_{j-1}$ aus dem Referenzgenom entfernt. Das bedeutet, dass die Deletionsvariante an der Position $j-k$ beginnt und eine Länge von k besitzt, da sie genau k Zeichen entfernt.

Es wird zunächst beschrieben, wie der naive Ansatz die Deletionsvariante behandeln würde und anschließend ein effizientes Verfahren abgeleitet, welches jedoch zum gleichen Endergebnis führt. Die naive Vorgehensweise würde die Matrix F für den Originaltext t und eine weitere Matrix $\tilde{F}$ für den Text $t_1 \ldots t_{j-k-1} t_j \ldots t_n$ berechnen. Dabei fällt auf, dass beide Matrizen bis zur $j-k-1$-ten Spalte identisch wären, weil sich die Texte bis dorthin nicht unterscheiden. Die ersten $j-k-1$ Spalten müssen

für $\tilde{F}$ also gar nicht explizit berechnet werden, sondern es genügt auf die Matrix F des Originaltexts zuzugreifen.

Da der Teilstring $t_{j-k} \dots t_{j-1}$ bei der Deletion wegfällt, ist die erste tatsächlich neu zu berechnende Spalte für $\tilde{F}$ die für das Zeichen t_j. Diese lässt sich durch die Spalte für t_{j-k-1} und dem Zeichen t_j aus der Rekursionsgleichung berechnen. F und $\tilde{F}$ enthalten also für das Zeichen t_j unterschiedliche Spalten, sodass die restlichen Spalten für beide Matrizes separat berechnet werden müssten. Die Schwierigkeit besteht nun darin, das Berechnen von separaten Matrizen zu verhindern, ohne dabei eine optimale Lösung zu verlieren.

Die Werte der j-ten Spalte von F sind die Edit-Distanzen ohne Verwendung der Deletionsvariante, die Werte von $\tilde{F}$ jene mit Verwendung der Variante. Da es für die Verwendung von Varianten keine zusätzlichen Bedingungen gibt (wie z. B. eine maximale Anzahl von verwendeten Varianten), ist es aber eigentlich unnötig zwischen Verwendung und Nichtverwendung zu unterscheiden. Bildet man über die beiden Spalten komponentenweise das Minimum, so erhält man jeweils Edit-Distanzen zwischen einem Präfix von p und t, die sowohl durch Verwendung als auch Nichtverwendung der Variante zustande kommen können — je nachdem, was zu einer besseren Lösung führt. Man kann also sagen: Das Minimum von $F[i, j]$ und $\tilde{F}[i, j]$ für ein festes $i \in \{0, \dots, m\}$ ist die Edit-Distanz zwischen $p_1 \dots p_i$ und einem Suffix von $t_1 \dots t_j$ mit optionaler Verwendung der Deletionsvariante. Ob die Variante verwendet wurde oder nicht, ist für die Berechnung der nächsten Spalte unerheblich.

Zusammengefasst ist also das naive Vorgehen mit der Berechnung zweier Edit-Distanz-Matrizen nicht nötig. Für die Behandlung der Deletionsvariante reichen folgende Schritte aus:

1. Finde die Spalte, die sich unmittelbar vor der Deletion befindet, also Spalte $j - k$.

2. Berechne ausgehend vom Zeichen nach der Deletion (also von t_j) und Spalte $j - k$ eine neue virtuelle Spalte gemäß der Rekursionsgleichung 4.9.

3. Bilde das komponentenweise Minimum zwischen der virtuellen Spalte und Spalte j, welche auf dem Originaltext berechnet wurde.

Das Vorgehen lässt sich auch auf mehrere Deletionsvarianten, die alle vor t_j enden, verallgemeinern. Für jede Deletionsvariante muss gemäß der obigen Schritte eine zusätzliche virtuelle Spalte berechnet und anschließend das komponentenweise Minimum aller Spalten für das Endergebnis gebildet werden.

In unserer Implementierung werden die virtuellen Spalten nicht explizit berechnet, sondern die zusätzlichen Werte, die sich für jedes Feld $F[i, j]$ ergeben, werden mit in die Rekursionsgleichung eingebunden. Sei $V := \{v_1, \dots, v_r\}$ eine Menge von De-

letionsvarianten, die genau vor t_j enden und sei $K := \{k_1, \ldots, k_r\}$ die Menge der entsprechenden Längen der Varianten. Dann gilt für die Felder der Spalte j:

$$F[i,j] = \min \begin{cases} F[i-1,j-1] & +[\![p_i \notin T_j]\!] \\ F[i-1,j] & +1 \\ F[i,j-1] & +1 \\ F[i-1,j-k_1-1] & +[\![p_i \notin T_j]\!] \\ F[i,j-k_1-1] & +1 \\ \ldots & \\ F[i-1,j-k_r-1] & +[\![p_i \notin T_j]\!] \\ F[i,j-k_r-1] & +1 \end{cases} \tag{4.10}$$

Die oberen drei Fälle sind identisch zur bisher bekannten Rekursionsgleichung 4.9. Für jede Deletionsvariante kommen zwei weitere Fälle hinzu, die sich auf die Spalte vor der Deletion beziehen. Man könnte also sagen, dass das Zeichen t_j nicht nur den Vorgänger t_{j-1} besitzt, sondern für jede Deletionsvariante v_l einen weiteren Vorgänger t_{j-k_l-1}. In den folgenden Abschnitten wird der Begriff des *Vorgängers* noch öfter auf diese Weise benutzt werden. Um die Vorstellung mehrerer Vorgänger zu illustrieren ist die Verwendung der erweiterten Rekursionsgleichung 4.10 in Abbildung 4.7 visuell veranschaulicht.

	-	**A**	**C**	**G**	**T**	**G**	**A**	**T**
-	0	0	0	0	0	0	0	0
A	1	0	1	1	1	1	0	1
C	2	1	0	1	2	2	1	1
G	3	2	1	0	1	2	1	2
A	4	3	2	1	1	2	0	1
T	5	4	3	2	2	2	1	0

Abbildung 4.7: Beispielmatrix für eine Deletionsvariante, die t_4 und t_5 entfernt (blaue Umrandung). Die grünen Pfeile deuten die Rekursionsfälle an, die sich durch die Deletionsvariante ergeben: Der Wert 0 kommt zustande durch „Überspringen“ der beiden vorigen Spalten.

Behandlung von Insertionen

Für Insertionen lassen sich ähnliche Überlegungen wie für Deletionen anstellen. Das Ziel ist auch hier, dass Insertionen lokal und unabhängig voneinander behandelt werden. Die naive Vorgehensweise wäre wiederum, dass für eine Insertionsvariante an der Stelle $j-1$ zwei Edit-Distanz-Matrizen F und $\tilde{F}$ berechnet werden, wobei für $\tilde{F}$ zwischen t_{j-1} und t_j der Insertionsstring $s = s_1 \ldots s_l$ eingefügt wird. Die Matrix $\tilde{F}$ soll wie zuvor nicht tatsächlich berechnet werden, sondern dient der Anschauung, wie unsere Implementierung funktioniert.

Analog zu den Deletionen hängt die Spalte für das Zeichen t_j direkt davon ab, ob man F oder $\tilde{F}$ betrachtet. Mit dem gleichen Argument für optimale Teillösungen kön-

nen auch hier beide Versionen dieser Spalte über ein komponentenweises Minimum vereinigt werden, sodass die Berechnung nur mit einer Lösung fortgesetzt wird.

Der Unterschied zu den Deletionen liegt darin, dass die Matrix $\tilde{F}$ bis zum Vorgänger von t_j noch nicht bekannt ist. Dies wäre die Spalte für das Zeichen s_l, welches jedoch nicht zum Originaltext t gehört. Der Aligner muss also die zusätzlichen Spalten, die durch s hinzukommen, berechnen und die Spalte zu s_l in die Berechnung für die Spalte zu t_j einfließen lassen. Dazu könnte der Aligner die Spalte zu t_{j-1} kopieren, anschließend die Alignierung gemäß der Rekursionsformel über den String s fortsetzen und sich schließlich die letzte resultierende Spalte als Vorgänger von t_j merken.

Neben dem Aufwand für das Alignieren von s müssen für t_j bei dessen Rekursionsformel zwei weitere Fälle berücksichtigt werden - so wie es bei den Deletionen der Fall war. Im Falle von mehreren Insertionen an der gleichen Position muss das beschriebene Vorgehen einmal pro Insertionsstring angewandt werden. Dadurch kommt pro Insertion ein Vorgänger für t_j hinzu, was zwei zusätzlichen Fällen in der Rekursionsformel entspricht. Das Vorgehen für Insertionen wird in Abbildung 4.8 veranschaulicht.

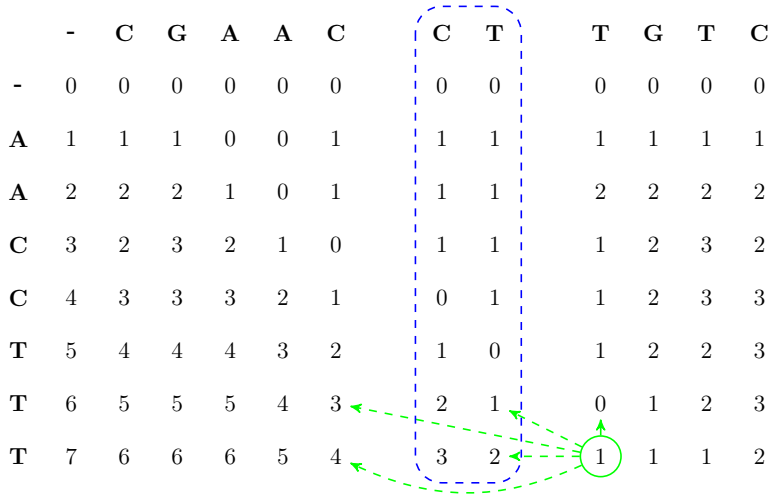


Abbildung 4.8: Erweiterung der Beispielberechnung aus Abbildung 4.6 mit einer zusätzlichen Insertionsvariante zwischen t_5 und t_6 mit $s = \texttt{CT}$. Die spekulative Berechnung für die Insertion ist blau umrandet. Die grünen Pfeile deuten die Rekursionsfälle an, die sich durch eine Insertionsvariante ergeben.

Es ist zu beachten, dass bei diesem Verfahren nicht mehrere Insertionsstrings an der gleichen Position konkateniert werden können. Gibt es an der Stelle t_j zwei Insertionen mit Insertionsstrings s und s', werden nur die variierten Strings $t_1 \dots t_{j-1} s t_j \dots t_n$ und $t_1 \dots t_{j-1} s' t_j \dots t_n$, nicht jedoch Kombinationen wie $t_1 \dots t_{j-1} s s' t_j \dots t_n$ berücksichtigt. Der Grund dafür ist, dass die Anzahl solcher Kombinationen exponentiell mit der Anzahl von Indels an der gleichen Position wächst. Um solche Kombinationen dennoch im Aligner zu verarbeiten, müssten diese vorher als separate Varianten erzeugt und dem Aligner als Eingabe übergeben werden.

Behandlung von Substitutionen

Bei Substitutionsvarianten wird ein Teilstring des Referenzabschnitts durch einen spezifizierten Insertionsstring beliebiger Länge ersetzt. Diese Art von Varianten lässt sich mit den Methoden für Insertionen und Deletionen behandeln, indem man eine Substitution als eine Deletion mit anschließender Insertion auffasst. Eine Substitution an der Stelle $j-k$, die die nächsten k Zeichen durch einen Insertionsstring $s = s_1 \dots s_l$ ersetzt, würde analog zu den Deletionen erst behandelt werden, sobald der Aligner das j-te Zeichen des Referenzstrings erreicht hat. Statt jedoch die Spalte des $j-k$-ten Zeichens als Vorgänger der j-ten Spalte zu definieren (wie das bei der Deletion gemacht wurde), aligniert er den Insertionsstring s und benutzt dabei die $j-k$-te Spalte als Vorgängerspalte für s_1. Schließlich wird wie bei den Insertionen das letzte Zeichen s_l von s zu einem Vorgänger für das aktuelle Originaltextzeichen t_j. Mit diesem Vorgehen können auch Substitutionen lokal und unabhängig voneinander behandelt werden.

Indizierung von Varianten

Bei der bisherigen Variantenbehandlung wurde die Information, an welcher Stelle welche Varianten vorkommen einfach vorausgesetzt. In der Praxis befinden sich die Informationen über bekannte Varianten in einer separaten VCF-Datei (siehe Unterabschnitt 2.4.4). Da sich die Menge der Varianten während der Alignierung mehrerer Reads nicht ändert, ist es sinnvoll die benötigten Informationen in einem Vorverarbeitungsschritt aus der VCF-Datei zu extrahieren und in einer geeigneten Datenstruktur zu speichern.

Es sprechen im Wesentlichen zwei Gründe für die Verwendung eines solchen Indexes:

1. Für die Behandlung von Insertionen benötigt der Aligner die Startposition selbiger, während er für Deletionen deren Endposition braucht. Letztere kann bei langen Deletionen sehr aufwändig zu bestimmen sein, da die VCF-Datei nur die Startposition der angegebenen Varianten enthält. Mit einem sequenziellen Scan können in linearer Zeit alle relevanten Referenzpositionen und die dort befindlichen Varianten erfasst werden. Zudem kann dieser Scan *offline*, also unabhängig vom Verarbeiten der Reads geschehen.

2. Die VCF-Spezifikation erzwingt keine eindeutige Indizierung der Varianten. Durch einen Index bekommen alle Varianten eine eindeutige Zuordnung.

Zur Behandlung von SNPs wurde ein IUPAC-kodierter Referenzabschnitt vorausgesetzt. Damit dieser nicht für jeden Alignierungsprozess neu berechnet werden muss, wird das gesamte Referenzgenom in die IUPAC-Kodierung überführt und die Informationen aller SNPs dort eingefügt. Durch eine logische, bitweise Oder-Operation können zwei IUPAC-Zeichen kombiniert werden. Da sich IUPAC-Zeichen als eine Menge von möglichen Basen auffassen lassen, entspricht das logische „oder“ hier einer Vereinigung zweier Basenmengen.

Der Aligner behandelt Insertionen vor Auftreten ihres Insertionsstrings im Genom, während Deletionen und Substitutionen am Ende des durch sie gelöscht bzw. ersetztes Substrings behandelt werden. Bei der Verarbeitung einer neuen Referenzposition,

benötigt der Aligner also die Information, ob an dieser Position eine Insertion beginnt oder eine Deletion bzw. Substitution endet. Zur Speicherung dieser Varianten haben wir uns für ein sortiertes Array entschieden, welches alle Varianten aufsteigend nach Position im Referenzgenom enthält. Dieses bildet den *Variantenindex*.

Da für die Abfrage, ob an einer bestimmten Position j eine Variante vorliegt, eine binäre Suche notwendig wäre, geht der Index anders vor. Er sucht zu Beginn des Alignierungsprozesses mit binärer Suche die erste Variante, die im verarbeiteten Referenzabschnitt liegt und merkt sich ihre Position j_{var}. Fragt der Aligner eine kleinere Position $j < j_{var}$ an, so liegt dort garantiert keine Variante vor, weil j_{var} die Position der ersten Variante war. Bei einer Anfrage für Position $j = j_{var}$, gibt der Index die dort befindliche Variante zurück und setzt j_{var} auf die nächstgrößere Position, an der eine Variante vorkommt. Diese lässt sich in konstanter Zeit finden, da alle Varianten in einem sortierten Array vorliegen. Da auch mehrere Varianten an einer Position vorkommen können, fragt der Aligner im Falle eines Treffers solange die aktuelle Position j an, bis $j < j_{var}$ gilt,d. h. keine Variante mehr an der aktuellen Position vorkommt. Fragt der Aligner eine größere Position $j > j_{var}$ an, kann der Index das sortierte Array sequenziell ab der Position der ersten Variante scannen statt binär zu suchen. Das liegt daran, dass der Aligner die Positionen seines Referenzabschnitts linear durchgeht.

Für die Verarbeitung eines Reads müssen also im schlimmsten Fall eine binäre Suche über dem sortieren Array durchgeführt werden, sowie ein sequenzieller Scan über alle Varianten, die in dem Referenzabschnitt vorkommen, für den eine Alignierung berechnet werden muss. Um die binäre Suche zu umgehen, stand außerdem die Verwendung einer Rank-Datenstruktur zur Diskussion, die in konstanter Zeit für eine beliebige Position feststellen kann, welche Varianten dort vorkommen. Allerdings war der Laufzeitvorteil in der Praxis nicht messbar, da der Variantenindex nur Indel-Varianten enthält und von diesen im Verhältnis zur Länge des Referenzgenoms nur wenige vorkommen. Da die Rank-Datenstruktur jedoch einiges mehr an Speicher[3] verbraucht, haben wir uns letztlich für das sortierte Array entschieden.

Da alle Varianten in der VCF-Datei nach Chromosomen gruppiert sind und die Position innerhalb des Chromosoms angegeben ist, gibt es ein Array pro Chromosom.

4.2.3 Backtracing und Ausgabe des Algorithmus

In den bisherigen Abschnitten ging es nur um die Berechnung der kleinsten Edit-Distanz zwischen einem Muster p und einem Substring des Textes t. Da ein Readmapper jedoch Alignierungen von Reads berechnet, ist noch ein Schritt erforderlich, um aus der Edit-Distanz-Matrix das genaue Alignment zu bestimmen. Dieses soll in Form eines Cigar-Strings (vgl. Abschnitt 2.4.6) ausgegeben werden.

Für die Konstruktion des Cigar-Strings werden für jedes Feld der Edit-Distanz-Matrix zwei weitere Informationen gespeichert:

1. Die Vorgängerspalte, welche für die Berechnung der Edit-Distanz im aktuellen Feld ausschlaggebend war. Bei mehreren möglichen Vorgängerspalten wird eine beliebige davon ausgewählt, sodass dieser Wert eindeutig ist.

[3] beim Humangenom einige 100MB

2. Die Richtung der Rekursion, d.h. ob die Edit-Distanz im aktuellen Feld durch ein Match, ein Mismatch, eine Insertion oder eine Deletion zustande kam. Bei einer Deletion war die aktuelle Zeile der Vorgängerspalte der ausschlaggebende Wert, bei einem Match bzw. einem Mismatch oder einer Insertion war es die Zeile darüber. Bei mehreren Möglichkeiten muss hier auf Konsistenz mit dem ersten Punkt geachtet werden.

Die Rekonstruktion des Cigar-Strings beginnt beim Feld der letzten Zeile, welches die kleinste Edit-Distanz enthält. Durch die gespeicherte Rekursionsrichtung und Vorgängerspalte lässt sich für jedes Matrixfeld das für die dynamische Programmierung ausschlaggebende Vorgängerfeld finden. Auf diese Weise springt der Algorithmus solange auf das jeweilige Vorgängerfeld, bis er in der ersten Zeile der Matrix ankommt. Für jeden Sprung merkt sich der Algorithmus die benutzte Rekursionsrichtung, sodass er alle Alignierungsschritte in inverser Reihenfolge erhält. Diese Reihenfolge wird invertiert, die Alignierungsschritte durch die jeweiligen Buchstaben des Cigar-Strings ersetzt und der resultierende String durch Zusammenfassen gleicher aufeinanderfolgender Zeichen komprimiert. Das Ergebnis ist der Cigar-String der berechneten Edit-Distanz.

Algorithmus 4.2 zeigt dieses Vorgehen im Pseudocode an. Abbildung 4.9 visualisiert es zusätzlich anhand eines kleinen Beispiels.

Eingabe: Edit-Distanz-Matrix F
Ausgabe: Cigar-String C
1: $C \leftarrow \epsilon$
2: $row \leftarrow m$ `// Länge des Reads`
3: $col \leftarrow$ Spalte mit minimaler Edit-Distanz in Zeile m
4: **while** $row > 0$ **do**
5: $rec \leftarrow$ Rekursionsfall von $F[row, col]$
6: **if** rec ist *match* **then**
7: $C \leftarrow$ `'M'` $\cdot\, C$
8: $row \leftarrow row - 1$
9: **else if** rec ist *mismatch* **then**
10: $C \leftarrow$ `'X'` $\cdot\, C$
11: $row \leftarrow row - 1$
12: **else if** rec ist *insertion* **then**
13: $C \leftarrow$ `'I'` $\cdot\, C$
14: $row \leftarrow row - 1$
15: **else if** rec ist *deletion* **then**
16: $C \leftarrow$ `'D'` $\cdot\, C$
17: **end if**
18: $col \leftarrow$ Vorgängerspalte von Rekursion
19: **end while**
20: **return** C

Algorithmus 4.2: Konstruktion des Cigar-Strings

Theoretisch käme der Aligner auch ohne die Zusatzinformationen über verwendete Rekursionsrichtung und Vorgängerspalte aus. Für jedes Matrixfeld, welches beim

	-	C	G	A	A	C	T	G	T	C
-	0	0	0	0	0	0	0	0	0	0
A	1	1	1	0	0	1	1	1	1	1
A	2	2	2	1	0	1	2	2	2	2
C	3	2	3	2	1	0	1	2	3	2
C	4	3	3	3	2	1	1	2	3	3
T	5	4	4	4	3	2	1	2	2	3
T	6	5	5	5	4	3	2	2	2	3
T	7	6	6	6	5	4	3	2	2	3

Abbildung 4.9: Beispielberechnung für ein semiglobales Alignment mit $p :=$ `AACCTTT` und $t :=$ `CGAACTGTC`. Der markierte Pfad gibt das beste Alignment und den Beitrag jedes Sprungs am Cigar-String an. Dieser lautet hier `3M1I1M1X1M`, wobei das `X` für ein Mismatch steht.

Backtracing erreicht wird, könnte sich der Aligner alle Vorgängerfelder anschauen und sich eines aussuchen, aus dem die Edit-Distanz des aktuellen Feldes hervorgeht. Folgt er in jedem Schritt einem solchen Vorgängerfeld, ergäbe sich ebenfalls ein korrektes, optimales Alignment. Wir haben uns allerdings aus zwei Gründen für die Zusatzinformationen pro Feld entschieden, auch wenn diese mehr Speicher benötigen:

1. Die Laufzeit wäre ohne das Speichern der Informationen sehr wahrscheinlich höher, weil der Code sehr viele zusätzliche Verzweigungen enthalten würde.
2. Man bräuchte für jede Spalte der Matrix eine Liste von Vorgängerspalten aufgrund von Varianten. Diese müssten entweder bei jedem Alignment zusätzlich berechnet werden oder in einem zweiten Variantenindex gespeichert sein.

Unabhängig von unserer Entscheidung, muss die gesamte Matrix bis zum Ende der Berechnung vorgehalten werden, um eine Rekonstruktion des Cigar-Strings zu ermöglichen. Das gilt auch für die Spalten, die bei einem Insertionsstring zusätzlich erzeugt werden, da Alignments insbesondere innerhalb einer Insertion beginnen oder enden können. Bei unserer Implementierung werden daher sowohl die Spalten, die zum Referenzstring gehören, als auch die Spalten, die zu einem Insertionsstring gehören, in der gleichen Matrix gespeichert. Das bedeutet natürlich, dass das j-te Zeichen des Referenzstrings nicht zwingend zur j-ten Spalte gehört. Um die Spalten eindeutig zuordnen zu können, enthält jede Spalte einen Header mit zusätzlichen Informationen:

- Die in der Spalte vorherrschende Variante bzw. die Information, dass die Spalte zum Referenzstring gehört.
- Die Position im Referenzstring, zu der die Spalte gehört bzw. die Position im Referenzstring, an der die vorherrschende Variante beginnt.
- Die Position innerhalb der aktuellen Variante (nur relevant, falls die Spalte zu einer Variante gehört).

Zusätzlich gibt es ein separates Array, welches für jede Position im Referenzstring den zugehörigen Spaltenindex speichert.

Bei der Berechnung des semiglobalen Alignments ohne Varianten besteht die Ausgabe aus einem Cigar-String und seiner Startposition im Referenzgenom. Bei der variantentoleranten Version ergibt sich jedoch das Problem, dass der Cigar-String keine Informationen darüber enthält, welche Varianten für das optimale Alignment genutzt wurden. Für eine spätere Visualisierung der Ergebnisse ist das unpraktisch, da sich der modifizierte Referenzstring, an den der Read tatsächlich aligniert wurde, nur schwer rekonstruieren lässt. Noch schwieriger wird es, wenn ein Alignment innerhalb eines Insertionsstrings beginnt, weil völlig unklar ist, welcher Wert als Startposition für das Alignment angegeben werden soll. Wir haben uns daher entschieden, dass der Algorithmus für jedes Alignment die folgenden zusätzlichen Informationen ausgibt:

- Eine Liste aller für das Alignment benutzter Varianten. Dazu muss die VCF-Datei gegebenenfalls so umgeschrieben werden, dass alle Varianten eindeutig indiziert sind.
- Falls zutreffend, den Index der Variante, in der das Alignment beginnt.
- Beginnt ein Alignment in einer Insertion bzw. Substitution, wird die Position vor der Insertion als Position im Referenzgenom ausgegeben. Es wird eine zweite Startposition gespeichert, die den Beginn des Alignments innerhalb des Insertionsstrings angibt.

Da das SAM-Format für diese Informationen keine dedizierten Felder hat, nutzen wir dafür das Zusatzfeld, welches im SAM-Format vorgesehen ist. Dadurch bleibt die Datei für andere Programme weiterhin lesbar und wir können alle nötigen Informationen aus unserer SAM-Datei rekonstruieren.

4.2.4 Laufzeitoptimierung

Mit Hilfe des beschriebenen Algorithmus lassen sich theoretisch optimale Alignments mit einer beliebigen Anzahl von Fehlern berechnen. In der Praxis ist man jedoch häufig nur an Alignments interessiert, die eine gewisse Mindestgüte besitzen, also z. B. eine Schranke für die Fehlerzahl nicht überschreiten. Dieser Umstand lässt sich ausnutzen, um die Laufzeit des Algorithmus zu reduzieren. Ist die Fehlerschranke im Verhältnis zur Readlänge sehr klein, müssen viele Spalten gar nicht vollständig berechnet werden, da alle Edit-Distanzen, die diese Schranke überschreiten, für das Endergebnis irrelevant sind. Um die Ideen der Optimierungen zu beschreiben, werden Indel-Varianten zunächst außen vorgelassen.

Pruning

Die Laufzeit des beschriebenen Algorithmus ist linear in der Größe der Edit-Distanz-Matrix, da jedes Feld für sich in konstanter Zeit berechnet werden kann. Für $|t| := n$ und $|p| =: m$ liegt die Laufzeit bei $\mathcal{O}(nm)$. Da aber nur Alignments mit einer gewissen Fehlerschranke interessant sind, könnte durch geschickte Pruning-Regeln die

Berechnung einiger Felder eingespart werden. In Abbildung 4.10 ist ein Beispiel dafür, welche Bereiche einer Edit-Distanz-Matrix potenziell gar nicht benötigt werden. Man kann sich vorstellen, dass bei einer im Verhältnis zur Readlänge kleinen Fehlerschranke die nicht benötigten Bereiche einen Großteil der Matrix einnehmen und sich somit ein großes Optimierungspotenzial ergibt. Es ist jedoch auch klar, dass diese Optimierungen nicht den *worst case* von $\mathcal{O}(nm)$ verbessern können, da im Falle von $t = N^n$ (also einer Referenz, die ausschließlich aus N's besteht) jedes Muster an jeder beliebigen Position fehlerfrei aligniert werden kann und somit auch quadratisch viele Felder berechnet werden müssen.

	-	C	G	A	A	C	C	T	T	G	T	C	G	G	A
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	1	1	1	0	0	1	1	1	1	1	1	1	1	1	0
A	2	2	2	1	0	1	1	1	2	2	2	2	2	2	1
C	3	2	3	2	1	0	1	1	1	2	3	2	3	3	2
C	4	3	3	3	2	1	0	1	1	2	3	3	3	4	3
T	5	4	4	4	3	2	1	0	1	2	2	3	4	4	4
T	6	5	5	5	4	3	2	1	0	1	2	3	4	5	5
T	7	6	6	6	5	4	3	2	1	1	1	2	3	4	5
C	8	7	7	7	6	5	4	3	2	2	2	1	2	3	4
G	9	8	7	8	7	6	5	4	3	2	3	2	1	2	3

Abbildung 4.10: Edit-Distanz-Matrix mit $t = CGAACCTTGTC$ und $p = AACCTTTC$. Die rot und blau markierten Felder wären bei einer Fehlerschranke von $k = 1$ irrelevant, da sie letztere überschreiten und somit niemals zu einer optimalen Lösung beitragen können.

Für weitere Überlegungen wird das folgende Lemma benötigt, welches aussagt, dass sich zwei horizontal oder vertikal benachbarte Felder einer Edit-Distanz-Matrix F um höchstens 1 unterscheiden können.

4.2.5 Lemma. *Seien $t = t_1 \cdots t_n$ ein Text, $p = p_1 \cdots p_m$ ein Muster und F die dazu berechnete Edit-Distanz-Matrix. Für alle Indizes $i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\}$ gilt:*

1. $|F[i-1, j] - F[i, j]| \leq 1$
2. $|F[i, j-1] - F[i, j]| \leq 1$

Beweis. Aus der Rekursionsgleichung für F folgt sofort:

1. $F[i-1, j] + 1 \geq F[i, j] \Leftrightarrow 1 \geq F[i, j] - F[i-1, j]$
2. $F[i, j-1] + 1 \geq F[i, j] \Leftrightarrow 1 \geq F[i, j] - F[i, j-1]$

Es bleibt zu zeigen:

1. $F[i-1,j] - F[i,j] \leq 1$
2. $F[i,j-1] - F[i,j] \leq 1$

Die Ungleichungen sagen aus, dass für einen Eintrag $F[i,j]$ der linke und obere Nachbar höchstens um 1 größer sein können. Der Beweis erfolgt durch Induktion:

Der Induktionsanfang ergibt sich aus der Initialisierung der Matrix, wodurch die genannten Ungleichungen für die erste Zeile und die erste Spalte erfüllt sind.

Angenommen die Aussage gilt für die Felder $F[i-1,j]$ und $F[i,j-1]$ mit $i \in \{1,\ldots,n\}, j \in \{1,\ldots,m\}$:

Es muss also gezeigt werden:

1. $F[i-1,j] - F[i,j] \leq 1$
2. $F[i,j-1] - F[i,j] \leq 1$

Da $F[i,j]$ durch einen Rekursionsfall berechnet worden sein muss, gilt auf jeden Fall:

$$F[i,j] \geq \min \begin{cases} F[i-1,j-1] & \\ F[i-1,j] & +1 \\ F[i,j-1] & +1 \end{cases}$$

Wir müssen also zeigen, dass die Werte der Vorgängerfelder von $F[i,j]$ nicht zu weit auseinanderliegen. Die Felder $F[i-1,j]$ und $F[i,j-1]$ haben wegen der Induktionsvoraussetzung höchstens Abstand 1 zum Feld $F[i-1,j-1]$. Daraus folgt per Dreiecksungleichung:

$$|F[i-1,j] - F[i,j-1]| \leq 2$$

Es folgt eine Fallunterscheidung:

1. Falls $F[i,j]$ aus $F[i-1,j-1]$ berechnet wurde, gilt $F[i,j] \geq F[i-1,j-1]$. Da $F[i-1,j]$ und $F[i,j-1]$ sich höchstens um 1 von $F[i-1,j-1]$ unterscheiden, kann $F[i,j]$ um höchstens 1 kleiner sein als $F[i-1,j]$ und $F[i,j-1]$. Es folgt die zu zeigende Aussage des Induktionsschritts.
2. Falls $F[i,j]$ aus $F[i-1,j]$ berechnet wurde, gilt $F[i,j] = F[i-1,j]+1$. Wie zuvor gezeigt, kann $F[i,j-1]$ wiederum um höchstens 2 größer sein als $F[i-1,j]$, sodass auch hier der Induktionsschritt für $F[i,j]$ folgt.
3. Falls $F[i,j]$ aus $F[i,j-1]$ berechnet wurde, gilt $F[i,j] = F[i,j-1]+1$. Analog zum zweiten Fall folgt auch der Induktionsschritt für $F[i,j]$. □

Mit Hilfe des Lemmas lässt sich das folgende Theorem beweisen.

4.2.6 Theorem. *Sei F eine Edit-Distanzmatrix für ein semiglobales Alignment mit Fehlerschranke k. Für jedes Feld $F[i,j]$ mit $i < m$, $j < n$ und $F[i,j] > k$ gilt:*

$$F[i+1, j+1] > k$$

Falls die Fehlerschranke k also in $F[i,j]$ überschritten wird, muss $F[i+1, j+1]$ nicht berechnet werden, weil dieser Wert garantiert auch größer als k ist.

Beweis. Wegen Lemma 4.2.5 gilt:

$$F[i+1, j], F[i, j+1] \geq F[i,j] - 1$$

Aus der Rekursionsgleichung folgt damit:

$$F[i+1, j+1] \geq \min\{F[i, j+1] + 1, F[i+1, j] + 1, F[i,j]\} = F[i,j] \qquad \square$$

RowCancel-Optimierung

Die erste algorithmische Optimierung, die wir in unserem Aligner implementiert haben, ist das Abschneiden von Spalten, die sogenannte *RowCancel-Optimierung*. An den Positionen, wo Read und Referenzabschnitt nicht zusammenpassen, liegen fast alle Spalteneinträge der Edit-Distanz über der Fehlerschranke k. Insbesondere gibt es in jeder Spalte j einen kleinsten Zeilenindex h_j, unterhalb dessen alle Einträge größer als k sind. Idealerweise würde der Aligner alle Spalten nur bis zu dieser Zeile berechnen.

Für die Spalte mit Index 0 gilt aufgrund der Initialisierung $h_1 = k$. Wegen Theorem 4.2.6 gilt $h_{j+1} \leq h_j + 1$ für jede Spalte j, da jede Überschreitung der Fehlerschranke in Spalte j auch eine Überschreitung in der darunterliegenden Zeile in Spalte $j+1$ impliziert. Unser Aligner berechnet die Edit-Distanz-Matrix spaltenweise und merkt sich bei der Berechnung einer Spalte j bis zu welchem Zeilenindex h_{j-1} die vorherige Spalte noch Werte unterhalb von k aufwies. Die Spalte wird anschließend nur bis zur $h_{j-1} + 1$-ten Spalte berechntet. Der Wert h_j ist der letzte Zeilenindex, für den sich noch ein Wert von k oder kleiner ergab.

Im Falle von Indelvarianten besitzt eine Spalte der Edit-Distanz-Matrix im Allgemeinen mehrere Vorgänger, sodass für Spalte j der maximale $h_{j'}$-Wert für alle Vorgängerspalten j' betrachtet werden muss, um Korrektheit zu garantieren.

In Abbildung 4.10 entsprechen die roten Felder gerade den Feldern, die sich durch die RowCancel-Optimierung einsparen lassen. Diese entspricht der Optimierung, die auch im Skript von Rahmann et al. (2013) beschrieben wird.

Um zusätzliche Laufzeit einzusparen, werden nur berechnete Felder in einem Alignierungsdurchgang beschrieben. Dazu wird die Backtracing-Matrix für alle Reads einmal allokiert und anschließend für jeden Read nur die benötigten Felder überschrieben, ohne die Matrix nach jeder Alignierung zurückzusetzen[4]. Dadurch lässt

[4]Es wird pro Thread eine Instanz des Aligners erzeugt, von denen jede unterschiedliche Reads aligniert. Jede Instanz besitzt allokiert eine eigene Backtracing-Matrix

sich bei der Laufzeit ein Best-Case von $\mathcal{O}(nk)$ erreichen, falls Read und Referenzabschnitt überhaupt nicht zusammenpassen.

Das Problem bei dieser Laufzeitoptimierung ist, dass Felder, die für die Verarbeitung des aktuellen Reads nicht beschrieben wurden, potenziell beliebige Werte beinhalten könnten, die ihnen bei der Verarbeitung eines vorherigen Reads zugewiesen wurden. Daher muss der Aligner so arbeiten, dass er jedes Feld, auf das er lesend zugreift, zuvor mit einem sinnvollen Wert beschreiben muss. Bei der RowCancel-Optimierung ist der einzige kritische Fall, dass eine Spalte j um eine Zeile weiter berechnet wird als ihr Vorgänger $j-1$. Um den letzten Eintrag von j zu berechnen, würde auf einen Zeilenindex der Spalte $j-1$ zugegriffen werden, der für den aktuellen Read nicht beschrieben wurde. Aus diesem Grund schreibt der Aligner am Ende jeder Spalte in die erste nicht berechnete Zeile den Wert $k+1$. Dadurch ist sichergestellt, dass der Aligner in der nächsten Spalte nicht über den selbst beschriebenen Bereich hinaus liest.

Trickreicher hingegen wird die Behandlung von Varianten. Falls eine Spalte j mehrere Vorgängerspalten besitzt und bis zum Zeilenindex h berechnet werden muss, so müssen alle Vorgängerspalten, falls notwendig, bis zum Zeilenindex h aufgefüllt werden. Da bereits bekannt ist, dass alle undefinierten Spalteneinträge größer als k sein müssen, können die Vorgängerspalten einfach mit dem Wert $k+1$ aufgefüllt werden. Dieser ist zwar nicht korrekt bezüglich der Definition von $F[i,j]$, allerdings haben alle Einträge, die größer als k sind, keinen Einfluss auf Alignments mit höchstens k Fehlern. Um beim Auffüllen keine gültigen Werte zu überschreiben, muss sich der Aligner für jede Spalte j den Wert h_j dauerhaft merken.

RowSkip-Optimierung

Ausgehend von der RowCancel-Optimierung haben wir den Aligner weiter optimiert. Als Motivation dafür dient Abbildung 4.10. Bei den blau markierten Feldern wird zwar die Fehlerschranke k überschritten, sie werden aber durch die RowCancel-Optimierung nicht erfasst, weil darunter noch Einträge mit kleinerer Fehlerzahl liegen. Bei üblichen Reads mit einer Länge von hundert Basenpaaren und einer verhältnismäßig kleinen Fehlerzahl können die „blauen“ Felder einen beträchtlichen Teil der Restmatrix ausmachen.

Die „blauen“ Felder lassen sich erneut mit Hilfe von Theorem 4.2.6 ausfindig machen. Falls in Spalte j der Edit-Distanz-Matrix in den Zeilen i_1 bis i_2 nur Werte größer als k vorliegen, so müssen in Spalte $j+1$ die Zeilen i_1+1 bis i_2+1 nicht berechnet werden, da sie auch größer als k sein müssen. Um diese Information auszunutzen, muss der Aligner in der Lage sein, bei der spaltenweisen Berechnung Zeilen beliebig überspringen zu können.

Unsere Implementierung fügt dazu für jede Zelle der Edit-Distanz-Matrix eine *Next-Row*-Information hinzu, welche den jeweils nächsten zu berechnenden Zeilenindex angibt. Im zuvor genannten Beispiel könnte in $F[i_1, j+1]$ der Wert i_2+2 gespeichert sein, um anzudeuten, dass i_2+2 der nächste zu berechnende Zeilenindex nach i_1 ist, da in $F[i_1+1, j+1]$ bis $F[i_2+1, j+1]$ nur Werte über k liegen können. Die Next-Row-Werte für eine Spalte j werden während der Berechnung der Spalte $j-1$

bestimmt. Abbildung 4.11 veranschaulicht diesen Optimierungsansatz anhand des bereits bekannten Beispiels aus Abbildung 4.10.

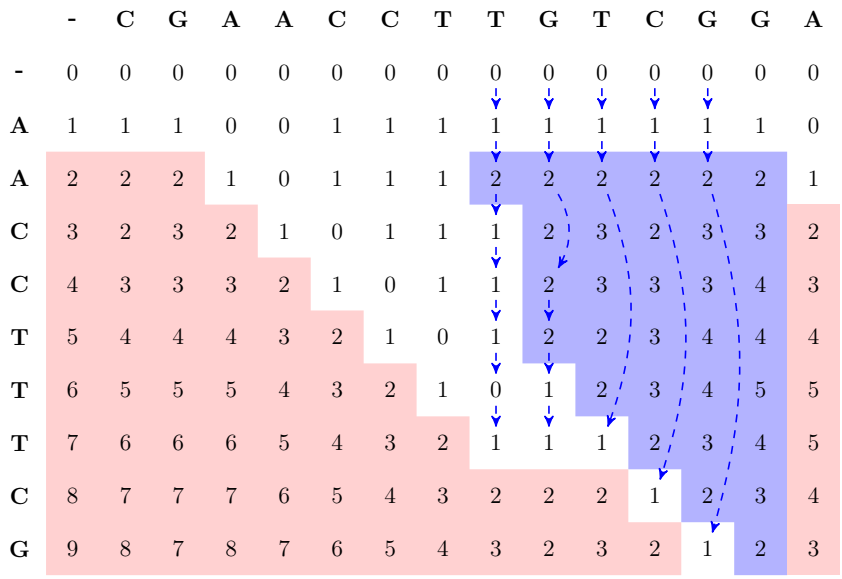


Abbildung 4.11: Edit-Distanz-Matrix mit $t = CGAACCTTGTC$ und $p = AACCTTTC$. Die Pfeile geben die Reihenfolge vor, in der die Zeilen einer Spalte gemäß der RowSkip-Optimierung berechnet werden. Sobald ein zu hoher Wert detektiert wird, wird in der nächsten Spalte das rechts unterhalb liegende Matrixfeld übersprungen. Dadurch können viele der blau markierten Felder bei der Alignierung ausgelassen werden.

Auch bei der RowSkip-Optimierung werden Felder, die übersprungen werden, nicht explizit beschrieben. Um Lesezugriffe auf undefinierte Felder zu vermeiden, ist mehr Aufwand als bei der RowCancel-Optimierung notwendig. Bei der Rekursionsgleichung können (ohne Varianten) folgende kritische Fälle eintreten:

1. Es gilt $F[i,j] \leq k$ und das Feld $F[i+1,j]$ wird übersprungen. Dadurch muss Feld $F[i+1,j+1]$ berechnet werden, greift aber auf das übersprungene Feld $F[i+1,j]$ zu.
2. Es gilt $F[i,j] \leq k$ und $F[i-1,j] > k$. Dadurch kann das Feld $F[i,j+1]$ übersprungen werden, allerdings wird es bei der Berechnung von $F[i+1,j+1]$ gelesen.

Es ist möglich diese Sonderfälle im Aligner direkt zu erkennen und die kritischen Felder erneut mit $k+1$ zu füllen. In der Praxis erwies es sich jedoch als effizienter, auf Verzweigungen im Code zu verzichten und bei der Berechnung jedes Feldes $F[i,j]$ den rechten und unteren Nachbarn (also $F[i,j+1]$ und $F[i+1,j]$) präventiv auf $k+1$ zu setzen. Dies ist im Sinne der Korrektheit unkritisch, da ein Matrixfeld wegen der allgemeinen Vorgehensweise der dynamischen Programmierung immer erst (durch einen korrekt berechneten Wert) beschrieben wird, bevor darauf lesend zugegriffen wird. Ausgenommen davon sind natürlich Felder, die aufgrund unserer Optimierungen übersprungen wurden. Für diese Felder wurde aber nachgewiesen, dass sie einen Edit-Distanz-Wert über k haben müssen, sodass diese Felder problemlos mit $k+1$ beschrieben werden können. Es lässt sich nachweisen, dass durch die präventiv

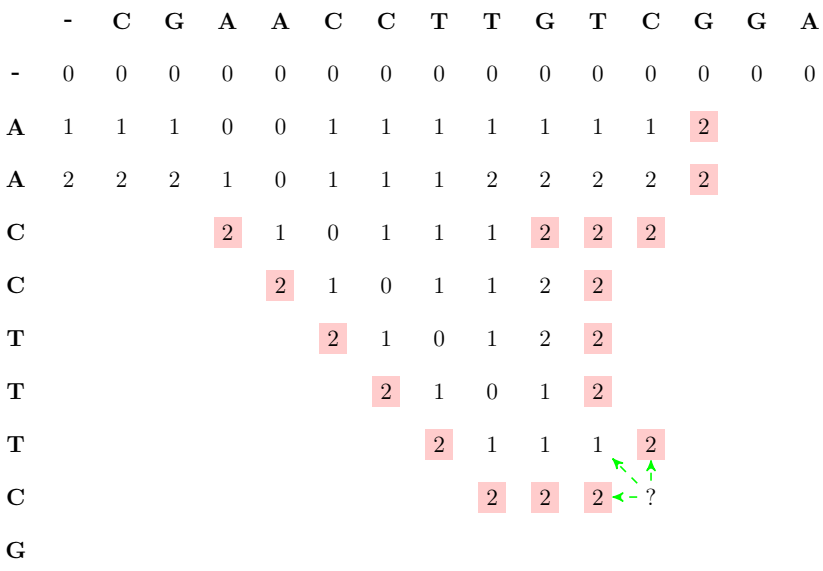


Abbildung 4.12: Tatsächlich beschriebene Matrixfelder für voriges Beispiel. Das Feld mit dem Fragezeichen ist das nächste zu berechnende Feld, die grünen Pfeile deuten die Lesezugriffe für diese Berechnung an. Die rot hinterlegten Felder wurden nicht berechnet, sondern ihr Inhalt wurde präventiv auf $k+1$ gesetzt.

geschriebenen Werte die oben erwähnten undefinierten Lesezugriffe nicht auftreten können. Das Vorgehen ist in Abbildung 4.12 nochmal dargestellt.

Bei Indels müssen zusätzlich zu den Sonderfällen, die bei der RowCancel-Optimierung auftreten, weitere beachtet werden:

1. Bei der Berechnung der Spalte j wird die Next-Row-Spalte für die Spalte $j+1$ berechnet. Falls j durch Insertionen und Substitutionen mehrere Nachfolgespalten besitzt, muss diese Next-Row-Spalte dupliziert werden, damit jeder Insertionsstring diese nutzen kann.

2. Wenn eine Spalte j mehrere Vorgängerspalten besitzt, liegen für j mehrere Next-Row-Spalten vor, die jeweils angeben welche Zeilen in Spalte j tatsächlich berechnet werden müssten. Diese Next-Row-Spalten müssen zu einer Next-Row-Spalte kombiniert werden, sodass keine relevante Zeile ausgelassen wird.

3. Falls vor einer Spalte j eine Deletionsvariante der Länge l vorkommt, wird bei der Berechnung von Spalte j auf die zurückliegende Spalte $j-l-1$ zugegriffen. Bei der RowSkip-Optimierung kann nun passieren, dass in Spalte $j-l-1$ Zeilen übersprungen wurden, auf die der Aligner bei der Berechnung von Spalte j zugreifen will.

Der erste Fall erfordert lediglich eine Kopieroperation auf Next-Row-Spalten, die im Falle von Varianten für die betroffenen Spalten aufgerufen wird. Die Kombination von Next-Row-Spalten für den zweiten Fall besteht bei unserem Aligner darin, dass eine Spalte mit mehreren Vorgängern immer lückenlos berechnet wird. Dies untergräbt zwar die RowSkip-Optimierung, ist dafür aber sehr einfach zu implementieren. Für den dritten Fall haben wir eine Operation implementiert, die eine Spalte der

Backtracing-Matrix anhand ihrer Next-Row-Verzeigerung traversiert und jede übersprungene Zeile mit dem Wert $k+1$ füllt. Falls eine Spalte j mehrere Vorgängerspalten besitzt - dies passiert immer wenn unmittelbar vor Spalte j Indel-Varianten behandelt wurden - wird die Auffülloperation für jede Vorgängerspalte von j aufgerufen.

4.2.5 Soft Clipping

Unser Aligner bietet die Möglichkeit an, sogenanntes *Soft Clipping* beim Alignieren zu benutzen. Soft Clipping bedeutet, dass der Anfang bzw. das Ende eines Reads bei der Alignierung ausgelassen wird. Dies kann nützlich sein, um Reads, die beispielsweise aufgrund der Sequenziertechnik am Anfang oder am Ende viele Fehler aufweisen, trotzdem mit einer geringen Fehlerzahl alignieren zu können. Basenpaare, die auf diese Weise bei der Alignierung abgeschnitten wurden, werden im Cigar-String mit dem Buchstaben „S“ vermerkt (vgl. auch Abschnitt 2.4.6).

Um Soft Clipping zu nutzen, muss der entsprechenden Parameter beim Aufruf des Aligners angegeben werden. Der Parameter gibt die maximale Länge des Soft Clippings am Anfang und am Ende des Reads an. Der Standardwert ist 0, wodurch kein Soft Clipping angewendet wird. Die Implementierung arbeitet nur sehr grob, da dieses Feature erst sehr spät diskutiert wurde.

Sei c die maximale Länge des Soft Clippings. Der Aligner führt für jeden Read p der Länge m zunächst die in den vorigen Abschnitten beschriebene Alignierung mit der vorgegebenen Fehlerschranke durch. Ist ein Read damit alignierbar, findet kein Soft Clipping statt. Falls ein Read nicht aligniert werden konnte, bestimmt der Aligner den größten Zeilenindex i, bei dem die Fehlerschranke eingehalten wurde. Das daraus resultierende Alignment würde das Suffix $p_{i+1}p_{i+2}\dots p_m$ des Reads also unberücksichtigt lassen. Falls die Länge dieses Suffixes kleiner gleich der maximalen Clipping-Länge c ist, gibt der Aligner das entsprechende Alignment aus und markiert den Suffix durch eine Sequenz von „S“ im Cigar-String. Der Zeilenindex i lässt sich bestimmen, indem sich der Aligner während der Berechnung den höchsten erreichten Zeilenindex sowie die Position des Minimums innerhalb dieser Zeile merkt.

Falls das abzuschneidende Suffix zu lang wäre, um die Clipping-Länge c einzuhalten, führt der Aligner einen zweiten Durchlauf für p aus. Dabei wird ein Präfix der Länge c weggelassen, sodass nur der hintere Teil des Reads aligniert wird. Für diesen Teil wird gegebenenfalls erneut ein Suffix-Clipping, wie im vorigen Absatz beschrieben, angewandt, sodass schließlich sowohl am Anfang, als auch am Ende des Reads bis zu c Zeichen unter das Soft Clipping fallen können. Das Präfix $p_1\dots p_c$ von p wird nicht zwingend komplett als *clipped* im Cigar-String aufgeführt. Bei einer erfolgreichen Alignierung werden die Zeichen p_c, p_{c-1}, usw.. nacheinander in invertierter Reihenfolge mit den Referenzzeichen verglichen, die vor dem Beginn der Alignierung stehen. Jede Übereinstimmung wird im Cigar-String als *match* aufgeführt, das eigentliche Soft Clipping beginnt erst vor dem ersten Mismatch. Dieses Vorgehen ist nicht optimal; vor allem berücksichtigt es keine Varianten. Es kürzt aber immerhin unnötig lange Clippings, falls sich offensichtliche Matches für den Read ergeben.

4.3 Programmstruktur und Implementierungsdetails

Die vorausgehenden Abschnitte haben die von uns verwendeten Verfahren zum Mapping und Alignment detailliert beschrieben. Zum Abschluss des Kapitels wird im Folgenden die Zusammenführung der Verfahren in unseren Readmapper VATRAM und dessen modulare Programmstruktur erläutert. Weiterhin werden die in der Implementierung verwendeten Sequenzdatentypen sowie deren Realisierung mittels der verwendeten Programmbibliothek *SeqAn* dargelegt.

4.3.1 Module

Das von uns entwickelte Programm VATRAM ist ein per *Command-Line Interface* (*CLI*) ansprechbarer, in Module aufgeteilter Readmapper. Die per Kommandos ansteuerbaren Module ergeben sich durch die zuvor beschriebene Trennung von Mapping und Alignment. Abbildung 4.13 zeigt die Module und deren durch die Vererbungshierarchie realisierten Beziehungen zueinander. Wie zu sehen ist, existieren vier konkrete Module, welche sich auf der Kommandozeile mittels

```
vatram <module-name> <module-options>
```

aufrufen lassen.

Das *AlignModule* stellt das Hauptmodul dar, welches im einfachsten Anwendungsfall durch

```
vatram align -r genome.fasta reads.fastq
```

mit seinen Standardparametern gestartet werden kann. Falls, wie im obigen Beispiel, keine explizite Ausgabedatei über den Parameter `-o` angegeben wird, wird die Standardausgabe der Konsole verwendet. Ein Aufruf des Moduls bewirkt eine vollständige Ausführung des Readmappers vom Einlesen und Indizieren des Referenzgenoms bis hin zum Mapping und Alignment der Reads. Die restlichen drei Module bieten jeweils nur einen Teil dieser Funktionalitäten an.

Das *IndexModule* erlaubt es, durch den Aufruf

```
vatram index -r genome.fasta genome.index
```

den für das Mapping notwendigen Index separat zu erzeugen. Somit kann die wiederholte Erzeugung des Index bei mehrfacher Ausführung des Programms vermieden werden. Durch

```
vatram align -r genome.fasta -i genome.index reads.fastq
```

kann der vorberechnete Index wiederverwendet werden.

Die beiden Module *MapModule* und *RealignModule* führen jeweils entweder das Mapping oder das Alignment der Reads separat aus.

Das *MapModule* gibt bei dem Aufruf

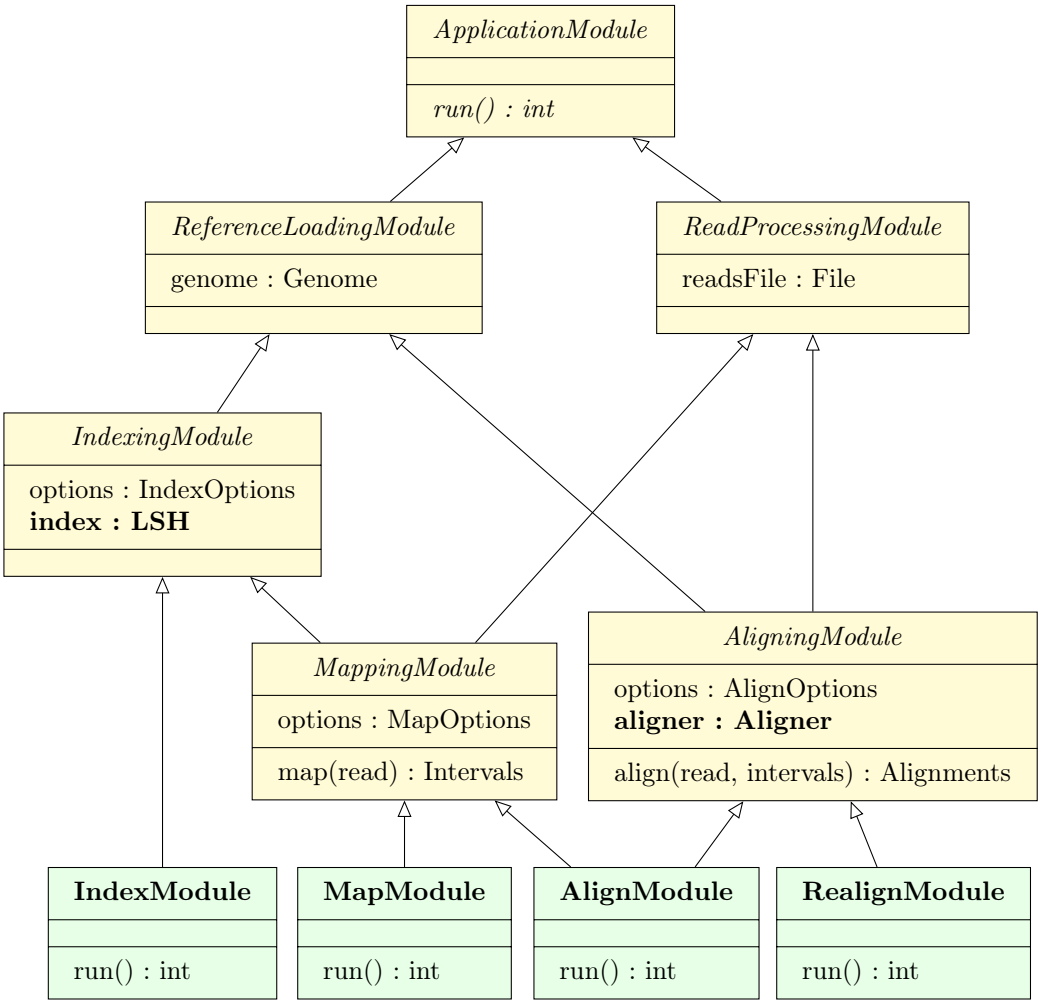


Abbildung 4.13: Programmmodule von VATRAM. Die grün hinterlegten nicht abstrakten Klassen repräsentieren die per CLI-Kommandos über ihre Namen (`index`, `map`, `align`, `realign`) ansprechbaren Module. Das Diagramm zeigt der Übersicht wegen eine reduzierte, vereinfachte Darstellung der Module.

```
vatram map -r genome.fasta -o mapped-reads.bam reads.fastq
```

hierzu in `mapped-reads.bam` für jeden Read die zugeordneten Referenzgenomabschnitte aus, ohne den Read an diese zu alignieren. Diese so erzeugte Ausgabedatei kann wiederum als Eingabe für das *RealignModule* durch den Aufruf

```
vatram realign -r genome.fasta mapped-reads.bam
```

verwendet werden. Die beiden zuletzt genannten Module sind nicht direkt für den Endanwendungsfall gedacht, sondern dienen vor allem der Reproduzierbarkeit von Experimenten und dem Vergleich mit Ergebnissen anderer Readmapper.

Die in Abbildung 4.13 dargestellte Vererbungshierarchie der Module wurde so gewählt, dass möglichst viel gemeinsam genutzter Code in die abstrakten Klassen ausgelagert wurde. Die in den Abschnitten 4.1 und 4.2 beschriebenen Verfahren zum Mapping und Alignment finden sich in der Implementierung in den Klassen *LSH*

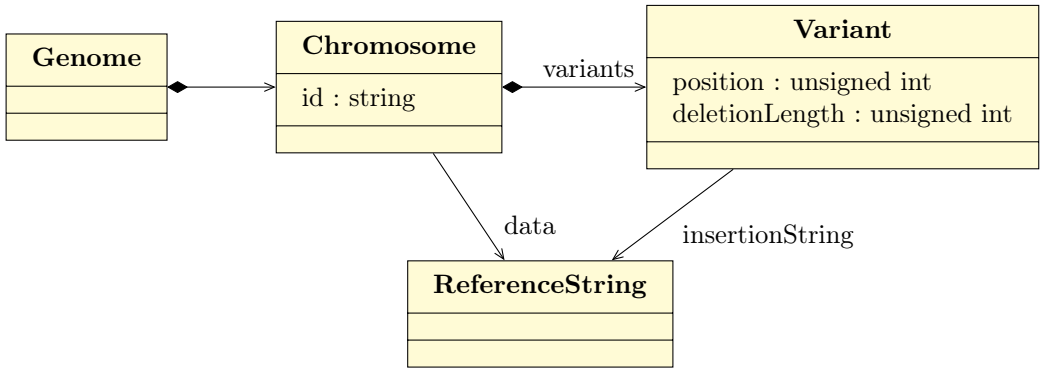


Abbildung 4.14: In VATRAM verwendete Datentypen für das Referenzgenom.

und *Aligner*, welche in den abstrakten Klassen *IndexingModule* und *AligningModule* durch die Attribute **index** respektive **aligner** benutzt werden. Somit verwendet sowohl das *MapModule* als auch das *AlignModule* das Mapping mittels LSH, da sie von *MappingModule* erben, welches wiederum ein *IndexingModule* ist. Ebenso wird zur Indizierung des Genoms im *IndexModule* das LSH benutzt. Neben dem LSH wird im *AlignModule* zusätzlich der Aligner verwendet. Das *RealignModule* benutzt hingegen nur den Aligner, da es schon zuvor berechnete Mappings verarbeitet. Weiterhin zeigt Abbildung 4.13, dass alleinig das *IndexModule* keine Reads verarbeitet, da es nicht von *ReadProcessingModule* erbt. Alle konkreten, ausführbaren Module benötigen für das Mapping bzw. Alignment ein vorhandenes Referenzgenom, welches zusammen mit dessen Varianten in jedem *ReferenceLoadingModule* geladen wird. Die Basisklasse *ApplicationModule* stellt letztlich die gemeinsame Schnittstelle zur Ausführung der Module zur Verfügung.

4.3.2 Implementierung der Sequenzdatentypen

In VATRAM werden einige grundlegende Datentypen zur Datenhaltung des Genoms und der Reads verwendet, über die hier ein kurzer Überblick gegeben wird.

Das Referenzgenom und dessen Varianten liegen in den in Kapitel 2.4 beschriebenen Dateiformaten FASTA und VCF vor. Zur Weiterverarbeitung werden die derart gegebenen Daten von VATRAM in eine Instanz der Klasse *Genome* geladen, dessen Klassendiagramm in Abbildung 4.14 dargestellt ist. Die Klasse *Genome* ist ein Array bestehend aus Instanzen der Klasse *Chromosome*. In den *Chromosome*-Instanzen liegen die Daten aus den FASTA- und VCF-Dateien in einem für das Readmapping aufbereiteten Format vor. Ein *Chromosome* besitzt neben einer eindeutigen Kennung, welche in der Regel die Nummer des Chromosoms angibt, die zwei Attribute *data* und *variants*. In *data* liegt das Referenzchromosom als IUPAC-String (siehe Abschnitt 2.4.1) kodiert in einer *ReferenceString*-Instanz vor. Neben dem reinen Referenzgenom sind auch alle Varianten, welche je nur ein Nukleotid betreffen (SNPs), in diesen IUPAC-Zeichen kodiert. Das Attribut *variants* ist die Liste von kurzen Indels und Substitutionen, welche aufsteigend nach den Positionen der Varianten im Chromosom sortiert ist. Die in *variants* vorliegenden Varianten besitzen als Instanzen der Klasse *Variant* neben ihren Positionen im Chromosom (*position*) noch die Information, wie viele Basen der Referenz entfernt wurden (*deletionLength*) und wel-

che zusätzlich hinzugefügt werden (*insertionString*). Die Liste der *Variant*-Instanzen wird vom Aligner verwendet.

Im Gegensatz zum Referenzgenom werden die einzelnen Reads alleinig über die Abfolge ihrer Basen beschrieben. Der Typ der Basenabfolge ist *ReadString*, was einer Zeichenkette über den IUPAC-Zeichen $\{A, C, G, T, N\}$ entspricht.

Den beiden Sequenztypen *ReferenceString* und *ReadString* liegen Datentypen aus der verwendeten Programmbibliothek *SeqAn* zugrunde, welche im nachfolgenden Abschnitt erläutert werden.

4.3.3 SeqAn

Bei der Implementierung des Readmappers wurde die Programmbibliothek *SeqAn* (Doring et al., 2008) als Grundlage verwendet. Durch sie werden in VATRAM insbesondere die grundlegenden Sequenzdatentypen sowie Funktionen zur Ein- und Ausgabe bereitgestellt.

SeqAn ist eine umfangreiche C++-Bibliothek zur Sequenzanalyse von biologischen Daten. Durch die in großem Umfang verwendete C++-Template-Metaprogrammierung erreicht die Bibliothek eine hohe Erweiterbarkeit und Konfigurierbarkeit. Die Metaprogrammierung ermöglicht es zudem, weitergehende Optimierungen durch den Compiler sowie Vorberechnung während der Compile-Zeit vorzunehmen, wodurch das endgültige Laufzeitverhalten des Programms verbessert werden kann. Im Folgenden soll ein einfaches Beispiel eines Datentyps, welcher auch so in VATRAM verwendet wird, die Verwendung der Templatebibliothek nahelegen.

Wie im vorherigen Abschnitt beschrieben, verwendet der Readmapper den Sequenztyp *ReferenceString* zur Repräsentation des Referenzgenoms. Dieser Datentyp ist folgendermaßen definiert:

```
typedef seqan::Iupac ReferenceChar;
typedef seqan::String<ReferenceChar,
                      seqan::Packed<seqan::Alloc<>>>
                     ReferenceString;
```

Zunächst wird *ReferenceChar* als *seqan::Iupac* definiert, was bedeutet, dass ein Zeichen des Referenzgenoms als IUPAC-Zeichen dargestellt werden soll. Der Typ *ReferenceString* ist eine Ausprägung des Templatetyps *seqan::String*, welcher zur Datenhaltung von Sequenzen benutzt wird. Hierbei wird dem Template als erster Parameter der Typ *ReferenceChar* übergeben, wodurch wir *ReferenceString* als Sequenz über IUPAC-Zeichen definieren. Durch den zweiten Templateparameter, *seqan::Packed*, wird die Art und Weise, wie die einzelnen Zeichen der Sequenz gespeichert und angesprochen werden, definiert. Der Parameter *seqan::Alloc<>* ermöglicht die dynamische Allokation von Strings auf dem Heap. Mittels *seqan::Packed* werden in einem Byte zwei IUPAC-Zeichen gespeichert, da diese jeweils nur 4 Bit belegen. Standardmäßig würde ein einziges Zeichen auf der x86-Prozessorarchitektur ein volles Byte belegen. *seqan::String* benutzt jedoch in seiner *seqan::Packed*-Ausprägung zusätzliche Informationen über die tatsächliche Anzahl an Bits, die die verwendeten Datentypen belegen. Diese werden für *seqan::Iupac* folgendermaßen bereitgestellt:

```
template <> struct BitsPerValue<Iupac> {
```

```
    typedef __uint8 Type;
    static const Type VALUE = 4;
};
```

Durch *seqan::BitsPerValue<ReferenceChar>* erfährt somit die Implementierung der gepackten Zeichenketten, dass nur 4 Bits pro Zeichen für den Typ *ReferenceString* verwendet werden müssen. Intern fasst SeqAn in einem Maschinenwort so viele Zeichen wie möglich zusammen und greift dann wortweise auf den zugrundeliegenden Speicher zu. Bei einer Wortgröße von 64 Bits werden demnach jeweils 16 IUPAC-Zeichen zugleich geladen bzw. gespeichert. Bei Benutzung der von SeqAn bereitgestellten Zugriffsfunktionen läuft die Extraktion und Zuweisung der einzelnen IUPAC-Zeichen innerhalb eines Wortes völlig automatisch ab. Somit können in den Algorithmen die gepackten und nicht gepackten Ausprägungen von *seqan::String* transparent ausgetauscht werden. In VATRAM findet dies etwa Anwendung bei der Signaturerstellung: Das gesamte Referenzgenom befindet sich in gepackter Form im Arbeitsspeicher, was zur Reduzierung des Speicherbedarfs verhilft und durch Speicherzugriffszeiten sowie Cachingeffekten die Laufzeit positiv beeinflusst. Bei der Signaturerstellung werden allerdings die Zeichen eines kleinen Genomabschnitts mehrfach direkt nacheinander gelesen. In diesem Fall ist die Zeit für den Speicherzugriff vernachlässigbar klein und der Mehraufwand durch die Extraktion der einzelnen IUPAC-Zeichen ausschlaggebend. Durch eine vorherige Konversion des Fensters von *seqan::String<ReferenceChar, seqan::Packed<> >* in *seqan::String<ReferenceChar>* kann der anfallende Mehraufwand vermieden werden, ohne dass weitere Änderungen am restlichen Code vorgenommen werden müssen.

Abgesehen von den beschriebenen Sequenztypen benutzt der Readmapper die Bibliothek SeqAn zur Ein- und Ausgabe der in Kapitel 2.4 beschriebenen Dateiformate und als Grundlage für das Command-Line Interface.

Insgesamt verwendet VATRAM jedoch nur ein Bruchteil der gesamten Bibiliothek. SeqAn stellt neben vielen weiteren Funktionalitäten zur Sequenzanalyse auch weitreichende Grundlagen und komplette Implementierungen für mehrere Textindexe und Aligner bereit. Das eigentliche Mapping und Alignment unseres Readmappers wurde hingegen von Grund auf neu programmiert, ohne die von SeqAn dazu vorgefertigten Konzepte zu verwenden. Dies ist zum einen dadurch begründet, dass das LSH zum Mapping und das variantentolerante Alignment Neuheiten darstellen, die in dieser Form nicht direkt durch die Bibliothek unterstützt werden. Zum anderen bedingt die in SeqAn verwendete Templatemetaprogrammierung nicht nur Schnelligkeit und Flexibilität, sondern erhöht auch die Komplexität des Codes und verschlechtert mitunter seine Lesbarkeit bzw. Verständlichkeit. Daher wurde letztlich das Konzept der Metaprogrammierung in unserem Projekt nur vereinzelt eingesetzt, was ebenfalls die Verwendung von SeqAn begrenzte.

Kapitel 5

Evaluation

Dieses Kapitel beschäftigt sich mit der Evaluation der Ergebnisse des programmierten Readmappers VATRAM. Um einen belastbaren Vergleich zu erzielen, gliedern sich die Experimente in vier große Bereiche:

Zu Beginn werden die beiden großen Bestandteile unserer Software, also der Einsatz des *Locality-Sensitive Hashing* und der Aligner, evaluiert. Dort werden Eigenschaften und Leistungen überprüft und in Anbetracht verschiedener Parameter optimiert. Anschließend werden die Ergebnisse unseres Readmappers mit denen zweier führender Konkurrenzprodukte verglichen: BWA (Li, 2013) und Bowtie2 (Langmead und Salzberg, 2012). Auch ein Fremdbenchmark namens *GCAT*[1] wird im folgenden Abschnitt eingesetzt. Dort wird auch diskutiert, welche Aussagekraft diese Ergebnisse haben.

5.1 Verwendete Hard- und Software

Bevor wir uns den einzelnen Teilergebnissen widmen, wird auf den globalen Aufbau der Experimente eingegangen. Lauffähig ist VATRAM auf einem handelsüblichen System, für welches eine Implementierung von GCC in der Version 4.8 (oder eines vergleichbaren C++-Compilers) existiert. Damit die Ausführung hochperformant wird, sollte eine hohe Menge an Arbeitsspeicher sowie ein leistungsstarker Prozessor zur Verfügung stehen.

Während der Evaluationsphase standen uns mehrere Server zur Verfügung. Der Compute-Server besitzt zwei Prozessoren des Typs *Intel Xeon E5-2640* mit einer Taktrate von 2.5 GHz und jeweils 6 Kernen mit 2 Threads[2]. Als Arbeitsspeicher standen 64 GB RAM zur Verfügung. Möglich war also die parallele Ausführung von 24 Threads. Für zeitunkritische Durchläufe standen uns weitere Server zur Verfügung, deren Ausstattung deutlich leistungsschwächer war. Ersterer besitzt einen Prozessor des Typs *Intel Core i7* mit einer Taktrate von 2.93 GHZ bei 4 Kernen mit 2 Threads und 12 GB Arbeitsspeicher. Letzterer besitzt zwei Prozessoren des Herstellers *AMD* mit der Bezeichnung *Opteron 2378*, welcher 4 Kerne mit einer jeweiligen Taktrate von 2.4 GHz besitzt. Als Arbeitsspeicher standen 16 GB zur Verfügung.

[1] Genome Comparison & Analytic Testing
[2] Diese unterstützen die Hyperthreading-Technologie

Im nächsten Abschnitt wird nun auf die Implementierung unseres *ReadGenerators* eingegangen, mit welchem wir synthetische Daten erzeugen können.

5.2 ReadGenerator

Mit Hilfe der Klasse *ReadGenerator* synthetisieren wir aus einer gegebenen Referenz und seinen zugehörigen Varianten einen Readstring, der mit einer Wahrscheinlichkeit, die durch einen Paramter *varProb* angegeben wird, eine Variante an einer einer Stelle einfügt. Damit haben wir einen Read, für den wir wissen, von welcher Position er stammt und welches perfekte Alignment er besitzt.

Dieser Punkt ist durchaus wichtig, denn erstens überprüfen wir unsere Fragestellung, zweitens testen wir die Funktionsweise und können somit sehen welche Anforderungen wir erfüllt haben. Der dritte Punkt ist die umfassende Information, woher der Read stammt und welche Varianten verwendet wurden, welche man bei Sequenzierungen von realen Daten nicht besitzt. Bei fremden Bibliotheken wissen wir meistens auch nur, wohin andere Mapper wie BWA sie aligniert haben. Damit erfahren wir auch, was Readmapper generell leisten können und ob wir mit unserem Vorhaben unser Ziel erreicht haben. Solche Informationen sind eigentlich vor der Evaluationsphase wichtig und man hätte bereits in der Phase der LSH-Entwicklung nutzen können und hätte den Aligner noch nicht benötigt, um zielgenaue Aussagen bezüglich der Tauglichkeit erlangen zu können. Nun benutzen wir die generierten Reads für die Evaluation und für statistische Aussagen.

Da wir von unserem Readmapper VATRAM äußerst überzeugt sind, wollen wir natürlich auch Varianten mappen. Vorherige Readmapper haben bei diesem Anwendungsfall Probleme, diese in angemessener Laufzeit zu mappen.

Wir generieren einen Read aus einer Referenz und wahlweise aus einer Variante. Im internen Ablauf lässt der *ReadGenerator* zwei Indizes laufen - einen für die Quelle in der Referenz und einmal für den Read selbst. Fügen wir eine Variante ein, so bleibt der Index für die Referenz unverändert. Die Deletions sind rückwärts gespeichert, dass heißt wir reduzieren den read index und kürzen den Read um die angegebene Länge. Bei einer Insertion hingegen hängen wir einfach die Insertion an und erweitern den read Index um die Insertion-Länge.

Das Abhandeln der Varianten wird in der Methode *handleVar* vollzogen. Dort wird die Information aus der Variantendatei befragt. Anschließend kehren wir wieder zurück, aktualisieren beide Indizes und arbeiten mit der Referenz weiter. Die Simulation von Sequenzierfehlern arbeiten wir nach dem Synthetisieren des Reads ein, da die Verarbeitung in der Hauptschleife zu kompliziert wäre und man mit den Varianten durcheinander käme. Die Bearbeitung erfolgt in der Methode *DoRand*, in welcher erneut durch den Read iteriert wird. Je nach Parametereinstellung für die Wahrscheinlichkeiten werden Insertionen, Deletionen oder Substitutionen der Länge 1 hinzugefügt.

Hier folgt nun der genaue Aufbau des ReadGenerators:

Damit keine Platzhalter aligniert werden müssen, wird eine Stelle gesucht, die nicht allzu viele N-SNP besitzt. Es werden dann solange Zeichen aus der Hauptreferenz

Suche Stelle im Genom, die nicht zu viele N-SNPs besitzt

```
1: procedure GenerateRead(varProb, insProb, delProb, subsProb, snpProb)
2:     for i ≤ referenceLength do
3:         handleVar(p, varProb)
4:         readString ← zeichenausReferenzanStellei
5:         readIndex++
6:     end for
7:     doRand(readString, insProb, delProb, subsProb, snpProb)
8: end procedure
9:
10: procedure handleVar(p, prob)
11:     if prob then
12:         variants ← alle Varianten an der Stelle p
13:         v ← Wähle eine Variante aus variants
14:         if v is del or subs then
15:             setze den readIndex um deletionLength zurück
16:             setze read auf Länge (readIndex+1)
17:         else if v is ins or subs then
18:             hänge insertion an read
19:             readIndex += insertionLength
20:         end if
21:     end if
22: end procedure
23:
24: procedure doRand(readString, insProb, delProb, subsProb snpProb)
25:     while p ≤ len(read) do
26:         if insProb then
27:             füge zufällige Insertion an Stelle p ein
28:         else if delProb then
29:             entferne den Abschnitt bei (p - deletionLength, p)
30:         else if subsProb then
31:             entferne den Abschnitt bei (p - deletionLength, p)
32:             füge zufällige Insertion an Stelle p - deletionLength ein
33:         end if
34:         p++
35:     end while
36: end procedure
```

kopiert, bis wir auf eine Variante stoßen. Diese Überprüfung erfolgt mit Hilfe der Methode *handleVar*. Die Wahrscheinlichkeit varProb entscheidet, ob wir eine der Varianten auswählen.

Wurde eine Variante ausgewählt und ist sie entweder eine Insertion, eine Deletion oder eine Substitution. Im ersten Fall kopieren wir die komplette Insertion an die Stelle. Ist der zweite Falle eingetreten, erfolgt die Behandlung der Deletion durch rückwirkendes Zurücksetzen des Strings um die angegebene Länge. Der letzte Fall ist die Substitution, wo zuerst analog der Deletion die Referenz zurückgesetzt wird und die alternativen Basen an dieser Stelle eingefügt werden.

Die bisherige Implementierung ist nicht sehr effizient bezüglich der Auswahlmöglichkeiten für Varianten. Auch die Laufzeit ist durch wiederkehrendes Zurücksetzen des Reads unglücklich gelöst. Eine Vorauswahl der Varianten durch ein ausgeklügeltes Auswahlkriterium und eine Stack-Datenstruktur, welchen man immer bis zur nächstgelegene Variante mit der Referenz auffüllt *(CopyToVariantPosition)*, wäre besser gewesen.

5.3 LSH-Experimente

Der LSH-Algorithmus beinhaltet eine Vielzahl von Parametern, die richtig konfiguriert werden müssen, damit VATRAM gute Ergebnisse erzielt. Zum einen sollten die zurückgegebenen Intervalle die korrekte Position des Reads beinhalten, zum anderen sollten die Größe und die Anzahl der Intervalle möglichst gering sein, um die Laufzeit für den Aligner zu minimieren. Des Weiteren möchte man den Speicherbedarf der LSH-Datenstrukturen minimieren sowie die Laufzeit für eine Suchanfrage gering halten. Letztere muss immer in Relation mit der Zeit gesehen werden, die der Aligner benötigt, um den Read in den zurückgegebenen Intervallen wiederzufinden. In diesem Abschnitt geht es darum, eine möglichst gute Parameterkonfiguration für den LSH-Algorithmus zu finden, die die oben beschriebenen Kriterien optimiert.

5.3.1 Versuchsaufbau

Datensätze Bei den LSH-Experimenten wurden drei Datensätze verwendet: Ein echter Datensatz mit Reads der Illumina-Sequenziermaschine (siehe Abschnitt 2.2.2), ein Benchmark-Datensatz namens GCAT[3] sowie selbst generierte Reads aus dem Referenzgenom. Bei den Illumina-Reads wurde ein Durchlauf mit Varianten als auch einer ohne Varianten ausgeführt. Die Berücksichtigung der Varianten macht bei diesem Datensatz Sinn, da es sich um die DNA eines tatsächlich existierenden Menschen handelt. Insbesondere sollten die Ergebnisse besser sein, wenn man die häufig vorkommenden Varianten miteinbezieht. Bei den GCAT-Reads macht eine Berücksichtigung von Varianten keinen Sinn, da beim Erstellen des Datensatzes ausschließlich auf das Referenzgenom zurückgegriffen wurde. Bei den von uns generierten Reads wurden die Experimente einmal ohne und einmal mit SNP-Varianten durchgeführt. Nur letzten Fall wurden die bekannten SNP-Varianten in die IUPAC-Codierung des

[3]Eine genauere Beschreibung und eine Analyse des GCAT-Datensatzes findet sich in Abschnitt 5.6.

Referenzgenoms eingefügt. Beim Erstellen der Reads wurde an jeder Position mit 2%-iger Wahrscheinlichkeit ein Fehler eingebaut. Bei 90% dieser Fehler handelt es sich um die Änderung einer Base, in den restlichen 10% der Fälle wurde eine Base eingefügt oder gelöscht. Durchschnittlich enthält damit jeder Read 1,8 SNP-Fehler und 0,2 Indel-Fehler. Wird bei der Read-Generierung eine SNP-Variante im Referenzgenom gelesen, wird eine zufällige der jeweils möglichen Basen ausgewählt (siehe auch Abschnitt 5.2). Auf das Einfügen bekannter Indel-Varianten wurde komplett verzichtet, da diese vom LSH-Algorithmus nicht berücksichtigt werden.

Durchführung Die Reads der jeweiligen Datensätze wurden bei den LSH-Experimenten ausschließlich gemappt, allerdings nicht aligniert. Um trotzdem eine Aussage darüber zu treffen, ob die Mappings vom LSH korrekt sind, ist es notwendig, die richtige Position des jeweiligen Reads zu kennen. Bei den selbst generierten Reads ist dies unproblematisch, da beim Erzeugen die jeweilige Position im Referenzgenom gespeichert werden kann. Bei den GCAT-Reads und insbesondere bei den realen Illumina-Reads ist uns die richtige Position jedoch unbekannt. Aus diesem Grund wurden die Datensätze zunächst mit BWA verarbeitet, um je eine Position der Reads im Referenzgenom zu bestimmen. Die entstandene SAM-Datei wurde dann als Eingabe für die LSH-Experimente verwendet. Dieses Vorgehen ist jedoch suboptimal: Zum einen existieren Reads, die BWA nicht findet, aber möglicherweise von unserem Readmapper hätten korrekt gemappt werden können, und zum anderen könnte es sein, dass bestimmte Reads von BWA an eine falsche Position gemappt wurden, zu denen unser Readmapper aber die richtige Position gefunden hätte. Warum wir uns trotzdem für dieses Vorgehen entschieden haben und welche Alternativen es gegeben hätte, wird in den nächsten beiden Absätzen näher erläutert.

Wenn man sich die unter http://www.bioplanet.com/gcat veröffentlichten Ergebnisse anschaut, stellt man fest, dass BWA etwa 0,8% der GCAT-Reads falsch gemappt hat und nur 0,3% gar nicht gefunden hat. Die restlichen 98,9% der Reads wurden korrekt aligniert. Die 0,3% tauchen in dem Datensatz für das LSH-Experiment nicht auf, die 0,8% könnte unser Readmapper möglicherweise korrekt mappen. Unser Readmapper könnte damit bis zu 0,8%-Punkte besser sein, als sich durch die Experimente ergibt[4]. Angesichts der Tatsache, dass in den meisten der folgenden Experimenten nur höchstens 97% der Reads an die von BWA zurückgegebene Position gemappt wurden, kann man davon ausgehen, dass unser Readmapper sicherlich nur wenige der schwierigen 0,8% korrekt positioniert hätte. Genauere Vergleiche zwischen unserem und anderen Readmappern finden sich in Abschnitt 5.5.

An dieser Stelle kann man sich die Frage stellen, warum die Reads bei den LSH-Experimenten nur gemappt und nicht zusätzlich aligniert wurden. Dies hat zum einen historische Gründe: Als angefangen wurde, die LSH-Experimente zu implementieren, war der Aligner noch sehr langsam (vgl. Abschnitt 4.2 und 5.4). Inzwischen dominiert je nach Anzahl der zurückgegebenen Intervalle sogar die Laufzeit für LSH-Anfragen.

[4]Wenn man die 0,3% der von BWA nicht gemappten Reads hinzunimmt und davon ausgeht, dass unser Readmapper diese korrekt positioniert hätte, erhöht sich die Gesamtzahl der gemappten Reads nicht um 0,3%-Punkte, da durch Hinzunahme dieser Reads auch die Gesamtzahl der Reads ansteigt. Die Anzahl der nicht alignierten Reads würde sich um den Faktor $1/1{,}003 \approx 0{,}997$ verringern, was vernachlässigbar wenig ist. Wenn man davon ausgeht, dass 95% korrekt gemappt werden, würde sich der Wert auf höchstens 95,015% verbessern können.

Ein Grund, warum nachträglich darauf verzichtet wurde, die Reads zu alignieren, ist, dass zumindest bei den selbst erzeugten Reads die korrekte Position bekannt ist. Der Aligner arbeitet deterministisch, so dass er einen Read stets alignieren kann, wenn das übergebene Intervall ein korrektes Mapping darstellt. Ein zusätzlicher Aligner-Aufruf würde also nur die Laufzeit erhöhen und damit effektiv die Anzahl der durchführbaren Experimente verringern, da diese durch die Rechenzeit limitiert werden. Darüber hinaus stellt sich die Frage, wie hoch die Fehlerschranke im Aligner zu wählen ist. Auch die Interpretation der Ergebnisse der anderen Datensätze ist problematisch: Wird ein Read an eine Position aligniert, wissen wir nicht, ob diese korrekt war oder ob es vielleicht eine bessere gegeben hätte.

Parameter Bei der Variation der Parameter wurde wie folgt vorgegangen: In der Regel wurde nur ein Parameter variiert, während die anderen konstant auf Standardwerte gesetzt wurden. Die benutzten Standardwerte sind durch viele vorangegangene Experimente ermittelt worden, bei denen die LSH-Parameter immer wieder auf bessere Werte gesetzt worden sind.[5] Einige Parameter hängen eng miteinander zusammen, wie beispielsweise die Fenstergröße (*windowSize*) und der Fensterabstand (*windowOffset*). Bei diesen Parametern wurden verschiedene Kombinationen von Werten ausprobiert. Allerdings steigt die Menge der auszuführenden Experimente mit der Anzahl der gleichzeitig betrachteten Parameter exponentiell an. Da ein Experiment etwa drei bis fünf Minuten dauerte, wurden möglichst wenig Parameter gleichzeitig verändert.

Prinzipiell kann man für eine solche Parameteroptimierung auch evolutionäre Algorithmen einsetzen. Problematisch dabei ist zum einen, dass mehrere Optimierungskriterien existieren (multikriterielle Optimierung), und zum anderen, dass die Anzahl der zu variierenden Parameter (siehe Tabelle 5.1) recht groß ist. Darüber hinaus dauert ein Experiment (trotz der Benutzung von 24 parallelen Threads) mehrere Minuten, wodurch ein evolutionärer Ansatz kaum noch in Frage kommt. Aus diesem Grund wurde auf den Einsatz eines evolutionären Algorithmus verzichtet und stattdessen die Parameterwahl und die Auswertung manuell durchgeführt.

Bei den LSH-Parametern, die variiert wurden, kann zwischen Index-Parametern und Anfrage-Parametern unterschieden werden. Erstere beeinflussen den Index (und damit auch die Anfragen auf diesen), letztere beeinflussen nur die Anfragen, d.h. es wäre nicht notwendig, einen neuen Index zu erstellen, wenn Anfrage-Parameter verändert werden. Als Beispiel für einen Index-Parameter könnte die q-Gramm-Länge (*qGramSize*) benannt werden: Bei einer Veränderung dieser, muss auch der Index neu berechnet werden. Wird dagegen beispielsweise die maximale Anzahl der zurückgegebenen Intervalle variiert (*maxSelect*), hat dies nur Auswirkung auf die Anfragen, der Index selbst bleibt derselbe.

Messung In den Experimenten wurden verschiedene Größen gemessen, die im folgenden kurz erläutert werden.

Es wurde zum einen gespeichert, wie lange es dauert, den LSH-Index zu erstellen, und zum anderen, wie lange durchschnittlich die Suche nach einem Read benötigt. Die an-

[5] Dazu wurden die unten beschriebenen Experimente mehrfach durchgeführt und die LSH-Parameter jeweils auf die besten ermittelten Werte gesetzt.

Parameter	**Bedeutung**	**Standard**-Wert
TQGramHash	Datentyp für q-Gramme	uint32_t (32 bit)
QGramHashFunction	q-Gramm-Hashfunktion	multiplikativ, $m = 33767$
qGramSize	q-Gramm-Länge	16
gapVector	q-Gramm-Lückenmuster	`####-#-####-##-#####`
windowSize	Fensterlänge	140
windowOffset	Fensterabstand	125
signatureLength	Signaturlänge	48
TBandHash	Datentyp für Bänder	uint32_t (32 bit)
bandSize	Anzahl der Bänder	1
bandHashFunction	Band-Hashfunktion	Identitäts-Funktion
hashFunctions	Initialisierung der Signatur-Hashwerte	Gleichverteilt zufällig
limit	Parameter für Variantenberücksichtigung	3
limitSkip	Parameter für Variantenberücksichtigung	8
maxiumOfReturnedWindows	Maximale Anzahl zurückgegebener Fenster	1000
minCount	Minimale Anzahl Treffer pro Fenster	2
maxSelect	Maximale Anzahl zurückgegebener Intervalle	64
nextFactor	Faktor für die Auswahl von Fenstersequenzen	4
contractBandMultiplicator	Band-Multiplikator für die Verkleinerung von Fensterpaaren (bei der Intervall-Bestimmung)	0,3
extendBandMultiplicator	Band-Multiplikator für die Vergrößerung von einzelnen Fenstern (bei der Intervall-Bestimmung)	0,3
extendNumber	Absolute Vergrößerung von einzelnen Fenstern (bei der Intervall-Bestimmung)	60

Tabelle 5.1: Die LSH-Parameter und ihre Standard-Werte. Eine genaue Beschreibung der einzelnen Parameter findet sich in den jeweiligen unten stehenden Abschnitten sowie im Kapitel 4.1. Die Index-Parameter sind von den Anfrage-Parametern durch eine doppelte horizontale Linie getrennt.

gegebenen Laufzeiten beziehen sich stets auf die verwendete Hardware (siehe Anfang des Abschnitts 5) und sind nicht als pro-Kern-Laufzeiten zu verstehen. Darüber hinaus wird der Speicherverbrauch des LSH-Index ausgegeben. Die Angaben beziehen sich dabei ausschließlich auf den Speicherverbrauch der SuperRank-Datenstruktur (bzw. der CollisionFreeBandHashTable) sowie für die Verwaltung der Fensterindizes. Der Speicherverbrauch für das Referenzgenom und die Datenstrukturen des Aligners werden nicht mitgezählt.

Die Mapping-Qualität kann auf zwei verschiedene Weisen bestimmt werden. Sei $R = [r_l, r_r]$ das Intervall auf dem Referenzgenom, welches dem Read entspricht, und $I = [i_l, i_r]$ das Intervall, das vom LSH-Verfahren zurückgegeben wurde. Wenn $R \cap I = R$ ist, beinhaltet I das richtige Read-Intervall, d.h. der Aligner wird den Read korrekt alignieren können. Alle Reads, für die diese Bedingung (für mindestens ein zurückgegebenes Intervall I) erfüllt wurde, werden bei diesem Maß als korrekt bewertet. Der Anteil der korrekt gemappten Reads wird in den Rohdaten als *Correct Interval Found* bezeichnet.

Wenn I bereits um eine Base zu klein ist, wird das Mapping bei diesem Qualitätswert als falsch gewertet. Aus diesem Grund gibt es noch ein zweites Maß, bei dem es ausreicht, wenn sich das zurückgegebene Intervall und der Read an mindestens einer Position überschneiden. Hierbei werden also alle Reads als richtig gemappt angesehen, für die $R \cap I \neq \emptyset$ gilt. In den Rohdaten wird dieses Maß als *Correct Intersection Found* bezeichnet. Die Differenz zwischen beiden Qualitätsmaßen gibt an, wie sehr man die Mapping-Qualität verbessern könnten, wenn man die zurückgegebenen Intervalle links und rechts entsprechend erweitern würde.

Darüber hinaus wurde die Anzahl der Reads gezählt, für die kein Intervall zurückgegeben wurde (*No Interval Found*). Dieser Anteil ist jedoch meistens so klein (in der Regel weniger als 0,01%), sodass er in den Auswertungen der einzelnen Experimente in der Regel nicht näher beschrieben wird.

Die durchschnittliche Anzahl und die Gesamtgröße der zurückgegeben Intervalle wurde ebenfalls festgehalten. Insbesondere die Gesamtgröße der Intervalle ist ein guter Anhaltspunkt für die Laufzeit des Aligners. Mit Hilfe der Messergebnisse aus Abschnitt 5.4.2 kann die Intervallgröße ungefähr in eine Laufzeit umgerechnet werden.

Sofern nicht anders angegeben wurde bei den Messungen stets derselbe Seed für die Zufallszahlengeneratoren verwendet. Trotzdem sind minimale Abweichungen möglich, sofern mit Varianten gearbeitet wurde. Die Ursache hierfür liegt darin, dass jeder Thread seinen eigenen Zufallsgenerator besitzt, der bestimmt, welche Kombinationen bei SNP-Varianten gewählt werden (sofern die Anzahl der Kombinationen zwischen *limit* und *limit-skip* liegt). Da wir bei der Indexerstellung auf die genaue Zuteilung der Fenster zu den Threads keinen Einfluss nehmen, sind geringfügige Abweichungen möglich. Diese spielen bei der Auswertung jedoch keine Rolle.

5.3.2 Index-Parameter

Dieser Abschnitt behandelt die Auswertung der Index-Parameter, also diejenigen Parameter, die Einfluss auf den LSH-Index haben.

5.3.2.1 q-Gramm-Parameter

In diesem Abschnitt geht es darum, die q-Gramm-Parameter zu variieren. Dazu gehört die Länge der q-Gramme, die verwendete Hashfunktion, um ein q-Gramm in ein 32- oder 64-Bit-Wort zu konvertieren, sowie die Verwendung sogenannter *gapped q-grams*, also q-Gramme mit Lücken.

q-Gramm-Hashfunktion Da der Standardparameter für die q-Gramm-Länge 16 beträgt und jedes Symbol (d.h. jede Base) zwei Bit benötigt, bietet es sich an, die Zeichen des jeweiligen q-Gramms in ein 32-Bit-Wort „hineinzuschieben“. Die beiden höchstwertigsten Bits repräsentieren dabei das erste Zeichen des q-Gramms, die beiden darauf folgenden Bits das zweite Zeichen, usw. Der entstandene Bit-String wird dann als Hash-Wert genutzt. Bei dieser Hashfunktion entstehen keine Kollisionen, d.h. jeder Hashwert gehört zu genau einem q-Gramm. Nachteilig könnte sein, dass die Zeichen im q-Gramm bei dieser Hashfunktion nicht durchmischt werden. In der LSH-Implementierung wird bei der Signaturberechnung keine echte Permutation aller q-Gramme erzeugt. Anstatt dessen werden die einzelnen Bits des q-Gramm-Hashwertes mit einem weiteren konstanten Wert exklusiv verodert (vgl. Abschnitt 4.1.5.2). Anschließend wird das Minimum der sich ergebenden Werte berechnet, das heißt, dass fast ausschließlich die höchstwertigsten Bits darüber entscheiden, welcher Wert das Minimum bildet. Die höchstwertigsten Bits entsprechen bei der oben beschriebenen, sogenannten Shifting-Hashfunktion dem Anfang des q-Gramms. Um auch Mitte und Ende der q-Gramme bei der Minimumsbildung stärker zu berücksichtigen, könnte es besser sein, eine multiplikative Hashfunktion zu verwenden:

$$h(x) = \sum_{i=1}^{q} \left(m^{q-i} \cdot x_i\right) \mod 2^{32} \tag{5.1}$$

Dabei ist x_i das i-te Zeichen des q-Gramms x, m eine beliebige Zahl und $h(x)$ der resultierende Hashwert. Da die Berechnung auf 32-Bit breiten Wörtern stattfindet, ist die explizite Berechnung der Modulo-Operation nicht notwendig. Für $m = 4$ ergibt sich die oben beschriebene Shifting-Hashfunktion. Für $m \neq 4$ kann es zu Kollisionen kommen, d.h. mehrere q-Gramme werden ggf. auf den gleichen Wert abgebildet.

Als weitere Alternative bietet sich die Hashfunktion der C++-Standardbibliothek an. Dazu wird das q-Gramm zunächst durch die Shifting-Hashfunktion in ein 32-Bit-Wort umgewandelt und auf dem danach *std::hash* angewendet wird.

Das linke Diagramm in Abbildung 5.1 zeigt, für wie viele Reads ein korrektes Intervall gefunden wurde. Die einfache Shifting-Hashfunktion schneidet dabei am schlechtesten ab: Durch Verwendung anderer Hashfunktionen steigt der Anteil gefundener Reads je nach Datensatz zwischen 0,3 und 0,8 Prozentpunkte. Bei unseren Reads und den GCAT-Reads schneidet die multiplikative Hashfunktion mit $m = 5$ am besten ab, bei den realen Reads liefert $m = 33767$ die besten Ergebnisse. Die Unterschiede sind aber so minimal, dass bei Verwendung eines anderen Seeds vermutlich jeweils eine andere Hashfunktion die besten Ergebnisse produzieren würde. In Abschnitt 5.3.4 gehen wir noch genauer auf die Streuung der Ergebnisse mit unterschiedlichen Seeds ein.

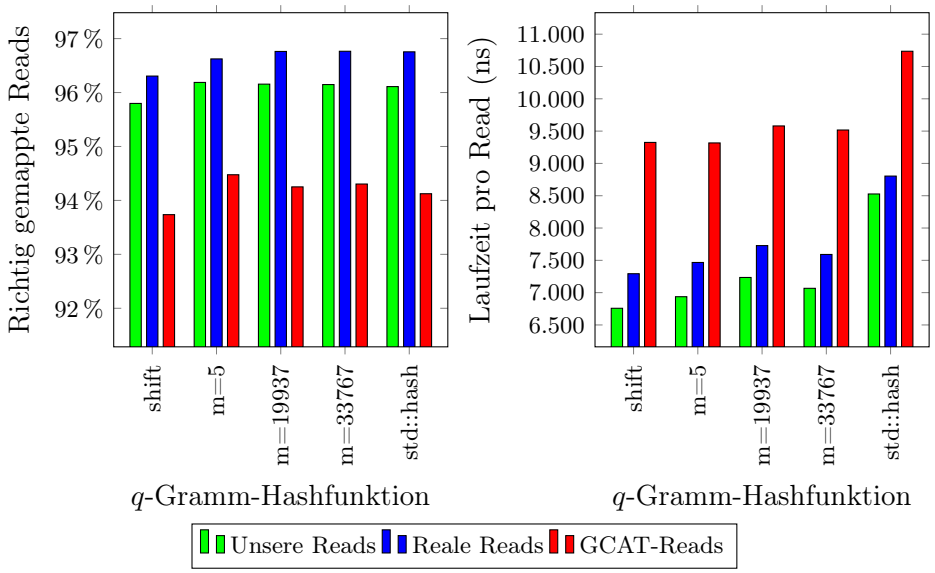


Abbildung 5.1: Variation der q-Gramm-Hashfunktionen.

Warum die Shifting-Hashfunktion nicht gut sein könnte, wurde bereits oben erörtert. Die Theorie war, dass die Bits für die Minimumsbildung nicht ausreichend durchmischt werden. Dagegen spricht, dass bereits $m = 5$ bei zwei der drei Datensätze die besten Ergebnisse liefert. Wenn man annimmt, dass für die Minimumsbildung von 85 q-Gramm-Hashwerten[6] durchschnittlich die ersten sieben Bits entscheidend sind, dann werden bei der Shifting-Hashfunktion nur die ersten 3 bis 4 Basen berücksichtigt. Bei $m = 5$ haben die ersten 5 Basen Einfluss auf die höchstwertigsten 7 Bits, denn $m^{(16-5)} = 48.828.125 > 33.554.432 = 2^{(32-7)}$ und $m^{(16-6)} < 2^{(32-7)}$. Dass diese zusätzlichen 1,5 Basen solch eine große Auswirkung haben, ist unwahrscheinlich.

Durch Verwendung von $m = 5$ entstehen Kollisionen, d.h. mehrere q-Gramme werden auf denselben Wert abgebildet. Für $q = 2$ und einer Wortlänge von vier Bit ergibt sich beispielsweise folgende Kollision, wobei die Basen (A, C, G, T) auf die Werte $(0, 1, 2, 3)$ abgebildet wurden:

$$h(TC) = 5 \cdot T + C = 5 \cdot 3 + 1 = 16 \equiv 0 \pmod{16}$$

$$h(AA) = 5 \cdot A + A = 5 \cdot 0 + 0 \equiv 0 \pmod{16}$$

Durch diese Kollisionen werden möglicherweise Übereinstimmungen in den Band-Hashtabellen gefunden, die aber auf eine q-Gramm-Kollision zurückzuführen sind. Als Folge sollte die false-positive-Rate steigen, d.h. die Größe der zurückgegebenen Intervalle müsste zunehmen. Betrachtet man aber die Messergebnisse, sinkt diese sogar bei Verwendung von $m = 5$ statt $m = 4$ (also der Shifting-Hashfunktion) oder bleibt zumindest konstant. Diese Theorie wäre damit auch widerlegt.

Um mehr über die Auswirkungen der q-Gramm-Hashfunktionen zu erfahren, hilft es einen Blick in die einzelnen Band-Hashtabellen zu werfen. Für jedes Fenster wird genau ein Paar aus Band-Hash und Fensterindex in jeder Band-Hashtabelle abgelegt. Dabei können je nach LSH-Konfiguration mehr oder weniger viele Kollisionen

[6] Dies ist die Anzahl der 16-Gramme in einem Read der Länge 100.

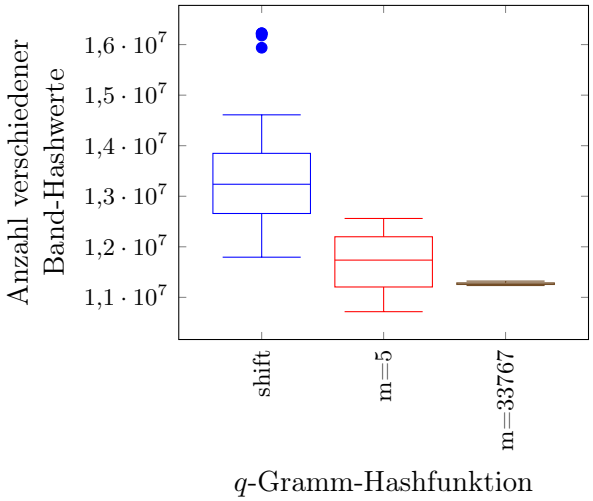


Abbildung 5.2: Anzahl verschiedener Band-Hashwerte in den einzelnen Band-Hashtabellen. Diese Messergebnisse beziehen sich nur auf den LSH-Index und sind damit unabhängig von den Read-Datensätzen.

auftreten. Die Boxplots in Abbildung 5.2 zeigen, wie viele unterschiedliche Band-Hashwerte in den einzelnen Band-Hashtabellen gespeichert wurden bzw. wie deren Verteilung aussieht. Betrachtet man jeweils den Median, fällt auf, dass die Anzahl der verschiedener Band-Hashwerte bei der Shifting-Hashfunktion am höchsten und bei $m = 33767$ am geringsten ist. Wie bereits vermutet, vergrößert also $m = 5$ und besonders $m = 33767$ die Anzahl der Kollision der q-Gramm-Hashfunktion. Darüber hinaus zeigt sich, dass die Streuung der Werte bei der Shifting-Hashfunktion am größten ist, während bei $m = 33767$ alle Band-Hashtabellen fast gleich viele verschiedene Band-Hashwerte beinhalten. Dabei enthält die kleinste Band-Hashtabelle der Shifting-Hashfunktion mehr Werte als die größte Tabelle von $m = 33767$. Doch offensichtlich wirkt sich dies bei $m = 33767$ nicht negativ auf die false-positive-Rate aus. Letztendlich bleibt offen, wodurch genau die besseren Ergebnisse erzielt werden.

Als weiteres Experiment könnte man eine zirkuläre Shifting-Hashfunktion verwenden, bei der z.B. in jedem Schritt um 10 Bits zirkulär nach links geschoben wird. Bei 16-Grammen wird so weiterhin das gesamte 32-Bit-Wort gefüllt, im Gegensatz zu $m = 5$ oder $m = 33767$ treten aber keine Kollisionen auf. Trotzdem findet im Vergleich zur normalen Shifting-Hashfunktion eine Durchmischung der Basenreihenfolge des q-Gramms statt. In C++ könnte diese Hashfunktion wie folgt realisiert werden, wobei s ein konstanter Parameter ist:

```
qGramHash = (qGramHash >> (32-s)) ^ (qGramHash << s) ^ nextChar;
```

Das zirkuläre Verschieben wird dabei durch den zusätzlichen Linksshift realisiert. Der Parameter s sollte als Vielfaches von zwei gewählt werden (da pro Zeichen im q-Gramm zwei Bits benötigt werden). Darüber hinaus müssen $\frac{32}{2}$ und $\frac{s}{2}$ teilerfremd sein, da sonst nicht alle Stellen des Hashwertes beschrieben werden. Für $s = 2$ ergibt sich die gewöhnliche Shifting-Hashfunktion. Die Hoffnung wäre, dass z.B. für $s = 10$ die Anzahl unterschiedlicher Band-Hashwerte so hoch ist wie bei $m = 4$, aber

trotzdem genau so gute oder bessere Qualitätswerte als bei $m = 5$ oder $m = 33767$ erreicht werden. An dieser Stelle besteht durchaus noch Forschungsbedarf.

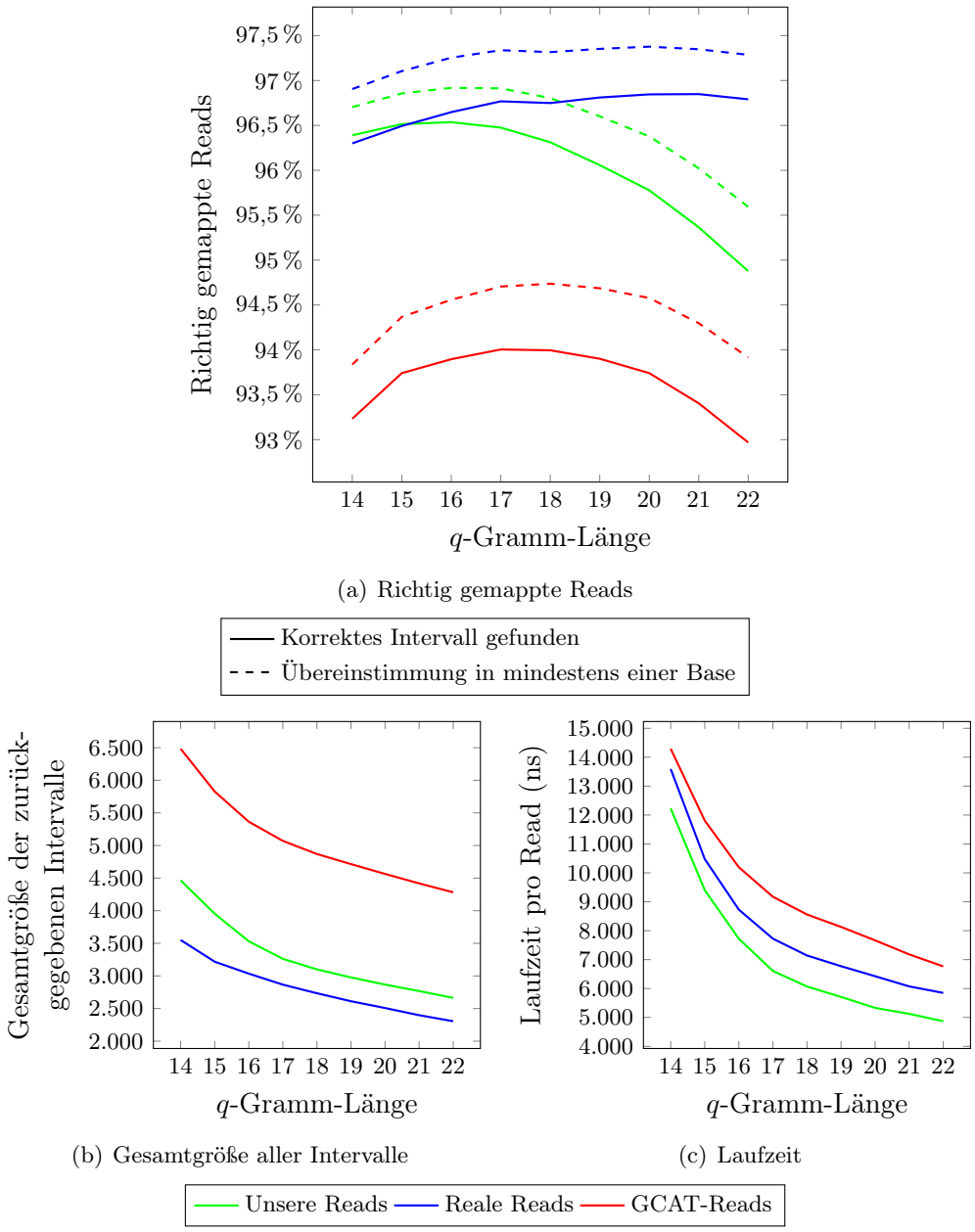
Durch die Verwendung einer komplexeren Hashfunktion steigt die Laufzeit pro Read geringfügig an (siehe rechtes Diagramm in Abbildung 5.1). Dies ist besonders bei Standard-Hashfunktion der C++-Bibliothek zu sehen. Die Unterschiede der multiplikativen Hashfunktionen im Vergleich zur Shifting-Hashfunktion sind aber so gering, dass man bedenkenlos eine multiplikative Hashfunktion verwenden kann. Neben der Laufzeit des Mappings pro Read steigt auch die Laufzeit für die Indexerstellung. Der Anstieg ist aber ähnlich stark bzw. schwach wie bei Laufzeit pro Read und deshalb nicht in den Diagrammen gezeigt.

Zusammenfassend kann man festhalten, dass die Verwendung einer multiplikativen Hashfunktion mit $m = 5$, $m = 19937$ oder $m = 33767$ am sinnvollsten ist. Bei *std::hash* steigt die Laufzeit merklich an, bei der Shifting-Hashfunktion werden die Ergebnisse qualitativ schlechter. Die Wahl von $m = 33767$ als Standardparameter ist also durchaus sinnvoll.

q-Gramm-Länge Da ein 32-Bit-Wort nur q-Gramme der Länge kleiner gleich 16 kollisionsfrei abbilden kann, wurde für das Variieren der q-Gramm-Länge die Länge eines q-Gramm-Hashwertes auf 64 Bit erhöht. Die Diagramme in Abbildung 5.3 zeigen die Anzahl korrekt gefundener Positionen und Intervalle sowie die Laufzeit pro Read als auch die durchschnittliche Gesamtgröße der zurückgegebenen Intervalle pro Read.

Bezüglich der Laufzeit pro Read und der Anzahl zu berechnender Spalten des Aligners ist das Ergebnis eindeutig: Bei allen Datensätzen verringert sich die Laufzeit bei längeren q-Grammen. Der Grund dafür liegt vermutlich darin, dass bei längeren q-Grammen weniger Treffer in den Band-Hashtabellen gefunden werden. Die Sortierung der Fensterindizes geht dadurch schneller (siehe Abbildung 5.3(b)). Darüber hinaus werden weniger Intervalle an den Aligner übergeben (siehe Abbildung 5.3(c)). Allerdings geschieht dies auf Kosten der Qualität (siehe Abbildung 5.3(a)). Werden bestimmte Intervalle durch Erhöhung der q-Gramm-Länge nicht mehr zurückgegeben, könnten unter ihnen auch Intervalle sein, die eigentlich korrekt gewesen wären. Dieser Effekt ist besonders bei unseren Reads sehr stark. Hier liegt das Optimum bei einer q-Gramm-Länge von 16 bis 17 Basenpaaren, die Anzahl richtig gemappter Reads fällt danach parabelförmig ab. Im Gegensatz dazu steigt die Qualität der zurückgegeben Intervalle bei realen Reads noch bis zu einer q-Gramm-Länge von 21. Dies könnte zum einen daran liegen, dass die Fehler (bzw. unbekannte Varianten) in echten Reads nicht gleichverteilt sind, sondern eher am Ende der Reads oder generell stark gehäuft vorkommen. Beträgt der Abstand zwischen zwei Fehlern beispielsweise 18 Basenpaare, so bilden q-Gramme der Länge $q \leq 18$ diesen Bereich noch ab, während bei längeren q-Grammen die Information zwischen den Fehlern ungenutzt bleibt. Eine andere Erklärung wäre, dass die Anzahl an Fehlern bei den realen Reads schlichtweg geringer ist, wodurch es letztendlich zu denselben Effekten kommt.

Bei den GCAT-Reads schneidet VATRAM erneut relativ schlecht ab: Der Anteil gefundener Reads ist am geringsten und die Laufzeit pro Read sowie die Gesamtgröße aller zurückgegebener Intervalle am höchsten, obwohl sie durchschnittlich genauso viele Fehler enthalten wie unsere Reads. Warum die GCAT-Reads so schlecht ab-



(a) Richtig gemappte Reads

(b) Gesamtgröße aller Intervalle

(c) Laufzeit

Abbildung 5.3: Variation der q-Gramm-Länge. Die gestrichelten Linien im ersten Diagramm zeigen den Anteil korrekt gefundener Positionen (Übereinstimmung an mindestens einer Base), die durchgezogenen Linien den Anteil korrekt gefundener Intervalle (d.h. das tatsächliche Read-Intervall liegt innerhalb des zurückgegebenen Intervalls).

schneiden, werden wir in Abschnitt 5.6 noch genauer analysieren. Die qualitativ besten Ergebnisse werden bei einer q-Gramm-Länge von 18 erreicht.

Die Schlussfolgerung, die man aus diesem Experiment ziehen kann, ist, dass es sich durchaus lohnt, längere q-Gramme zu verwenden. Ein guter Kompromiss wäre die q-Gramm-Länge von 16 auf 18 zu erhöhen. Abhängig vom Datensatz könnten sich dadurch die Ergebnisse qualitativ geringfügig verschlechtern (unsere Reads) oder aber auch verbessern (reale Reads und GCAT-Reads). Die Gesamtlaufzeit pro Read würde sich um 10 bis 20% verringern. Alternativ könnte man auch zwei Indizes erstellen (der eine mit kleinem, der andere mit großen q), um so die Güte bei einem unbekanntem Datensatz zu maximieren.

Gapped q-grams Bisher wurden nur sogenannte vollständige q-Gramme ohne Lücken betrachtet. Bei sogenannten lückenhaften q-Grammen (engl. *gapped q-grams*) werden einige Stellen des q-Gramms nicht berücksichtigt (siehe auch Abschnitt 4.1.2). Ein solches q-Gramm könnte beispielsweise wie folgt aussehen:

$$\#\#\#\#-\#$$

Bei diesem q-Gramm der Länge 6 wird das fünfte Zeichen stets ignoriert und für die Berechnung des Hashwertes nicht verwendet. Prinzipiell stellt sich die Frage, ob lückenhafte q-Gramme bessere Ergebnisse erzielen als q-Gramme ohne Lücken. Bei fehlerlosen Reads verringert sich die Anzahl der q-Gramme um eins, wenn zusätzlich eine Lücke ins q-Gramm eingebaut wird, weil dadurch die Länge des q-Gramms zunimmt. Selbst bei Reads mit genau einem SNP-Fehler sinkt die Anzahl der mit dem Original übereinstimmenden q-Gramme um eins: Links vor dem Fehler verringert sich die Anzahl um eins, genauso wie dahinter. Im Gegenzug kann ein q-Gramm so über den Fehler gelegt werden, dass Lücke und Fehler übereinander liegen. Auch wenn die Gesamtzahl der q-Gramme um eins sinkt, kann das lückenhafte q-Gramm einen Vorteil bringen, da es im Read einen Zusammenhang zwischen dem Bereich vor dem Fehler und dem Bereich hinter dem Fehler herstellt, die bei vollständigen q-Grammen getrennt wären. Folgendes Beispiel soll dies verdeutlichen:

5.3.1 Beispiel. In der unten stehenden Tabelle sind zwei Referenzausschnitte sowie ein Read darstellt. Der Read passt zur ersten Referenz, wobei die fünfte Base des Reads einen Fehler enthält (T statt G). Die Edit-Distanz zwischen Referenz 2 und dem Read ist deutlich höher. Betrachtet man vollständige 3-Gramme, stellt man fest, dass der Read und die beiden Referenzen jeweils vier gemeinsame q-Gramme besitzen (blau dargestellt), obwohl der Read-String deutlich besser zur ersten Referenz passt. Wird bei dem 3-Gramm eine Lücke hinter der zweiten Position eingebaut, so stimmen Read und Referenz 2 nur in zwei q-Grammen überein. Die richtige Referenz 1 hat hingegen 3 gemeinsame q-Gramme mit dem Read und wird somit bei der Suche bevorzugt. In diesem Beispiel bringt das lückenhafte q-Gramm somit ein Vorteil beim Readmapping.

Referenz 1:	ACGTTTGCA
q-Gramm ###:	ACG, CGT, GTT, TTT, TTG, TGC, GCA
q-Gramm ##-#:	AC-T, CG-T, GT-T, TT-G, TT-C, TG-A
Referenz 2:	TGCAAACGT
q-Gramm ###:	TGC, GCA, CAA, AAA, AAC, ACG, CGT
q-Gramm ##-#:	TG-A, GC-A, CA-A, AA-C, AA-G, AC-T
Read:	ACGTGTGCA
q-Gramm ###:	ACG, CGT, GTG, TGT, GTG, TGC, GCA
q-Gramm ##-#:	AC-T, CG-G, GT-T, TG-G, GT-C, TG-A

Noch interessanter wird es bei Reads mit zwei eng zusammen liegenden SNP-Fehlern. Während ein gewöhnliches q-Gramm den Bereich zwischen den Fehlern nicht abdecken kann, kann sich ein lückenhaftes q-Gramm so über die Fehler legen, dass es auch im Bereich der Fehler übereinstimmende q-Gramme zwischen Read und Original gibt. Folgendes Beispiel soll dies verdeutlichen:

Angenommen, zwischen zwei Fehlern liegen 15 Basenpaare und die q-Gramm-Länge beträgt 16. In dem Bereich zwischen den beiden Fehlern hat kein q-Gramm eine Übereinstimmung mit dem Original. Fügt man aber in der Mitte des 16-Gramms eine Lücke ein, kann jeweils im Bereich der Fehler das q-Gramm so positioniert werden, dass die Fehler in der Mitte des q-Gramms liegen. Es ergeben sich somit zwei zusätzliche q-Gramme im Bereich der Fehler. Im Bereich links vom ersten Fehler und rechts hinter dem zweiten Fehler verringert sich die Anzahl der passenden q-Gramme wiederum jeweils um eins (wegen der Zunahme der Gesamtlänge des lückenhaften q-Gramms). Die Gesamtanzahl der q-Gramme, die mit dem ursprünglichen Text übereinstimmen, bleibt somit bei Verwendung des lückenhaften q-Gramms konstant. Darüber hinaus beinhalten die lückenhaften q-Gramme auch Information über den Bereich zwischen den Fehlern. Dieser wurde bei Verwendung des gewöhnlichen q-Gramms nicht abdeckt, bzw. zeigte keine Übereinstimmung mit dem Original.

Bei drei oder mehr Fehlern kann sich dieser Effekt noch verstärken. Die beschriebenen möglichen Vorteile von lückenhaften q-Grammen beziehen sich nur auf SNP-Fehler. Gegen Indel-Fehler bringen lückenhafte q-Gramme keinen Vorteil. Indel-Fehler kommen allerdings im Vergleich zu SNP-Fehlern auch deutlich seltener vor (siehe Abschnitt 5.5).

Als Basis für die Experimente wurde ein q-Gramm der Länge 16 verwendet, bei dem zusätzlich Lücken eingebaut wurden. Da die Anzahl verschiedener lückenhafter q-Gramme exponentiell mit der Anzahl der Lücken ansteigt (die Anzahl beträgt $\binom{15+k}{k}$) und die Laufzeit pro Experiment bei drei bis fünf Minuten liegt, war es nicht möglich, alle Kombinationen für beispielsweise $k \leq 4$ auszuprobieren. Daher wurde eine kleine Auswahl von q-Grammen manuell gewählt. Um die Notation zu vereinfachen, wurde den getesteten q-Grammen jeweils eine ID zugewiesen (siehe Tabelle 5.2). q-Gramm A entspricht dabei dem vollständigen q-Gramm der Länge 16.

Bei q-Gramm B wurde eine längere Lücke kurz vor Ende des q-Gramms eingebaut. Die q-Gramme C bis E besitzen eine Lücke unterschiedlicher Länge in der Mitte. Bei den q-Grammen F und G wurden die Lücken gleichmäßig verteilt. Die q-Gramme H bis L enthalten eine unregelmäßige Verteilung der Lücken, die teilweise zufällig erzeugt wurde.

ID	# Lücken	Gapped q-Gram
A	0	`################`
B	4	`##############--##`
C	1	`########-########`
D	2	`########-########`
E	4	`########--########`
F	4	`###-###-####-###-###`
G	7	`##-##-##-##-##-##-##-##`
H	5	`####-##-###-##-####`
I	4	`####-####-##-####-##`
J	5	`#-##-####-#####-####`
K	7	`#-##-####-##-#-##-####`
L	8	`#-###-#-###-##-#-###-#-#`
M	5	`####-#-####-##-#####`

Tabelle 5.2: Die getesteten lückenhaften q-Gramme und ihre IDs.

Das q-Gramm `####-#-####-##-#####` (ID=M) wurde bewusst gewählt. Es kann besonders viele Abstände zwischen zwei Fehlern abdecken: Die doppelte Lücke deckt direkt aufeinander folgende SNP-Fehler ab. Die Struktur `-#-` deckt einen Abstand von zwei Basen ab. Genauso finden sich Stellen für sämtliche Abstände bis neun sowie für einen Abstand von elf. Bei größeren Abständen gibt es hinreichend viele Positionen, bei denen nur einer der beiden Fehler abgedeckt wird, d.h. das q-Gramm befindet sich links vom rechten oder rechts vom linken Fehler.

Die Qualität der Ergebnisse hängt stark vom betrachteten Datensatz ab (siehe erstes Diagramm in Abbildung 5.4. Bei den GCAT-Reads verbessert sich der Anteil der Reads, für die ein passendes Intervall gefunden wurde, von 94,3% beim vollständigen q-Gramm A um bis zu 1,2%-Punkte auf 95,5% beim bewusst gewählten q-Gramm M. Im Gegensatz dazu sinkt der Anteil korrekt gemappter Reads bei realen Reads von 96,4% auf $96,2\%$. Besonders groß sind die Unterschiede beim q-Gramm L: Die Ergebnisse der GCAT-Reads verbessern sich um 0,9%-Punkte, während sich die Qualität bei den realen Reads um dieselbe Prozentzahl verschlechtert. Die Ursache könnte die mit $k = 8$ recht große Anzahl an Lücken im q-Gramm L sein, denn ähnliche Ergebnisse zeigen sich auch beim q-Gramm G und K, die jeweils sieben Lücken beinhalten. Bei unseren Reads zeigt sich eine geringfügige Verbesserung der Qualität (etwa 0,3%-Punkte) bei Verwendung der richtigen q-Gramme (z.B. M oder H).

Das q-Gramm B scheint ein Beispiel für ein schlechtes q-Gramm zu sein: Bei allen Datensätzen sinkt der Anteil der korrekt gemappten Reads. Insgesamt schneidet das q-Gramm M am besten ab. Dieses wurde deshalb als Standardparameter für die LSH-Experimente gewählt. Da das q-Gramm M bei entsprechenden Datensätzen schlechtere Ergebnisse erzeugt als ein vollständiges q-Gramm, verwendet VATRAM standardmäßig keine lückenhaften q-Gramme.

Generell verbessern lückenhafte q-Gramme die Laufzeit. Dabei reduziert sich nicht nur die Laufzeit für die LSH-Anfrage (wie im zweiten Diagramm von Abbildung 5.4 gezeigt), sondern etwa auch im gleichen Maße die Gesamtlänge der zurückgegebenen Intervalle und damit die Laufzeit für den Aligner. Besonders gut schneiden

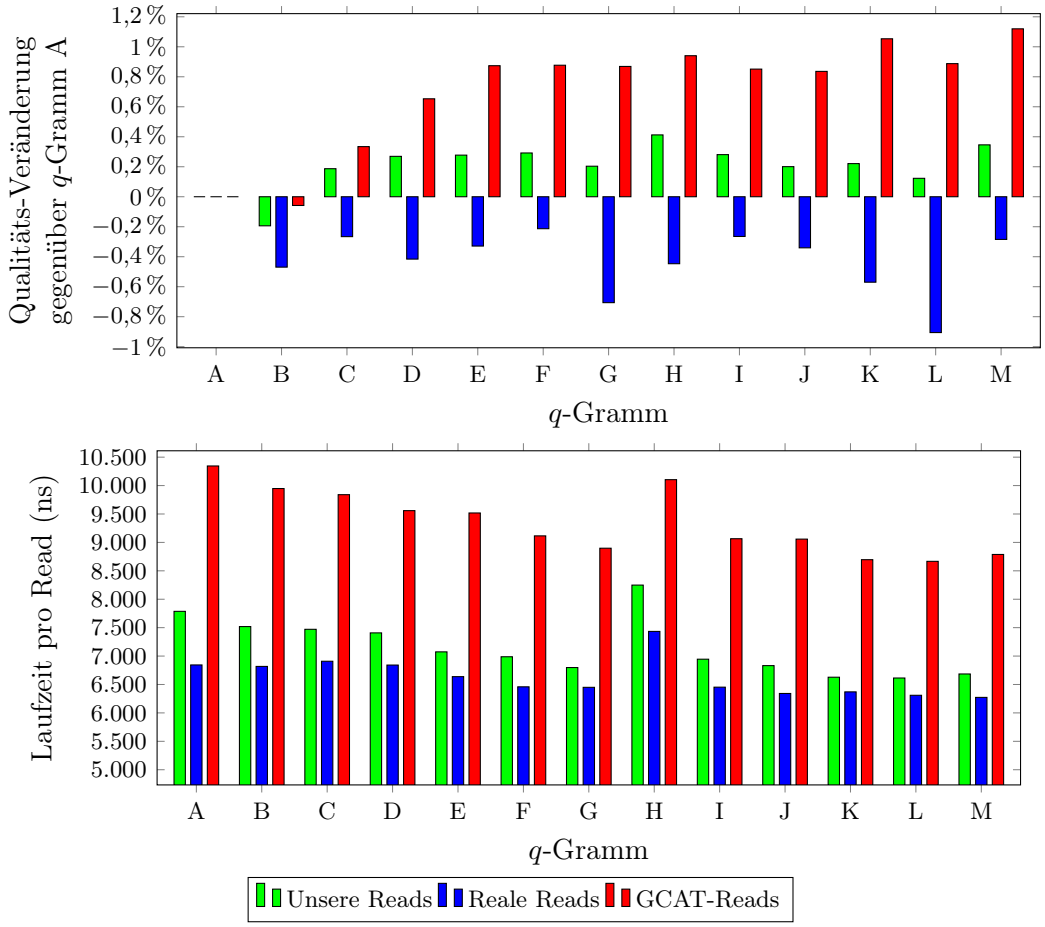


Abbildung 5.4: Ergebnisse für verschiedene lückenhafte q-Gramme. Das erste Diagramm zeigt, um wie viele %-Punkte sich der Anteil der Reads verändert hat, für die ein korrektes Intervall gefunden wurde. Das zweite Diagramm zeigt die durchschnittliche Laufzeit pro Read für eine LSH-Anfrage.

die q-Gramme K bis M ab. Die Laufzeitreduktion ist dabei beim GCAT-Datensatz am stärksten ausgeprägt. Auch bei den realen Reads zeigt sich noch eine minimale Verbesserung der Laufzeit.

Besonders negativ fällt das q-Gramm H auf, da es das einzige getestete q-Gramm ist, bei dem die Laufzeit größer ist als beim vollständigen q-Gramm A. Diese Anomalie ist auf einen Fehler bei der Eingabe der q-Gramme für das Experiment zurückzuführen: Während alle anderen q-Gramme genau 16 lesende Stellen besitzen, basiert q-Gramm H auf einem gewöhnlichen 15-Gramm. Wie wir bereits in Abbildung 5.3 gesehen haben, steigt die Laufzeit bei Verringerung der q-Gramm-Länge drastisch an. An dieser Stelle würde man erwarten, dass sich auch bei dem Anteil korrekt gemappter Reads eine Verschlechterung bei q-Gramm H gegenüber dem q-Gramm A zeigt. Umso überraschender ist es, dass das q-Gramm H dennoch beinahe genauso gute Ergebnisse wie das q-Gramm M erzeugt hat, welches bei dem Qualitätstest am besten abgeschnitten hatte. Die Position der Lücken im q-Gramm H müssen also anscheinend recht gut gewählt worden sein.

Bisher wurden die q-Gramm-Länge und Lücken im q-Gramm nur getrennt voneinander analysiert. Beim folgenden Experiment sollen nun auch längere lückenhafte q-Gramme getestet werden. Um die Datenmenge gering zu halten, benutzen wir als Basis ausschließlich q-Gramm M, da es bisher die besten Resultate lieferte, sowie q-Gramm A als Referenz, um die Ergebnisse vergleichen zu können. Zur Verlängerung des q-Gramms M werden vorne und hinten #-Zeichen (d.h. lesende Stellen) angehängt, sodass sein Kern mit der entsprechenden Lückensequenz gleich bleibt. Diese Form der Verlängerung eines q-Gramms wird im folgenden als erweitertes q-Gramm bezeichnet.

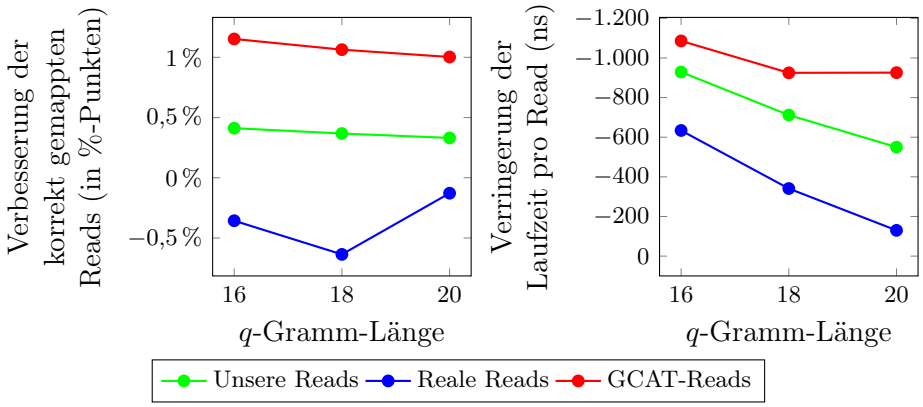


Abbildung 5.5: Veränderung der Mapping-Qualität und der Laufzeit bei Verwendung des q-Gramm-Musters M im Vergleich zum vollständigen q-Gramm A.

Die Diagramme in Abbildung 5.5 zeigen wie sich Güte und Laufzeit bei verschiedenen q-Gramm-Längen verändern, wenn das erweiterte q-Gramm M statt eines vollständigen q-Gramms verwendet wird. Bezüglich der Güte bleiben die Ergebnisse bei unseren Reads und den GCAT-Reads konstant (die Abweichungen betragen weniger als 0,1%-Punkte), d.h. die q-Gramm-Länge und das Muster der Lücken im q-Gramm verhalten sich weitestgehend unabhängig voneinander. Bei den realen Reads

liefern die lückhaften q-Gramme bei allen getesteten q-Gramm-Längen schlechtere Ergebnisse als die jeweiligen vollständigen q-Gramme.

Der Effekt der Laufzeitreduktion durch lückenhafte q-Gramme nimmt tendenziell mit zunehmender q-Gramm-Länge ab, insbesondere bei den realen Reads. Bei den GCAT-Reads liegt die Laufzeit von q-Gramm-Muster M etwa konstant 1000 ns unter der Laufzeit des vollständigen q-Gramms. Hier bringen lückenhafte q-Gramme mit Längen von 18 lesenden Positionen besonders viele Vorteile, wenn die Ergebnisse aus Abbildung 5.3 berücksichtigt werden. Die realen Reads profitieren jedoch in keinem Fall von den lückenhaften q-Grammen. Bei $q = 20$ sinkt der Laufzeitvorteil auf 130 ns, das heißt die Laufzeit verringert sich lediglich um 3%. Diese Verbesserung ist somit irrelevant, die Vorteile der lückhaften q-Gramme sind bei $q = 20$ also nicht mehr vorhanden.

Letztendlich kann man aus den q-Gramm-Experimenten zusammenfassen, dass die Wahl der q-Gramm-Länge und die Entscheidung, ob lückenhafte q-Gramme verwendet werden sollen, abhängig vom Datensatz getroffen werden muss. Bei einem unbekannten Datensatz kann es durchaus Sinn ergeben, verschiedene Konfigurationen auszuprobieren, um so eventuell qualitativ bessere Mappings in kürzerer Zeit zu generieren.

5.3.2.2 Fenstergröße, -abstand und Anzahl der Bänder

Die Parameter Fenstergröße und Fensterabstand hängen eng zusammen und werden deshalb gemeinsam in diesem Abschnitt behandelt. Die Differenz aus Fenstergröße und -abstand bestimmt, um wie viele Basen sich zwei benachbarte Fenster überschneiden. Eine gewisse Überschneidung ist sinnvoll, da sonst q-Gramme im Übergangsbereich nicht berücksichtigt werden würden.

Wird der Abstand zwischen zwei Fenstern verringert, kann man allgemein davon ausgehen, dass sich die Ergebnisse verbessern, da es mehr Fenster gibt und somit die Wahrscheinlichkeit größer ist, dass ein Großteil des Reads vollständig in einem Fenster liegt. Durch die höhere Anzahl von Fenstern erhöht sich aber auch der Speicherbedarf um denselben Faktor. Um den Einfluss der Fensterabstände sinnvoll messen zu können, wurde die Anzahl der Bänder jeweils so verringert, dass der Speicherbedarf bei jedem Experiment gleich bleibt[7]. Aus diesem Grund wurden bei diesem Experiment alle drei Parameter gleichzeitig variiert. Die Ergebnisse sind in Abbildung 5.6 zu sehen.

Wie bereits vermutet, führen nicht überlappende Fenster zu qualitativ schlechten Ergebnissen, da ein Teil der q-Gramme der Referenz bei der Index-Erstellung nicht berücksichtigt wird. Für $q = 16$ reicht eine Überlappung von 15 Basen aus, um jedes q-Gramm der Referenz zu benutzen. Dementsprechend ist der Anteil der Reads, für die ein korrektes Intervall gefunden wurde, deutlich größer. Dies erklärt jedoch nicht, warum sich die Ergebnisse bis zu einer Überlappung von 25 Basen weiter verbessern und zwar für alle getesteten Fensterabstände (diese werden im Diagramm durch die Helligkeit der Kurven dargestellt). Vermutlich profitieren von der größeren Überlappung vor allem Reads, die genau in der Mitte von zwei benachbarten Fenstern

[7] Der Quotient aus Bandanzahl geteilt durch den Fensterabstand ist konstant gleich 0,32. Dieser Wert ergibt sich durch die oben festgelegten Standardparameter.

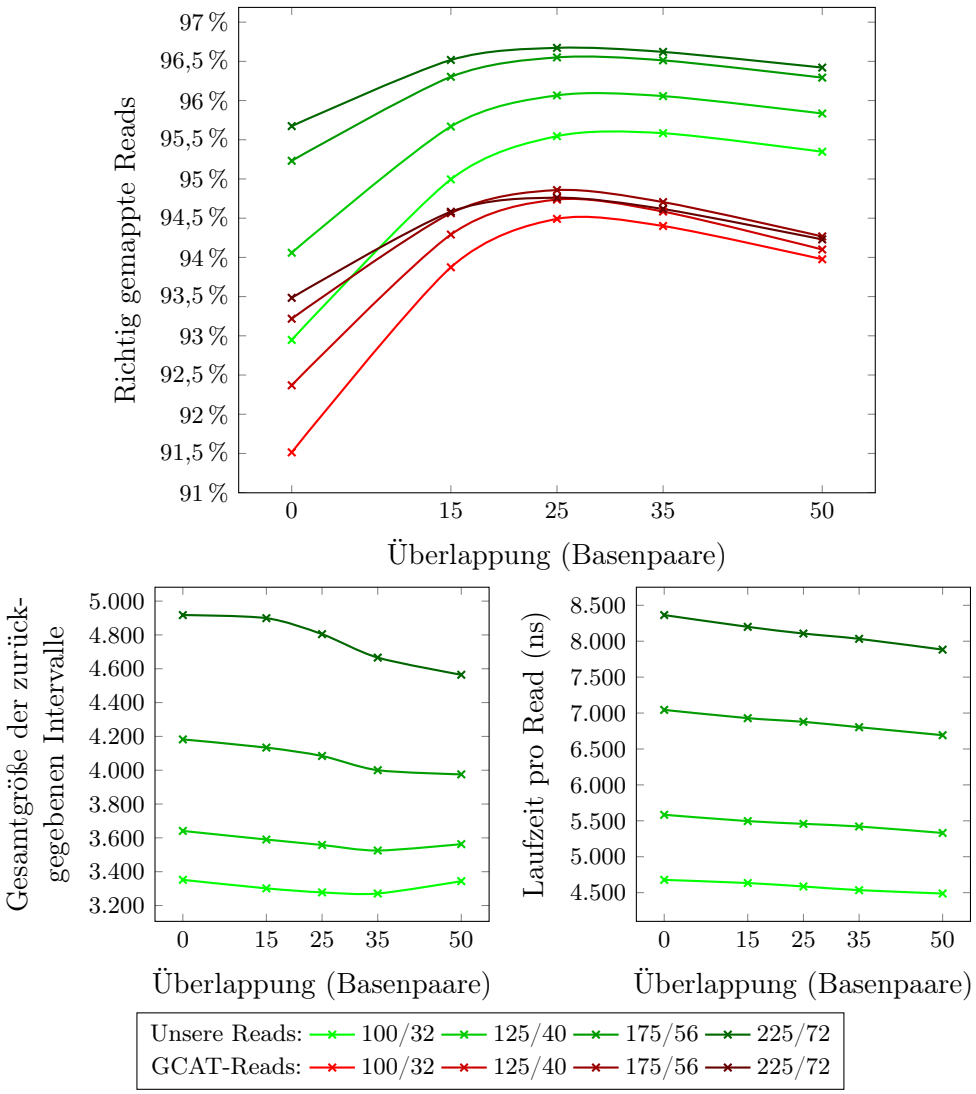


Abbildung 5.6: Variation der Fensterparameter. Die x-Achse zeigt jeweils um wie viele Basen sich zwei benachbarte Fenster überlappen. Die Angaben in der Legende beziehen sich auf den Fensterabstand (erster Wert) und die Anzahl der Bänder (zweiter Wert).

liegen. Bei Überlappungen von 35 und mehr Basenpaaren verschlechtern sich die Ergebnisse wieder. Dies könnte daran liegen, dass die Fensterlänge selbst (also die Summe aus Überlappung und Abstand) sehr viel größer wird als die Readlänge (100 Basen). Die Menge der q-Gramme wird damit entsprechend größer, sodass seltener die Minima von exklusiv veroderten q-Gramm-Mengen von Referenzfenster und Read kollidieren.

Gegen diese Theorie spricht scheinbar, dass der Anteil korrekt gemappter Reads mit zunehmender Fensterlänge steigt. Man muss dabei allerdings berücksichtigen, dass nicht nur die Fenster größer werden, sondern auch die Anzahl der Bänder zunimmt (siehe Legende unter den Diagrammen). Letzteres hat definitiv positiven Einfluss auf die Anzahl richtig positionierter Reads. Offensichtlich dominiert dieser positive Effekt gegenüber dem negativen Einfluss der längeren Fenster. Dies gilt aber nicht für

beliebig lange Fenster. Bei den GCAT-Reads kann man bei einem Fensterabstand von 225 Basen feststellen, dass die Ergebnisse wieder geringfügig schlechter werden als bei einem Abstand von 175. Insgesamt wirkt sich die Veränderung der Fensterabstände bzw. der Bänderanzahl bei den von uns generierten Reads besonders stark aus. Die Ergebnisse der hier nicht gezeigten realen Reads sind ähnlich zu den GCAT-Reads mit dem Unterschied, dass sie insgesamt bessere Werte erzielen.

Betrachtet man die beiden unteren Diagramme in Abbildung 5.6, stellt man fest, dass eine Erhöhung des Fensterabstandes und der Bandanzahl nicht nur Vorteile bringt. So steigt beispielsweise die durchschnittliche Suchzeit unserer Reads von 3400 ns bei 32 Bändern um etwa 50% auf bis zu 4900 ns bei Verwendung von 72 Bändern. Die Ursache hierfür ist vermutlich die höhere Anzahl von Bändern: Es muss auf mehr Band-Hashtabellen zugegriffen werden, wodurch mehr Fenster zurückgegeben werden. Der Aufwand für das Sortieren und Filtern der Fenster erhöht sich dadurch entsprechend.

Neben der Suchzeit steigt auch die Gesamtgröße der zurückgegebenen Intervalle um 75%. Dies liegt an den größeren Fenstern, wodurch auch die zurückgegebenen Intervalle größer werden, da der LSH-Algorithmus nicht feststellen kann, wo sich ein Read in einem Fenster befinden könnte (vgl. Abschnitt 4.1.9). Neben der Gesamtgröße der Intervalle liefern die LSH-Experimente auch die Anzahl an Intervallen. Diese sinkt bei einer konstanten Überlappung von 25 Basen von durchschnittlich 14,5 bei Verwendung von 32 Bändern auf ein Minimum von 13,9 bei 56 Bändern. Diese Zahlen bestätigen die gerade beschriebene Theorie der größeren Intervalle.

Bei der Wahl der Fensterparameter muss man sich fragen, ob einem qualitativ bessere Ergebnisse wichtiger sind als eine geringe Laufzeit. Bei VATRAM haben wir uns auf einen Kompromiss geeinigt und als Standardparameter einen Fensterabstand von 125 mit einer Überlappung von 15 Basen gewählt, sodass sich als Fensterlänge 140 Basenpaare ergeben. Die Anzahl der Bänder wurde später von 40 auf 48 angehoben, da der Speicherbedarf durch die SuperRank-Datenstruktur stark gesenkt werden konnte. Im Nachhinein hätte man als Überlappung auch 25 wählen können, da damit für alle Datensätze bessere Ergebnisse erzielt werden. Die Laufzeit bleibt davon unbetroffen, bzw. sinkt minimal.

5.3.2.3 Bandanzahl

Hier betrachten wir, wie sich die Anzahl an Bändern auf die Genauigkeit und die Laufzeit unseres Readmappers auswirken. Es lässt sich feststellen, dass bei steigender Bandanzahl die Genauigkeit erhöht wird. Des Weiteren zeigen wir die Unterschiede zwischen der Super-Rank-Datenstruktur, die eine erhebliche Reduktion des Speicherverbrauchs einführt, und die Collision-Free-Bandhash-Table(CFBHT), die eine Beschleunigung der Suchanfragen zur Folge hat. Generell lässt sich über die Daten sagen, dass die Super-Rank-Datenstruktur sehr viel Platz spart. Dies gilt generell bei allen Datensätzen (Real, Real-NoVariant, GCAT, OwnReads, OwnReads-NoVariant).

Grundsätzlich entsteht ein höherer Speicherverbrauch und eine höhere Suchanfragenzeit bei steigender Bandzahl. Die Suchanfragenzeit steigt linear.

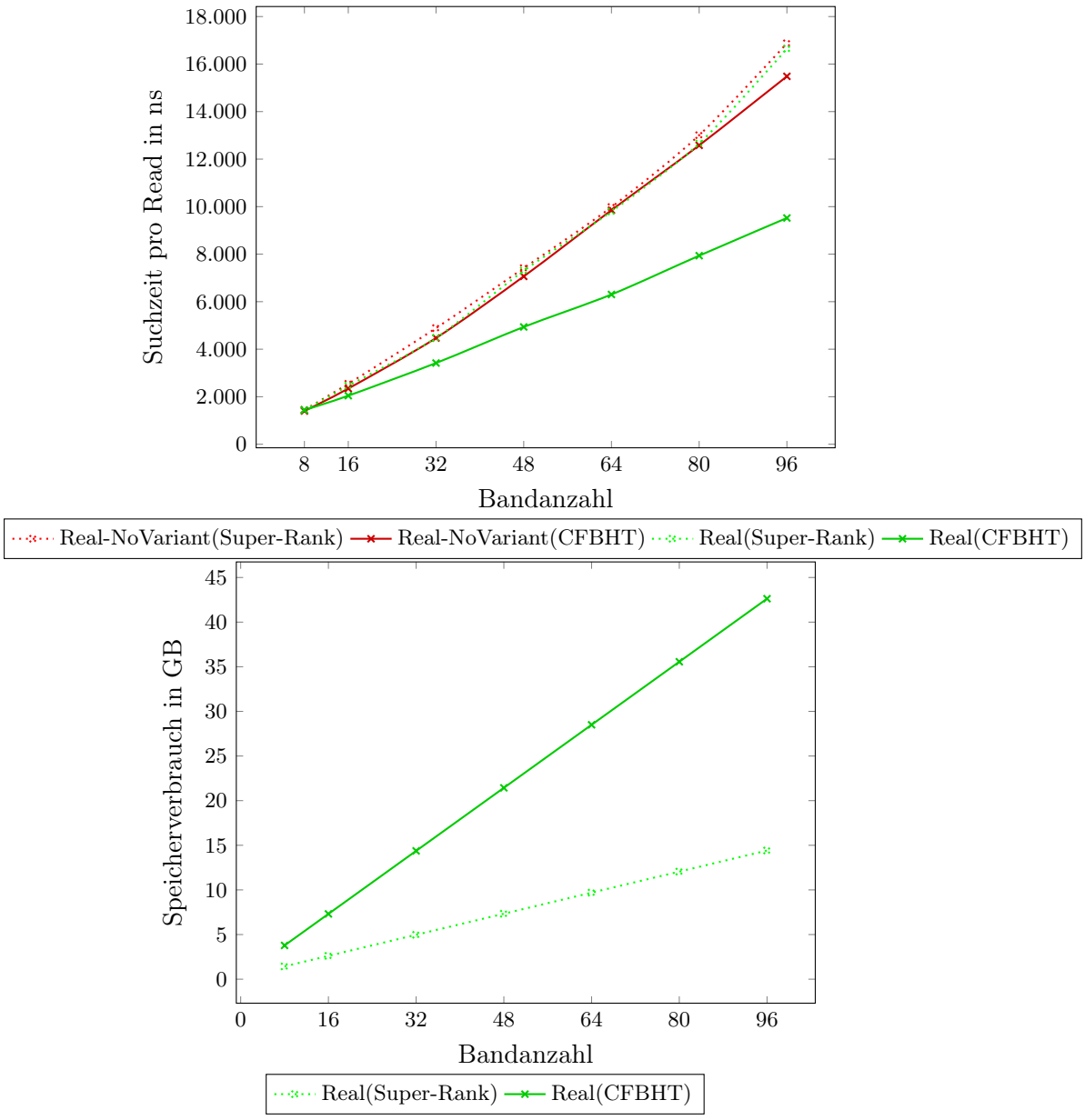


Abbildung 5.7: Speicherverbrauch und Suchzeit für Superrank (grün) und CFBHT (dunkelgrün) für echte reads.

In der Abbildung 5.7 erkennt man einen leicht stärkeren Anstieg als linear (rote Kurve stärker als linear).

Eine genaueres Mappen der Intervalle lässt sich jedoch auch vermerken (Abbildung 5.7).

Für die Collision-Free-Bandhash-Tablezeigt sich in Abbildung 5.7 eine Geschwindigkeitsverbesserung bezüglich der Suchanfagezeiten. Der Speicherverbrauch bleibt jedenfalls sehr hoch.

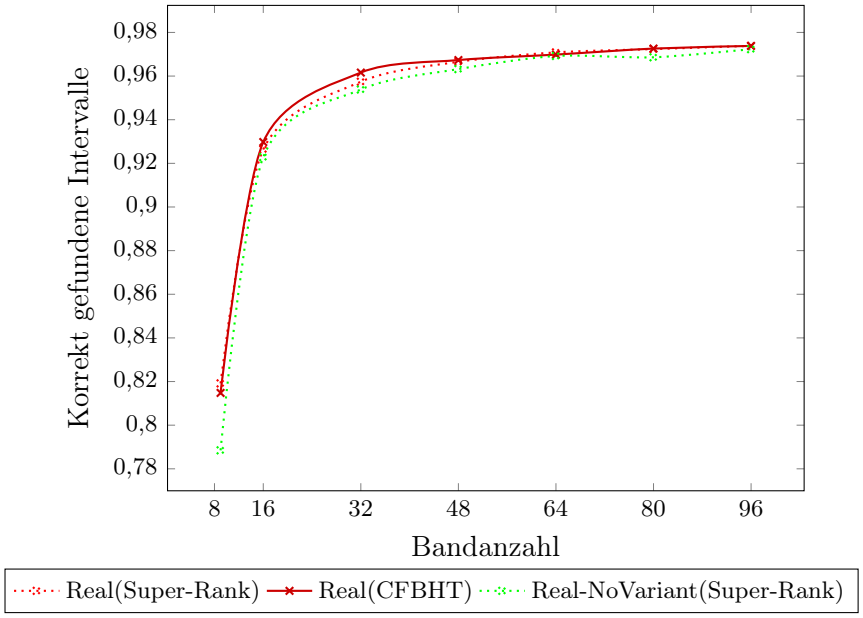


Abbildung 5.8: Prozent der korrekt gefundenen Intervalle.

Bei geringen Bandanzahlen werden mehr Reads in derselben Zeit verarbeitet, jedoch weniger Reads korrekt gemappt. Mit einer Bandanzahl von 8 werden nur 81% der realen Reads gefunden. Da der Speicherverbrauch jedoch nur 1.4 GB beträgt, ist dieses Ergebnis bereits als positiv zu werten. Bei unseren Reads (OwnReads) sind es 58%, bei GCAT 66%

Ist der Speicher jedoch verfügbar, zeigt sich bei der Verwendung der Collision-Free-Bandhash-Table eine Verbesserung. Mit der Collision-Free-Bandhash-Table lässt sich in vielen Fällen eine Halbierung der Suchzeit erreichen.

Eine Bandanzahl von 16 verbessert die Güte auf 93% (reale Reads). Bei unseren Reads sind es 86?%, bei GCAT 85%.

Mit 48 Bändern (dies entspricht dem Standard-Wert) werden 97% der Reads gefunden. Eine Verdopplung der Bandanzahl verbessert die Ergebnisse um 2%-Punkte. Bei höheren Bandanzahlen ist die Verbesserung nicht mehr so signifikant.

5.3.2.4 Bandgröße und Band-Hashfunktion

In diesem Abschnitt geht es um den Einfluss der Bandgröße und der Band-Hashfunktion auf die Qualität der Ergebnisse. Standardmäßig wird eine Bandgröße von eins verwendet. Dementsprechend wurde als Band-Hashfunktion die Identitätsfunktion gewählt, da sowohl die Signaturen als auch die Bänder 32 Bit groß sind. Das Diagramm in Abbildung 5.9 zeigt den Anteil der Reads, für die der LSH-Algorithmus ein korrektes Intervall zurückgegeben hat. Um die Ergebnisse vergleichen zu können, wurde nicht nur die Bandgröße sondern auch die Signaturlänge erhöht, sodass die Anzahl an Bänder und damit auch der Speicherbedarf konstant bleibt. Als Band-

Hashfunktion wurde analog zu den q-Grammen eine multiplikative Hashfunktion mit $m = 33767$ verwendet (siehe Gleichung 5.1).

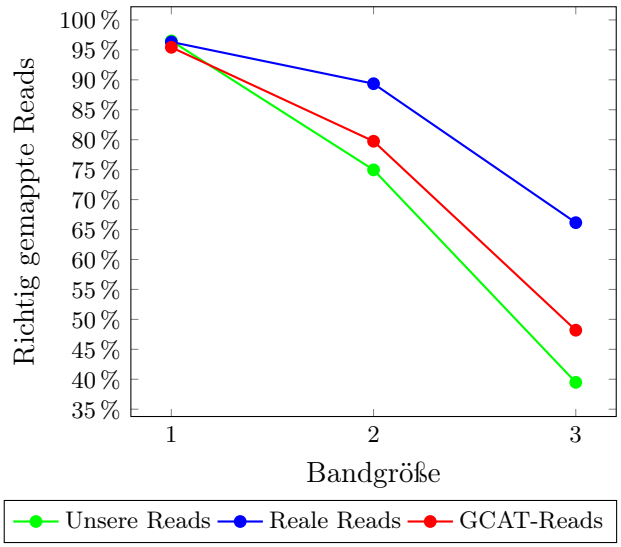


Abbildung 5.9: Anteil der Reads, für die ein korrektes Intervall bestimmt wurde, in Abhängigkeit von der Bandgröße. Die Anzahl an Bändern wurde jeweils konstant gehalten.

Im Gegensatz zu den vorherigen Experimenten zeigt sich hier ein klares Ergebnis: Die Verwendung von eins als Bandgröße liefert eindeutig die besten Resultate. Je nach Datensatz werden 95% bis 97% der Reads gefunden. Bereits bei einer Bandgröße von zwei sinkt diese auf etwa 90% bei den realen Reads. Unsere Reads und die GCAT-Reads schneiden noch deutlich schlechter ab (unter 80%). Bei einer Bandgröße von drei sind die Ergebnisse nochmals dramatisch schlechter.

Möglicherweise liegt das schlechte Abschneiden von Bandgröße zwei auch an der verwendeten Hashfunktion. Um dies auszuschließen wurde unter anderen ein Test mit 64-Bit-Bändern durchgeführt. Die Signaturen eines Reads bzw. eines Fensters bestehen aus 32 Bits. Sollen zwei Signaturen zu einem Band zusammengefügt werden, könnte man sie einfach aneinanderhängen, um so ein 64-Bit-Band zu erhalten. Bei diesem Verfahren geht keinerlei Information verloren, d.h. wenn es eine Hashfunktion gäbe, bei der eine Bandgröße von zwei einen Vorteil bringt, dann würde man diesen Vorteil insbesondere bei 64-Bit-Bändern beobachteten.

Die Experimente mit 64-Bit-Bändern zeigten aber keine Verbesserungen gegenüber der multiplikativen Hashfunktion mit $m = 33767$. Somit sollte stets eins als Bandgröße verwendet werden.

5.3.2.5 Parameter für die Variantenberücksichtigung

In unserer Implementierung des *Locality-Sensitive Hashing*s werden SNP-Varianten direkt berücksichtigt. Dazu werden alle q-Gramm-Kombinationen in einem Fenster gebildet und zur Indexdatenstruktur hinzugefügt. Dies führt bei einer großen Anzahl von SNP-Varianten zur kombinatorischen Explosion, die Zahl der hinzugefügten q-Gramme muss also begrenzt werden. Dazu wurden die beiden Parameter

`limit` und `limit-skip` eingeführt. `Limit` beschränkt die Anzahl der hinzugefügten q-Gramme. Werden mehr q-Gramm-Kombinationen erzeugt, werden alle weiteren verworfen. Es wird also eine zufällige Auswahl von `limit` vielen q-Grammen berücksichtigt. `Limit-skip` gibt eine generelle Grenze für die Zahl der Kombinationen vor. Wird diese für einen q-Gramm-Ausschnitt erreicht, werden gar keine q-Gramme für die Varianten hinzugefügt. Um die Auswirkungen von `limit` und `limit-skip` zu erfassen, wurden diese für reale und generierte Reads variiert. Auswirkungen auf die Qualität und Laufzeit sind in Abbildung 5.10 dargestellt.

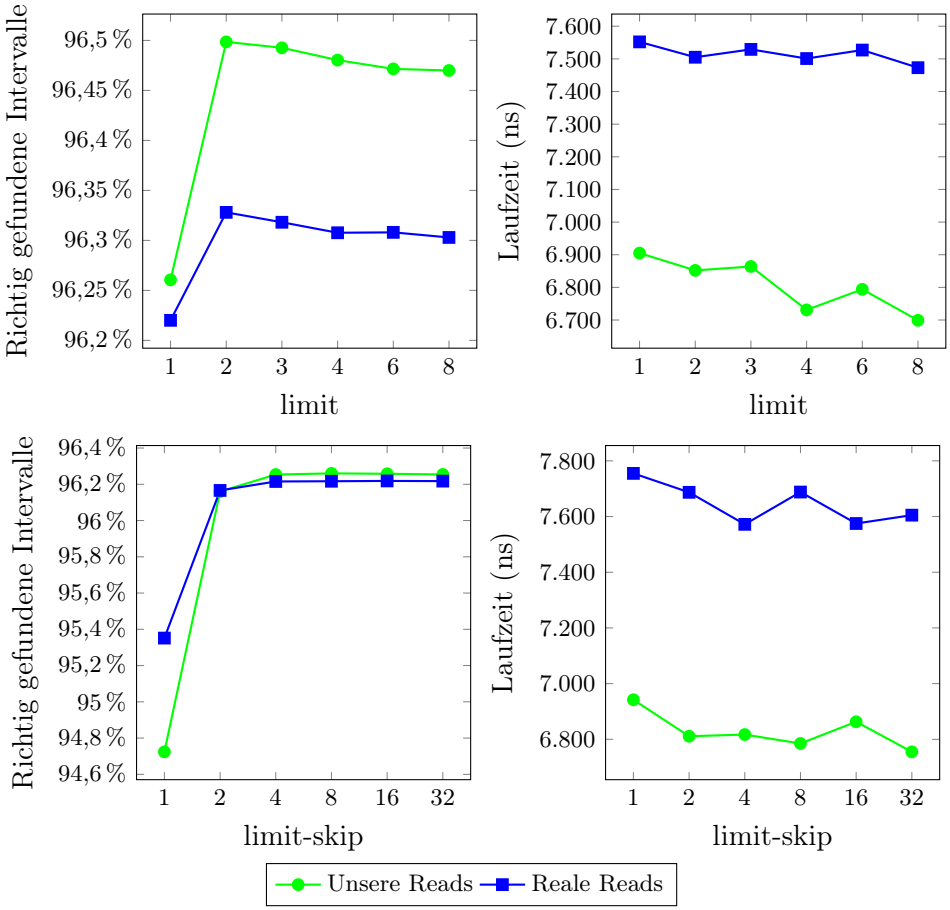


Abbildung 5.10: Einfluss der Parameter `limit` und `limit-skip` auf Qualität und Laufzeit des LSH.

Bei der Qualität wirkt sich vor allem der Wert 1 bei beiden Parametern negativ aus. Durch diese Einstellung werden SNP-Varianten komplett ignoriert. Im weiteren Verlauf ändern sich die Werte nur noch geringfügig. Bei `limit` ist eher eine Verschlechterung zu beobachten, wenn es zu groß gewählt wird. Aufgrund der Messungen wurde als Standardwert für `limit` 3 gewählt und für `limit-skip` ein Wert von 8.

5.3.2.6 Initialisierung der Hashfunktionen

Zur Berechnung der Signaturen werden die q-Gramm-Hashwerte eines Fensters oder eines Reads mit den konstanten Zufallswerten im Array *hashFunctions* exklusiv verodert (siehe Abschnitt 4.1.5.2). Diese konstanten Zufallswerte werden im Folgenden als Signatur-Hashwerte bezeichnet. Die Signatur-Hashwerte werden standardmäßig zufällig gewählt. Beim Min-Hashing sind wegen der Minimumsbildung vor allem die höchstwertigsten Bits der exklusiv veroderten Werte ausschlaggebend. Da bei der Signaturberechnung die Bitreihenfolge nicht durcheinander gewürfelt wird, sind somit auch bei den Signatur-Hashwerten die vorderen Bits entscheidend. Würden sich im Extremfall zwei Signatur-Hashfunktionen gleichen, wird in den entsprechenden Band-Hashtabellen dieselbe Zuordnung von Band-Hashwerten zu Fensterindizes gespeichert. Durch Weglassen einer der beiden Signaturen würde man damit weiterhin genauso gute Ergebnisse erzielen. Dieser Extremfall taucht in der Praxis allerdings nie auf. Durch die zufällige Wahl kann es jedoch durchaus passieren, dass sich zwei Signatur-Hashwerte in den vorderen Bits stark ähneln, wie folgende Rechnung zeigt:

Die Wahrscheinlichkeit, dass die ersten k Bits von zwei gleichverteilt zufälligen Bitstrings identisch sind, beträgt $\left(\frac{1}{2}\right)^k$. Sei S eine Menge von s unabhängigen Bitstrings. Es existieren $\binom{s}{2} = \frac{s(s-1)}{2}$ verschiedene Bitstringpaare $\{a, b\}$ mit $a, b \in S$. Die Wahrscheinlichkeit, dass kein Paar $\{a, b\}$ aus S existiert, dessen erste k Bits identisch sind, beträgt

$$\left(1 - \frac{1}{2^k}\right)^{\frac{s(s-1)}{2}}$$

. Als Signaturlänge wird von VATRAM standardmäßig $s = 48$ verwendet. Mit 68%-iger Wahrscheinlichkeit besitzen somit zwei dieser Hashwerte zehn gemeinsame Bits. Für elf gemeinsame Bits beträgt die Wahrscheinlichkeit 44%, für zwölf noch fast 25%. Das theoretisch erreichbare Minimum (das bei einer äquidistanten Verteilung der Hashwerte erreicht wird) liegt bei sechs gemeinsamen Bits. Daher ist es naheliegend, dass durch die deterministische Wahl der Signatur-Hashfunktionen möglicherweise bessere Ergebnisse erzielt werden.

Nichtsdestotrotz wäre es wünschenswert ein gewisses Maß an Zufall einzubauen, damit beispielsweise nicht alignierte Reads durch einen zweiten LSH-Index mit gleichen Parametern, aber anderem Zufalls-Seed, erneut verarbeitet werden können, um so gegebenenfalls weitere Reads zu finden. Bei Verwendung von deterministischen Signatur-Hashfunktionen würde sich stets dasselbe Mapping ergeben.

Die erste Idee ist, den 32- bzw. 64-Bit-Raum in s gleich große Teile zu teilen. Innerhalb dieser Teile wird dann der genaue Wert des jeweiligen Signatur-Hashwertes durch eine Gleichverteilung gewählt. Dieses Verfahren wird im Folgenden als Intervall-Methode bezeichnet. Formal ergibt sich folgende Formel für die Berechnung der Hashwerte: Sei m die Anzahl der Bits in einem Hashwert und $random(n)$ eine Funktion, die eine zufällig gleichverteilte ganze Zahl zwischen 0 und n zurückliefert.

$$h_i = \left\lfloor \frac{2^m}{s} \cdot i \right\rfloor + random\left(\left\lfloor \frac{2^m}{s} \right\rfloor\right)$$

Trotzdem kann es vorkommen, dass sich zwei Hashwerte stark ähneln, nämlich wenn der Hashwert h_i sich am rechten Rand seines Intervalls befindet und h_{i+1} an dem lin-

ken Rand des darauf folgenden Intervalls. Um diesem unerwünschten Effekt entgegen zu wirken, wurden die Signatur-Hashwerte innerhalb ihres Intervalls nicht gleichverteilt gewählt, sondern mittels einer symmetrischen Dreiecksverteilung erzeugt. Diese erreicht man einfach durch Addition zweier Gleichverteilungen, wie in folgender Formel gezeigt:

$$h_i = \left\lfloor \frac{2^m}{s} \cdot i \right\rfloor + random\left(\left\lfloor \frac{2^m}{2s} \right\rfloor\right) + random\left(\left\lfloor \frac{2^m}{2s} \right\rfloor\right)$$

. Dieses Verfahren nennen wir Dreiecks-Methode.

Abbildung 5.11 zeigt die Ergebnisse für die verschiedenen Initialisierungsverfahren. „Random“ steht dabei für die triviale, vollständig zufällige Berechnung der Signatur-Hashwerte.

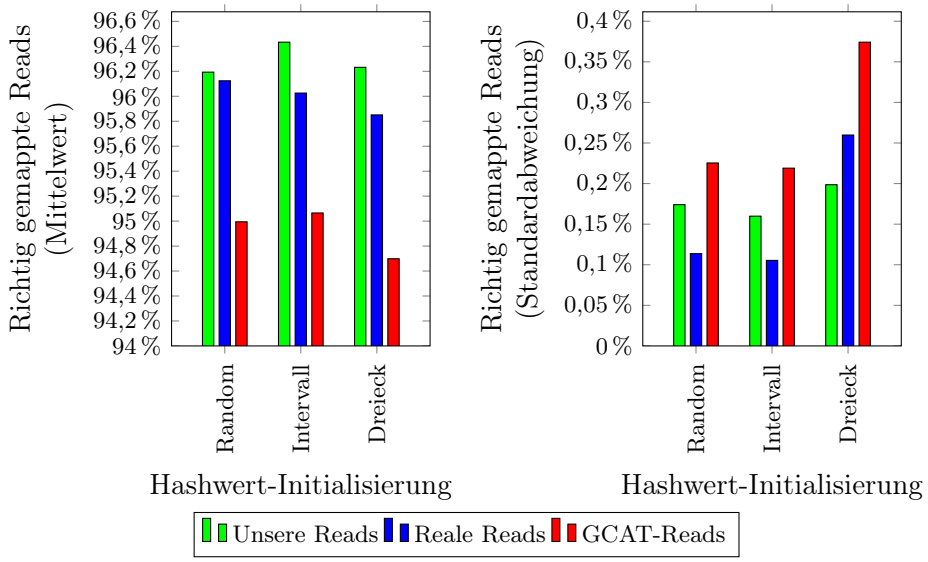


Abbildung 5.11: Verschiedene Initialisierungsverfahren für die Signatur-Hashwerte. Gemessen wurde der Anteil der Reads, für die ein korrektes Intervall gefunden wurde. Gezeigt sind die Mittelwerte und Standardabweichungen von fünf Experimenten.

Bei der Intervall-Methode wird die Mapping-Qualität bei unseren Reads und den GCAT-Reads gegenüber „Random“ geringfügig besser (0,24%-Punkte bzw. 0,07%-Punkte). Der Anteil der korrekt gemappten realen Reads verschlechtert sich dagegen um 0,1%-Punkte. Bei der Dreiecks-Methode verschlechtern sich die Ergebnisse aller drei Datensätze gegenüber der Intervall-Methode und sind mit Ausnahme unserer Reads schlechter als die Random-Methode. Dies widerspricht der oben geäußerten Vermutung, nach der die Dreiecks-Methode eigentlich am besten abschneiden müsste.

Das rechte Diagramm in Abbildung 5.11 zeigt die Standardabweichung der Qualitätswerte von fünf Versuchen, wenn die Zufallszahlen mit unterschiedlichen Seed-Werten erzeugt werden. Die Streuung der Ergebnisse bei „Random“ und „Intervall“ sind etwa gleich groß; bei der Dreiecks-Methode sind sie am größten. Dies ist sehr verwunderlich, da man erwarten würde, dass bei „Dreieck“ die Streuung am geringsten sein müsste. Schließlich ist die Wahl der Signatur-Hashwerte weniger zufällig als bei „Random“ oder „Intervall“. Warum die „Dreiecks“-Methode trotzdem solch stark

schwankende Ergebnisse liefert, bleibt unklar. Generell sind die Standardabweichungen so groß, dass die oben gemessenen Verbesserungen oder Verschlechterungen auch reine Zufallsprodukte sein könnten.

Bezüglich der Laufzeit pro Read und der durchschnittlichen Gesamtgröße der zurückgegebenen Intervalle zeigt sich keine signifikante Änderung bei den verschiedenen Initialisierungsmethoden.

Zusammenfassend kann man festhalten, dass die wohlüberlegten alternativen Initialisierungsmethoden keinen Vorteil und vielleicht sogar Nachteile bringen. Die Wahl von „Random“ als Standard-Initialisierungsmethode scheint somit angemessen zu sein. An dieser Stelle soll noch erwähnt werden, dass bei Verwendung der Shifting-Hashfunktion als q-Gramm-Hashfunktion (siehe Abschnitt 5.3.2.1) die Dreiecks-Methode sehr wohl Vorteile gegenüber „Random“ bringt. Es lohnt sich jedoch vielmehr, eine bessere q-Gramm-Hashfunktion zu verwenden.

5.3.3 Anfrage-Parameter

In diesem Abschnitt geht es um die Auswertung der Anfrage-Parameter. Werden Anfrage-Parameter geändert, ist es nicht nötig, einen neuen Index zu erstellen. Die Laufzeit für die Indexerstellung sowie der Speicherverbrauch bleiben somit konstant und sind deshalb für die folgenden Experimente irrelevant.

5.3.3.1 Maximale Anzahl zurückgegebener Fenster

Zur Optimierung des Speicherverbrauchs, und um Schranken zur Begrenzung der Laufzeit festlegen zu können, wurde ausgemessen, wie viele Buckets der Hashtabelle mehrere Fenster enthalten und vor allem wie hoch der Erwartungswert für die Anzahl der Fenster pro Bucket ist. Abbildung 5.12 zeigt, wie viele Buckets mit wie vielen Fenstern aufgetreten sind. Für die Messungen wurden ansonsten Standardparameter verwendet.

Bei den Buckets mit sehr vielen Fenstern gibt es nur geringe Varianz zwischen den einzelnen Read-Datensätzen. Die Mehrheit der Buckets enthält weniger als 1 000 Fenster. Mit steigender Fensterzahl fällt die Anzahl der Buckets sehr stark ab. Es gibt jedoch noch vereinzelte Buckets mit über 150 000 Fenstern. Diese stellen ein großes Hindernis für die Laufzeit dar, da sie einerseits sehr selten auftreten, andererseits sehr viel Zeit zur Verarbeitung benötigen. Um diesem Effekt entgegenzuwirken wurde die Anzahl der maximal zurückgegebenen Fenster begrenzt. Buckets mit mehr Fenstern werden verworfen.

Um herauszufinden ob es allgemein sehr viele Band-Hashtabellen gibt, die viele Ergebnisse liefern, wurde zusätzlich die Verteilung der Anzahl an Treffern auf die einzelnen Band-Hashtabellen gemessen. Dazu wurden für jeden Read die Anzahl an Treffern in jeder Band-Hashtabelle gezählt und diese Liste aufsteigend sortiert. Abbildung 5.13 zeigt nun die durchschnittliche und maximale Anzahl an allen Reads mit dieser Sortierung. *BHT Index 1* ist dabei die minimale Anzahl Treffer, die eine beliebige Band-Hashtabelle für einen Read liefert, *BHT Index 95* die maximale.

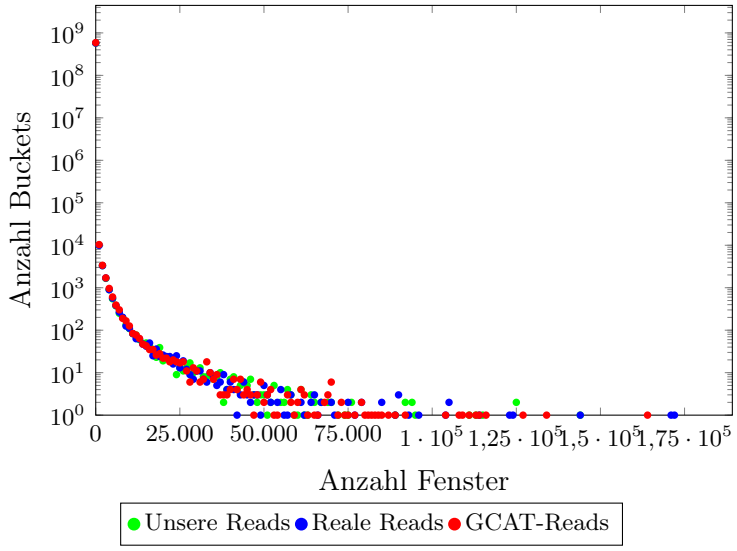


Abbildung 5.12: Anzahl der Buckets, die sehr viele Fenster enthalten.

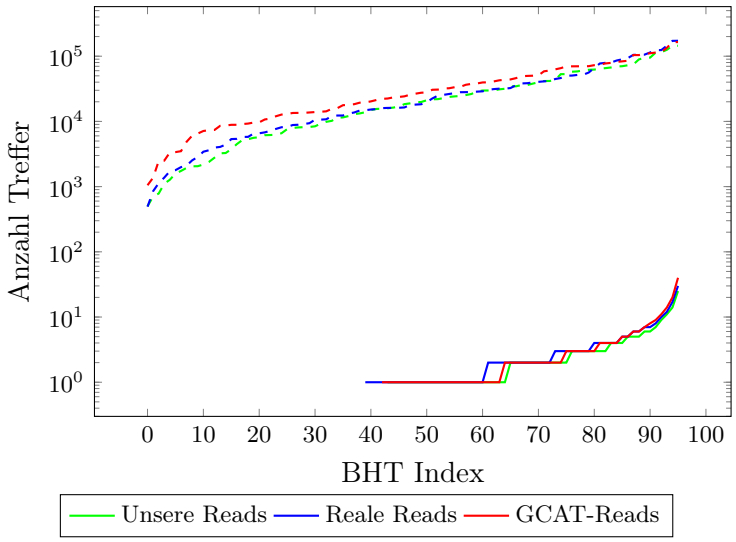


Abbildung 5.13: Median der Anzahl an Treffern in verschiedenen Band-Hashtabellen, absteigend sortiert. Die gestrichelten Linien sind die Maximalwerte.

Dabei zeigt sich, dass die kleinste Anzahl im Schnitt sehr niedrig ist, bei einem Median von weniger als Null. Das Maximum unter den kleinsten Anzahlen liegt bei etwa 1 000. Die maximale Anzahl, die von einer Band-Hashtabelle geliefert werden kann, ist hingegen mit Werten in der Region um 100 000 bis 200 000 deutlich größer. Im Schnitt liefert eine Band-Hashtabelle je nach Quelle der Reads 8 000 bis 20 000 Treffer mit entsprechenden Auswirkungen auf die Laufzeit.

Die Begrenzung der Anzahl der Fenster hat aber auch Auswirkungen auf die Anzahl der gefundenen Reads. Um diesen Effekt zu messen, wurden verschiedene Begrenzungen getestet und die Anzahl der korrekt gefundenen Reads gezählt. In Abbildung 5.14 werden diese Messungen grafisch dargestellt. Das linke Diagramm zeigt die Auswir-

kung der Begrenzung der maximal gefundenen Fenster auf die Anzahl der korrekt gefundenen Intervalle, was direkt der Qualität der Ergebnisse entspricht. Das rechte Diagramm zeigt den Einfluss auf die Laufzeit.

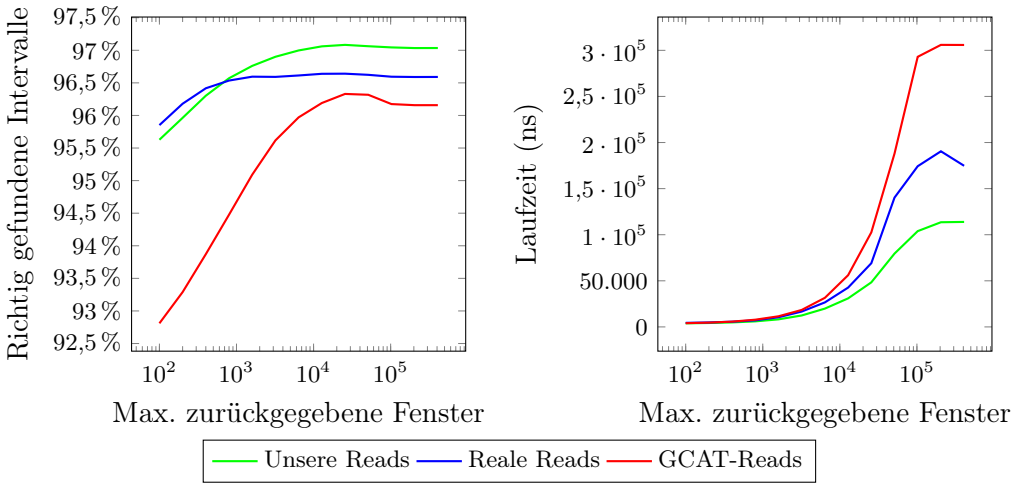


Abbildung 5.14: Auswirkungen der Bregrenzung der maximal zurückgegebenen Fenster auf die Qualität der Ergebnisse und die Laufzeit der Suche.

Man sieht, wie die Laufzeit bei großen Begrenzungen stark ansteigt, während die Qualität der Ergebnisse schon sehr früh ihr Maximum erreicht und dort verbleibt. Man erkennt außerdem, dass sich die Werte für die GCAT-Reads deutlich unterscheiden. Die Gründe dafür werden in Abschnitt 5.6 genauer beleuchtet. Allgemein empfiehlt es sich, die Anzahl der maximal zurückgegebenen Fenster zu begrenzen, da hierdurch die Laufzeit der Suche beschränkt wird, ohne dass sich große Qualitätseinbußen ergeben. Als Standardwert wurde die Begrenzung auf 10^3 gesetzt. Dieser Wert liegt bei unseren generierten und den getesteten realen Reads so, dass durch höhere Werte nur noch geringe Verbesserungen erzielt werden. Bei den GCAT-Reads würde eine höhere Begrenzung, etwa 10^4, eine Verbesserung bringen. Die würde allerdings auf Kosten der Laufzeit geschehen, sodass sich die Suchzeit durch diese Änderung verdoppeln würde.

5.3.4 Streuung der Ergebnisse

Locality-Sensitive Hashing ist ein randomisiertes Verfahren und VATRAM nutzt Zufallswerte zur Initialisierung des LSH. Dies bedeutet, dass das Alignment von VATRAM nicht deterministisch ist, sondern die Ergebnisse Schwankungen unterliegen, wenn verschiedene Werte zur Initialisierung des Zufallszahlengenerators verwendet wurden. Große Unterschiede in den Ergebnissen sind natürlich unerwünscht, da dies den Eindruck auf den Nutzer verfälscht und das Programm mehrfach ausgeführt werden müsste. Um die Streuung bei verschiedenen Zufallszahlen zu bestimmen, wurden für jeden Read-Datensatz zehn Durchläufe durchgeführt und jeweils Laufzeit und Qualität der Ergebnisse gemessen. Die Messungen sind in Abbildung 5.15 dargestellt.

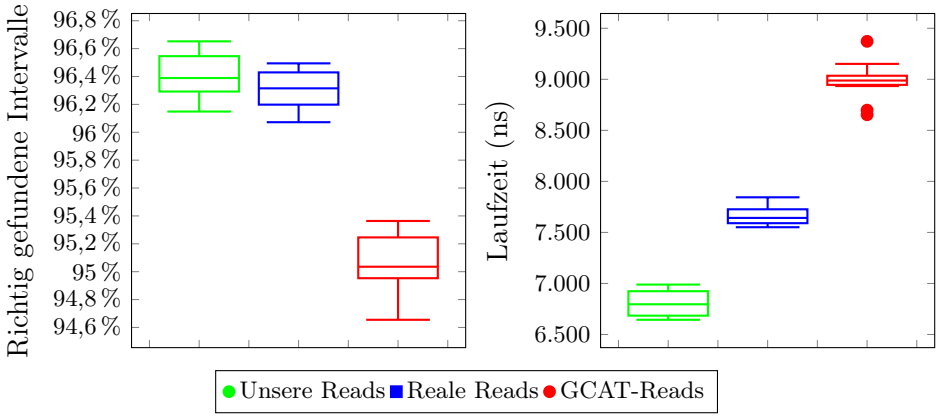


Abbildung 5.15: Streuung der Qualität der Ergebnisse und der Suchzeit mit zehn verschiedenen Durchläufen für unterschiedlichen Zufallszahlen.

Es zeigt sich, dass die Qualität der Ergebnisse nur sehr wenig Streuung aufweist, zumindest bei Betrachtung der uns generierten und der realen Reads. Bei den GCAT-Reads ist die Qualität allgemein schlechter und auch die Streuung stärker. Sie bewegt sich aber trotzdem nur im Bereich von einigen Promille. Bei der Laufzeit ist die Spannweite ebenfalls bei GCAT-Reads am größten.

5.3.5 Intervall-Parameter

In diesem Abschnitt werden die Auswirkungen unterschiedlicher Parameter diskutiert, die Einfluss auf die Eingabe des Aligners haben. Der Aligner erhält ein exaktes Intervall im Referenzgenom. Manche Parameter können die Übergabe eines Intervalls komplett verhindern, während andere Einfluss auf die Größe des Intervalls haben. Beide Arten der Parametervariation haben Einfluss auf die Verarbeitungszeit im LSH-Modul sowie Berechnungszeit im Aligner.

5.3.5.1 Minimale Trefferschranke für eine Fenstersequenz

Im Mapping-Modul werden Reads per LSH-Verfahren auf Fenster abgebildet. Da ein Read häufig auch über zwei Fenster mittig gemappt werden kann, können dementsprechend auch mehrere Fenster bei einer Anfrage gefunden werden. Falls eine Teilmenge der gefundenen Fenster nebeneinander liegt, werden diese zu einer Fenstersequenz zusammengefasst. Für eine Fenstersequenz wird außerdem die Anzahl der Hash-Treffer gespeichert. Um die Anzahl der *false positives* zu verringern, kann eine Minimalschranke für die Trefferanzahl in dieser Fenstersequenz verwendet werden. Abbildung 5.16 zeigt die Auswirkungen dieses Parameters auf die Mapping-Laufzeit und den Overhead im Aligner. Da bei Unterschreiten der Schranke kein Intervall für den Aligner berechnet wird, sinkt die Laufzeit mit zunehmender Schranke. Gleichzeitig sinkt aber auch der Overhead im Aligner, denn es werden durch eine größer werdende Schranke auch weniger *false positives* gefunden, die Overhead im Aligner erzeugen. Die Anzahl der korrekt gemappten Reads nimmt zwar auch ab, unterscheidet sich nur um 1 Promille für die Schrankenwerte 1 und 4.

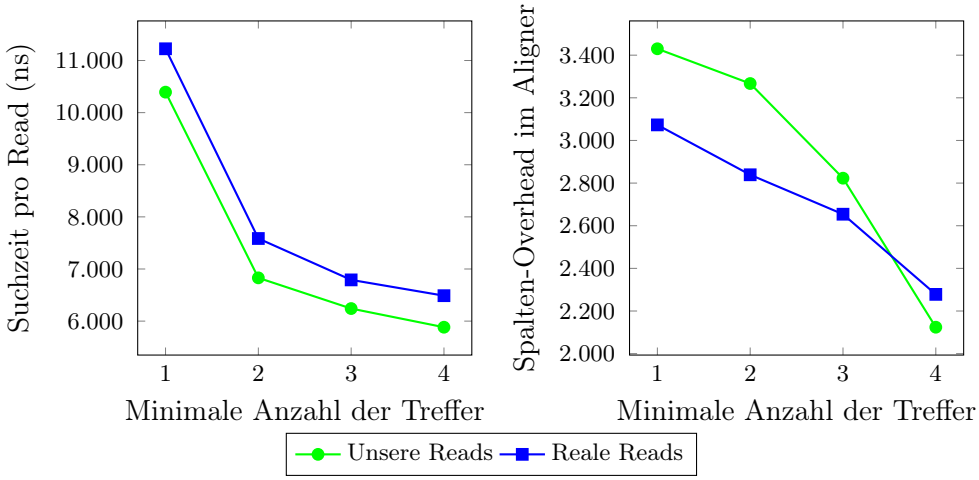


Abbildung 5.16: Auswirkungen der minimalen Trefferschranke in einer WindowSequence auf die Laufzeit pro Read und den Overhead im Aligner

5.3.5.2 Begrenzung der Anzahl zurückgegebener Fenster

Auch wenn die minimale Trefferschranke gut gewählt wird, kann es weiterhin passieren, dass sehr viele Fenstersequenzen als Kandidaten gefunden werden. Aus diesem Grund wurden zwei weitere Parameter eingeführt. Der Parameter *maxSelect* beschränkt die Gesamtanzahl der betrachteten Fenstersequenzen ohne Berücksichtigung der Trefferanzahlen.
In der Verarbeitung werden die gefundenen Fenstersequenzen anfangs nach Trefferanzahlen sortiert. Ein weiterer Parameter *nextFactor* bricht die Berechnung ab, wenn die aktuell betrachtete Fenstersequenz um den Faktor *nextFactor* weniger Treffer im Vergleich zur besten Fenstersequenz hat. Abbildung 5.17 zeigt die Auswirkungen

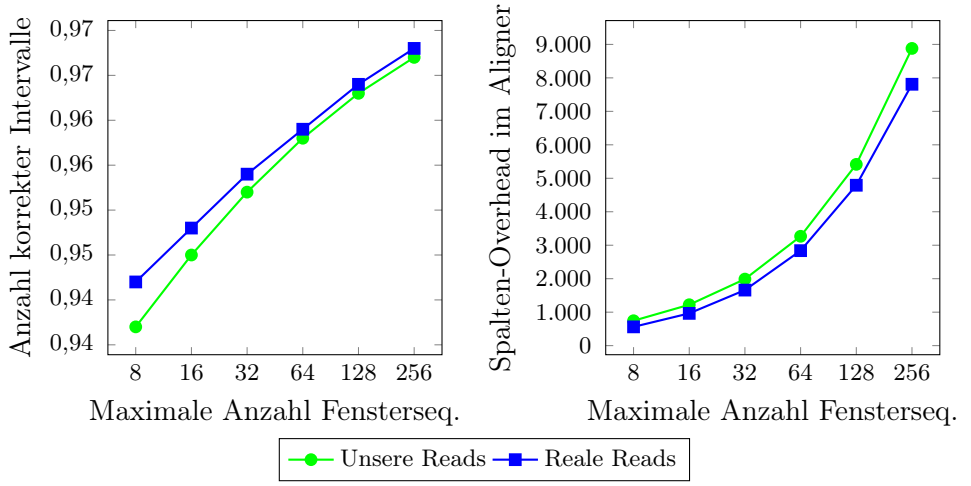


Abbildung 5.17: Auswirkungen der Paramter *maxSelect* und *nextFactor* auf die Anzahl der korrekt gefundenen Intervalle und den Spalten-Overhead im Aligner.

von *maxSelect* auf die Anzahl der korrekt gefundenen Intervalle und den Spalten-

Overhead im Aligner bei einem festen *nextFactor* von 4. Offensichtlich werden mehr *false positives* an den Aligner übergeben, die sich zwar negativ auf die Gesamtlaufzeit des Readmappers auswirken, aber insgesamt auch die Qualität (also die korrekt gefundenen Intervalle) um einige Promille verbessern.

5.3.5.3 Vergrößerung der Intervalle

Bevor das Intervall, das sich aus der Kandidaten-Fenstersequenz ergibt, an den Aligner übergeben wird, wird es noch an beiden Seiten vergrößert um ein möglichst gutes Alignment zu ermöglichen. Diese Vergrößerung findet allerdings nur bei Fenstersequenzen bis zu einer Maximallänge von 2 statt. Der Parameter *extendNumber* bestimmt die maximale Vergrößerung des Intervalls abhängig von der Trefferanzahl in der Fenstersequenz, wenn sie nur aus einem Fenster besteht. Die letztendliche Größe der überstehenden Abschnitte berechnet sich mittels *extendNumber* − *countHits* ∗ *extendMult*. Abbildung 5.18 zeigt die Auswirkungen des Parameters *extendNumber* auf die Anzahl der korrekt gefundenen Intervalle bei einem festen Parameter *extendMult* von 0.3. Für den Fall, dass eine Fenstersequenz aus 2 Fenstern besteht, wird der Abschnitt mindestens um die q-Gramm-Länge auf beiden Seiten vergrößert. Zusätzlich werden die Abschnitte um den Faktor *contractMult* ∗ *countHits* vergrößert. Die Wahl des Parameters *contractMult* zeigte dabei allerdings keine relevanten Auswirkungen auf die Qualität des Readmappers oder auf den Spalten-Overhead im Aligner.

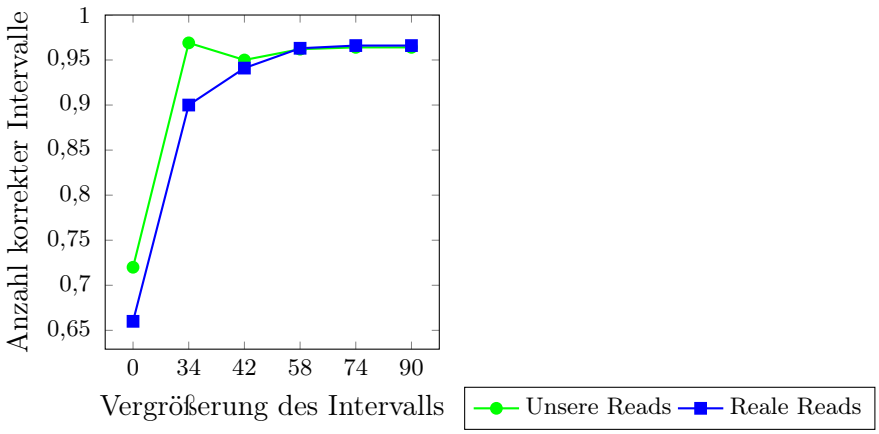


Abbildung 5.18: Auswirkungen der Vergrößerung der Intervalle auf die Qualität des Readmappers.

5.4 Alignment-Experimente

Im Gegensatz zum LSH-Verfahren gibt es beim Aligner keine externen Parameter, die zur Optimierung verwendet werden können. Der Aligner garantiert, dass er das optimale Alignment zwischen einem gegebenen Read und einem Referenzabschnitt in der vorgegebenen Fehlerschranke findet, sodass es keinen Trade-Off zwischen Laufzeit und Ergebnisgüte gibt. Bei dieser Evaluation geht es daher um Laufzeitmessungen

in verschiedenen Szenarien. Zum einen soll getestet werden, wie sich die Optimierungen „RowCancel“ und „RowSkip“ hinsichtlich der Laufzeit und der Anzahl explizit berechneter Felder im Vergleich zu einem naiven Aligner verhalten. Zum anderen ist es interessant zu beobachten, wie gut die einzelnen Alignervarianten mit steigender Readlänge und steigender Fehlerschranke skalieren.

5.4.1 Aufbau der Experimente

Für die Tests wurde ein künstliches Genom mit 10 Millionen Basenpaaren erzeugt. Für das erzeugte Genom wurden wiederum verschiedene Read- sowie Variantendatensätze generiert, sodass alle erzeugten Reads aus dem generierten Referenzgenom stammen. Fast alle Readdatensätze beinhalteten 10000 Reads mit einer erwarteten Anzahl von drei Fehlern pro Read, unabhängig von ihrer Länge (davon erwartet 2,7 SNP-Fehler sowie 0,3 Indel-Fehler). Ausnahmen davon sind:

- Ein Datensatz mit Reads der Länge 1000, welcher für die Fehlerskalierung verwendet wird und im Erwartungswert sechs Fehler pro Read enthält.
- Der Datensatz mit Reads der Länge 100 für Clipping-Tests, welcher 100000 Reads beinhaltet und im Erwartungswert neun Fehler pro Read enthält.

Für fast alle Tests wurde ein Variantendatensatz verwendet, der durchschnittlich an 1% der Positionen eine Indel-Variante enthält. Die Ausnahme sind hier explizite Tests mit unterschiedlicher Variantendichte. Für die Tests wurden bewusst künstliche Daten ausgewählt, um mehr Kontrolle über die Parameter der erzeugten Daten zu haben. Außerdem müssten bei realen Daten vorher alle N-Sequenzen entfernt werden, da diese im normalen Anwendungsfall des LSH herausgefiltert werden. Dies ist wichtig, weil lange N-Sequenzen zu einer garantierten quadratischen Laufzeit des Aligners führen würden. Die Länge der erzeugten Varianten liegt gleichverteilt zwischen 1 und 6.

Zur Durchführung wurden in einem Prozess zuerst alle benötigten Daten erstellt und anschließend ausschließlich der Aligner verwendet. Ein Mapping ist wegen der durch die Generierung bereits bekannten Positionen nicht notwendig. Der Aligner bekam als Referenzabschnitt immer jenen Ausschnitt aus dem Genom, aus dem der Read entnommen wurde. Zusätzlich wurde der Ausschnitt um die halbe Readlänge nach vorne und hinten erweitert. Für jeden Aufruf des Aligners wurde die Zeit gemessen, die in diesem Aufruf verbracht wird, sowie die Anzahl explizit berechneter Felder[8]. Bei den Messungen wird zusätzlich unterschieden, ob ein Read zu einer gültigen Alignierung führte oder nicht. Um den Fall zu berücksichtigen, dass der Aligner in der Praxis auch falsche Suchintervalle für Reads erhält, wird jeder Read mit einer Wahrscheinlichkeit von 5% in einem falschen Intervall gesucht. Dadurch wird eine gewisse Zahl von nicht alignierten Reads garantiert, sodass auch gemessen wird, wie teuer falsch gemappte Reads für den Aligner tatsächlich sind.

Die Implementierung der RowSkip-Optimierung entspricht der aktuellen Implementierung, wie sie VATRAM standardmäßig für das Alignieren verwendet. Um die an-

[8] Dazu zählen nicht die Felder, die gemäß der RowSkip-Optimierung präventiv beschrieben werden.

deren Alignervarianten zu simulieren, wurde die Referenzimplementierung in eine eigene Testklasse kopiert und die jeweiligen Optimierungen entfernt. Dabei wurde darauf geachtet, möglichst jeden Overhead mit zu entfernen, der durch die Optimierungen anfällt, wie beispielsweise das präventive Schreiben von Fehlerwerten (vgl. Abschnitt 4.2.4). Die Behandlung von Indel- und SNP-Varianten bliebt bei allen Alignerimplementierungen erhalten. Alle Tests wurden - sofern nicht anders angegeben - mit einer Fehlerschranke von sechs durchgeführt, was der Standardeinstellung des Aligners entspricht.

5.4.2 Vergleich zwischen Aligneroptimierungen

Der Fokus liegt zunächst auf den verschiedenen Aligner-Optimierungen. Dazu wurden alle Implementierungen mit Reads der Länge 100, die durchschnittlich drei Fehlern enthielten, aligniert, da dies dem ursprünglichen Anwendungsfall von Illumina-Reads entspricht. Die Ergebnisse sind in Abbildung 5.19 dargestellt.

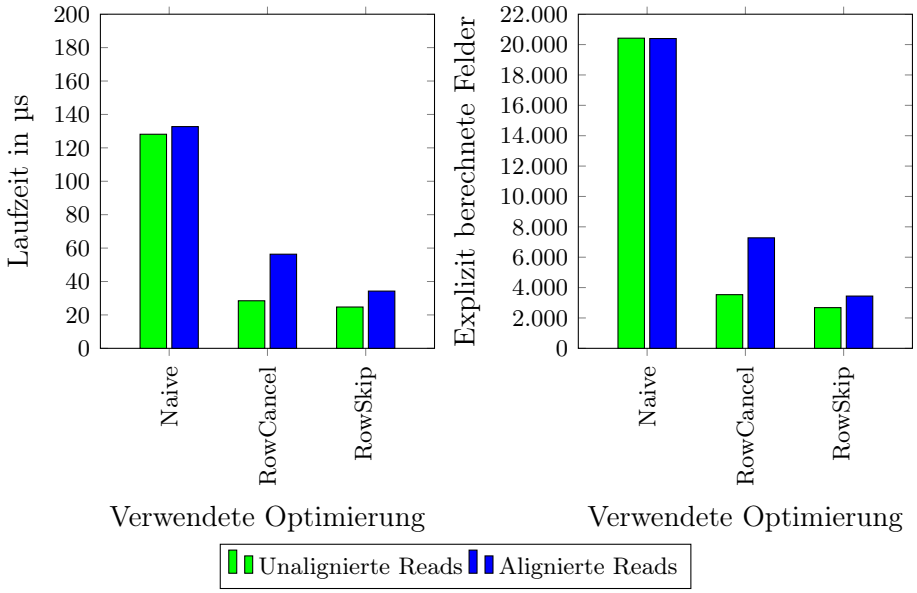


Abbildung 5.19: Übersicht über die verschiedenen Aligner-Optimierungen in Bezug auf Laufzeit und Anzahl berechneter Matrixfelder pro Aufruf. Readlänge 100, maximal 6 erlaubte Fehler.

Der naive Aligner berechnet immer die gesamte Matrix, was sich gut an der Anzahl berechneter Felder ablesen lässt. Diese liegt bei 20000, was der Readlänge von 100 multipliziert mit der Ausschnittsgröße von 200 entspricht. Die RowCancel-Optimierung berechnet deutlich weniger Felder, wobei der Unterschied zwischen alignierten und nicht alignierten Reads hier besonders auffällt. Bei dem Versuch einen Read an ein unpassendes Genomintervall zu alignieren, werden kaum Matches gefunden, sodass die Edit-Distanz innerhalb der Spalten sehr schnell die vorgegebene Fehlerschranke überschreitet. Diesen Fall verarbeitet die RowCancel-Optimierung offensichtlich relativ effizient. Bei Reads, die zu einem Alignment geführt haben, müssen alle Felder berechnet werden, die unterhalb des Pfades des optimalen Alignments liegen. Dadurch müssen deutlich mehr volle Spalten berechnet werden.

Da die RowSkip-Optimierung auch innerhalb einer Spalte Felder überspringen kann, zeigt sie erwartungsgemäß kein solches Verhalten und kommt mit gut halb so vielen berechneten Feldern wie die RowCancel-Optimierung aus. Der Unterschied zu nicht alignierten Reads ist hier relativ gering.

Bei der Laufzeit ergibt sich die gleiche Rangfolge, auch wenn die Abstände etwas kleiner ausfallen. Die RowCancel-Optimierung ist sehr leicht zu implementieren und gegenüber der naiven Variante im gleichen Maße schneller wie sie weniger Felder berechnen muss. Die RowSkip-Optimierung muss zusätzlich zu den Edit-Distanz-Werten noch RowSkip-Spalten berechnen, die für die Optimierung an sich benötigt werden. Durch diesen Overhead fällt der Vorsprung gegenüber den anderen Optimierungen nicht ganz so stark aus. Bei nicht alignierten Reads ist der Vorsprung sogar komplett verschwunden. Der Grund, warum die naive Variante bei alignierten Reads minimal langsamer ist, ist die zusätzliche Berechnung des Cigar-Strings per Backtracing.

Insgesamt lässt sich aus diesem Experiment aber ableiten, dass sich die RowSkip-Optimierung in der Praxis selbst bei Reads der Länge 100 durchaus lohnt.

5.4.3 Skalierung mit steigender Readlänge

In diesem Abschnitt soll untersucht werden, wie der Aligner mit zunehmender Readlänge skaliert. Da der Aligner im *worst case* eine quadratische Laufzeit besitzt, könnte sich der Aligner als ungeeignet für längere Reads herausstellen. Für die naive Variante ist das offensichtlich der Fall, da diese die komplette Matrix berechnet; bei den Optimierungen ist es nicht ganz so offensichtlich. Da die RowSkip-Optimierung bereits bei 100 Basenpaaren durchschnittlich deutlich weniger Matrixfelder berechnet, könnte man erwarten, dass sich diese Variante bei längeren Reads noch stärker von den anderen Varianten absetzen kann.

Für den Test wurden Reads mit 100, 250, 350, 500, 750, 1000, 1500 und 2000 Basenpaaren erzeugt und die Laufzeit der Alignierung gemessen. Das Ergebnis ist in Abbildung 5.20 zu sehen.

Der naive Algorithmus skaliert wie erwartet quadratisch mit der Readlänge, sodass die Ergebnisse für Reads mit mehr als 1000 Basenpaaren der Übersicht wegen nicht auf dem Plot zu sehen sind. Die RowCancel-Optimierung skaliert ebenfalls annähernd quadratisch mit der Readlänge. Das Problem dieser Optimierung besteht erneut darin, dass bei einer erfolgreichen Alignierung alle Matrixfelder unterhalb des optimalen Pfades berechnet werden. Während sich dieser Effekt bei kurzen Reads nicht so stark auf die Laufzeit auswirkt, dominiert er mit zunehmender Readlänge immer mehr die Laufzeit.

Die RowSkip-Optimierung besitzt dieses Problem nicht und kann beliebig lange Passagen einer Spalte überspringen. Die Messungen belegen, dass diese Aligner-Optimierung deutlich besser mit der Readlänge skaliert und bei der maximalen getesteten Readlänge bereits eine Größenordnung schneller als die anderen Varianten ist. Die Skalierung fällt dabei fast linear aus, wobei der *worst case* natürlich weiterhin quadratisch ist.

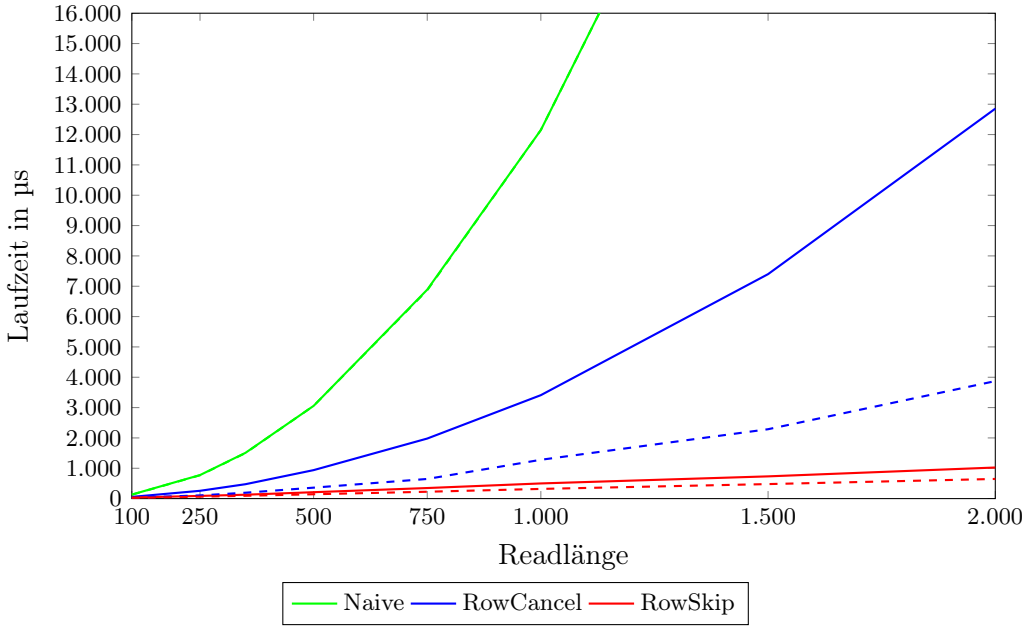


Abbildung 5.20: Laufzeitskalierung mit steigender Readlänge für die Alignervarianten. Gestrichelte Linien stehen für die Laufzeit von nicht alignierten Reads, die durchgezogenen Linien für alignierte Reads.

Um eine etwas genauere Aussage über die Skalierung des Aligners treffen zu können, wurden die Laufzeiten anhand der Readlänge normiert. Abbildung 5.21 zeigt die Laufzeiten der RowSkip-Optimierung pro hundert Basenpaare an.

Man erkennt leicht, dass die normierte Laufzeit mit steigender Readlänge nur sehr langsam wächst. Von 100 auf 2000 Basenpaaren pro Read findet weniger als eine Verdopplung der normierten Laufzeit statt, sodass der Aligner durchaus in der Lage ist, auch deutlich längere Reads effizient zu verarbeiten.

Bei Reads mit mehr als 2000 Basenpaaren stößt die verwendete Implementierung jedoch an ihre Grenzen. Der Grund dafür ist, dass der Aligner immer die gesamte Edit-Distanz-Matrix vorhält. Der Speicherverbrauch des Aligners steigt somit quadratisch mit der Readlänge an, sodass die Matrix für Reads der Länge 2000 und einem Referenzabschnitt der Länge 4000 bereits mindestens 96MB Speicher benötigt. Da VATRAM in der Praxis für jeden Thread eine eigene Aligner-Instanz erzeugt und die Referenzabschnitte durch die LSH-Implementierung bis zu zehn Mal so lang wie der zugehörige Read werden können, ist diese Aligner-Implementierung bei langen Reads nicht mehr verwendbar.

Auf mögliche Lösungsansätze wird in Abschnitt 6.2.2 noch genauer eingegangen.

5.4.4 Skalierung mit steigender Fehlerzahl

Als nächstes wurde untersucht, wie sich die vorgegebene Fehlerschranke auf die Laufzeit des Aligners auswirkt. Der naive Aligner wurde dabei nicht getestet, da er unabhängig von diesem Parameter immer die gleiche Laufzeit besitzt. Die anderen beiden

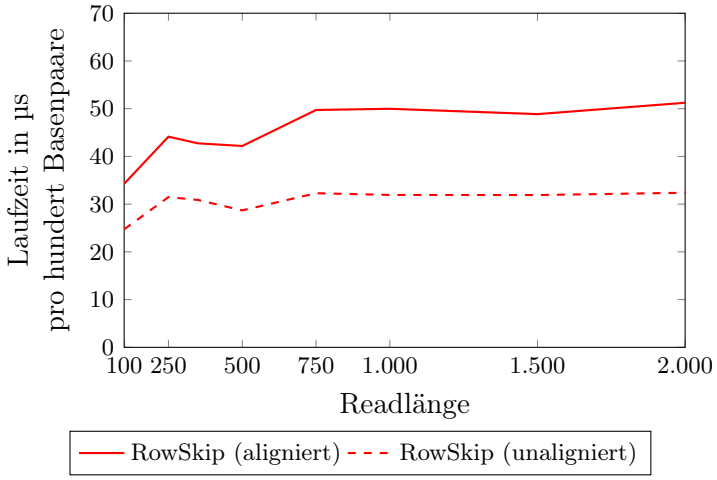


Abbildung 5.21: Laufzeitskalierung mit steigender Readlänge für die RowSkip-Optimierung. Die angegebene Laufzeit ist anhand der Readlänge normiert.

Alignervarianten nutzen die Fehlerschranke aus, um überflüssige Berechnungen auszulassen. Es ist somit naheliegend, dass sich eine steigende Fehlerschranke negativ auf die Laufzeit auswirkt. Falls die Fehlerschranke der Länge des Reads entspricht, würden beide Optimierungen zu einem naiven Aligner degenerieren. Da solche hohen Fehlerschranken aber keine praktische Relevanz besitzen, wurden maximal zehn Fehler (für Reads mit hundert Basenpaaren) bzw. 20 Fehler (für Reads mit tausend Basenpaaren) getestet. Der Datensatz mit 100er-Reads besaß durchschnittlich drei Fehler pro Read, während der 1000er-Datensatz durchschnittlich sechs Fehler pro Read besaß. Die Ergebnisse sind in Abbildung 5.22 dargestellt.

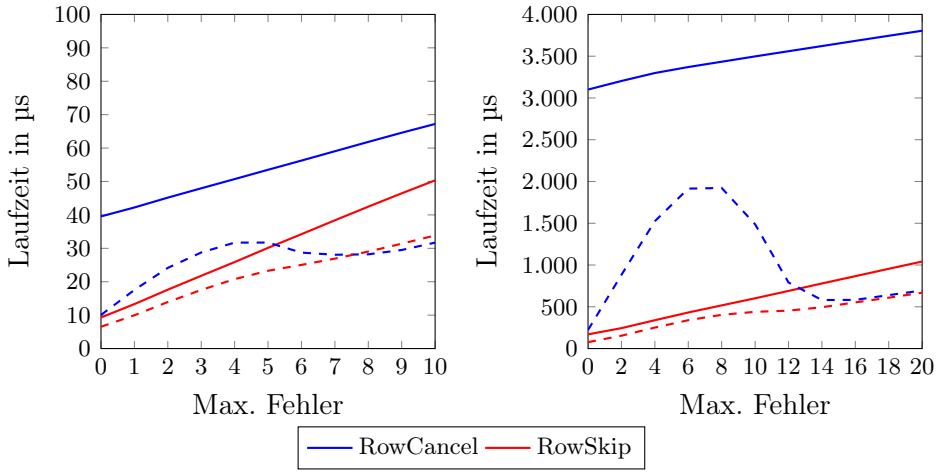


Abbildung 5.22: Skalierung mit steigender Fehlerschranke für die Alignervarianten. Readlängen sind 100 Basenpaare (links) bzw. 1000 Basenpaare (rechts). Gestrichelte Linien stehen für die Laufzeit von nicht alignierten Reads, die durchgezogenen Linien für alignierte Reads.

Die Laufzeit nimmt in allen Fällen mit steigender Fehlerschranke zu. Bei der RowSkip-Optimierung entsteht dabei eine fast lineare Skalierung mit der Fehlerschranke, wenn

man von einem kleinen konstanten Overhead absieht. Dabei spielt es kaum eine Rolle, ob ein Read zu einer Alignierung geführt hat oder nicht.

Die RowCancel-Optimierung verhält sich gegenüber der steigenden Fehlerschranke bezüglich der alignierten Reads sehr ähnlich. Auch hier bildet die Laufzeit annähernd eine Gerade, die eine leicht kleinere Steigung als die der RowSkip-Optimierung hat. Allerdings ist hier der konstante Anteil an der Laufzeit deutlich größer und dominiert bei Reads mit 1000 Basenpaaren faktisch die Laufzeit. Dies lässt sich erneut dadurch erklären, dass bei alignierten Reads viele lange Spalten berechnet werden müssen, die bei der RowSkip-Optimierung wegfallen. Je länger der Read desto weniger macht sich also die Fehlerschranke anteilsmäßig bemerkbar. Insgesamt scheinen sich beide Optimierungen mit steigender Fehlerzahl anzunähern, doch in dem getesteten Bereich ist die RowCancel-Optimierung durchgehend schlechter.

Die Laufzeit der nicht alignierten Reads verhält sich bei der RowCancel-Variante anders als die der alignierten Reads. Die Kurven beginnen ähnlich der RowSkip-Optimierung fast beim Ursprung, steigen dann relativ stark an und sinken anschließend wieder auf das Niveau der RowSkip-Kurve. Der Grund für die Steigung ist der unterschiedliche Grad der Nicht-Alignierung. Je kleiner die Fehlerschranke ist, desto weniger Readzeichen muss der Aligner verarbeiten, um den Read aufgrund zu vieler Fehler verwerfen zu können. Wenn die Fehlerschranke k etwa so groß ist wie die erwartete Fehleranzahl der Reads, kann es sein, dass ein Read mit mehr als k Fehlern fast komplett aligniert wird. Die Fehlerschranke wird letztendlich aber doch überschritten, sodass der Read verworfen wird. Dadurch ist es bei solchen Fehlerschranken wahrscheinlicher, dass der RowCancel-Aligner viele lange Spalten berechnen muss und so die Laufzeit übermäßig stark ansteigt. Ab einer gewissen Fehlerschranke gibt es jedoch kaum noch Reads im getesteten Datensatz, die sich damit nicht mehr alignieren lassen. An dieser Stelle fallen dann die 5% der Reads ins Gewicht, die absichtlich an ein falsches Intervall aligniert wurden und gemäß der Optimierungen extrem schnell verarbeitet werden. Daher nähern sich beide Kurven für nicht alignierte Reads am Ende an.

Die nahezu lineare Skalierung der RowSkip-Optimierung mit der Readlänge, die sich in Abschnitt 5.4.3 ergab, bezog sich auf eine konstante Fehlerschranke von 6. Falls man davon ausgeht, dass mit steigender Readlänge auch die Fehlerschranke in gleichem Maß erhöht wird, so entsteht wieder ein quadratischer Zusammenhang zwischen Laufzeit und Readlänge.

5.4.5 Skalierung der Variantendichte

Zur Verarbeitung von Varianten muss der Aligner zusätzliche Matrixspalten berechnen bzw. bei der Rekursionsgleichung zusätzliche Spalten berücksichtigen. Das bedeutet, dass der Mehraufwand zur Behandlung von Varianten etwa ihrer Länge im Verhältnis zur Referenzausschnittslänge entspricht. Um den Einfluss von Varianten auf die Laufzeit zu bestimmen, wurde die Dichte der Varianten schrittweise erhöht, während die Readlänge und Fehlerschranke konstant blieben. Die *Variantendichte* ist dabei definiert als Anzahl der Indel-Varianten im Verhältnis zur Genomlänge. Die Dichte gibt also die erwartete Anzahl von Indel-Varianten pro Position im Referenzgenom an. Für den Test wurden erneut Reads mit hundert Basenpaaren verwendet,

die mit bis zu 6 Fehlern aligniert wurden. Die Variantendichte wurde zwischen 0% und 20% variiert.

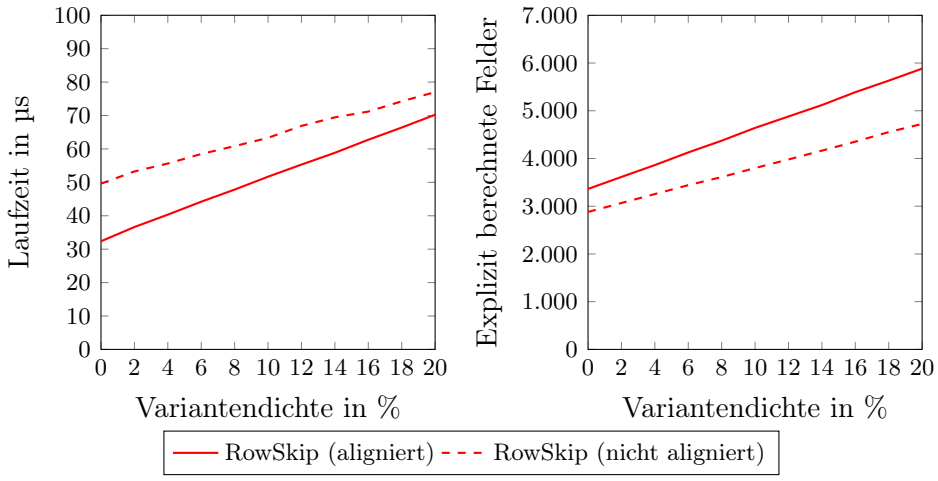


Abbildung 5.23: Skalierung mit steigender Variantendichte für den RowSkip-Aligner. Die Readlänge beträgt 100 Basenpaare, Varianten sind Insertionen, Deletionen oder Substitutionen mit einer maximalen Länge von sechs Basen.

Sowohl die Laufzeit als auch die Anzahl berechneter Felder steigen monoton mit der Variantendichte an, wie in Abbildung 5.23 zu sehen ist. Der Anstieg verläuft linear, was bedeutet, dass der Einfluss einer einzelnen Variante nicht davon abhängt, wie viele weitere Varianten verarbeitet wurden. Das entspricht den Erwartungen, da alle Varianten unabhängig voneinander verarbeitet werden und sich keine kombinatorischen Einflüsse ergeben.

Der Anstieg zu berechnender Felder liegt ebenfalls im Rahmen der Erwartungen. Eine Variantendichte von 20% führt zu einem Anstieg von 73,7%. Da Varianten eine erwartete Länge von 3,5 Basenpaaren besitzen, ergibt sich ein theoretischer Anstieg von etwa 70%. Anders dagegen sieht es bei der Laufzeit aus: Sie verdoppelt sich bei der größten getesteten Variantendichte im Vergleich zu einem Datensatz ohne Indelvarianten. Der Grund dafür liegt im Overhead der RowSkip-Optimierung. Bei einer Indelvariante müssen mehrere NextRow-Spalten zu einer Spalte verschmolzen werden. Zur Vereinfachung wurde in diesen Fällen festgelegt, dass Spalten nach einer Indelvariante immer komplett berechnet werden. Dazu müssen in den Vorgängerspalten viele Dummy-Werte in der Matrix eingetragen werden, um undefinierte Lesezugriffe zu verhindern. Diese Vereinfachung wirkt sich bei einer großen Variantendichte negativ auf die Skalierung aus. Bei längeren Reads würde sich das Problem vermutlich weiter verschärfen, da die Spalten dort verhältnismäßig noch dünner besetzt sind.

Für nicht alignierte Reads hat der Aligner durchschnittlich länger gebraucht als für alignierte Reads. Dieses Ergebnis erscheint wenig plausibel, da für unalignierte Reads weniger Felder berechnet werden müssen. Die durchschnittliche Anzahl von behandelten Indels pro Durchlauf ist den Messungen nach fast identisch und damit auch keine mögliche Ursache. Auffällig ist dagegen, dass die Standardabweichung der Laufzeit bei unalignierten Reads deutlich höher ist. Das könnte darauf hindeuten, dass

es einige Ausreißer gibt, die die Durchschnittszeit verfälschen, allerdings konnten wir dafür keine Ursache finden. Die Messungen sind reproduzierbar.

5.4.6 Laufzeit des Soft Clippings

Abschließend wurde das Soft Clipping des Aligners auf seine Laufzeit untersucht. Da das Soft Clipping möglicherweise zu einem zweiten internen Alignierungsvorgang führen kann, wird sich die Laufzeit sehr wahrscheinlich im Durchschnittsfall erhöhen. Abbildung 5.24 zeigt einen Vergleich zwischen Reads, die direkt aligniert wurden und solchen, bei denen Soft Clipping zum Einsatz kam. Um für letzteren Fall genügend Reads zur Verfügung zu haben, wurde die durchschnittliche Fehlerzahl für diesen Test von drei auf neun Fehler pro Read erhöht. Eine Unterscheidung zwischen alignierten und nicht alignierten Reads wurde nicht vorgenommen.

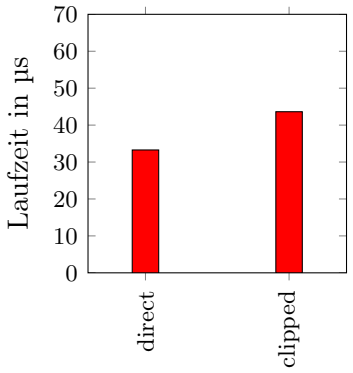


Abbildung 5.24: Laufzeitvergleich zwischen direkt alignierten Reads und Reads, die mit Hilfe von Soft Clipping aligniert wurden. Die erwartete Fehlerzahl pro Read lag hier bei 9. Die Fehlerschranke des Aligners lag beim Standardwert von 6, die maximale Länge für Soft Clipping lag bei 30 Basenpaaren.

Der Laufzeitzuwachs fällt insgesamt gering aus, wobei hier natürlich anzumerken ist, dass das Soft Clipping des Aligners nicht optimal ist und möglicherweise zu viele Stellen abschneidet.

5.5 Vergleich mit anderen Readmappern

In diesem Abschnitt werden die Ergebnisse von VATRAM mit den Ausgaben verschiedener führender Readmapper verglichen. Dabei wird zunächst auf den Aufbau des Experiments eingegangen. Anschließend werden die verwendeten Daten und die zur Analyse nötigen Metriken vorgestellt. Abgeschlossen wird dieser Teil durch Testergebnisse und deren Interpretation.

5.5.1 Aufbau des Experiments

Auf dem Testsystem wurde VATRAM, sowie die beiden Vergleichsprogramme *BWA* (Li, 2013) und *Bowtie2* (Langmead und Salzberg, 2012) eingerichtet. Jedes Einzel-

experiment setzt sich aus einem Datensatz zusammen, der sequenziell von allen drei Programmen verarbeitet wird. Die Ausgabe liegt danach jeweils als SAM-Datei vor, welche dann mit Hilfe der in Abschnitt 5.5.2 beschriebenen Metriken analysiert wird.

Datensätze

Es wurden drei verschiedene Kategorien an Datensätzen verwendet, deren Eigenschaften im Folgenden beschrieben werden.

Synthetische Datensätze Die einfachste Art an Testdaten lässt sich durch einen Datengenerator (wie in Abschnitt 5.2 beschrieben) erstellen. Eine triviale Möglichkeit, eine synthetische Referenz zu erstellen, ist eine Konkatenation von zufällig ausgewählten Basen einer vorgegebenen Länge. Für die Erstellung der Reads lässt sich die Referenz in gleichlange Abschnitte unterteilen. Damit sich die Abschnitte auch von der Referenz unterschieden, werden zufällig einzelne oder mehrere Basen hinzugefügt, gelöscht oder verändert. Dieser naive Ansatz simuliert Sequenzierfehler und Varianten.

Es wurden 2 Millionen Reads mit einer Länge von jeweils 100 Basenpaaren generiert, um einen Kompromiss zwischen Gesamtlaufzeit und Qualität der Ergebnisse zu erhalten. Ein weiterer Paramter ist die Wahrscheinlichkeit, dass eine Base durch eine Insertion, Deletion oder Substitution verändert wurde. Diese wurde auf 5% gesetzt. Mit einer Wahrscheinlichkeit von 90% wurde ein SNP erzeugt, ansonsten ein Indel. Dort wurde in der Hälfte der Fälle eine Insertion generiert, ansonsten wurden Basen entfernt.

Halbsynthetische Datensätze Um Testdaten zu erstellen, welche näher an der Realität liegen, kann man, analog zu obigen Vorgehen, auch aus dem Humangenom synthetische Reads generieren. Sequenzierfehler und unbekannte Varianten werden analog wie bei synthetischen Datensätzen erzeugt. Jedoch gibt es bei authentischen Varianten nun eine weitere Möglichkeit: Durch Informationen von Datenbanken, die bereits bekannte Varianten enthalten, können Reads erzeugt werden, die genau einer Variante der Referenz entsprechen. Diese sollte ein variantentoleranter Readmapper perfekt alignieren können.

Analog zu der Erstellung der voll-synthetischen Reads wurden 2 Millionen Reads mit einer Länge von 100 Basenpaaren erstellt. In diesem Falle wurden nun die Informationen einer VCF-Datei zur Hilfe genommen, um Varianten zu erzeugen. Mit einer Wahrscheinlichkeit von 25% wurde diese Information für eine bestimmte Position befragt. Da nicht jede Position bekannte Varianten enthält, sinkt der reale Anteil der Varianten. Mit einer Wahrscheinlichkeit von jeweils 0.1% werden außerdem Basen gelöscht oder hinzugefügt, sowie mit einer Wahrscheinlichkeit von 1.8% verändert, um Sequenzierfehler und unbekannte Varianten zu simulieren.

Realdaten Für praxisnahe Experimente werden aber reale Daten verwendet, welche durch Sequenziermaschinen erstellt wurden. Hierbei zeigt sich die Praxistauglichkeit eines Readmappers. Bei der Evaluation ergibt sich aber ein Problem, was mit

der Problemstellung des Readmappings zusammenhängt. Man unterschiedet zwischen dem mathematischen und dem biologischen Problem (Holtgrewe et al., 2011). Das biologische Problem besteht darin, die Reads so zusammenzusetzen, dass die reale Reihenfolge erhalten bleibt. Das mathematische Problem jedoch hat keine Information über die reale Ursprungsposition und muss versuchen, die passendsten Stellen der Referenz zu finden. Idealerweise existiert nur eine Stelle, jedoch ist es durch repetitive Stellen der DNA möglich, dass multiple Alignments möglich sind.

Bei der Evaluation von Realdaten können wir also nicht überprüfen, ob ein Read an die korrekte Position aligniert wurde. Jedoch ist es möglich, mit einem Readmapper mit voller Sensitivität die jeweils beste Position für einen Read zu finden (Holtgrewe et al., 2011). Da die Berechnung eines solchen globalen Alignments sehr rechenintensiv ist, wurde darauf aus Zeitgründen verzichtet.

Im Folgenden werden nun die verwendeten Realdatensätze vorgestellt:

Der Datensatz **SRR062634** stammt aus den Datensätzen des *1000 Genomes Project*[9]. Er enthält 24 Millionen Reads und wurde an der Universität Washington erstellt. Bei dem sequenzierten Individuum handelt es sich um einen britischen Staatsbürger.

ERX358059 wurde an der Universität Stuttgart mit einer Sequenziermaschine des Typs *Illumina HiSeq 2000* generiert[10].

Am 4. Januar wurde der Datensatz **SRR1145796** heruntergeladen[11]. Zur Zeit ist der Datensatz leider nicht auf den Seiten des *NCBI* abzurufen. Er enthält 38 Millionen Reads.

5.5.2 Metriken

Für den Vergleich der Qualität eines Readmappers benötigt man Qualitätsmaße. Im Folgenden werden nun einige Metriken definiert, um Leistung messbar zu machen.

Laufzeit Eine Metrik ist die Messung der Laufzeit. Gemessen wird sie vom Startpunkt der Ausführung bis zum Zeitpunkt, an dem das Programm beendet ist oder eine Ausgabedatei vollständig erzeugt wurde. Um Abweichungen externer Faktoren, wie beispielsweise parallele Ausführung von anderen Prozessen, auszuschließen, werden mehrere Durchläufe unternommen und der Mittelwert gebildet.

Mapped / Unmapped Im Kontext des Readmappings lassen sich zwei wichtige Eigenschaften betrachten. Ziel ist es, möglichst alle Reads an die Referenz zu alignieren. Ein einfaches Qualitätskriterium ist also bereits die absolute Anzahl der Reads, welche das Programm nicht an die Referenz alignieren kann. Wenn ein Read gemapped wurde, bedeutet dies jedoch nicht automatisch, dass er auch an die korrekte Position gesetzt wurde. Diese Überprüfung ist, wie bereits oben beschrieben, nur bei synthetisch erzeugten Daten möglich.

[9] http://sra.dnanexus.com/runs/SRR062634/experiments
[10] http://sra.dnanexus.com/experiments/ERX358059
[11] http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSM1316301

Precision / Recall Im Bereich des Information Retrieval finden sich diverse Metriken, die sich mit der Analyse der korrekten Klassifikation befassen. Für den Bereich des Readmappings haben Siragusa et al. (2013) für die beiden populären Metriken **Precision** und **Recall** eine alternative Definition beschrieben. Danach definiert die Trefferquote *(engl. recall)* den Anteil der korrekt gemappten Reads an der Gesamtzahl:

$$recall = \frac{\text{Anzahl korrekt zugeordneter Reads}}{\text{Gesamtzahl zugeordneter Reads}}$$

Einige Readmapper alignieren einen Read an mehrere verschiedene Positionen. Die Präzision betrachtet nun die *unique reads*, welche nur an eine einzige Position des Referenzgenoms aligniert wurden:

$$precision = \frac{\text{Anzahl einmalig korrekt zugeordneter Reads}}{\text{Gesamtzahl aller Reads}}$$

F1-Metrik Die F1-Metrik schafft einen gewichteten Vergleich zwischen den Metriken *Precision* und *Recall* und bildet sie auf einem Wert im Intervall $[0, 1]$ ab (Sarwar et al., 2000). Sie berechnet sich nach folgender Formel:

$$F1 = \frac{2 \cdot Recall \cdot Precision}{Recall + Precision}$$

5.5.3 Testergebnisse

In diesem Abschnitt werden die Testergebnisse der drei Readmapper diskutiert. Begonnen wird mit der Laufzeit. Danach folgen detailliertere Experimente mit Datensätzen aus vorher genannten Kategorien.

5.5.3.1 Laufzeit

Zunächst wird die Laufzeit der drei Readmapper diskutiert. Die Gesamtlaufzeit setzt sich zusammen aus der Dauer für die Indizierung der Referenz und der Dauer für die eigentliche Alignierung.

Indizierung Die Indizierung der Referenz muss für jede Referenzdatei lediglich ein einziges Mal ausgeführt werden und kann dann im Alignierungsschritt für alle Reads verwendet werden. Es wurden folgende Laufzeiten gemessen:

	VATRAM	BWA	Bowtie2
Laufzeit	235 s	5 740 s	4 486 s

Tabelle 5.3: Laufzeiten für die Indizierung des Humangenoms.

Jedoch ist zu beachten, dass VATRAM die Referenzdatei bei jedem Programmstart neu einliest. Da die Berechnung des Indexes sehr schnell ist und das Laden einer

Index-Datei fast genauso lange dauert, ergibt sich durch die Speicherung des Indexes kein wirklicher Vorteil.

Bezüglich der Laufzeiten fällt auf, dass VATRAM erheblich schneller ist als BWA oder Bowtie2. Wenn also nur wenige Reads aligniert werden sollen und kein Index vorliegt, lohnt sich die Verwendung von VATRAM besonders.

Alignierung Im nächsten Schritt werden die Zeiten für die Alignierung an die Referenz gemessen. Dabei werden halbsynthetische und reale Datensätze betrachtet. Bei den halbsynthetischen (generiert mit Varianten aus einer VCF-Datei) zeigten sich folgende Ergebnisse:

Anzahl Reads	Readlänge	VATRAM	BWA	Bowtie2
2 Millionen	100 bp	396 s	208 s	218 s

Tabelle 5.4: Laufzeiten für die Alignierung eines halbsynthetischen Datensatz an das Humangenom.

Wie schon erwähnt, enthält die Laufzeit von VATRAM hier die Dauer der Indexerstellung von ca. 230 Sekunden. Zieht man diese Zeitdauer von den Ergebnissen ab, so ergeben sich bei VATRAM durchschnittlich 83 μs pro Read, während BWA und Bowtie 104 bzw. 109 μs pro Read benötigen. VATRAM ist also bei halbsynthetischen Reads schneller als herkömmliche Readmapper. Bei den realen Datensätzen wurden folgende Laufzeiten gemessen:

Datensatz	Anzahl Reads	Readlänge	VATRAM	BWA	Bowtie2
SRR062634	24 476 109	100 bp	1 371 s	780 s	780 s
SRR1145796	38 020 000	50 bp	780 s	540 s	880 s
ERX358059	33 393 586	50 bp	660 s	540 s	480 s

Tabelle 5.5: Laufzeiten für die Alignierung von realen Reads an das Humangenom.

Die Laufzeiten pro Read hängen stark vom verwendeten Datensatz ab. Obwohl *SRR062634* am wenigsten Reads enthält, dauert die Verarbeitung am längsten. Nichtsdestotrotz nimmt die Alignierung der realen Reads weniger Zeit in Anspruch als die Alignierung der halbsynthetischen Reads. So benötigt VATRAM beispielsweise etwa 47 μs pro Read des *SRR062634*-Datensatzes (ohne Indexerstellung), BWA und Bowtie sind jeweils sogar 32% schneller. Bei *SRR1145796* erreichen VATRAM und BWA jeweils ähnliche Laufzeiten, bei *ERX358059* ist VATRAM 20% schneller als BWA und immerhin 10% schneller als Bowtie. Bezüglich der Laufzeiten kann man zusammenfassend sagen, dass VATRAM mit der Geschwindigkeit von BWA und Bowtie mithalten und sie teilweise sogar übertreffen kann.

	VATRAM	BWA	Bowtie2
korrekt aligniert	99.11%	97.52%	96.56%
inkorrekt aligniert	0.55%	2.43%	0.75%
nicht aligniert	0.34%	0.05%	2.69%
Precision	0.9945	0.9757	0.9922
Recall	0.9965	0.9995	0.9728
F1	0.9955	0.9874	0.9824

Tabelle 5.6: Testergebnisse für synthetische Reads.

	VATRAM	BWA	Bowtie2
korrekt aligniert	93.47%	94.63%	92.24%
inkorrekt aligniert	5.91%	5.33%	4.06%
nicht aligniert	0.62%	0.04%	3.7%
Precision	0.9404	0.9466	0.9578
Recall	0.9933	0.9996	0.9614
F1	0.9662	0.9724	0.9596

Tabelle 5.7: Testergebnisse für halbsynthetische Reads mit zufälliger Veränderung einzelner Basen.

5.5.3.2 Synthetische Reads

Nachdem die einzelnen Programme auf ihre Laufzeit untersucht wurden, wird nun auf die Testergebnisse der synthetischen Reads eingegangen. Dabei wurden sowohl die Referenz als auch die Reads mit Hilfe des in Kapitel 5.2 beschriebenen Generators erstellt.

Dabei erzielte VATRAM, wie in Tabelle 5.6 dargestellt, den besten Anteil an korrekt gemappten Reads, welches sich auch an den anderen Metriken widerspiegelt. Bei Anwendung von BWA können die meisten Reads an die Referenz alignieren werden, jedoch sind dann 2.43% der Reads falschen Startpositionen zugeordnet. Gegenteiliges passiert bei Bowtie, welcher etwas vorsichtiger vorgeht und Reads lieber gar nicht aligniert, bevor diese er diese falsch zuordnet.

5.5.3.3 Halbsyntetische Reads

Zur Erinnerung sei bei diesen Reads gesagt, dass nun aus der realen Referenz des Humangenoms *GRCh37* Reads generiert wurden. Zunächst gehen wir dabei auf die Reads ein, die ohne Hinzunahme von bekannten Varianten generiert wurden. Nichtsdestotrotz wurden zufällig an jeder Stelle mit einer Wahrscheinlichkeit von 1% Fehler eingefügt. Bei 90% dieser Fehler handelt es sich um Substitutionen einzelner Basen (SNP), bei den restlichen 10% wurde eine Base entfernt (Deletion) oder hinzugefügt (Insertion).

	VATRAM	BWA	Bowtie2
korrekt aligniert	93.8%	93.71%	90.45%
inkorrekt aligniert	5.6%	6.23%	4.57%
nicht aligniert	0.6%	0.06%	4.98%
Precision	0.9436	0.9376	0.9518
Recall	0.9936	0.9993	0.9478
F1	0.9680	0.9675	0.9498

Tabelle 5.8: Testergebnisse für halbsynthetische Reads mit Veränderung einzelner Basen nach Informationen von bekannten Varianten.

Die Ergebnisse sind in Tabelle 5.7 dargestellt. Bei der Betrachtung der korrekt zugeordneten Reads schneidet BWA mit ca. 95% am besten ab; gefolgt von VATRAM und Bowtie2 mit jeweils einem Prozentpunkt weniger. Bei den inkorrekt alignierten Abschnitten liegt Bowtie2 vorne, kann aber auch 3.7% der Reads gar nicht an die Referenz alignieren.

Dies spiegelt sich bei den anderen Metriken wider. Bei der Präzision siegt Bowtie2. Für den Wert des Recalls liegen VATRAM und BWA dicht beieinander. Die Reihenfolge für die kombinierte F1-Metrik beginnt mit BWA, gefolgt von VATRAM und Bowtie2.

Für den zweiten Teil der halbsyntetischen Daten wurden nun bekannte Varianten hinzugefügt. Diese werden VATRAM durch eine weitere Parameterangabe als *VCF-Datei* übergeben. Dabei bestand eine Wahrscheinlichkeit von 25%, dass eine Variante verwendet wurde (siehe auch Abschnitt 5.2). Die Testergebnisse finden sich in Tabelle 5.8.

Mit Anteilen von ca. 93% korrekt zugeordneten Reads liegen VATRAM und BWA wieder Kopf an Kopf in derselben Größenordnung. Bowtie2 kann 3% weniger zuordnen, weist aber auch den geringsten Wert an falsch alignierten Reads auf. Jedoch ist bei Bowtie2 der Anteil von unalignierten Reads mit 5% am höchsten. VATRAM und BWA schneiden dort am besten ab.

Dies spiegelt sich auch in den anderen Metriken wider, wobei VATRAM und BWA abermals einen fast-perfekten Recall-Wert erreichen. Der Wert für die Präzision ähnelt sich ebenfalls bei diesen beiden Programmen - Bowtie ist nur einige Nuancen besser. Der Wert der kombinierten F1-Metrik weist VATRAM als Sieger aus, dicht gefolgt von BWA.

Bei Betrachtung der Ergebnisse stellt man fest, dass VATRAM mit der Qualität von *state-of-the-art*-Readmappern mithalten und diese auch teilweise übertreffen kann. Man sieht, dass VATRAM einen Read, welcher aligniert wird, mit hoher Wahrscheinlichkeit auch richtig aligniert. Bei BWA fällt auf, dass dieser Reads eher inkorrekt mappt, anstatt Reads gar nicht zuzuordnen.

	VATRAM			
SRR1145796	mit Varianten	ohne Varianten	BWA	Bowtie2
aligniert	96.7%	99.8%	97.1%	96.14%
nicht aligniert	3.3%	0.2%	2.9%	3.86%

	VATRAM			
ERX358059	mit Varianten	ohne Varianten	BWA	Bowtie2
aligniert	99.3%	99.92%	99.03%	98.1%
nicht aligniert	0.7%	0.08%	0.97%	1.9%

	VATRAM			
SRR062634	mit Varianten	ohne Varianten	BWA	Bowtie2
aligniert	97.4%	98.76%	99.49%	97.1%
nicht aligniert	2.6%	1.24%	0.51%	2.9%

Tabelle 5.9: Testergebnisse der Datensätze mit realen Reads.

5.5.3.4 Reale Reads

Wie bereits beschrieben, wurden drei Datensätze von realen Daten verwendet. Dabei wird nur auf die statischen Metriken *aligniert* und *nicht aligniert* eingegangen.

Bei den Ergebnissen der Realdaten in Tabelle 5.9 fällt auf, dass VATRAM bei der absoluten Anzahl an gemappten Reads bei den beiden Datensätzen *SRR1145796* und *ERX358059* die besten Ergebnisse erzielt. Bei ersterem sogar, wenn das Programm gar keine Informationen über bekannte Varianten hat. Werden diese zusätzlich verwendet, kann die Qualität sogar noch weiter gesteigert werden. Somit ist es möglich, VATRAM bei realen Reads in der Praxis einzusetzen.

5.5.4 Interpretation

Bei den Experimenten mit Realdaten werden Ergebnisse erzielt, die für die Praxistauglichkeit unseres Readmappers sprechen. Besonders die geringe Dauer der Indizierung für VATRAM sorgt dafür, dass eine schnelle Alignierung bei gleichzeitiger Wahrung der Ergebnisqualität, möglich ist. Die kombinierte Laufzeit von Indizierung und Alignierung bei VATRAM ist dabei nur um einige Augenblicke länger als die Laufzeit für die Alignierung bei BWA und Bowtie2. Diese benötigen darüber hinaus die 20-fache Laufzeit für die Indizierung der Referenz, welche allerdings nur einmal ausgeführt werden muss. Ein besonderer Einsatzzweck könnte daher die Alignierung an verschiedene Referenzen sein, da die Zeitdauer der Indizierung nur einen Bruchteil der von BWA und Bowtie2 entspricht.

Bei den Qualitäten lässt sich feststellen, dass die Ergebnisse von VATRAM dieselbe Größenordnung wie die Ergebnisse von BWA besitzen, bei einigen Realdatenexpe-

rimenten werden sogar bessere Ergebnisse erzielt. Besonders positiv ist, dass die Alignierung von Realdaten bei Berücksichtigung von bekannten Varianten zu einem höheren Anteil von alignierten Reads führt.

Im nächsten Abschnitt wird nun auf einen weiteren Benchmark zur Qualitätsüberprüfung eingegangen, welcher einen objektiven und unabhängigen Blick auf die Ergebnisse werfen soll.

5.6 Analyse der GCAT-Reads

Genome Comparison & Analytic Testing (GCAT) ist eine Webseite, die einen komfortablen Weg um verschiedene Readmapper gegeneinander zu testen bietet. Die Seite stellt mit dem Read-Generator `dwgsim` aus dem Humangenom Reads erzeugte Reads zur Verfügung. Beim Starten eines Tests wird eine Fastq-Datei mit 12 Millionen Reads zum Download angeboten, die Positionen von denen diese Reads stammen sind unbekannt. Mit dieser Datei führt man anschließend seinen Readmapper aus und lädt die resultierende BAM-Datei zu GCAT hoch, wo sie ausgewertet wird. Die Auswertung besteht aus einer grafischen Aufbereitung der gemappten Reads und Informationen, wie viele Reads falsch oder nicht aligniert wurden. Die Werte lassen sich dann mit anderen ausgeführten Tests vergleichen.

Es zeigte sich, dass unser Readmapper mit Reads von der GCAT-Webseite deutlich schlechtere Ergebnisse lieferte als mit Reads aus anderen Quellen. Insbesondere war die Anzahl der Reads, die nicht gefunden wurden, 10 bis 20 Prozentpunkte höher als bei von unserem Read-Generator generierten Reads oder Reads aus echten Sequenziervorgängen. Um diesem Problem zu begegnen wurde die Wahl der LSH-Parameter optimiert, wodurch das Problem entschärft wurde, sodass nur noch etwa 6 % der Reads nicht gefunden werden. Die Ergebnisse sind allerdings immer noch schlechter als bei anderen Datensätzen.

Herauszufinden, warum gerade die Daten von GCAT „anders“ sind als die Reads aus allen anderen getesteten Quellen, gestaltet sich schwierig, da wenig Information über den Erstellungsprozess dieser Reads bekannt ist. Ein erster Ansatz war die Analyse der in den Reads vorhandenen eindeutigen q-Gramme (Tabelle 5.10).

Die deutlich geringere Anzahl an verschiedenen q-Grammen ist ein erster Hinweis auf eventuell ungleich verteilte Reads. Um diesen Verdacht weiter zu erhärten wurde mit BWA ein unabhängiger Test durchgeführt. Dieser Mapper hat auf den GCAT-Reads nur sehr geringe Fehlerquoten (weniger als 5 % nicht gefunden, weniger als 2 % fehlerhaft). Auf diesen Ergebnissen konnte nun eine Auswertung der gefundenen Positionen durchgeführt werden. Zuerst ist dabei die Verteilung der Reads auf die jeweiligen Chromosome interessant. Die im Folgenden angegebenen Daten beziehen

reads	unique q-grams ($q = 16$)
GCAT	273 965 337
random	366 456 231

Tabelle 5.10: Q-Gramm-Verteilung in GCAT- und Random-Reads.

Mapper	BWA 0.7.10		VATRAM (19.01.2015)	
Reads gesamt	11 945 249	(100 %)	11 945 249	(100 %)
Mapped Reads	11 822 544	(99 %)	11 206 203	(94 %)
Unmapped Reads	127 848	(1 %)	739 046	(6 %)
Mapped in Chr. 19	11 636 973	(97 %)	10 965 283	(92 %)
Mapped in Chr. 1	20 861	(0 %)	25 366	(0 %)

Tabelle 5.11: Ergebnisse für GCAT-Benchmark und Verteilung der Mapping-Positionen.

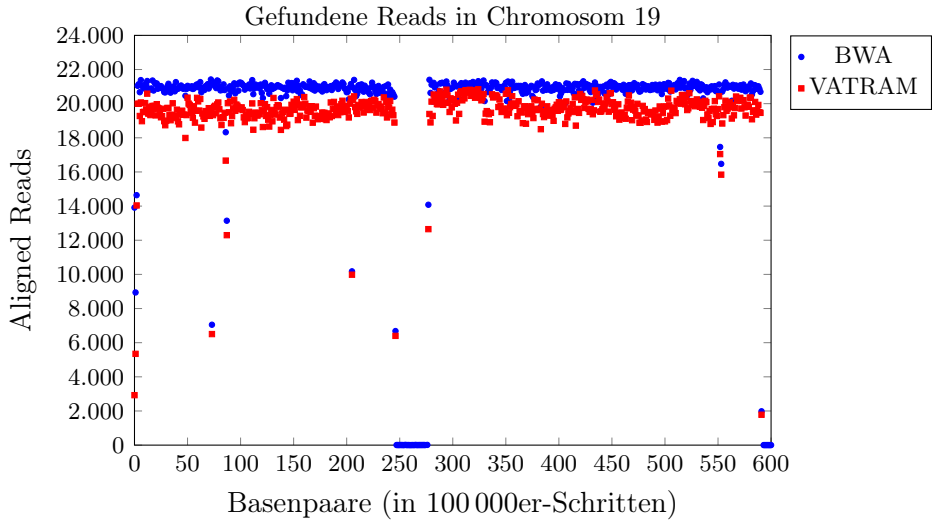


Abbildung 5.25: Anzahl der gefundenen Reads in 100 000er-Abschnitten von Chromosom 19.

sich dabei auf den Datensatz *GCAT_Reads_039*, andere getestete Datensätze von GCAT zeigen allerdings ähnliche Verteilungen. Für unseren Readmapper wurden nur Reads der Länge 100, single-ended mit geringer Fehlerzahl getestet. Werden die GCAT-Reads mit BWA aligniert, ergeben sich durchschnittlich 2,2 Fehler pro Read.

Wie man an den Ergebnissen von BWA in Tabelle 5.11 sieht, befindet sich mit 97 % die überwältigende Mehrheit der gefundenen Reads in Chromosom 19. Das Chromosom mit den zweitmeisten Treffern, Chromosom 1, ist weit abgeschlagen und in allen anderen Chromosomen sind noch weniger Treffer zu finden. Bei der weiteren Analyse kann man sich also auf Chromosom 19 fokussieren, dort ist noch interessant zu erfahren, wo sich die Reads befinden. Hierzu wurde das Chromosom 19 in 100 000er-Abschnitte eingeteilt und jeweils die Anzahl der dort gefundenen Reads ausgegeben. Das Ergebnis ist in Abbildung 5.25 dargestellt.

Man sieht, dass die Reads abgesehen von dem Ausreißer zwischen den Positionen 250 000 000 und 275 000 000 gleichmäßig auf dem Chromosom 19 verteilt sind. Das besagte Intervall auf Chromosom 19 enthält dabei ausschließlich unbekannte Basen, sodass dort keine sinnvollen Reads erzeugt werden können. BWA findet im Schnitt 21 000 Reads auf 100 000 Basenpaare, VATRAM etwa 19 500. Dies ist eine enorme Dichte an Reads in der Eingabe, da so jedes Basenpaar in durchschnittlich 20 Reads enthalten ist. Deshalb stellt sich die Frage, wie repräsentativ diese Daten für Reads

aus echten Sequenzierungsvorgängen sind. Eine hohe Abdeckung kann dort durchaus auftreten, die Beschränkung auf ein Chromosom macht den Benchmark aber weniger interessant. Die gleichmäßige und sehr dichte Verteilung der Reads lässt aber auch darauf schließen, dass die Ursache für die schlechten Ergebnisse nicht die Reads sind, sondern vielmehr die Referenz, aus der sie stammen. Anscheinend sind die kurzen Ausschnitte aus Chromosom 19 entweder untereinander sehr ähnlich oder es gibt viele Ähnlichkeiten mit anderen Chromosomen, sodass VATRAM Schwierigkeiten hat, die richtige Position zu finden.

Kapitel 6

Ausblick

Neben den umgesetzten Ideen für unseren Readmapper gibt es noch eine ganze Reihe weiterer Verbesserungen bzw. Erweiterungen, die es aus zeitlichen Gründen nicht mehr in das finale Programm geschafft haben. Diese weiterführenden Ideen sollen hier gesammelt und im Ansatz beschrieben werden, damit sie nicht verloren gehen. Dieses Kapitel kann daher auch Anreize für weitergehende Forschung oder Projektarbeit bieten.

6.1 Erweiterungen der vorhandenen Algorithmen

6.1.1 Verwendung mehrerer q-Gramm-Hashfunktionen

Es wurde in Kapitel 4.1.3 erklärt, wie MinHash zur Approximation der Jaccard-Ähnlichkeit verwendet werden kann. Die q-Gramme der betrachteten Referenzgenomabschnitte wurden nicht direkt zur Berechnung benutzt, sondern nur deren Hashwerte gemäß einer festen q-Gramm-Hashfunktion. Eine dadurch entstehende Problematik wurde in Abschnitt 5.3.2.1 bereits angesprochen: Beim Verwenden eines Rolling-Hashes beeinflussen vor allem die Anfangsbuchstaben im q-Gramm die höherwertigen Bits des entstehenden Hashwertes. Per XOR mit vorher festgelegten Zufallswerten werden verschiedene Permutationen der q-Gramm-Menge simuliert, die dann zur MinHash-Bestimmung verwendet werden. Es stand zur Diskussion, ob per LSH nicht gefundene Reads erneut mit anderen Parametern gesucht werden sollten. Dabei können sich beispielsweise die Zufallswerte ändern. Das Problem ist an dieser Stelle, dass eine erneute Suche nur mit nicht gefundenen Reads nicht sinnvoll im Kontext der WindowSequence-Berechnung wäre. So könnte beispielsweise im ersten Durchlauf der erste Teil des Reads in einem Fenster gefunden werden, das dann anschließend nicht mehr im zweiten Durchlauf verwendet werden würde. Das optimale Mapping auf eine längere WindowSequence hätte somit nicht entdeckt werden können. Stattdessen könnten mehrere q-Gramm-Hashfunktionen simultan in einem Durchlauf verwendet werden. Ein denkbar einfache zweite Hashfunktion könnte beispielsweise die vordere und hintere Hälfte des ersten Hashwerts vertauschen oder den gesamten ersten Hashwert rückwärts betrachten.

6.1.2 Mapping von Paired-End-Reads

Bei Paired-End-Reads wird ein DNA-Abschnitt erst von dem 5'-Ende aus sequenziert und dann anschließend vom 3'-Ende aus. Die sequenzierten Teile nennt man Tags. Die Mitte des Abschnitts bleibt unsequenziert. Damit hat man einen *Tag1* mit den initialen Basenpaaren des 5'-Ende und einen *Tag2* mit den Basenpaaren des 3'-Endes und desweiteren kennt man den Abstand zwischen den beiden Tags. Der Abstand zwischen *Tag1* und *Tag2* wird *insert_size* genannt. Bei einer PET-Sequenzierung[1] weiß man, dass eine Deletion stattfand, wenn die beiden Tags weiter auseinander aligniert wurden. Analog dazu kann man eine Insertion erkennen, wenn der PET-Abstand enger zusammen liegt.

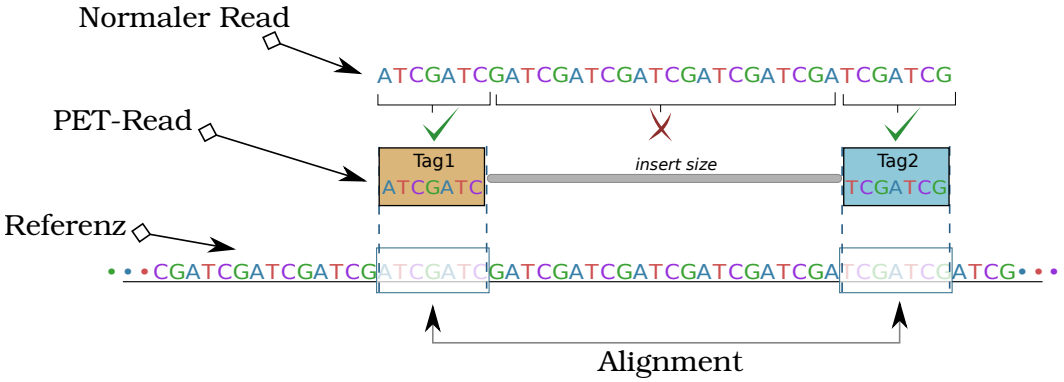


Abbildung 6.1: Oberer Read ist ein Single-End(normaler) Read. Darunter wird dieser Read als PET-Read dargestellt. Anfangbereich und Endbereich werden sequenziert und die Abstand zwischen diesen, der sogenannten *insert_size* wird vermerkt. Darunter wiederum befindet sich die Referenz und markierte Stellen, an denen das Anfangstag und das Endtag an der Referenz aligniert wurden.

Bei Paired-End geht man davon aus, dass man einen DNA-Abschnitt mit einer ausreichend eindeutigen Sequenz mit zwei Abschnitten und einer Angabe der Länge dazwischen eindeutig identifizieren kann. Eine Länge von 100 bp ist heutzutage üblich. Aufgrund der geringen Kosten und Aufwand ist diese Methode sehr beliebt. Man kann Insertions und Deletions durch die strukturellen Information (Linker/Abstand zwischen den Tags) erschließen. In den meisten Bibliotheken finden sich deshalb Paired-End-Reads wieder. Dies macht diese Variante von Reads für unsere PG interessant. Die Unterstützung für PET ist bisher noch nicht eingebaut.

Abbildung 6.1 zeigt einen normalen DNA-Abschnitt der mit PET sequenziert wurde. Es werden nur Anfangs- und Endbereiche des Reads sequenziert und anstelle der Mitte die *insert_size* angegeben. Man aligniert Anfang und Ende der PET-Reads. Der sequenzierte Bereich wird durch Anfangs- und End-Tags gefunden. Mit der zugehörigen *insert_size* lässt sich dann ermitteln, ob sich indels in der Referenz befinden. PET-Reads sind dort effizient, wo sich DNA-Abschnitte nicht durch einfache Single-End-Reads ausfindig machen lassen. Beispielsweisen an Stellen, wo sich die Stellen extrem wiederholen. Ein Single-End-Read-Alignment würde wiederholt wiedergefunden werden, ohne, dass wirklich der gewünschte Abschnitt in dem Bereich der Single-End-Reads befindet. Durch die Hinzunahme eines weiteren Bereichs kann damit der DNA-Abschnitt „umklammert" werden und ein Fund ist nur

[1] Abkürzung für Paired-End-Tagging

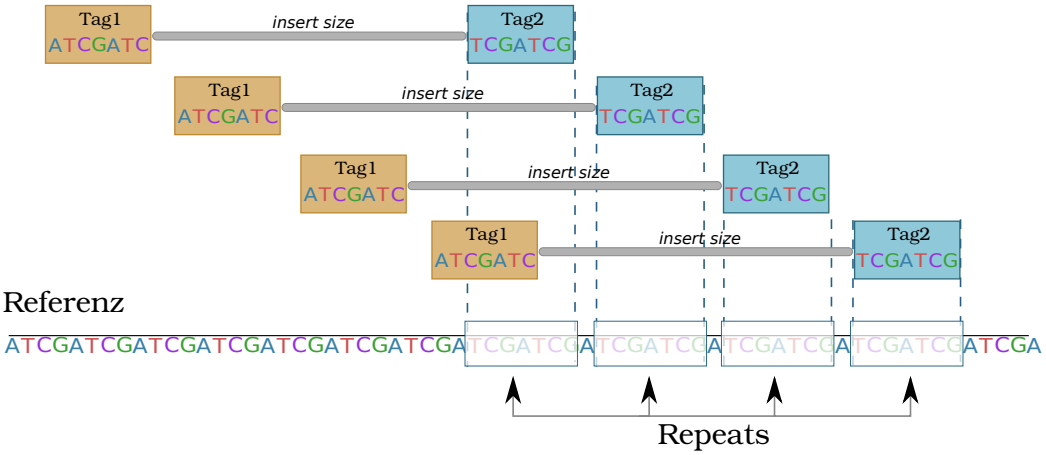


Abbildung 6.2: In diesem Szenario wird eine Refenzabschnitt dargestellt der hohe Wiederholungsraten(Repeats) aufweist. Hier wird das End-Tag (*Tag2*) eines PET-Reads an mehreren Stellen in der Referenz gefunden. Das Start-Tag (*Tag1*) hingegen ist nicht in diesem Teil der Referenz. Eine PET-Alignment wäre damit kein Fund. Bei Single-End-Reads ginge man bei einer Genanalyse davon aus, dass der gesuchte DNA-Abschnitt womöglich häufiger gefunden wurde. Dies erhört in der Praxis die Genauigkeit gegenüber Single-End-Reads. (In Anlehnung an: http://www.illumina.com/content/dam/illumina-marketing/images/technology/paired-end-sequencing-figure.gif)

dann ein Fund, wenn er beide Reads beinhaltet. Abbildung zeigt eine Situation mit häufigen Repeats. Im Gegensatz zu langen Reads braucht man nur einen kleinen Teil zu sequenzieren und bei der Suche ergibt sich damit ein Verschnellerung und Vereinfachung.

VATRAM und PET-Reads Die Reads werden als Dokumente in den LSH eingegeben. Es wird betrachtet, wo diese gemapped wurden. Danach aligniert der Aligner die PET-Reads. Die die Tags eines PET-Reads können wir wie gewöhnlich, voneinander getrennte Dokumente behandeln, bis nach dem Alignment. Am Ende müssen wir natürlich eine Information speichern, sodass wir den anderen Teil eines Tags finden. Dazu müsste die Read-Datenstruktur um eine PET-ID erweitert werden.

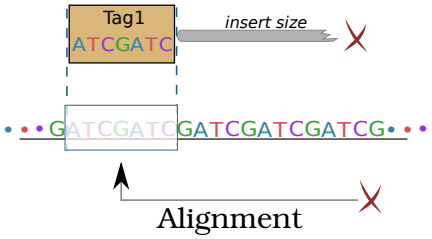


Abbildung 6.3: Ein Alignment eines Tags ohne zugehörigen anderen Tag wird nicht als Alignment gewertet.

An dieser Stelle kann man eine Optimierung vornehmen, die für die PET-Reads (genauer gesagt für die Laufzeit) von Vorteil ist, jedoch nicht für das PET-Read-Alignment notwendig ist. Im LSH-Algorithmus lassen sich die Vorkommen eines PET-Reads in einem Fenster zählen. Liegen die Fenster nicht zu weit auseinander, so kann man für *Tag1* nachsehen, ob *Tag2* gemapped wurde. Wurde *Tag1* beispiels-

weise in Fenster 1337 fünf Mal gefunden und in Fenster 1330-1336 kein Mal, so kann man davon ausgehen, dass der PET-Read nicht gemapped wurde (Abbildung 6.3 zeigt ein Mapping für *Tag1* ohne *Tag2*). Finden sich Mappings auf die Fenster 1337 und 1333, so kann man, wenn beispielsweise *Tag1* zwei Mal und *Tag2* zwei Mal gemapped wurden, die möglichen Kombinationen für ein Alignment durchgehen.

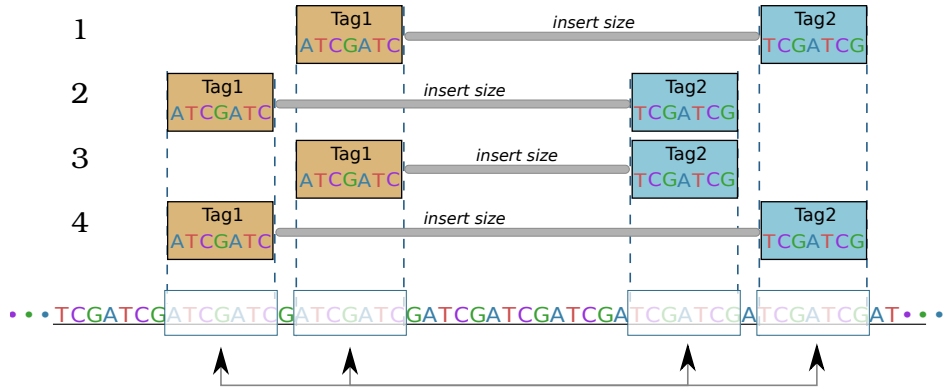


Abbildung 6.4: Darstellung mehrer möglichen Alignments für mehrfache Funde für *Tag1* (zwei Mal links) und *Tag2* (zwei Mal rechts).

Nach dem Mapping betrachten wir die Alignments der Tags. Es kann zu Fällen kommen bei denen mehrere Alignments beider *Tags 1* und *2* eines PET-Reads vorkommen. In diesem Fall muss man sich mit der Kombinationen der mehrfachvorkommenden *Tag1* und *Tag2* Kombinationen beschäftigen. Abbildung 6.4 zeigt ein Szenario, bei dem dies der Fall ist. Die Fälle 1 und 2 sind normale Alignments des PET-Reads. Bei einer PET-Sequenzierung weiß man, dass eine Insertion stattfand, wenn die beiden Tags enger zusammen aligniert wurden. Aus diesem Grund ist Fall 3 eine PET-Alignment mit möglicher Insertion. Man kann erkennen, dass eine Deletion stattfand, wenn der PET-Read weiter auseinander aligniert wurde. Aus diesem Grund ist Fall 4 ein PET-Alignment mit möglicher Deletion.

6.1.3 Behandlung von Indels beim LSH

Zurzeit werden keine Indels im LSH berücksichtigt. Um Indels bereits im LSH zu bearbeiten könnte man eine Variante in Fenster aufteilen und fügt links und rechts halbüberlappende Fenster hinzufügen.
Ist ein Read kürzer als ein Fenster oder nicht in volle Fenster teilbar, so fügt man Zeichen aus der Referenz hinzu und fügt dann die halbüberlappenden Fenster hinzu.

Der Rand ließe sich durch eine einfache Ganzzahlige Division mit Rest errechnen:
$r = var\%FensterLaenge$, wobei r der Rest ist.
Ist $r > 0$ dann addiere $f + 1$ und fülle um $FensterLaenge - r$ Zeichen auf. Die Zeichen sind aus der Referenz.

Die neu generierten Fenster übergibt man dann dem LSH wie gewöhnliche Dokumente. Die Erstellung dieser Fenster lässt sich in $\mathcal{O}(m + (\frac{m}{w} + 1))$ generieren (mit

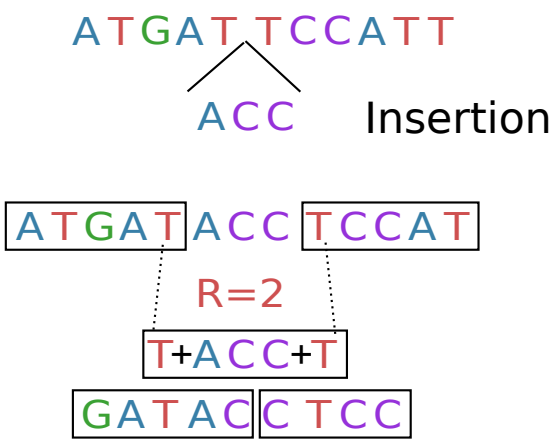


Abbildung 6.5: Diese Abbildung zeigt einen DNA-Abschnitt mit einer Varianten und darunter einen ohne Variante. Üblicherweise werden die Sequenzen in Fenster unterteilt, um diese dem LSH zu übergeben. Im Fall der Variante werden für jede Variante neue Dokumente erstellt. In dieser Abbildung zeigt sich eine Insertion. Für den Insertion-String wird ein Fenster generiert und für den Fall einer nicht genau passenden Länge zur Fensterlänge Zeichen aus der Referenz (In diesem Bild die obigen Sequenz) übernommen. Links und rechts werden nun halbüberlappende Fenster erstellt.

gleicher zusätzlicher Speicherzunahme). Bedingt durch die Häufigkeit der Varianten ist dieser Mehraufwand nicht problematisch.

6.2 Verarbeitung von langen Reads

Wir haben uns während unserer Projektgruppe hauptsächlich auf die Reads konzentriert, die mit der zu der Zeit aktuellen Illumina-Sequenziertechnik gewonnen wurden. Das bedeutet, dass sich unsere Optimierungen fast ausschließlich auf Readlängen von etwa 100 Basenpaaren beschränken. In diesem Abschnitt soll diskutiert werden, welche Probleme bei der Verarbeitung von längeren Reads mit VATRAM auftreten können und wie diese Probleme behoben werden könnten.

6.2.1 Mapping von langen Reads

Beim Verarbeiten längerer Reads müsste man einige LSH-Parameter anpassen. Ursprünglich hatten wir das Ziel, auch eine Parameteroptimierung für 1000er Reads durchzuführen, dies war jedoch aus Zeitmangel leider nicht mehr möglich. Dieser Abschnitt behandelt theoretisch, was bei der Verarbeitung von längeren Reads beachtet werden muss oder wie man alternativ vorgehen kann.

Wahl der Parameter Fenster und Reads sollten ungefähr die gleiche Länge haben, da sich sonst die Minima der exklusiv veroderten q-Gramme zu sehr unterscheiden. Naheliegend wäre, für Reads der zehnfachen Länge auch die Fensterlänge und der Fensterabstand entsprechend um Faktor zehn zu erhöhen. Durch die geringe Anzahl von Fenstern, kann man die Signaturlänge entsprechend erhöhen, um so die Menge der Paare aus Band-Hash und Fensterindex und damit letztendlich den

Speicherverbrauch konstant zu halten. Dies bereitet bei Verwendung der SuperRank-Datenstruktur jedoch Probleme, da die oberste Schicht pro Band 12,6 MB benötigt. Für 480 Bänder ergibt sich so ein Speicherverbrauch von 6 GB für eine eigentlich leere Datenstruktur. Abhilfe könnte das Benutzen einer dreischichtigen Rankdatenstruktur schaffen. Man kann so den Speicherverbrauch für die leere Datenstruktur um Faktor 64 senken und käme dann auf 100 MB, was akzeptabel wäre. Alternativ würde sich auch wieder die CollisionFreeBandHashTable eignen, da sie mit dünn besetzten Datenmengen erheblich besser umgehen kann und schneller ist als eine dreischichtigen Rankdatenstruktur. Bei längeren Reads sollte auch der Parameter *extend-number*, der angibt um wie viele Basenpaare ein einzelnes Fenster erweitert werden soll, angepasst werden. Naheliegend wäre, *extend-number* um denselben Faktor zu erhöhen wie die Fensterlänge. Die Parameter *extend-band-multiplicator* und *contract-band-multiplicator* sind so implementiert, dass sie unabhängig von Read- und Signaturlänge sinnvolle Ergebnisse liefern, und müssen daher nicht verändert werden.

Ob es sinnvoll ist, bei längeren Reads auch längere q-Gramme zu verwenden bleibt unklar und müsste getestet werden. Prinzipiell reicht eine q-Grammlänge von $q = 16$ aus, da sich das Referenzgenom nicht vergrößert und die Anzahl der verschiedenen 16-Gramme mit 4 Milliarden bereits die Länge der Referenz übersteigt. Wenn man davon ausgeht, dass die Fehlerdichte bei langen Reads geringer ist, könnte die Wahl längerer q-Gramme aber durchaus bessere Ergebnisse liefern. Dabei muss man beachten, dass als Wörtlänge für ein q-Gramm wenigstens 64-Bit benutzt werden sollte.

Bei einer größeren Signaturlänge erhöht sich auch die durchschnittliche Anzahl Treffer für ein Fenster, weswegen *min-count* vermutlich vergrößert werden muss. Um welchen Faktor und wie sich die Wahl der q-Grammlänge auf die Wahl von *min-count* auswirkt, ist ebenfalls offen.

Read-Splitting Ein alternativer Ansatz ist, lange Reads in einzelne Teile aufzuteilen. Für jedes dieser Teile startet man eine Anfrage an den LSH-Index. Beim Zusammenfügen der Fensterindizes einer Anfrage berücksichtigt man dann alle Teilergebnisse und sucht Häufungen von Fensterindex-Bereichen. Dieser Ansatz entspricht weitestgehend der Berücksichtigung von Paired-End-Reads (siehe Abschnitt 6.1.2).

6.2.2 Alignierung von langen Reads

Bei der Evaluierung des Aligners wurde bereits das Speicherplatzproblem des Aligners bei langen Reads mit mehreren tausend Basenpaaren angesprochen (vgl. Abschnitt 5.4.3). Um den Speicherverbrauch zu reduzieren, wird eine Datenstruktur benötigt, die nur die tatsächlich benutzten Matrixfelder speichert, aber dennoch einen effizienten Lese- und Schreibzugriff in konstanter Zeit ermöglicht.

Der einfachste Ansatz wäre statt eines großen Arrays eine Hashmap zu benutzen und diese nach jedem Alignierungsvorgang wieder zu leeren. Ein kurzer Test ergab, dass der Speicherverbauch damit tatsächlich deutlich langsamer ansteigt, dafür allerdings die Laufzeit zur reinen Array-Lösung auf mehr als das Zwanzigfache ansteigt. Zusätzlich zur Hashmap wurde danach ein Array benutzt, welches die letzten t Spalten der Matrix enthält und zyklisch überschrieben wird. Technisch gesehen bildet dieses

Array also einen FIFO-Cache für die Hashmap, der immer dann eine Matrixspalte in die Hashmap kopiert, wenn sie aus dem Cache verdrängt wird. Für $t = 100$ beschleunigt der Cache bereits nahezu alle Lese- und Schreibzugriffe, allerdings ist der Overhead für das Füllen und Leeren der Hashmap immer noch so groß, dass diese Implementierung nach groben Messungen trotzdem noch die Zehnfache Laufzeit der reinen Array-Implementierung benötigt.

Eine deutlich bessere Lösung könnte die Verwendung einer Rank-Datenstruktur sein. Bei einer solchen Datenstruktur kann man (genau wie bei einer Hashmap) die Existenz eines Matrixfeldes erfragen. Dadurch entfallen bei der Rowskip-Optimierung des Aligners die präventiven Schreibzugriffe, die ein Lesen von undefinierten Feldern verhindern sollen. Der Aligner besitzt die günstige Eigenschaft, dass alle Schreibzugriffe in aufsteigend sortierter Reihenfolge erfolgen, d. h. die Matrix wird spaltenweise beschrieben und innerhalb einer Spalte mit monoton wachsendem Zeilenindex. Außerdem erfolgen keine Löschoperationen. Dadurch wäre es problemlos möglich die Matrix platzsparend in einer Rank-Datenstruktur zu speichern. Die Vorteile gegenüber einer Hashmap wären der geringere Overhead bei der Adressierung (je nach Kollisionsstrategie der Hashmap) und eine bessere Cache-Effizienz, da die Rank-Datenstruktur im Wesentlichen von vorne nach hinten gelesen und beschrieben wird.

Da sich mit der Länge der Reads auch deren erwartete Fehlerzahl erhöht, könnte die Laufzeitskalierung problematisch werden. Statt einen Read in einem Durchlauf komplett mit voller Fehlerschranke zu alignieren, könnte der Read auch aufgesplittet werden, wobei jeder dieser Teile auch nur mit einem Bruchteil der Fehlerschranke aligniert wird. Da sich die Fehler nicht gleichmäßig auf die gesamte Länge des Reads verteilen, müssen einige Splits eventuell mehrfach mit steigender Fehlerschranke aligniert werden. Die Alignments der einzelnen Splits müssen anschließend korrekt zusammengefügt werden. Dieses Vorgehen bietet einige Vorteile:

- Nicht jeder Split muss gegen das gesamte Referenzintervall aligniert werden. Sobald die Position einiger Splits bestimmt wurde, kann man das Intervall der restlichen Splits stark eingrenzen.
- Die Fehlerschranke für jeden Split ist relativ klein. Da die Laufzeit annähernd linear mit der Fehlerzahl steigt, sind Splits sehr effizient zu alignieren.
- Falls sich einzelne Splits aufgrund hoher Fehlerkonzentration nicht alignieren lassen, kann für den Read trotzdem ein gültiger Cigar-String berechnet werden, in dem der nicht alignierte Split entsprechend markiert wird.

Hier ergibt sich sowohl bezüglich der Fehlertoleranz als auch der Laufzeit viel Optimierungspotenzial seitens eines Readsplitters.

6.3 Verwendung von q-Gramm-Indizes

Locality-Sensitive Hashing hat eine gewisse Ähnlichkeit mit einem normalen q-Gramm-Index, wenn die Anzahl der Bänder auf 48 festgelegt wird. Die q-Gramm-Länge bleibt bei 16. Es ist zu untersuchen, wie sich ein solcher q-Gramm-Index in Laufzeit

und Speicherverbrauch gegenüber LSH verhält. LSH bietet aber eine höhere Skalierbarkeit durch die vielen Parameter, q-Gramm-Indizes können aber auch kompakter dargestellt werden, etwa durch den q-Group-Index in PEANUT (Köster und Rahmann, 2014).

6.4 Wahrscheinlichkeiten von Sequenzierfehlern

Die FASTQ-Dateien der Reads enthalten neben der DNA-Sequenz auch für jede ihrer Basen je eine Qualitätsangabe. Die Qualitätswerte können als Wahrscheinlichkeiten für Sequenzierfehler interpretiert werden. Ist einer Position im Read ein niedriger Qualitätswert zugewiesen, besteht eine hohe Wahrscheinlichkeit, dass die angegebene Base durch einen Sequenzierfehler nicht korrekt gelesen worden konnte. Beide Teile unseres Readmappers, der LSH-Mapper ebenso wie der Aligner, verarbeiten diese Zusatzinformation noch nicht.

In der Mapping-Phase wäre es möglich, die Read-Strings ebenfalls als IUPAC-Strings zu behandeln, wie es bei der Indizierung des Referenzgenoms geschieht. Hierbei können Basen mit geringer Qualität in das IUPAC-Zeichen N umgewandelt werden. Somit werden zusätzliche q-Gramm-Hashwerte erzeugt, wie es bei den SNPs in der Referenz der Fall ist. Bei vielen Basen mit niedrigen Qualitätswerten müsste natürlich auch hier die „kombinatorische Explosion“ beachtet werden, damit nicht zu viele zusätzliche q-Gramme verarbeitet werden.

Beim Alignment der Reads kann einerseits ebenfalls auf das IUPAC-Zeichen N zurückgegriffen werden, wodurch die betroffenen Basen effektiv von der Zählung der Fehler ausgenommen werden, da ein N immer als Match zählt. Andererseits könnten die Qualitätswerte auch direkter verwendet werden, indem sie in die Bewertung nicht als ganzzahlige Fehler sondern Bruchteile dieser eingehen. So könnte bei einem Match mit der Referenz dennoch ein Fehler von 0,25 registriert werden, falls die Base im Read eine hohe Fehlerwahrscheinlichkeit aufweist. Genauso könnten bei einem Mismatch in diesem Fall auch nur 0,75 Fehler gezählt werden. Für die entsprechenden Fehlerbruchteile bietet es sich natürlich auch an, diese abhängig vom Qualitätswert zu definieren, anstatt konstanter Werte zu benutzen.

Kapitel 7

Fazit

Zum Abschluss wird die Arbeit unserer Projektgruppe hier zusammengefasst und beurteilt. Das Ziel der Projektarbeit war die Entwicklung eines variantentoleranten Readmappers im Hinblick auf aktuelle Sequenziertechnologien. Der Schwerpunkt lag dabei auf dem Umgang mit Varianten, da dieser Aspekt von den meisten aktuellen Readmappern nicht explizit berücksichtigt wird.

Genvarianten wurden bereits bei der Planung unseres Readmappers VATRAM bedacht. Das verwendete LSH-Verfahren ist auch bei starken lokalen Abweichungen eines Reads in der Lage eine geeignete Position im Referenzgenom für diesen zu finden. Der entwickelte Aligner bietet eine Garantie dafür, dass Alignments mit beschränkter Fehlerzahl auch unter Verwendung vieler Varianten gefunden werden. Das Konzept von VATRAM war für die Aufgabenstellung also angemessen.

Die Evaluierung des Readmappers zeigt, dass die finale Version von VATRAM auch auf großen Datenmengen funktioniert und sich qualitativ durchaus mit anderen Referenz-Readmappern messen lassen kann. Die Laufzeit bewegt sich auf einem guten Niveau, auch wenn sie tendenziell höher ausfällt als bei den getesteten Referenz-Readmappern. Dies liegt jedoch zum großen Teil daran, dass die Laufzeitmessung von VATRAM zusätzlich noch die Indexerstellung enthält, da die Ladezeiten des Indexes im Gegensatz zu den anderen Readmappern annähernd so hoch ist wie die Berechnungszeit. Dies wäre ein erster Kritikpunkt, da sich VATRAM ohne eine effiziente Ladefunktion für den Index bei kleinen Datensätzen unnötig träge verhält. Im Gegenzug macht sich die Indexerstellung bei hinreichend großen Datensätzen nicht so stark bemerkbar, da sie im Vergleich zu den anderen getesteten Readmappern sehr schnell ist. VATRAM ist dort praktisch ohne Vorberechnungen nutzbar, während die anderen Algorithmen ein Vielfaches an Zeit brauchten, um ihren Index über dem Referenzgenom zu erstellen.

Bezüglich der Variantentoleranz ergibt sich ein gemischtes Bild. Zwar legt VATRAM beim Mapping von Reads, die Varianten enthalten, etwas gegenüber den anderen Readmappern zu, doch reicht dies nicht aus, um jene immer zu überbieten, wie wir es uns erhofft hatten. Hier offenbart sich eine mögliche Schwäche des LSH-Verfahrens, da dieses bei der Indexerstellung keinerlei Indelvarianten berücksichtigt. Dieses Ergebnis ist angesichts unserer ursprünglichen Zielsetzung etwas ernüchternd.

Darüber hinaus gab es noch viele Ideen, die in Kapitel 6 beschrieben wurden, es aber nicht mehr in die finale Version geschafft haben. Die Umsetzung einiger Ideen hätte das Endergebnis nochmals aufwerten können und wäre durch besseres Zeitmanagement oder andere Priorisierung vielleicht möglich gewesen. Auch der Arbeitsstil war verbesserungswürdig, wie wir in späteren Sitzungen festgestellt haben.

Betrachtet man das Ergebnis unserer Projektgruppe im Kontext, so fällt das Fazit allerdings überwiegend positiv aus. Zum einen fiel unsere Projektgruppe mit sieben Studierenden sehr klein aus — frühere Projektgruppen hatten bis zu zwölf Teilnehmer. Außerdem war der Fachbereich der Bioinformatik, insbesondere der DNA-Analyse für uns alle weitestgehend unbekannt. Dies machte eine Einschätzung von Konzepten und Ideen sehr schwierig. Den von uns gewählten Ansatz gab es unseres Wissens nach in dieser Form noch nicht, sodass wir zu Beginn der Projektgruppe nicht absehen konnten, ob unsere Ideen aufgehen würden. Dennoch stand am Ende ein fertiger Readmapper, welcher als durchaus konkurrenzfähig zu anderen Referenz-Readmappern bezeichnet werden kann.

Literaturverzeichnis

1000 Genomes Project. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, 2010.

A. Ahmadian, M. Ehn, and S. Hober. Pyrosequencing: history, biochemistry and future. *Clinica chimica acta*, 363(1):83–94, January 2006.

A. J. Brookes. The essence of SNPs. *Gene*, 234(2):177–186, July 1999.

M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM, 2002.

P. J. A. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Research*, 38(6):1767–1771, 2010.

P. Danecek, A. Auton, G. Abecasis, C. A. Albers, E. Banks *et al.* The variant call format and VCFtools. *Bioinformatics*, 27(15):2156–2158, 2011.

A. Doring, D. Weese, T. Rausch, and K. Reinert. SeqAn An efficient, generic C++ library for sequence analysis. *BMC Bioinformatics*, 9(1):11, 2008.

J. Eid, A. Fehr, J. Gray, K. Luong, J. Lyle *et al.* Real-Time DNA Sequencing from Single Polymerase Molecules. *Science*, 323(5910):133–138, 2009.

B. Ewing and P. Green. Base-calling of automated sequencer traces using phred. II. Error probabilities. *Genome research*, 8(3):186–194, 1998.

A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

J. C. Hansen. Human mitotic chromosome structure: what happened to the 30-nm fibre? *The EMBO Journal*, 31(7):1621–1623, 2012.

M. Holtgrewe, A.-K. Emde, D. Weese, and K. Reinert. A novel and well-defined benchmarking method for second generation read mapping. *BMC bioinformatics*, 12(1):210, 2011.

P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, Dallas, TX, USA, 1998. ACM.

P. Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et du Jura. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37(142), 1901.

M. Jansohn, S. Rothhämel, A. Aigner, and S. Rothhamel. *Gentechnische Methoden: Eine Sammlung von Arbeitsanleitungen für das molekularbiologische Labor*. Spektrum Akademischer Verlag, 2011.

J. Jiricny. Molekulare Zellbiologie. Vorlesungsskript, Zurich, Switzerland, März 2013.

P. Kaatsch, C. Spix, S. Hentschel, A. Kalalinic, S. Luttmann *et al. Krebs in Deutschland 2009/2010*. Robert Koch-Institut, Berlin, Germany, 2013.

A. Kassen and G. Hofmockel. Molekulargenetische und zellbiologische Grundlagen der Entstehung von malignen Tumoren. *Der Urologe*, 39(3):214–221, 2000.

R. Knippers. *Molekulare Genetik*. Thieme, 2006.

J. Köster and S. Rahmann. Massively parallel read mapping on GPUs with PEANUT. *CoRR*, abs/1403.1706, 2014.

B. Langmead and S. L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359, April 2012.

J. Leskovec. Finding Similar Items: Locality Sensitive Hashing. Vorlesung, Stanford, CA, USA, January 2014.

J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2014.

H. Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. unpublished, May 2013.

W.-H. Li, D. L. Ellsworth, J. Krushkal, B. H.-J. Chang, and D. Hewett-Emmett. Rates of Nucleotide Substitution in Primates and Rodents and the Generation–Time Effect Hypothesis. *Molecular Phylogenetics and Evolution*, 5(1):182 – 187, 1996.

X. Luo, W. Najjar, and V. Hristidis. Efficient near-duplicate document detection using FPGAs. In *Big Data, 2013 IEEE International Conference on*, pages 54–61. IEEE, 2013.

G. S. Manku, A. Jain, and A. Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the 16th international conference on World Wide Web*, pages 141–150. ACM, 2007.

M. Margulies, M. Egholm, W. E. Altman, S. Attiya, J. S. Bader *et al.* Genome sequencing in microfabricated high-density picolitre reactors. *Nature*, 437(7057): 376–380, September 2005.

M. L. Metzker. Sequencing technologies—the next generation. *Nature Reviews Genetics*, 11(1):31–46, 2010.

S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448, 1988.

S. Rahmann, D. Kopczynski, T. Marschall, and M. Martin. Algorithmen auf Sequenzen. Vorlesung, Dortmund, Germany, 2013.

D. E. Reich, S. F. Schaffner, G. McVean, J. C. Mullikin, J. M. Higgins *et al.* Human genome sequence variation and the influence of gene history, mutation and recombination. *Nature Methods*, 32:135–142, August 2002.

A. Rump. Medizinische Genetik für Biologen: Mutationen und deren Diagnostik. Vorlesung, 2009.

The Variant Call Format (VCF) Version 4.2 Specification. The SAM/BAM Format Specification Working Group, 2013.

Sequence Alignment/Map Format Specification. The SAM/BAM Format Specification Working Group, 2014.

F. Sanger, S. Nicklen, and A. R. Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74(12):5463–5467, 1977.

B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Analysis of recommendation algorithms for e-commerce. In *Proceedings of the 2nd ACM Conference on Electronic Commerce (EC'00)*, pages 158–167, Minneapolis, MN, USA, 2000. ISBN 1-58113-272-7.

J. Shendure and H. Ji. Next-generation DNA sequencing. *Nature biotechnology*, 26 (10):1135–1145, 2008.

J. Siqueira Jr, A. Fouad, and I. Rocas. Pyrosequencing as a tool for better understanding of human microbiomes. *Journal of oral microbiology*, 4, 2012.

E. Siragusa, D. Weese, and K. Reinert. Fast and accurate read mapping with approximate seeds and multiple backtracking. *Nucleic Acids Research*, 43(2), 2013.

M. Slaney and M. Casey. Locality-Sensitive Hashing for Finding Nearest Neighbors. *Signal Processing Magazine, IEEE*, 25(2):128–131, March 2008.

M. Stanke. Genomanalyse: Sequenzierung und Assemblierung. Vorlesung, Greifswald, Germany, 2013.

E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6(1): 132 – 137, 1985.