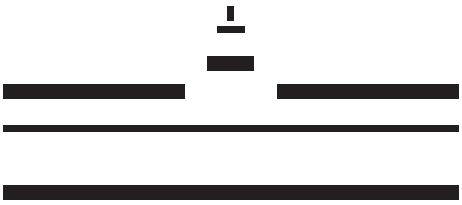


Westfälische
Wilhelms-Universität
Münster

An efficient error correction algorithm using FM-index

Seminararbeit

vorgelegt von:
Eike Gebauer

Matrikelnummer: 408126

Studiengang: MSc. Informatik

Inhaltsverzeichnis

1 Einleitung

DNA Sequenzierung ist aus der biologischen und medizinischen Forschung nicht mehr wegzudenken. Die Entwicklung immer günstigerer und leistungsstärkerer Sequenzierungsverfahren hat in den letzten Jahrzenten für einen rapiden Anstieg der zu verarbeitenden Datenmengen gesorgt. Hinzu kommt, dass die verwendeten Verfahren nicht fehlerfrei sind, was die Verarbeitung der Daten weiter erschwert.

In dieser Arbeit wird ein Paper [1] [2] behandelt, in dem der Fehlerkorrekturalgorithmus FMOE vorgestellt wird. Zu erst wird aber der Hintergrund in den Bereichen Biologie und Bioinformatik erläutert.

1.1 DNA-Sequenzierung

Das Ziel der de-novo DNA-Sequenzierung ist die Bestimmung der Nukleotidfolge in DNA ohne ein bereits bekanntes Referenzgenom. Wo bei alten Verfahren noch alle Basen nacheinander bestimmt wurden, werden seit dem sogenannten Shotgun Sequencing[3] [4] viele Teilstücke der DNA parallel analysiert. Das macht es möglich, lange DNA Stränge mit Verfahren zu sequenzieren, die nur kurze Fragmente verarbeiten können. Gleichzeitig ermöglicht es die parallele Sequenzierung von mehreren Fragmenten, was den Vorgang extrem beschleunigt.

Die Reihenfolge der DNA-Fragmente aufrecht zu erhalten, ist möglich, aber relativ langsam. Die sogenannten next-gen Sequenzierungsverfahren, welche den aktuell höchsten Durchsatz erzielen, verzichten daher darauf. Um trotzdem auf die ursprüngliche Reihenfolge und damit auf die gesamte DNA-Sequenz zurückschließen zu können, wird das Verfahren für mehrere Kopien desselben DNA-Strangs durchgeführt.

Wie in Abbildung 1.1 zu sehen, trennt man also mehrere Kopien des Strangs an zufälligen Positionen in ungefähr gleich lange Fragmente. Alle Fragmente können dann unabhängig von einander - und damit parallel - sequenziert werden. Das Ergebnis der Sequenzierung eines Fragments wird Read genannt. Aufgrund der zufälligen Trennungspositionen überlappen sich viele dieser Reads. Vorausgesetzt die Abdeckung[1] reicht an allen Stellen der Sequenz aus, kann man mit den Überlappungen die Reads in die ursprüngliche Reihenfolge brin-

[1] Abdeckung (Englisch Coverage) beschreibt die Anzahl einzigartiger Reads, die dieselbe Base enthalten.

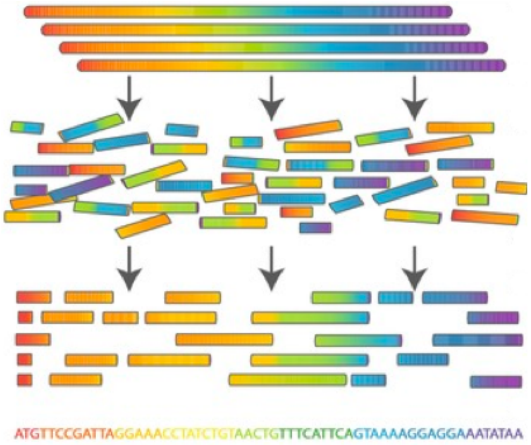


Abbildung 1.1: Shotgun Sequenzierung: Kopien des DNA-Strangs (oben) werden an zufälligen Positionen fragmentiert und sequenziert(darunter), in die korrekte Reihenfolge gebracht (3. von oben) und zur Gesamtsequenz zusammengesetzt (unten). Quelle [5]

gen und so die gesuchte DNA-Sequenz bestimmen. Dieser Prozess heißt Assembly.

Wie bereits erwähnt, sind die aktuellen Verfahren zur Bestimmung der Reads nicht fehlerfrei. Es kann zum Überspringen (Deletion) oder Vertauschen (Substitution) richtiger Basen oder zur Einfügung (Insertion) falscher Basen kommen. Kollektiv werden diese Fehler Indels genannt.

Die Assembly üblicherweise durch Überlappungsgraphen oder de Bruijn Graphen bewältigt. Überlappungsgraphen können Fehler bis zu einem gewissen Maß kompensieren, bringen allerdings hohe Speicher- und Rechenanforderungen mit sich. De Bruijn Graphen sind deutlich resourcensparender, tolerieren aber keine Fehler. Aus diesem Grund wird bei großen Datenmengen oft auf de Bruijn Graphen mit vorheriger Fehlerkorrektur zurückgegriffen.

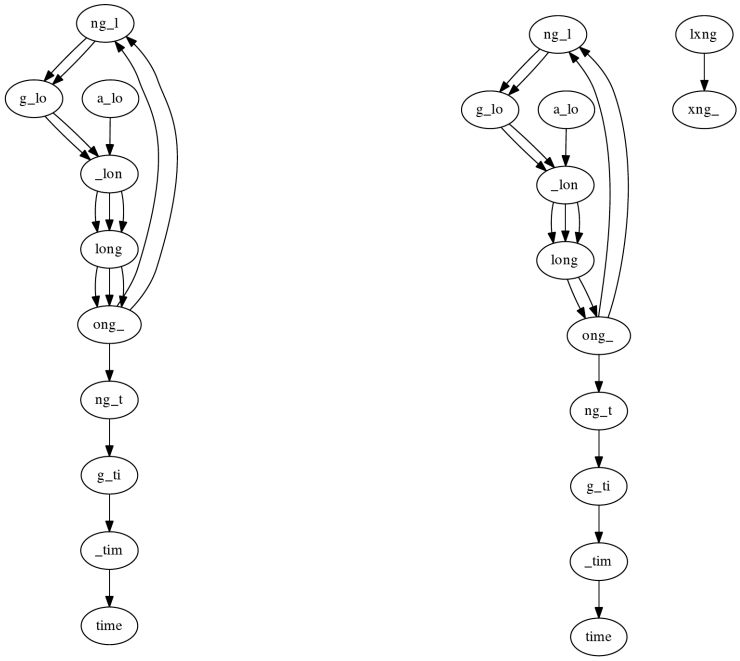
Fehler in den Reads sorgen für verschiedene Probleme bei der Assembly mithilfe eines de Bruijn Graphen:

Fragmentierte Assembly
: Fehlerhafte Reads verhindern das Erkennen von Überlappungen und sorgen für Graphen, die keine valide Assembly zulassen. Ein Beispiel für eine fragmentierte Assembly ist in Abbildung 1.2(b) zu sehen.

Misassembly
: Fehlerhafte Reads überlappen sich in manchen Fällen mit korrekten Reads und sorgen für eine valide Assembly, die aber eine falsche Sequenz ergibt.
Dieser Fall ist im Nachhinein schwer zu erkennen.

Zusätzlich erhöht sich die Komplexität des Graphen bei fehlerhaften Reads

unter Umständen enorm.



(a) Intakter de Bruijn Graph (b) Fragmentierter de Bruijn Graph

Abbildung 1.2: De Bruijn Graph des Texts „a long long time“. Oben der Graph aus fehlerfreien Fragmenten, unten der fragmentierte Graph, hervorgerufen durch Substitutionsfehler.

1.2 Fehlerkorrektur

Aufgabe der Fehlerkorrektur ist es, diese Komplikationen zu verhindern.
Da bei der de-novo Sequenzierung kein Referenzgenom zur Verfügung steht, kann die Fehlerkorrektur nur auf Grundlage der Reads geschehen. Die üblichen Verfahren stützen sich dabei auf statistische Modelle, um fehlerhafte Reads zu finden und zu verwerfen oder zu korrigieren.

Das hier behandelte Paper teilt Fehlerkorrekturalgorithmen in drei Kategorien:

k-mer Frequenzsprektrum Methoden erzeugen eine Erhebung aller k-mere[2] und ihrer Häufigkeiten aus den Reads und ersetzen seltene k-mere durch Häufige.
Diese Verfahren sind empfindlich gegenüber Wiederholungen in der DNA-Sequenz.

[2]k-mer wird eine k Zeichen lange Teilkette einer Zeichenkette genannt. Beispiel: Die 3-mere von „ABCDE“sind „ABC“„BCD“und „CDE“.

Suffix-Tree / -Array Methoden erzeugen einen Index der Suffixe der Reads in Form eines Arrays oder Baums und verhalten sich ansonsten ähnlich zu den k-mer-basierten Methoden.
Die variable Länge der Suffixe macht diese Verfahren etwas resistenter gegen Wiederholungen.

Überlappungsbasierte Methoden suchen für jeden zu korrigierenden Read alle überlappenden Reads und korrigieren potenzielle Fehler mit einem MSA[3] der gefundenen Reads.
Durch die Verwendung der ganzen Reads sind diese Verfahren nicht anfällig für Wiederholungen, allerdings ist die berechnung des MSA Berechnung sehr langsam.

Weitere Vor- und Nachteile der verschiedenen Vorgehensweisen werden später noch erläutert.

[3]MSA ist die Abkürzung für Multiple Sequence Alignment und beschreibt das alignen von mehr als drei DNA-Sequenzen. Mit alignment ist an dieser Stelle der methodische Vergleich von Sequenzen gemeint.

2 Methodik

FMOE fällt in die dritte Kategorie und unterscheidet sich von anderen Verfahren derselben Kategorie hauptsächlich durch die Verwendung einer FM-Index Implementierung [6] und damit verbundenen Optimierungen.

Im Folgenden werden daher zuerst die Datenstrukturen behandelt.

2.1 Datenstrukturen

Der FM-index [7] [8] ist eine Datenstruktur für Zeichenfolgen, die auf der Burrows Wheeler Transformation [9] (BWT) basiert.

Die BWT ist ein Algorithmus, der aus einer beliebigen Zeichenfolge eine Permutation dieser Folge und einen Index erzeugt, mit dem Effekt, dass die erhaltene Permutation in der Regel leichter zu komprimieren ist. Eine Rücktransformation ermöglicht dann die Wiederherstellung der ursprünglichen Zeichenfolge.

Der FM-Index wird ähnlich zur BWT konstruiert, speichert aber zusätzliche Daten, um schnelle Patternsuche bei immer noch sehr geringer Speicherkomplexität zu ermöglichen. Die Funktionsweise und die Gründe für die Resourcensparsamkeit lassen sich am besten im Vergleich zum Suffix-Array erläutern.

Wie in den Tabellen 2.1 und 2.2 zu sehen ist, beginnt man bei beiden Datenstrukturen damit, alle Suffixe der Zeichenfolge zu erzeugen. Im Fall des FM-Index wird jede Zeile mit dem Anfang der Zeichenfolge aufgefüllt, sodass sich die Rotationen der Zeichenfolge ergeben. Zusätzlich werden die Suffixpositionen in der Zeichenfolge in einer extra Spalte festgehalten.

Das Ergebnis dieser Operation wird dann lexikographisch sortiert. In Tabelle 2.4 ist der fertige Suffix Array zu sehen. Während bei dem Suffix-Array das gesamte Array gespeichert werden muss, kann beim FM-Index auf einen großen Teil der Daten verzichtet werden.

- Aus den Rotationen müssen nur die Spalten F und L gespeichert werden.
- Die Spalte F ist aufsteigend sortiert und kann daher bei einem kleinen Alphabet wie $\{A, C, G, T, \$\}$ sehr stark komprimiert werden.
- Für den Suchalgorithmus müssen die Ränge der Zeichen in der letzten Zeile (in Tabelle 2.5 als tiefgestellte Indizes sichtbar) bekannt sein. Diese kann man bei jeder Suche neu berechnen oder speichern. In den meisten Fällen wird ein hybrider Ansatz verwendet, bei dem ein Bruchteil

0	a	t	g	a	t	t	a	t	c	$
1	t	g	a	t	t	a	t	c	$	
2	g	a	t	t	a	t	c	$		
3	a	t	t	a	t	c	$			
4	t	t	a	t	c	$				
5	t	a	t	c	$					
6	a	t	c	$						
7	t	c	$							
8	c	$								
9	$									

Abbildung 2.1: Suffixe

0	a	t	g	a	t	t	a	t	c	$
1	t	g	a	t	t	a	t	c	$	a
2	g	a	t	t	a	t	c	$	a	t
3	a	t	t	a	t	c	$	a	t	g
4	t	t	a	t	c	$	a	t	g	a
5	t	a	t	c	$	a	t	g	a	t
6	a	t	c	$	a	t	g	a	t	t
7	t	c	$	a	t	g	a	t	t	a
8	c	$	a	t	g	a	t	t	a	t
9	$	a	t	g	a	t	t	a	t	c

Abbildung 2.2: Rotationen

Abbildung 2.3: Vorstufen von Suffix-Array und FM-Index für die Zeichenfolge „atgattatc$“

der Ränge vorberechnet und gespeichert wird, um die Berechnung der Übrigen zu beschleunigen.

- Die Positionen in der usprünglichen Zeichenfolge (in den Tabellen 2.6 als getrennte Spalte sichtbar) können genau wie die Ränge teilweise vorberechnet werden, um einen guten Kompromiss zwischen Speicherverbrauch und Rechenaufwand zu erhalten.

Diese Optimierungen sorgen dafür, dass ein Suffix-Array in der Regel ein vielfaches an Speicher benötigt als ein FM-Index.

2.1.1 Patternsuche

Die Patternsuche im Suffix-Array ist trivial, da es für jedes Vorkommen des Patterns einen Suffix geben muss, der mit dem Pattern beginnt. Weil alle

9	$									
6	a	t	c	$						
0	a	t	g	a	t	t	a	t	c	$
3	a	t	t	a	t	c	$			
8	c	$								
2	g	a	t	t	a	t	c	$		
5	t	a	t	c	$					
7	t	c	$							
1	t	g	a	t	t	a	t	c	$	
4	t	t	a	t	c	$				

Abbildung 2.4: Suffix-Array

	F								L
9	$	a	t	g	a	t	t	a	t	c_1
6	a_1	t	c	$	a	t	g	a	t	t_1
0	a_2	t	g	a	t	t	a	t	c	$
3	a_3	t	t	a	t	c	$	a	t	g_1
8	c_1	$	a	t	g	a	t	t	a	t_2
2	g_1	a	t	t	a	t	c	$	a	t_3
5	t_1	a	t	c	$	a	t	g	a	t_4
7	t_2	c	$	a	t	g	a	t	t	a_1
1	t_3	g	a	t	t	a	t	c	$	a_2
4	t_4	t	a	t	c	$	a	t	g	a_3

Abbildung 2.5: FM-Index

Abbildung 2.6: Resultierender Suffix-Array und FM-Index für die Zeichenfolge „atgattatc$“

Suffixe lexikographisch sortiert sind, liegen alle Suffixe, die mit dem Pattern beginnen hintereinander im Array. Es genügt also den Ersten und den Letzten der Suffixe durch wiederholte binäre Suche zu finden und mithilfe der ersten Spalte die Position aller dazwischen liegenden herauszufinden.
Beispiel: Suche nach „att“(siehe auch Tabelle 2.7)

1. Durch binäre Suche das Intervall der Suffixe finden, dass mit „a“beginnt (blau). Wenn die erste Spalte des Arrays durch RLE[1] komprimiert wurde, kann dieser Schritt ohne binäre Suche in konstanter Zeit stattfinden.

2. Mit binärer Suche das Teilintervall finden, dessen zweites Zeichen „t“ist

[1]RLE steht für Run Length Encoding (auf Deutsch Lauflängenkodierung) und ist eine Kompression, die bei Daten mit vielen Symbolwiederholungen eingesetzt wird. Beispiel: „aaaaabbb“wird komprimiert zu „a5b3“

(orange).

3. Schritt 2 für das nächste Zeichen wiederholen (rot).

4. Den gefundenen Suffix anhand der extra Spalte seiner Position in der Zeichenfolge zuordnen (Index 3)

9	$									
6	a	t	c	$						
0	a	t	g	a	t	t	a	t	c	$
3	a	t	t	a	t	c	$			
8	c	$								
2	g	a	t	t	a	t	c	$		
5	t	a	t	c	$					
7	t	c	$							
1	t	g	a	t	t	a	t	c	$	
4	t	t	a	t	c	$				

Abbildung 2.7: Patternsuche in Suffix-Array

Die Patternsuche in einem FM-Index ist etwas komplizierter. Man sucht das Pattern von hinten nach vorne durch ein sogenanntes LF-Mapping. Die Eigenschaft, die dieses Mapping ermöglicht, ist, dass der Zeichenrang in der letzten Zeile, dem der ersten Zeile entspricht. Das dritte „a“in der Spalte L ist in der ursprünglichen Folge also dasselbe Zeichen, wie das dritte „a“in der Spalte F.

Im FM-Index liegen alle Vorkommen des Patterns in dem gleichen zusammenhängenden Intervall, wie im Suffix Array. Aus diesem Grund ist auch bei Resultaten einer Patternsuche in einem FM-Index oft von Suffix-Array-Intervallen die Rede.
Das selbe Beispiel: Suche nach „att“(siehe auch Tabelle 2.8)

1. In konstanter Zeit das Intervall von F suchen, dass mit „t“beginnt (blau).

2. In dem selben Intervall in L linear nach dem nächsten Zeichen „t“suchen (orange).

3. Das gefundene „t“anhand des Rangs in konstanter Zeit in F wieder finden.

4. Schritt 2 und 3 mit dem Zeichen „a“wiederholen (rot).

5. Den gefundenen Eintrag anhand der extra Spalte seiner Position in der Zeichenfolge zuordnen (Index 3)

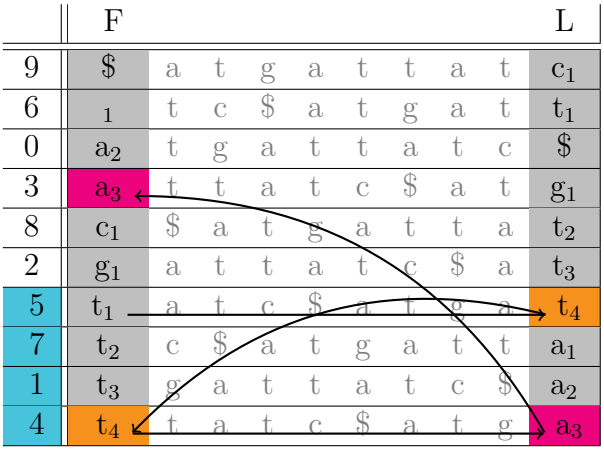


Abbildung 2.8: Patternsuche in FM-Index

Per LF-Mapping ist man mit den erwähnten Optimierungen also in der Lage mit geringer Komplexität nach Patterns zu suchen und die Speichereffizienz ermöglicht es im Vergleich zum Suffix-Array bei der selben Speichermenge deutlich größere Datenmengen zu verarbeiten.

Ein FM-Index des Menschlichen Genoms (ca. 3Mbp[2] mit 2 Bits pro Base) braucht ca. 1,5GB Speicher, während ein Suffix-Array ca. 12GB braucht [10].

2.2 Fehlerkorrektur

Im folgenden geht es um den Ablauf des Fehlerkorrekturalgorithmus. Dieser sieht es vor, dass jeder Read einzeln auf Fehler überprüft und bei Auftreten eines Fehlers dieser korrigiert wird.
Das Verfahren für den aktuellen - Query genannten - Read lässt sich in folgende grobe Schritte einteilen:

Potenzielle Fehler identifizieren
: Fehler werden anhand von Thresholding gesucht.

Reads finden, die sich mit der Query überlappen
: Aus allen Reads werden die gefunden, die sich in der Nähe des identifizierten Fehlers mit dem Query Read überlappen.

Gefundene Reads mit dem Query-Read alignen
: Ausgehend von der Überlappung werden alle gefundenen Reads schrittweise in beide Richtungen mit der Query aligned.

Korrektur des Fehlers
: Durch k-mer Voting aus den alignten Reads wird der Ersatz für die fehlerhafte Base gewählt.

[2]Mbp steht für Mega Basenpaare, also 1 Milliarde Basenpaare

Bevor dieser Ablauf beginnen kann, müssen aber ein paar Vorbereitungen getroffen werden.

2.2.1 Vorbereitung

Als erstes wird ein FM-Index aus allen Reads erzeugt und ein weiterer, der alle Reads rückwärts enthält. Die Verwendung des umgekehrten FM-Index wird in während des Alignment-Schritts erläutert.

Wie bei den meisten anderen Verfahren, wird auch ein Frequenzspektrum von k-meren erzeugt. In diesem Fall allerdings nicht von allen k-meren sondern aus 10.000 zufällig gezogenen k-meren aus allen Reads. Die Hoffnung ist, dass diese zufällige Stichprobe die gesamten Daten ausreichend repräsentiert, um Sonderfälle zu erkennen. k ist im Normalfall 31, kann aber angepasst werden. Die optimale Größe der k-mere hängt von der Länge der Reads, der erwarteten Größe der Überlappungen und weiteren Faktoren ab.

Aus diesem k-mer Frequenzspektrum werden weitere statistische Werte, wie der Mittelwert und Median der k-mer Häufigkeit, berechnet, die im Verlauf des Algorithmus Verwendung finden.

2.2.2 Fehleridentifikation

Für einen gegebenen Query Read werden nun alle k-mere erzeugt. Wenn die Häufigkeit von einem dieser k-mere im Frequenzspektrum unter einem bestimmten Schwellenwert (standardmäßig die Hälfte des Medians) fällt, enthält es wahrscheinlich Fehler.

2.2.3 Überlappungen finden

Der größte, an das fehlerhafte k-mer grenzende, Bereich, welcher entsprechend des Schwellenwerts ausreichende Qualität hat, wird identifiziert (in Abbildung 2.9 High Quality Region genannt).

Das k-mer innerhalb dieses Bereichs, das dem Fehler am nächsten ist, wird Seed genannt.

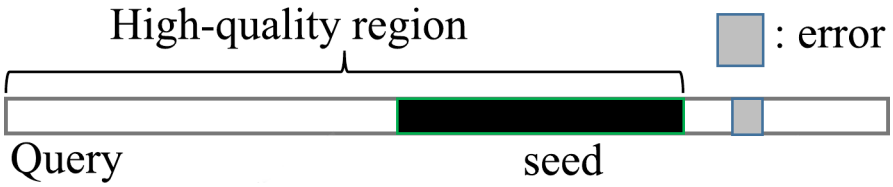


Abbildung 2.9: Visualisierung der Seed Position in einem fehlerhaften Query Read

Nun werden in dem normalen FM-Index alle Reads gesucht, die den Seed enthalten. Dies wird durch eine Patternsuche erreicht und resultiert in einem zusammenhängenden Suffix-Array-Intervall aller Vorkommen des Seeds.

2.2.4 Alignment der gefundenen Reads

Zusammen mit dem vorherigen Schritt wird diese Art von Alignment als Seed-and-Extend Strategie bezeichnet. Bereits vor dem erscheinen von FMOE wurde die FM-Index Variante [11] dieser Strategie in der Bioinformatik verwendet. Nachdem alle Reads mit dem Seed gefunden wurden, beginnt hier also die Extension Phase.

In dieser Phase werden die gefundenen Reads Schritt für Schritt mit der Query aligned. Man beginnt also mit dem Suffix-Array-Intervall aus dem vorherigen Schritt und führt in jedem Schritt das LF-Mapping für alle möglichen Basen (A, C, G und T) aus. Daraus ergeben sich vier neue Intervalle, die mit dem Seed beginnen und jeweils eine weitere Base enthalten.

Auf diese Art und Weise wird ein Baum aus allen auftretenden Basen erzeugt, welche mit ihrer Intervallgröße annotiert werden (siehe Abbildung 2.10). Das sorgt dafür, dass eine beliebige Menge von Reads in einen Pfad komprimiert werden, solange sie sich nicht voneinander unterscheiden. Während dieses Prozesses werden Pfade, die zu sehr kleinen Intervallen gehören, entfernt, um das Verfahren zu beschleunigen. Die Extension wird abgebrochen, wenn die gesamte Query enthalten ist oder die Anzahl der Pfade einen festgelegten Schwellenwert überschreitet.

Ein Beispiel für die FM-Index Extension ist in Abbildung 2.11 zu sehen.

An dieser Stelle wird auch deutlich, weshalb der zweite, umgekehrte, FM-Index notwendig ist. Die FM-Index Extension funktioniert aufgrund des LF-Mappings nur in eine Richtung. Um die Überlappung in beide Richtungen zu erweitern, ist also auch ein FM-Index der alle Reads rückwärts enthält nötig.

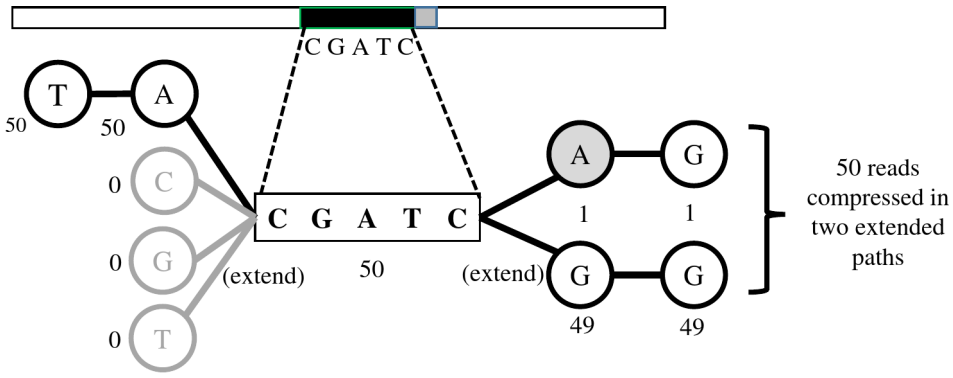


Abbildung 2.10: Extension Baum von 50 gefundenen Reads.

Anstatt das Alignment aus dem erhaltenen Baum mit herkömmlichen Verfahren aus dem Feld der dynamischen Programmierung zu berechnen, wurde eine heuristische Lösung entworfen. Noch vor der Extension werden die Suffix-Array-Intervalle aller k-mere des Query Reads berechnet. Während der Extension wird dann jedem neuen Pfad sein Intervall zugeordnet.

i	L													F
0	$	a	a	t	g	$	a	a	t	g	$	c	a	t	g
1	$	a	a	t	g	$	c	a	t	g	$	a	a	t	g
2	$	c	a	t	g	$	a	a	t	g	$	a	a	t	g
3	a	a	t	g	$	a	a	t	g	$	c	a	t	g	$
4	a	a	t	g	$	c	a	t	g	$	a	a	t	g	$
5	a	t	g	$	a	a	t	g	$	a	a	t	g	$	c
6	a	t	g	$	a	a	t	g	$	c	a	t	g	$	a
7	a	t	g	$	c	a	t	g	$	a	a	t	g	$	a
8	c	a	t	g	$	a	a	t	g	$	a	a	t	g	$
9	g	$	a	a	t	g	$	a	a	t	g	$	c	a	t
10	g	$	a	a	t	g	$	c	a	t	g	$	a	a	t
11	g	$	c	a	t	g	$	a	a	t	g	$	a	a	t
12	t	g	$	a	a	t	g	$	a	a	t	g	$	c	a
13	t	g	$	a	a	t	g	$	c	a	t	g	$	a	a
14	t	g	$	c	a	t	g	$	a	a	t	g	$	a	a

Abbildung 2.11: FM-Index der Reads „aatgc“, „aatgc“und „catgc“mit Trennzeichen „$“. Von dem Intervall des Seeds „atgc“, $[5, 7]$ ausgehend, entstehen durch das LF-Mapping mit allen vorhandenen Zeichen zwei Pfade: „a“, $[3, 4]$ und „c“, $[8, 8]$

Ist das Intervall des Pfads äquivalent zu dem Intervall eines k-meres aus der Query mit gleichem Abstand vom Seed, so überlappt sich die Query mit den Reads in diesem Pfad.

An dieser Stelle müssen allerdings mögliche Indels beachtet werden, welche den effektiven Abstand des gemeinsamen k-mers vom Seed beeinflussen können. Um eine gewisse Anzahl Indels zu tollerieren, müssen also auch Suffix-Array-Intervalle der benachbarten k-mere mit denen der Pfade verglichen werden.

2.2.5 Fehlerkorrektur durch k-mer Voting

Üblicherweise wird nach der Berechnung eines MSA, wie im vorherigen Schritt, eine Matrix aller alignten Reads erstellt (siehe Abbildung 2.12(a)). Aus dieser Matrix lässt sich die Häufigkeit jeder Base für jede alignte Position der Query berechnen.

Eine einfache Lösung, um Fehler zu beheben, wäre also, die seltenen Basen in der Query durch die Häufigsten der jeweiligen Position zu ersetzen. Diese Vorgehensweise ist zwar weit verbreitet, sorgt unter Umständen aber für Probleme.

Die chemischen Verfahren, die aktuellen Sequenzierungsmethoden zugrunde liegen, identifizieren nicht alle Basen gleich gut. Regionen mit hohen Guanin

und Cytosin Konzentrationen lassen sich nicht so gut verstärken, wie Adenin und Thymin reiche Regionen. Aus diesem Grund enthalten solche Regionen verstärkt Fehler.

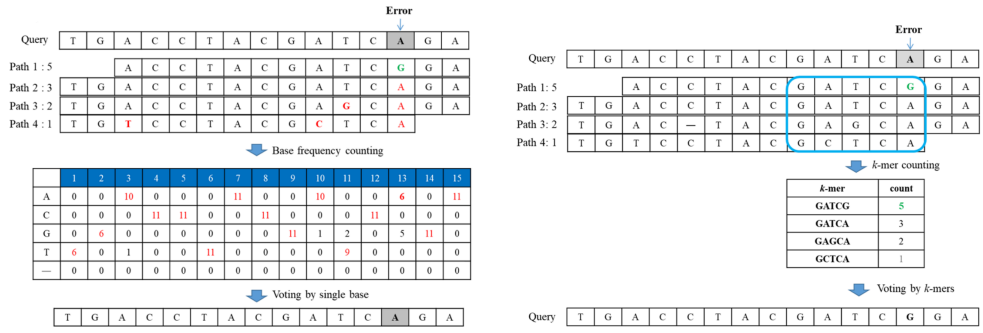
Diese und ähnliche systematische Fehler können bei herkömmlichen MSA Matrix Verfahren dafür sorgen, dass die Häufigkeit falscher Basen, die der korrekten übersteigt.

Um dem entgegen zu wirken, verwendet FMOE an dieser Stelle sogenanntes k-mer Voting (siehe Abbildung 2.12(b)). Beim k-mer Voting werden die k-mere aller Reads erzeugt, die die Fehlerposition enthalten. Die Häufigkeit aller unterschiedlichen k-mere wird berechnet und das Häufigste ersetzt die Fehlerposition.

Der Vorteil von k-mer Voting ist, dass selbst Reads mit hohen Fehleranteilen stückweise korrekte Sequenzen enthalten. Diese sind statistisch häufiger als identische Fehler in mehreren Reads und sorgen so für Wahl der korrekten Base.

k ist im Fall von FMOE standardmäßig 5, abhängig von der Länge des Alignments können aber andere Werte optimaler sein. Hier müssen neben dem Rechenaufwand wieder mehrere Faktoren abgewogen werden. Es wäre es auch möglich verschiedene Werte aus einem fetgelegten Intervall zu testen und die eindeutigste Entscheidung zu übernehmen.

Ein Algorithmisch bedingter Vorteil dieser Methode ist, dass die Häufigkeit der k-mere bereits durch die Intervallgrößen aus dem Alignmentschritt gegeben sind. Das offensichtlich komplexere Verfahren benötigt an dieser Stelle also nicht signifikant mehr Rechenleistung.



(a) MSA Matrix (b) k-mer Voting

Abbildung 2.12: Single-Base-Voting per MSA vs. k-mer Voting

3 Ergebnisse

FMOE wurde in dem behandelten Paper mit sechs anderen Korrekturalgorithmen vergleichen (siehe Tabelle 3.1). Alle zählten entweder zu den überlappungsbasierten oder k-mer Frequenzspektrum Verfahren.

Algorithmus	Erscheinungsjahr	Überlappungsbasiert	k-mer Spektrum	Read Qualität
FMOE	2017	x		Nein
Coral	2011	x		Ja
Karect	2015	x		Ja
QuorUm	2015		x	Ja
RACER	2013		x	unbkannt
BLESS2	2016		x	Ja
SGA	unbekannt	x	x	unbekannt

Abbildung 3.1: Übersicht der verglichenen Fehlerkorrekturalgorithmen. Quellen [12] [13] [14] [15]

Als Testdaten wurden vier Datensätze aus der GAGE-b Datenbank verwendet. Die GAGE-b Datenbank enthält Daten aus echten Sequenzierungen, durchgeführt mit den verbreitetsten Sequenzierverfahren. Das Ziel der Datenbank ist es, Sequenzierungs-, Assembly- und Korrekturverfahren mit realistischen Anwendungsfällen vergleichbar zu machen.

Die Größe der Datensätze variert zwischen 4,6Mb und 100,2Mb mit Readlängen von 110bp bis 251bp.

Die Güte eines Verfahrens wird im Allgemeinen durch die Korrekturleistung (Menge korrigierter Basen) und die Genauigkeit (Anteil richtiger Korrekturen) bestimmt.

Zu erwarten war, dass die überlappungsbasierten Verfahren - und damit auch FMOE - bei den größeren Read Längen eine höhere Korrekturleistung aufweisen und grundsätzlich, aufgrund des Alignments, deutlich langsamer sind als k-mer Frequenzspektrum basierte Verfahren.

Die Ergebnisse scheinen die Annahme mindestens für FMOE zu bestätigen, denn oft haben Coral oder Karect und besonders häufig hat FMOE relativ hohe Korrekturleistungen bei den Datensätzen mit 251bp Readlänge. Die Korrekturleistungen bei Datensätzen mit kurzer Readlänge sind kaum voneinander zu unterscheiden.

Bis auf einen Aussetzer von Coral bewegt sich die Identität der korrigierten Datensätze mit dem korrekten Genom in allen Fällen deutlich über den

Rohdaten.

Die Laufzeit der k-mer basierten Methoden beträgt, wie erwartet nur einen Bruchteil der Laufzeit der überlappungsbasierten Verfahren. Der Unterschied zwischen Coral auf der einen und Karect und FMOE auf der anderen Seite ist allerdings enorm. Die längste Laufzeit von Coral beträgt 43 Stunden, während Karect und FMOE für denselben Datensatz ca. 2 Stunden benötigen.

4 Diskussion und Fazit

Das behandelte Paper stellt FMOE im Detail vor und erläutert die Funktionsweise an vielen Stellen sehr anschaulich.

An einigen Stellen fehlen allerdings Details:

Qualität der Reads

Aktuelle Sequenzierungsverfahren stellen nicht nur die rohen Reads zur Verfügung, sondern auch weitere relevante Daten. Unter anderem ist im Normalfall ebenso eine Qualitätsmetrik für jeden einzelnen Read oder sogar jede Base enthalten. Diese ermöglicht es, die Fehlerwahrscheinlichkeit der Basen des Reads einzuschätzen und so Basen mit hohen Fehlerquoten bei der Korrektur oder Statistik schwächer zu gewichten.

Das Paper gibt keinen Aufschluss darüber, ob FMOE diese Daten ignoriert, andere Verfahren nutzen Sie allerdings erfolgreich [16] [17] (siehe 3.1).

Stichprobemgröße

Die Größe der k-mer Stichprobe ist laut dem Paper konstant vorgegeben. Das kann bei großen Datensätzen für Probleme sorgen, da eine relativ kleine Stichprobe die Daten nicht mehr ausreichend repräsentieren kann.

Ob FMOE eine Möglichkeit bietet, diese Größe anzupassen oder sie dynamisch angepasst wird, ist nicht durch das Paper ersichtlich.

Einstellungen der konkurrierenden Algorithmen

Das Paper enthält keine Informationen über die verwendeten Einstellungen bei den Vergleichsverfahren.

Das SGA Paket bietet mehrere Fehlerkorrekturalgrithmen an. Hier ist nicht klar, welches dieser Verfahren angewendet wurde.

Darstellung der Ergebnisse

Die Ergebnisse des Vergleichs befinden sich in vielen Fällen in den letzten zwei Nachkommastellen und sind daher selbst bei so geringen Mengen schwer zu überblicken.

An dieser Stelle währe eine andere Skalierung oder eine Visualisierung durch Graphen übersichtlicher gewesen.

Abgesehen davon macht das Paper aber einen guten Eindruck.

Besonders die Verwendung von GAGE-b Datensätzen für den Vergleich der Verfahren ist lobenswert, da hier auf synthetische Datensätze verzichtet wurde.

Literaturverzeichnis

[1] Huang, Yao-Ting ; Huang, Yu-Wen: An efficient error correction algorithm using FM-index. In: *BMC Bioinformatics* 18 (2017), Nov, Nr. 1, 524. http://dx.doi.org/10.1186/s12859-017-1940-1. – DOI 10.1186/s12859–017–1940–1. – ISSN 1471–2105

[2] Huang, Yao-Ting ; Huang, Yu-Wen: *An Efficient Error Correction Algorithm Using FM-index (Supplementary Material).* https://static-content.springer.com/esm/art%3A10.1186%2Fs12859-017-1940-1/MediaObjects/12859_2017_1940_MOESM1_ESM.pdf, Abruf: 14.01.2019

[3] R., Staden: A strategy of DNA sequencing employing computer programs. In: *Nucleic Acids Res.* (1979)

[4] S., Anderson: Shotgun DNA sequencing using cloned DNase I-generated fragments. In: *Nucleic Acids Res.* (1981)

[5] Commins, Toft C. Fares M. A. J.: *Whole genome shotgun sequencing versus Hierarchical shotgun sequencing.* https://en.wikipedia.org/wiki/Shotgun_sequencing#/media/File:Whole_genome_shotgun_sequencing_versus_Hierarchical_shotgun_sequencing.png, Abruf: 13.01.2019

[6] Li, Heng: Fast construction of FM-index for long sequence reads. In: *Bioinformatics* 30 (2014), Nr. 22, 3274-3275. http://dx.doi.org/10.1093/bioinformatics/btu541. – DOI 10.1093/bioinformatics/btu541

[7] Ferragina, P. ; Manzini, G.: Opportunistic data structures with applications. In: *Proceedings 41st Annual Symposium on Foundations of Computer Science*, 2000. – ISSN 0272–5428, S. 390–398

[8] T Simpson, Jared ; Durbin, Richard: Simpson JT, Durbin R.. Efficient construction of an assembly string graph using the FM-index. Bioinformatics 26: i367-i373. In: *Bioinformatics (Oxford, England)* 26 (2010), 06, S. i367–73. http://dx.doi.org/10.1093/bioinformatics/btq217. – DOI 10.1093/bioinformatics/btq217

[9] Burrows, David J. M.; Wheeler W. M.; Wheeler: *A block-sorting lossless data compression algorithm.* http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.html

[10] Langmead, Ben: *Burrows-Wheeler Transform and FM Index*. https://www.cs.jhu.edu/~langmea/resources/lecture_notes/bwt_and_fm_index.pdf, Abruf: 13.01.2019

[11] Huang, Yao-Ting ; Liao, Chen-Fu: Integration of string and de Bruijn graphs for genome assembly. In: *Bioinformatics* 32 (2016), Nr. 9, 1301-1307. http://dx.doi.org/10.1093/bioinformatics/btw011. – DOI 10.1093/bioinformatics/btw011

[12] Marinier, Eric: *Error Correction of Second-Generation Sequencing Reads*. https://uwspace.uwaterloo.ca/bitstream/handle/10012/8996/Marinier_Eric.pdf?sequence=1, Abruf: 14.01.2019

[13] Salmela, Leena ; Schröder, Jan: Correcting errors in short reads by multiple alignments. In: *Bioinformatics* 27 (2011), Nr. 11, 1455-1461. http://dx.doi.org/10.1093/bioinformatics/btr170. – DOI 10.1093/bioinformatics/btr170

[14] Allam, Amin ; Kalnis, Panos ; Solovyev, Victor: Karect: Accurate Correction of Substitution, Insertion and Deletion Errors for Next-generation Sequencing Data. In: *Bioinformatics (Oxford, England)* 31 (2015), 07. http://dx.doi.org/10.1093/bioinformatics/btv415. – DOI 10.1093/bioinformatics/btv415

[15] Marçais G, Zimin A. Yorke JA J. Yorke JA: QuorUM: An Error Corrector for Illumina Reads.

[16] Li, Heng: *Correcting Illumina sequencing errors: extended background*. https://lh3.github.io/2015/02/13/comments-on-illumina-error-correction. Version: 2015, Abruf: 13.01.2019

[17] Li, Heng: BFC: correcting Illumina sequencing errors. In: *Bioinformatics* 31 (2015), Nr. 17, 2885-2887. http://dx.doi.org/10.1093/bioinformatics/btv290. – DOI 10.1093/bioinformatics/btv290