Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: 12th of Jabuary 2018
Manuscript Category

Algorithmische Bioinformatik

Projekt 2

Paul Vogler, Tobias Mechura & Franziska Rau

Abstract

Projekt 2 des Moduls Algorithmische Bioinformatik.
Ziel ist das Erstellen eines Programms, basierend auf dem De Bruijn Graph Ansatz, das alle Contigs des erstellten Assemblies ausgibt.

1 Einführung

Genom Assemblierung dient dazu, Fragmente aus einer längeren DNA Sequenz, sogenannte Reads, wieder zusammenzuführen, um so die ursprüngliche Sequenz zu rekonstruieren.
Dies ist notwendig, da das Einlesen von ganzen Sequenzen durch morderne Sequenzierungsmaschinen noch nicht möglich ist. Diese können das Genom nur zerteilen und so kuŕzere Abschnitte generieren.
Um eine Sequenz anschließend zu rekonstruieren, werden die Reads überlappt und Contigs erzeugt.
Nachfolgend erhält man aus den Contigs bestimmte Regionen aus der DNA, wie zum Beispiel Consensussequenzen.

2 Ansatz

Gegeben ist ein Datensatz von Nasuia deltocephalinicola, der insgesamt 34.000 fehlerfreie Reads der Länge 100 enthält.
Ziel des Projektes ist es, einen Genome Assembler zu implementieren, der mit diesem Single-Read Datensatz funktioniert.
Die Reihenfolge der Reads ist unbekannt, deshalb sind diese alphabetisch geordnet.
Der aus der Vorlesung bekannte De Bruijn Graph Ansatz wurde in der Programmiersprache Java umgesetzt.
Dazu wurde zunächst ein Graph und dann der MaximalNonBranchingPaths Algorithmus implementiert.

3 Methoden

Um den Pseudocode MaximalNonBranchingPaths aus der Vorlesung zu implementieren, musste zunächst ein Graph erzeugt werden.
Hierzu wurde die Java Bibliothek "JGraphT" verwendet, die mathematische Graphen-Theorie Objekte bereitstellt und verschiedene Arten von Graphen unterstützt.
Für die Umsetzung des De Bruijn Graph Ansatzes, wurde der gerichtete Pseudograph ausgewählt.
Ein gerichteter Pseudograph ist ein nicht einfacher gerichteter Graph, in dem sowohl Graphenschleifen als auch mehrere Kanten erlaubt sind.
Die Graphenerstellung erforderte die Unterteilung der Reads in Präfixe und Suffixe, um so Eulersche Wege zu konstruieren, also Kantenfolgen eines Graphen, bei denen jede Kante genau einmal durchlaufen wird und Start- und Endknoten nicht identisch sind.
Der erstellte Graph ist anschließend Eingabe für den Algorithmus MaximalNonBranchingPaths, um die Contigs zu erstellen.
Jeder Knoten im Graphen wird besucht und überprüft, ob es sich um einen 1-in-1-out Knoten handelt, also ob es nur eine eingehende und ausgehende Kante gibt, dieser Weg wird dann in eine Liste geschrieben.
Handelt es sich nicht um einen 1-in-1-out Knoten, so wird die Liste um die ausgehenden Kanten erweitert. Dieser Vorgang wird wiederholt, bis alle Wege besucht wurden.
Im Anschluss wurde noch überprüft, ob isolierte Kreise existieren, dazu wurde der Tarjan Algorithmus zur Bestimmung starker Zusammenhangskomponenten verwendet.
Tarjans Algorithmus basiert auf der Tiefensuche, die einzelnen Knoten werden beim DFS Durchlauf indiziert. Bei der Rückkehr der Rekursion von der DFS wird jedem Knoten v ein Knoten w als Repräsentant zugewiesen.
Knoten mit dem selben zugewiesenen Repräsentanten befinden sich in der selben stark zusammenhängenden Komponente. Die gefundenen isolierten Kreise werden dann auch in die Liste der Wege geschrieben.
Das Ergebnis des MaximalNonBranchingPaths Algorithmus sind Contigs in Form einer Liste aus Readlisten (eine Readliste entspricht einem Contig). Anschließend wurden diese Readlisten zu einzelnen Contigsequenzen zusammengefügt.
In der Mainmethode wird der Datensatz zunächst eingelesen, die Algorithmen darauf angewendet und das Ergebnis im FASTA Format gespeichert.

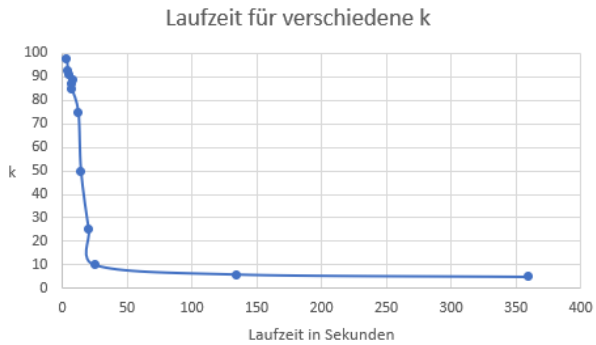
4 Diskussion

Um die Aufgabenstellung zu lösen, mussten die Reads zerbrochen werden, sogenanntes Read-Breaking.
k sollte dabei so gewählt werden, dass der De Bruijn Graph nicht zu komplex wird. Das Finden eines geeigneten k's umfasste mehrere Durchläufe des Programms, indem unterschiedlich große k's getestet wurden.
Außerdem wurden jeweils die durchschnittlichen Contiglängen berechnet, um einen passenden Zahlenbereich für k zu finden.
Es wurde darauf geachtet, dass die Gesamtcontiglänge möglichst groß war, und dass vorzugsweise viele Contigs länger als die ursprüngliche Readlänge 100 waren.
Der optimale Bereich umfasste k's der Größen 84 bis 93, k wurde dann auf 89 festgelegt.
Die ausgegebenen Contigs waren durchschnittlich 4 Basen länger als die Readlänge. Beim genauen Betrachten der Contigs konnte festgestellt werden, dass die Contiglänge unterschiedlich war und auch teilweise wesentlich längere Contigs vorhanden waren.
Lange Contigs zu erhalten ist von großer Bedeutung, denn dadurch kann im späteren Verlauf das ganze Genom rekonstruiert werden, indem man nach einem Eulerkreis sucht. Wichtig ist dafür auch, dass die Fragmente des entsprechenden k's klein genug sind. Das finden eines Eulerkreises wurde allerdings nicht implementiert und spielt somit vordergründig keine Rolle.
Um den Gesamtalgorithmus zu testen wurde ein Beispieldatensatz der Plattform ROSALIND benutzt, dieser umfasste Reads mit einer k-mer Länge von 68.
Der gegebene Output wurde mit dem Output des erstellten Programms verglichen und die Korrektheit konnte bestätigt werden.
Die erstellten Contigs des Genome Assemblers wurden als FASTA Ausgabedatei mit entsprechender Beschriftung der Contiganzahl gespeichert.

4.1 Laufzeitanalyse

Table 1. Laufzeiten für verschiedene k

k	Laufzeit in Sekunden
5	359
6	134
10	25
25	20
50	14
75	12
85	7
87	7
89	8
91	5
93	4
98	3



Für k ab 10 gibt es keine großen Änderungen in der Laufzeit, aber wenn k kleiner als 10 gewählt wird, steigt die Laufzeit exponentiell an. Unsere längste gemessene Laufzeit bei einem k von 5 betrug 359 Sekunden. Im optimalen Bereich für k zwischen 84 und 93 benötigt der Algorithmus durchschnittlich 6 Sekunden und ist damit sehr schnell.

4.2 Probleme

Probleme traten bei der Graphenerstellung auf, diesen Algorithmus selbst zu implementieren stellte sich als sehr aufwendig heraus, deshalb wurde auf die Java Bibliothek JGraphT zurückgegriffen.
Die Bibliothek umfasst sehr viele verschiedene Möglichkeiten Graphen zu erstellen und erforderte Einarbeitungszeit, um die passenden Eigenschaften eines De Bruijn Graphen auf einen der gegebenen Graphen in der Bibliothek zu übertragen.

4.3 Arbeitsaufteilung

Größtenteils wurde für das Projekt gemeinsam an einem PC gearbeitet. Lediglich kleinere Aufgaben wurden vereinzelt alleine bearbeitet.

5 Fazit

Das Projektziel wurde erfüllt und die benötigten Algorithmen, um ein Programm zu erstellen, das auf dem De Bruijn Graph Ansatz basiert, implementiert.
Die Länge der k-mere hängt von der Länge der Reads ab und sollte nicht zu gering gewählt werden, da sonst zu viele und zu kurze Contigs entstehen und sich somit das Genom nicht mehr rekonstruieren lässt.
Für die Readlänge empfiehlt sich daher ein k zwischen 84 und 93 festzulegen, um gute Ergebnisse zu erhalten.
Die Gesamtlaufzeit liegt weit unter dem vorgegebenem Zeitlimit für die Programmiersprache Java und zeugt von der Benutzerfreundlichkeit des Programms, denn schnelle Ergebnisse können auch schneller evaluiert werden.
Insgesamt ist der erstellte Genome Assembler praxisfähig und kann auch erweitert werden, um das Gesamtgenom zu rekonsturieren.