Санкт-Петербургский государственный университет

**Кафедра математической теории игр и статистических решений**

**Казачкова Екатерина Сергеевна**

**Магистерская диссертация**

**Программная реализация сравнения последовательности ДНК с эталонным геномом**

Направление ВМ.5504.2015

«Прикладная математика и информатика»

Магистерская программа «Исследование операций и системный анализ»

Научный руководитель,  
кандидат физ.-мат. наук,  
доцент  
Громова Е. В.

Санкт-Петербург

2017

Оглавление

[Введение 3](#_Toc481618519)

[*1.* *Введение в предметную область* 3](#_Toc481618520)

[*2.* *Процесс секвенирования* 4](#_Toc481618521)

[*3.* *Форматы fasta и fastq* 7](#_Toc481618522)

[*4.* *Выравнивание* 10](#_Toc481618523)

[*5.* *Обзор биоинформатических инструментов* 11](#_Toc481618524)

[Глава 1. Сравнение последовательности ДНК с эталонным геномом. 12](#_Toc481618525)

[*1.* *Основные принципы работы программы FastqScreen* 12](#_Toc481618526)

[*2.* *Описание работы алайнеров bowtie и bowtie2 и их роль в программе FastqScreen* 15](#_Toc481618527)

[*3.* *Критерий принятия решения* 17](#_Toc481618528)

[*3.1.* *Введение* 17](#_Toc481618529)

[*3.2.* *Описание критерия* 19](#_Toc481618530)

[*3.3.* *Реализация* 22](#_Toc481618531)

[*4.* *Программная реализация FastqScreen* 23](#_Toc481618532)

[*4.1.* *Исходная реализация на языке Perl* 23](#_Toc481618533)

[*4.2.* *Реализация по средствам языка Java* 24](#_Toc481618534)

[*4.3.* *Результаты замеров производительности* 27](#_Toc481618535)

[Глава 2. Применение метода нечеткого поиска к задаче поиска повторяющихся нуклеотидов последовательности ДНК 29](#_Toc481618536)

[*1.* *Описание форматов SAM/BAM/CRAM* 29](#_Toc481618537)

[*2.* *Описание набора инструментов Picard Tools* 32](#_Toc481618538)

[*3.* *Базовый алгоритм* 34](#_Toc481618539)

[*4.* *Метод нечеткого поиска* 36](#_Toc481618540)

[*5.* *Реализация метода нечеткого поиска* 40](#_Toc481618541)

[Заключение 43](#_Toc481618542)

[Список литературы 44](#_Toc481618543)

Введение

1. *Введение в предметную область*

За последние несколько лет все большую популярность набирает такое направление в науке как биоинформатика. Это обусловлено тем, что в данной области огромную роль играет использование современных компьютерных технологий для анализа и манипуляции большими биологическими данными. В широком смысле биоинформатика — это использование программного обеспечения для решения различных биологических задач, таких как­­ анализ экспериментальных данных, визуализация биологических визуализация, поиск вариаций в геноме. Целью этой науки является не только развитие алгоритмов использования биологических данных, но и подготовка данных для получения наиболее достоверных результатов. Таким образом, биоинформатика включает в себя как естественно-научные дисциплины, такие как генетика, молекулярная биология, так и точные науки — математика, статистика, информатика.

Биоинформатика берет свое начало из биологии, а точнее, молекулярной биологии, одной из задач которой является изучение генетической информации. Генетическая (наследственная) информация хранится в клетках живых существ. Единицами хранения такой информации являются молекулы ДНК, РНК и белки.

Молекула ДНК (дезоксирибонуклеиновая кислота) представляет собой двухцепочечную спиралевидную молекулу, состоящую из нуклеотидных звеньев, которые удерживаются водородными связями. Каждый нуклеотид последовательности ДНК содержит в себе азотиcтое основание одного из четырех видов: аденин (A), гуанин (G), тимин (T) и цитозин (C). В природе пары оснований соединены между собой согласно принципу комплементарности: аденин соединяется только с тимином, гуанин — только с цитозином [1].

Приведем некоторые термины из области биологии:

*Ген* — это некоторый участок ДНК, представляющий собой последовательность нуклеотидов. Количество генов у организмов различно. *Геном* — это общая генетическая информация, содержащаяся в клетках организма. Например, человеческий геном состоит из 23 пар хромосом и митохондриальной ДНК. Геномы большинства организмов состоят из ДНК, однако геном некоторых вирусов построен из РНК. Подробную информацию по этому вопросу можно почерпнуть, например, в [1].

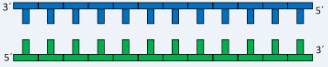
1. *Процесс секвенирования*

Одной из первых задач, которая встает перед исследователями, является получение генетической информации некоторого организма, пригодное для компьютерного представления. Процесс формального описания первичной структуры линейной макромолекулы в виде последовательности мономеров в текстовом виде называется секвенированием. Иными словами, это процесс определения последовательности нуклеотидов, из которых состоят изучаемые данные, и приведение их к цифровому виду. Прибор, с помощью которого экспериментатор из некоторого организма получает текстовый файл для последующей компьютерной обработки, называется секвенатором. Результат работы секвенатора — это текстовый файл, содержащий огромное количество ридов. Рид — это последовательность нуклеотидов, представляющая собой небольшой фрагмент ДНК. Размер этого фрагмента варьируется настройками секвенатора.

В основе процесса секвенирования лежит метод полимеразной цепной реакции, или, сокращенно, ПЦР. Этот метод является мощным инструментом для подготовки данных к формату, в цифровом виде. ПЦР — это метод, при котором происходит амплификация ДНК, то есть увеличение числа копий фрагментов ДНК, с помощью термических и некоторых химических воздействий. Этот метод цикличен и каждая итерация удваивает концентрацию фрагментов, поэтому число копий растет со скоростью , где n — количество циклов. Это происходит для того, чтобы секвенатор смог определить позиции нуклеотидов в фрагменте ДНК.

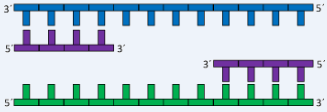
Перечислим стадии ПЦР:

* Выделение ДНК из биологического материала (пробы).
* Фрагментация, то есть деление ДНК на короткие фрагменты, с помощью ультразвука или ферментов.
* После чего, над полученными фрагментами, производится термическое воздействие, с целью разрыва водородных связей и, как следствие, получения одноцепочечных фрагментов.



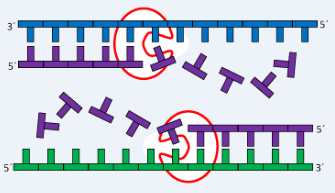
*Рис. 0.2.1. Фрагмент ДНК с разорванными водородными связями.*

* Далее, происходит подбор праймеров. Праймер — это очень короткая последовательность нуклеотидов, которая получается путем добавления комплементарных нуклеотидов на концы фрагмента.



*Рис. 0.2.2. Добавление праймеров на концы фрагментов.*

* Затем, к фрагменту ДНК добавляется ДНК-полимераза, которая доводит синтез до конца фрагмента.



*Рис. 0.2.3. Добавление ДНК-полимеразы.*

* Таким образом, мы получили два одинаковых фрагмента и можем повторить процесс.

Существует множество подходов к секвенированию биологических данных: полупроводниковое секвенирование, секвенирование синтезом, секвенирование при помощи нанопор, одномолекульное секвенирование в реальном времени и другие. Самым популярным на данный момент является метод секвенирования по Сэнгеру. Этот метод был изобретен Фредериком Сэнгером в 1977 и пользуется большой популярностью по сей день, поскольку он обладает достаточно высокой точностью. Более детально познакомиться с данным механизмом можно в [2].

Развитие современных технологий положило начало методам секвенирования нового поколения (next-generation sequencing, или сокращенно NGS). К наиболее популярным NGS секвенаторам относятся:

* Illumina. Основан на методе секвенирования синтезом, обрабатывающий короткие риды длиной 100bp[[1]](#footnote-1)-300bp. Отличительной особенностью Illumina является мостиковая амплификация, которая позволяет производить чтения фрагментов ДНК с обоих концов, что существенно облегчает многие задачи, связанные со сборкой генома. Кроме того этот секвенатор обладает высокой точностью — 99.9% и хорошей производительностью. Однако, имеет достаточно низкую скорость секвенирования: полная сборка генома человека может длиться неделю, что критично для клинических исследований. В настоящее время является наиболее широко используемым.
* IonTorent. Полупроводниковый секвенатор,­­ также работающий с короткими ридами, имеет высокую скорость секвенирования, но обладает более низкой точностью — 99%.
* К более современным технология секвенирования относятся секвенаторы Oxford nanopore и PacBio, основанные на секвенирование при помощи нанопор и одномолекульном секвенирование в реальном времени соответственно. Отличительно особенностью этих подходов является то, что они способны обрабатывать риды длиной до нескольких тысяч оснований. Тем не менее, эти инструменты имеют низкую точность (приблизительно 85%).

1. *Форматы fasta и fastq*

Результатом работы секвенатора является файл, доступный для чтения компьютеру и даже человеку. Существует два формата представления секвенированных данных: это форматы *fasta* и *fastq*. Файлы в данных форматах содержат информацию об изучаемых фрагментах ДНК в виде нуклеотидной последовательности. Отличие этих форматов заключается в том, что в файле *fastq*, в отличие от *fasta*, содержится еще и информация о качестве. Рассмотрим эти форматы подробнее.

Формат *fasta* — это формат представления данных секвенирования в виде последовательности биологической информации совместно с данными о качестве. Ниже представлен пример представления одного рида в формате *fasta*:

**>SEQ\_ID**

**GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAA**

Этот формат достаточно прост и строится следующим образом: первая строка начинается с символа '**>**' и содержит уникальный идентификатор последовательности, после которого может идти некоторая дополнительная информация; вторая строка содержит саму последовательность нуклеотидов. Для последовательности ДНК буквы A, C, G и T кодируют, соответственно, аденин, цитозин, гуанин и тимин. Кроме того, в файле также может присутствовать буква N, что означает, что секвенатор по какой-либо причине не смог однозначно определить нуклеотид на данной позиции.

Формат *fastq* очень похож на *fasta*, но представляет рид с помощью четырех строк:

**@SEQ\_ID**

**GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAA**

**+**

**!''\*((((\*\*\*+))\%\%\%++)(\%\%\%\%).1\*\*\*-+\*''))\*\*55CCF>>>>>**

Первые две строки содержат аналогичную информацию, что и *fasta*, с тем лишь отличием, что идентификатором начала первой строки является символа '**@**'. Третья строка начинается с символа '**+**' и может содержать дополнительную служебную информацию или же оставаться пустой, если такая информация не требуется. Четвертая строка содержит символы, кодирующие уровень качества для каждого из нуклеотидов. Её длина должна совпадать с длиной последовательности нуклеотидов. Рассмотрим подробнее как вычисляется качество каждого нуклеотида.

Уровень качества *Q* — это некоторое целое число, соответствующее вероятности того, что данный нуклеотид некорректен. Существует два подхода вычисления *Q*:

**Шкала PHRED**. В этой шкале каждому нуклеотиду соответствует значение, которое логарифмически зависит от вероятности ошибки:

= ,

где есть вероятность того, что соответствующий нуклеотид ошибочный.

**Шкала Solexa.** В этом случае уровень качества вычисляется следующим образом:

= ,

Для высокого уровня качества оба этих подхода дают одинаковый результат, но для низкого уровня качества различаются.

Для кодирования символов, характеризующих качество, используется таблица ASCII. ASCII-код символа равен уровню качества *Q* плюс некоторая константа. В случае шкалы PHRED эта константа равна 33, а для шкалы Salexa — 64. В обоих случаях код ASCII-код символа не должен превышать 127.

1. *Выравнивание*

Одним из основных инструментов биоинформатики является выравнивание. Выравнивание позволяет сопоставить полученный файл *fastq* некоторому эталонному референсному геному *fasta.* Программа, выполняющая выравнивание, называется алайнером (от англ. aligner) или выравнивателем. Существует около сотни алайнеров и многие из них имеют низкий уровень доверия или основаны на алгоритмах, решающих узкий круг задач. Наиболее популярными алгоритмами, решающими задачу выравнивания являются хэш-таблицы, суффиксные деревья и алгоритм слияния-сортировки. В рамках данной работы мы будем рассматривать только алайнеры, основанные на построении суффиксного дерева, а именно, *bowtie* и *bowtie2* [4]*.* Также, к этому классу программ-выравнивателей относится весьма популярный алайнер *BWA.*

*Bowtie*, *bowtie2* и *BWA* являются высокопроизводительными алайнерами, обладающими хорошей точностью и высоким индексом популярности, предназначенные для выравнивания коротких ридов на большой референсный геном. Для проведения процедуры выравнивания рассматриваемым алайнерам обязательно требуется построение FM-индекса. FM-индекс — это сжатый индексный файл, позволяющий производить навигацию по референсу и экономить оперативную память. В основе построения FM-индекса лежит преобразование Барроуза-Уилера [6].

В результате выравнивания мы получаем файлы в формате SAM, BAM или CRAM, с которыми подробнее познакомимся во второй главе настоящей работы. Эти форматы играют ключевую роль в биоинформатических исследованиях и являются наиболее популярными в данной области. Они открывают широкий спектр возможностей по манипуляции, анализу, визуализации биологических данных. Именно эти форматы используются в лабораторных и клинических исследованиях, в диагностике раковых опухолей, наследственных заболеваний и многом другом.

1. *Обзор биоинформатических инструментов*

Рассмотренные выше технологии являются только подготовительной частью большинства биоинформатических исследований. Наибольший интерес представляет дальнейший анализ данных. В данном разделе мы проведем краткий обзор различных инструментов, которые используются биоинформатиками, и возможностей, которые они предоставляют.

Обычно, анализ секвенированных данных, а именно, файла в формате *fastq,* начинается с проверки качества полученных данных. Причины плохого качества входных данных могут быть самые разные: ошибки работы прибора, грязные пробирки. Некоторые инструменты позволяют проверить это. Например программа *FastQC*  позволяет провести быстрый обзор файла и сообщить пользователю где могут быть проблемы. Кроме того, этот инструмент может собрать некоторую статистическую информацию о ридах. Еще одним инструментом проверки качества первичных секвенированных данных является программа *FastqScreen*, которая позволяет получить за достаточно короткий срок информацию о содержащихся в файле примесях.

Дальнейшим этапом исследования обычно является выравнивание, после чего, анализируются полученные файлы в формате SAM/BAM/CRAM. Одними из самых популярных программ для обработки биоинформатических данных являются *SamTools* и *Picard Tools*. Эти инструменты позволяют решать огромное количество биоинформатических задач, например, рассчитать покрытие нуклеотидов на референсе, собрать некоторую статистическую информацию о данных, удалить дубликаты и многое другое.

В настоящей работе, мы сфокусируем внимание на следующих задачах:

1. Детальное изучение программы *FastqScreen* и её оптимизация. Кроме того, мы предложим новое расширение функционала инструмента с целью упрощения его использования.
2. Улучшение одной из утилит *Picard Tools* путем оптимизации существующего алгоритма обработки данных.

Глава 1. Сравнение последовательности ДНК с эталонным геномом.

1. *Основные принципы работы программы FastqScreen*

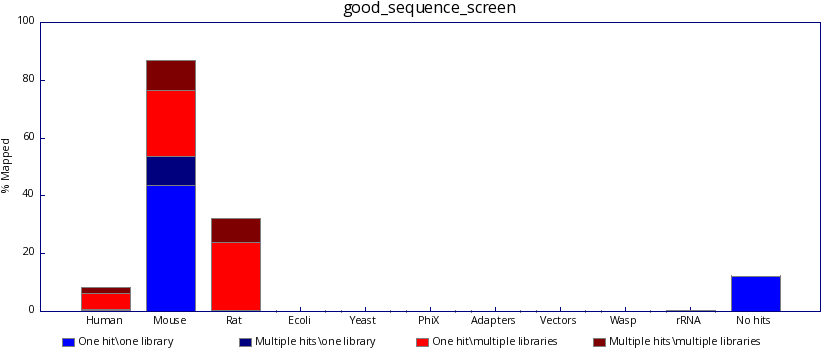
Анализ секвенированных данных стоит начинать с изучения их качества. Существует множество способов проверки качества и какому способу придерживаться зависит исключительно от поставленной задачи. В настоящей главе предлагается рассмотреть инструмент, позволяющий проверить, насколько входные данные соответствуют нашим ожиданиям, а именно, получить информацию о том, к какому организму относится изучаемая выборка и каков процент содержание примесей в ней.

*FastqScreen* — это программный продукт с открытым исходным кодом, разработанный группой биоинформатиков Бабрахамского института (Babraham Institute). Он является полезным помощником для предварительной оценки результатов секвенирования и позволяет решать некоторые задачи биоинформатики.

Данная программа позволяет определить процент содержания различных примесей в изучаемом организме. Примеси, в свою очередь, могут быть как ожидаемыми, так и нет. Например, небольшой процент содержания праймеров или адаптеров, являющимися артефактами процесса секвенирования, вполне допустим. Однако, если примесей такого рода слишком много, то этот факт может свидетельствовать о каких-либо сбоях в настройках оборудования. Кроме того, не нужно исключать и человеческий фактор, ведь пробирки могут быть не идеально промытыми и содержать, к примеру, ДНК бактерии, что может значительно испортить общую картину исследования. Такую информацию хотелось бы получать на начальных этапах изучения биологических данных и программа *FastqScreen* может успешно справиться с этой задачей.

Кроме того, данный программный инструмент позволяет решить задачу определения принадлежности изучаемых данных секвенирования некоторому конкретному организму путем сопоставления изучаемой выборки ряду интересующих нас эталонных геномов. Также, эталонные геномы, каждый из которых принадлежит определенному организму, принято называть библиотеками.

Результатом работы программы *FastqScreen* является текстовый файл, содержащий информацию о сопоставлении изучаемых данных и библиотек и диаграмма, визуализирующая полученную информацию. На Рис. 1.1.1 представлен пример вывода программы, который иллюстрирует выравнивание некоторого генома по библиотекам Mouse, Rat и Human. Как видно из диаграммы, наиболее вероятно, что рассматриваемый организм является мышью, ведь именно этой библиотеке соответствует самый высокий столбец. Кроме того, достаточно большой процент исходных данных выровнялся по библиотеке генома крысы (Rat), что ожидаемо, ведь эти животные принадлежат одному и тому же семейству мышиных. Также, заметим, что в результирующей диаграмме присутствует выравнивание по библиотеке генома человека (Human), что не случайно, так как некоторые участки последовательности ДНК человека могут быть схожи с участками ДНК других живых существ.



*Рис. 1.1.1. Пример результат работы программы FastqScreen.*

Каждый столбец диаграммы характеризует процент выровненных ридов по каждой из перечисленных библиотек. Как мы видим, каждый столбец может включать с себя до четырех цветов и каждый из этих цветов несет в себе информацию о выравнивании. Внизу диаграммы содержится краткое описание для каждого цвета:

* **One hit \ one library** — означает, что рид, попавший в диапазон данного цвета, выровнялся только один раз и исключительно по одной библиотеке.
* **Multiple hits \ one library** — говорит о том, что риды из данной цветовой области выровнялись несколько раз по одной библиотеке.
* **One hit \ multiple libraries —** означает, что риды выровняли по одному разу но по нескольким библиотекам.
* **Multiple hits \ multiple libraries —** говорит о том, что каждый рид, соответствующий данной цветовой области выровнялся по нескольким библиотекам и но несколько раз.

В качестве начальных данных для исследования *FastqScreen* принимает на вход файл в формате *fastq*. Этот файл содержит генетическую информацию о изучаемом организме. Помимо изучаемого организма, нам так же требуется информация о библиотеках. Каждая библиотека также хранит в себе генетическую информацию, только в формате *fasta*. Этот формат схож с форматом *fastq*, за исключением того, что содержит только последовательность нуклеотидов, исключая информацию о качестве. Оба этих формата были подробно описаны во введении.

В основе работы программы лежит выравнивание, которое достигается путем запуска алайнеров *bowtie* или *bowtie2*, в зависимости от пользовательских настроек. *Bowtie* и *bowtie2* являются популярными и высокопроизводительными инструментами для выравнивания генома и занимаю большую часть работу программы. Поэтому, ниже, мы подробно остановимся на их изучении.

Изначально, программа *FastqScreen* была написана на языке программирования *Perl.* Это вызвано тем, что этот язык изначально был предназначен для работы со строками и имеет широкий функционал работы с текстовыми файлами. Кроме того, язык *Perl* достаточно прост в изучении и применении и пользуется огромной популярностью среди специалистов в области биоинформатики.

Целью работы в настоящей главе обусловлены следующие задачи: изучение исходной программы *FastqScreen,* портирование её на высокоуровневый язык программирования *Java*, оптимизация работы алгоритма посредствам данного языка и внедрение новой функциональности для обеспечения более гибкой работы с рассматриваемым приложением.

1. *Описание работы алайнеров bowtie и bowtie2 и их роль в программе FastqScreen*

*Bowtie* и *bowtie2* являются высокоэффективными алайнерами, разработанными для выравнивания коротких последовательностей нуклеотидов ДНК, так же называемых ридами или контигами, на большой референсный геном. Оба инструмента активно используются и развиваются в настоящее время. *Bowtie2* относительно свежий программный продукт, по сравнению с *bowtie*: первый его релиз состоялся в 2013 году, что на шесть лет позже выхода первой версии *bowtie*. Кроме того, *bowtie2* имеет более расширенный функционал возможностей, включающий в себя такие моменты как:

* Обработка ридов больше чем 50bp занимает меньше времени, обладает большей чувствительностью и требует меньше памяти, по сравнению с работой *bowtie*. Более того, *bowtie2* не имеет ограничений на длины контигов, в то время как, *bowtie* способен обрабатывать исключительно риды длиной не превышающей 1000bp.
* *Bowtie2* способен предоставить более точное и более чувствительное выравнивание, нежели *bowtie*.

Однако, нужно отметить, что *bowtie2* не является заменой *bowtie*, так их аргументы запуска все же отличаются.

Для корректной работы аллайнеров *bowtie* и *bowtie2* перед запуском программы FastqScreen требуется сгенерировать вспомогательный файл называемый FM‑индексом. Этот индексный файл нужен для осуществления работы алгоритма выравнивания.

Как уже отмечалось ранее, в основе программы *FastqScreen* лежит выравнивание с помощью описанных выше алайнеров. Для того чтобы лучше понять что происходит при работе этих инструментов рассмотрим аргументы запуска алайнеров. Нам не требуется проводить детальный разбор каждого из аргументов, так как большинство из них не влияют на структуру вывода, поэтому сфокусируемся только на некоторых из них, а именно, на опциях **-k** и **--very-fast-local**.

*Опция -k <int>*

Правильное размещение ридов на референсе может быть неоднозначным, например, один и тот же рид может быть выровнен несколько раз. Такой случай в биоинформатике называется множественным выравниванием (multiple mapping). Один из таких ридов принято называть первичным (primary), остальные риды называются вторичными (secondary). Обычно, первичным ридом становится рид с наилучшим качеством, но решение может быть произвольным, в зависимости от настроек алайнера.

Информация о таких рида регулируется с помощью опции **-k.** Эта команда присутствует как в *bowtie*, так и в *bowtie2*. Опции **-k** соответствует некоторое целое число, характеризующие максимально допустимое количество вторичных ридов. Этот параметр способен оказывать воздействие на производительность приложения. Так, при увеличении значения параметра **-k** уменьшается скорость работы алайнера.

Программа *FastqScreen* запускает *bowtie* и *bowtie2* со значением опции **-k** равной **2**. Наличие именно этой команды влияет на размер и количество цветовых областей в столбцах вывода программы *FastqScreen*. Причем, значение данного параметра **2**, действительно является оптимальным, ведь нас не интересует точное количество выравниваний каждого рида, а лишь информация о проценте присутствия множественного выравнивания.

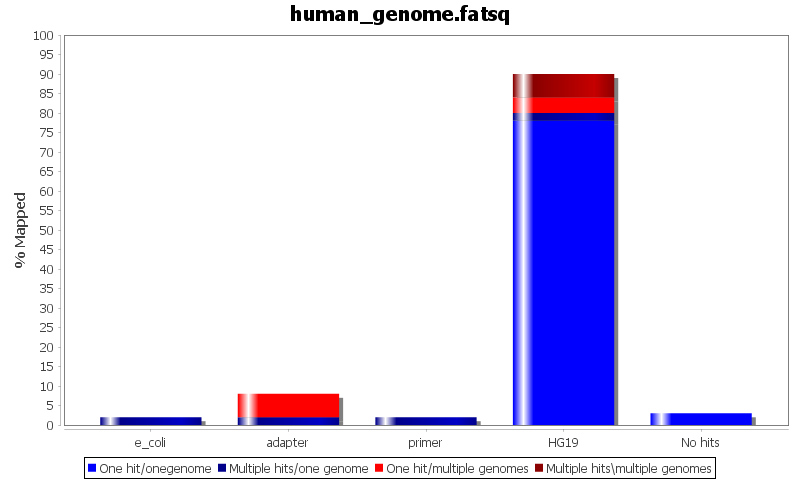
*Опция --very-fast-local*

Данная опция доступна только для алайнера *bowtie2* и именно благодаря возможности поддерживать эту команду *bowtie2* работает быстрее чем *bowtie*. Этот параметр позволяет проводить процедуру выравнивания теряя точность, но при этом выигрывая в производительности. Потеря точности заключается в том, что мы не выравниваем рид целиком на референс, а только его начало и конец. Но для задач, поставленных в рамках программы *FastqScreen* мы вполне можем довольствоваться анализом такого уровня.

1. *Критерий принятия решения*
   1. *Введение*

В настоящее время, многие биологические исследования, будь то опыты в лаборатории (с целью тестирования нового лекарства) или же медицинские учреждения (с целью выявления раковых опухолей), работают по принципу пайнлайн (*pipline*): исследователь выстраивает в ряд различные инструменты. Таким образом, за одно нажатие кнопки мы можем получить из пробирки всю доступную нам информацию об исследуемом организме. *FastqScreen* является инструментом первичного анализа биологических данных, но он жестко завязан на человеческом вмешательстве: именно сам исследователь должен делать выводы о качестве входных данных. Этот факт мешает применять данное приложение в пайплайне, а значит может повлиять на широту его использования. Хотелось бы иметь возможность автоматизировать эту процедуру.

Кроме того, может сложиться так, что по результатам работы программы сложно сделать вывод о качестве данных. Например, предположим что мы получили такой результат:



*Рис.3.1.1 Результат работы программы FastqScreen: пример хорошего выравнивания.*

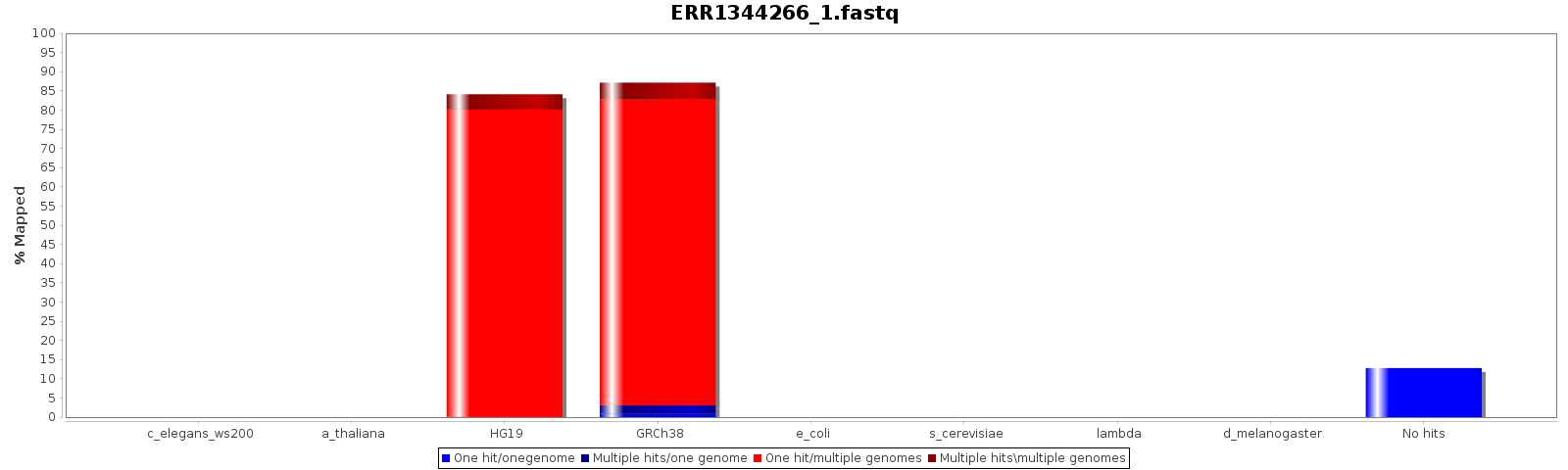
С одной стороны мы видим, что один столбец диаграммы явно выше от остальных, а с дугой стороны примесей достаточно много. Можно ли считать такие результаты приемлемыми?

Полностью исключать человека из данного процесса не следует, ведь для решения некоторых задач необходим анализ глазами человека. Но было бы неплохо построить так называемый критерий принятия решения и помочь исследователю сделать выводы.

Фактически, программа *FastqScreen* нацелена на выявление уровня качества данных. Но что считать хорошими данными, а что "мусором", негодным к дальнейшему анализу? Именно на это вопрос мы постараемся ответить в этом разделе.

Как уже упоминалось ранее, биологические данные бывают разные, цели исследования бывают разные и поэтому уровни качества тоже бывают разные. В связи с этим, изначально стоит определиться с уровнем значимости, который зависит от чувствительности анализа. Такое пороговое значение может выбрать исключительно исследователь, ибо только человек решает, насколько аккуратно ему следует проводить работу. Программно мы можем задать лишь значение по умолчанию.

Напомним, что программа *FastqScreen*выдаетинформацию о соответствии одного входного *fastq* файла какого-то организма различным библиотекам. Этих библиотек может быть много, и *fastq* файл может выровняться по нескольким из этих библиотек. В большинстве случаев мы не ожидаем такого поведения и такая ситуация может говорить о низком качестве данных. Но бывает, что именно такое поведение мы и ожидаем. Например, если мы выравниваем  некоторый геном (к примеру человека) по двум библиотекам каждая из которых является геномом человека, но разных сборок (например 19 и 38). Тогда мы можем получить примерно такую картину:

*Рис. 3.1.2. Пример результата работы программы FastqScreen: выравнивание по нескольким библиотекам.*

Такой результат связан с тем, что исследовался один и тот же организм, но эталонные геномы были взяты из разных источников. Такого рода анализ явно не предусматривает каких-либо пороговых значений, так как этот анализ нацелен больше на исследование, а не на получение информации о качестве продукта.

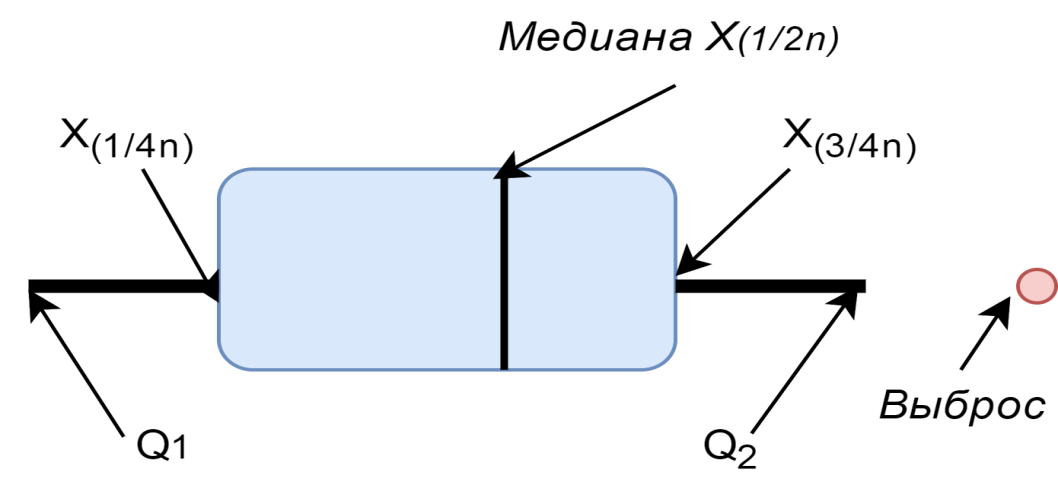
Зачастую, в качестве библиотек мы подаем геном предполагаемого организма и различные геномы бактерий, адаптеров или примесей, которые как нам кажется могут присутствовать в данных. Тогда, целью программы, является, показать процент содержания посторонних организмов в изучаемом геноме.

Таким образом, в большинстве случаев нам все же интересна информация о качестве входных данных, поэтому программная реализация принятия решения имеет место быть.

* 1. *Описание критерия*

Теперь перейдем непосредственно к построению критерия принятия решения. Рассмотрим типичный результат работы программы представленный на Рис. 3.1.1. Результатом хорошего секвенирования могут считаться только те данные, которые выровнялись преимущественно только по одной библиотеке. Из рисунка видно, что рассматриваемый организм лучше всего выровнялся по библиотеке HG19, в отличие от всех остальных. Поэтому построение диаграммы размаха здесь вполне уместно: значение, соответствующее самому высокому столбцу диаграммы, можно считать выбросом. Данный метод подробно описан в публикации [3] и активно применяется для анализа данных.

Для начала разберемся с тем что такое *box plot* или "ящик с усами". Рассмотрим выборку и построим для нее порядковые статистики  . Диаграмма размаха — это вид отображения графических данных, использующийся в описательной статистике. Диаграмма размаха показывает медиану выборки, верхний и нижний квартили, а также выбросы. Выброс — это значение, которое расположено аномально далеко от остальных. Визуально диаграмма размаха выглядит так:



*Рис. 3.2.1. Диаграмма размаха.*

Для построения диаграммы размаха необходимо знать всего три величины: медиану, первый и третий квартили. Напомним, что медиана (или 50-ый персентиль) это , первый квантиль (или 25-ый персентиль) это , а третий квантиль (или 75-ый персентиль) это . Интерес представляет вычисление длины усов, ведь именно благодаря этой характеристике мы можем делать выводы о количестве выбросов. Границы усов — это края статистически значимой выборки, то есть выборки без выбросов. Нижняя и верхняя граница усов вычисляется по формулам:

где *k* обычно принимается равным 1.5.

Значения, которые вышли за пределы границы усов считаются выбросами.

Вернемся к рассмотрению результатов работы *FastqScreen*. Пусть выборке соответствуют результаты работы программы, а именно процентное содержание ридов выровненных по каждой из библиотек. Тогда максимальной порядковой статистике будет соответствовать предполагаемы эталонный геном. Основываясь на вышесказанном, для того чтобы сделать выводы о качестве данных, нужно потребовать, чтобы выброс присутствовал и при том ровно один (и этот выброс будет ).

Кроме того, для того чтобы считать результаты секвенирования хорошим, нам необходимо чтобы количество примесей было минимальным. Для этого нам нужно ввести некоторое пороговое значение , согласно которому мы будем принимать решение. Этот порог будет зависеть от требований к строгости анализа и выбираться пользователем самостоятельно. Результаты работы программы показывают процентное соотношение, в котором риды выровнялись по той или иной библиотеке, а значит величина каждого столбца варьируется от 0 до 100, следовательно пороговое значение лежит в рамках отрезка [1..100].

Как уже упоминалось ранее, мы допускаем, что видов примесей может быть достаточно много, главное чтобы их в процентном соотношении было как можно меньше. Для того чтобы убедиться, что примесей небольшое количество, предлагается взять среднее по всем примесям и проверить что оно не превосходит заданного порогового значения.

Теперь мы можем формально описать критерий. Пусть имеется выборка и соответствующий ей вариационный ряд . И пусть для выборки выполнены следующие условия:

1. У диаграммы размаха существует единственный выброс и он является максимальным элементом выборки, а именно:

! *k*: < и при этом *k = n. (1)*

1. Среднее выборки не должно превосходить заданного критического значения:

*< . (2)*

Будем говорить, что данные не противоречат гипотезе на заданном уровне значимости , если для ряда выполнены условия *(1), (2).* В противном случае будем говорить, что гипотеза отвергается.

Таким образом, нулевой гипотезе будет соответствовать предположение о том, что входные данные являются качественными и если нулевая гипотеза отвергается, то и данные признаются непригодными для дальнейшего анализа.

* 1. *Реализация*

Для имплементации данного критерия нам потребуется два объекта: массив с результатами работы программы *sample* и критическое значение *criticalValue*, заданное пользователем. Если пользователь не указал это число, то передается значение по умолчанию. В нашем случае достаточно будет задать 5%, что допускает небольшое количество примесей. На Листинге 3.3.1 представлена реализация данного критерия.

1. **int** n = sample.length;
2. **double** k = 1.5;
3. **int** median = sample[Math.round(n / 2) - 1];
4. **int** percentile\_25 = sample[Math.round(n / 4) - 1];
5. **int** percentile\_75 = sample[Math.round(3 \* n / 4) - 1];
6. **int** maxValue = Arrays.stream(sample).max().getAsInt();
7. **double** mean = Arrays.stream(Arrays.stream(sample)
8. .filter(value -> (value != maxValue))
9. .toArray()).sum() / (n-1);
11. **double** q1 = percentile\_25 - k \* (percentile\_75
12. - percentile\_25);
13. **double** q2 = percentile\_75 + k \* (percentile\_75
14. - percentile\_25);
16. **return** (mean < criticalValue)
17. && (Arrays.stream(sample)
18. .filter(value -> (value >= q2)).count() == 1);

*Листинг 3.3.1. Имплементация критерия принятия решения.*

В итоге, мы построили и реализовали критерий, позволяющий облегчить процесс принятия решения и предоставляющий возможность автоматизировать процедуру обработки первичных биологических данных.

1. *Программная реализация FastqScreen*
   1. *Исходная реализация на языке Perl*

Алгоритм исходной программы *FastqScreen* построен следующим образом:

* Во-первых происходит загрузка конфигурационных данных. Ввод входных конфигураций возможен как из командной строки, так и из конфигурационного файла.
* Затем производится проверка разрядности системы, что является важным параметром для языка *Perl*, поскольку в ходе выполнения программы значения некоторых переменных хранятся в одном адресном пространстве. В языке *Perl* используется скалярный тип данных, который занимает либо 32, либо 64 бита в оперативной памяти (размер зависит от архитектуры операционной системы).
* Далее идет открытие входного файла в формате *fastq* и вычисление некоторой полезной информации: количество ридов в файле, длина кротчайшего рида и расчета интервала, для того, чтобы в дальнейшем передать ее в строки запуска алайнеров.
* Далее осуществляется повторное открытие входного файла для записи ридов во временный файл, который получен в результате работы одного из алайнеров. При этом для экономии памяти результаты записываются в одно адресное пространство.
* После этого, производится обработка файла, полученного в результате выравнивания и собирается информация о том, по каким библиотекам выровнялись риды рассматриваемого входного файла.
* Затем, строится финальная диаграмма с результатами и сопровождающий ее текстовый файл.

Кроме того, Perl версия программы использует вспомогательные модули языка: GD::Graph и IO::Uncompress::Gunzip. GD::Graph — инструмент, предназначенный для построения графиков, IO::Uncompress::Gunzip — модуль, позволяющий работать с сжатыми файлами.

* 1. *Реализация по средствам языка Java*

Как упоминалось выше, изначально *FastqScreen* была написана на языке *Perl*. Этот язык является высокоуровневым интерпретируемым динамическим языком программирования общего назначения. Основной особенностью данного языка является широкий спектр возможностей работы с текстовыми данными. Как и во многих других языках программирования, *Perl* имеет механизм поддержки внешних вспомогательных модулей, то есть программ, ранее написанных на языке *Perl* и позволяющих использовать себя для решения текущих задач. Все такие модули хранятся в архиве репозитория CPAN (Comprehensive Perl Archive Network) и должны быть предварительно загружены перед пользовательским запуском любого *Perl*-приложения. Примером таких модулей служат ранее упомянутые GD::Graph и IO::Uncompress::Gunzip. Одним из недостатков данного языка является тот факт, что *Perl* сохраняет в себе ранее популярные, но, к сожалению, уже устаревшие подходы.

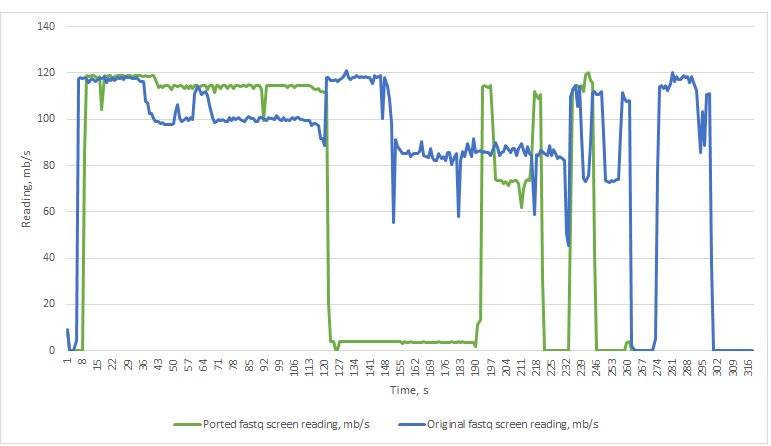
Одной из задач настоящей работы было портирование рассматриваемой программы *FastqScreen* с языка *Perl* на язык *Java*. *Java* является высокоуровневым объектно-ориентированным языком программирования, который активно используется и развивается в настоящее время, что позволяет использовать современные эффективные подходы для решения поставленных задач.

В рамках настоящей работы программа *FastqScreen* была полностью переписана на язык *Java*, была проведена оптимизация работы алгоритма с помощью возможностей этого языка и расширен функционал возможностей, основанный на применении критерия принятия решения. Перечислим основные моменты оптимизации:

1. Код на *Perl* обращается к файлу *fastq* дважды: в первый раз для получения общего числа ридов, длины кротчайшего рида и расчета интервала, во второй — для создания временного усеченного файл с определенным количеством ридов. Оба раза файл построчно просматривается от начала до конца и происходит обработка всех строк , даже тех, которые не войдут во временный файл.

В коде на *Java* при первом чтении *fastq* файла создается еще один вспомогательный файл, содержащий позиции всех ридов. Благодаря этому, при создании усеченного файла с ридами программа просматривает не весь файл *fastq*, а перемещается от одного рида к другом.

На Рис. 4.2.1 четко видно, что новый подход заметно снижает нагрузку на жесткий диск и уменьшает количество информации, которую необходимо считать, и, как следствие, сокращает общее время работы программы.

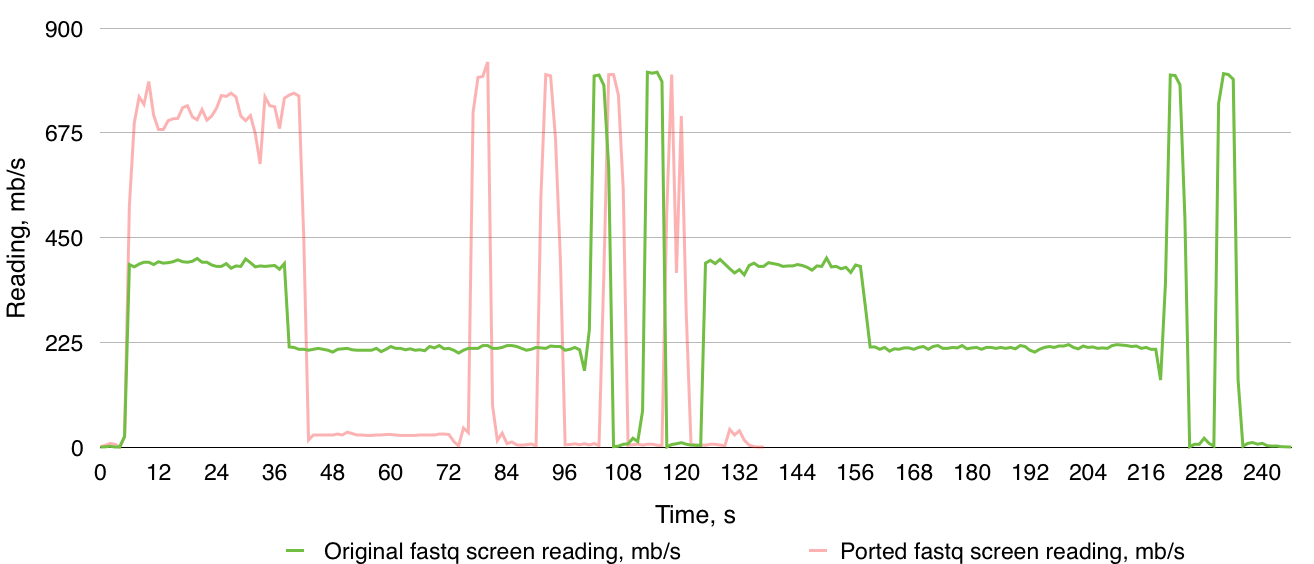


*Рис. 4.2.1. Результаты исследования скорости работы с диском. Синим цветом обозначены результаты для версии на Perl, зеленым на Java.*

1. Обработка нескольких файлов кодом на перле идентична последовательным запускам приложения с одинаковыми параметрами. Недостатком этого подхода является отсутствие возможности кэширования индексов *bowtie* операционной системой, а достоинством — пользователь максимально быстро получает результаты обработки первого файла.

В портированной программе на *Java* после записи временных файлов, каждый из них проверяется сначала по первой библиотеке, затем по второй и т.д., что позволяет кэшировать индексы *bowtie* и сокращает время работы приложения.

1. Для многопоточной обработки данных предложен новый конфигурационный параметр **--ssd**. Его следует указывать при работе с несколькими файлами находящимися либо на твердотельном накопителе, либо на отдельных жестких дисках. В этом случае начальная обработка двух исходных файлов происходит параллельно.



*Рис. 4.2.2. Результаты исследования работы с диском в случае обработки двух файлов.*

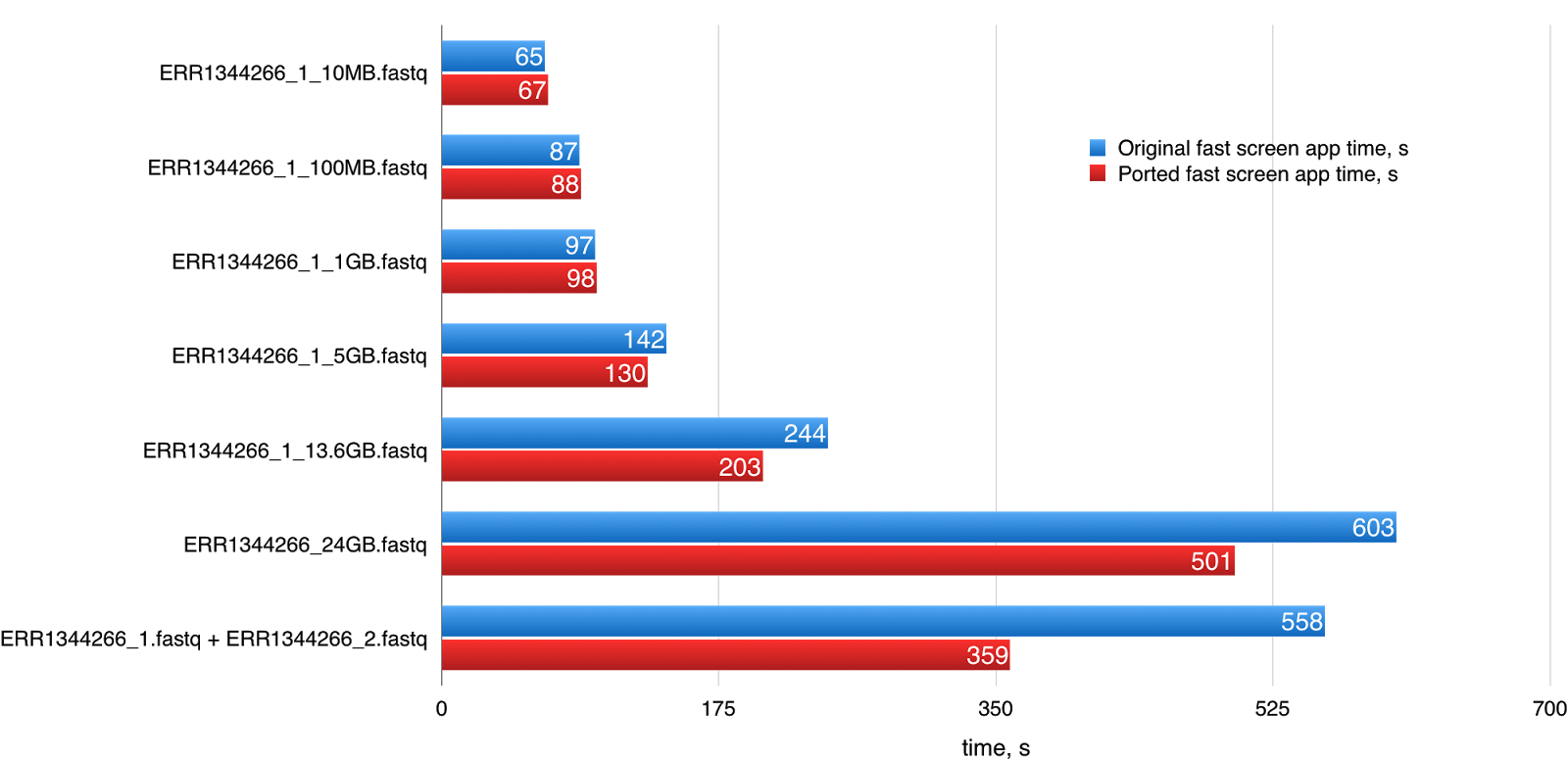
Рис. 4.2.2 иллюстрирует повышение эффективности использования носителя информации портированной программы с параметром **--ssd** по сравнению с кодом на *Perl*. При обработке файла *Perl*-приложением скорость чтения данных не превышает 400мб/с, на *Java* же сразу считывает два файла с скоростью около 800мб/с за тот же временной промежуток.

1. Пропала необходимость предварительной загрузки модулей для построения графического вывода и работы со сжатыми файлами, так как *Java* не требует подключения сторонних библиотек пользователем*.*
2. В коде на *Perl* для хранения информации о выравнивании ридов по библиотекам используется тип данных *Scalar*. Его размер зависит от архитектуры ОС и ограничивает максимальное количество подключаемых библиотек: 15 для 32-х битных ОС и 32 для 64-х битных.

В *Java* для подобных процедур предусмотрен специальный класс *BitSet*. Используемый им объем оперативной памяти прямо пропорционален количеству подключенных библиотек и не налагает ограничения на их количество, а внесение в него новой информации легко осуществляется с помощью встроенных методов.

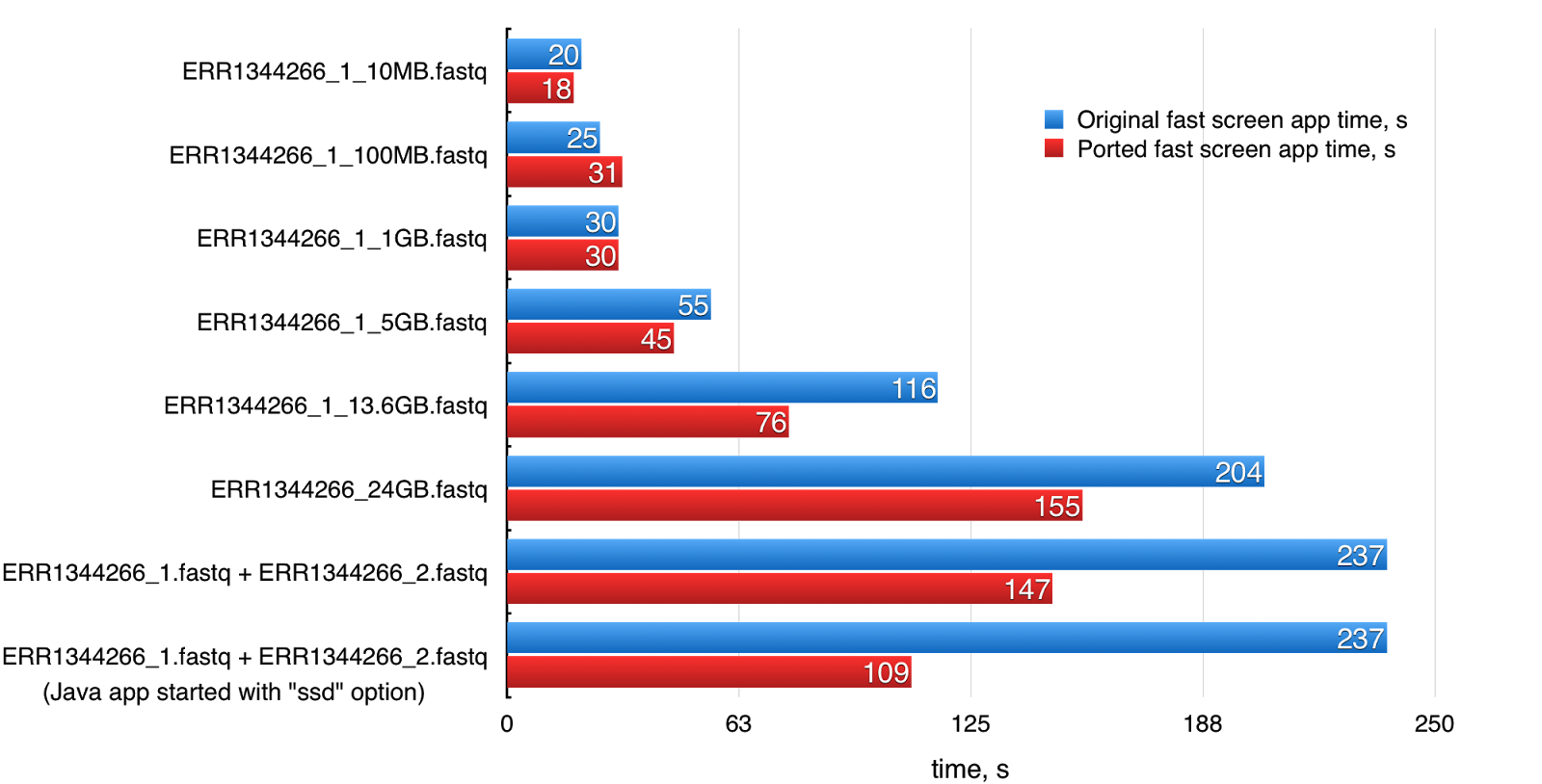
* 1. *Результаты замеров производительности*

На Рис. 4.3.1 представлены результаты сравнения скоростей обработки файлов различного объема для программы *FastqScreen* написанной на *Perl* и *Java*. Тест проводился на жестком диске. Время работы программы на Java показано красным цветом (а на *Perl* — синим). На файлах небольшого объема разница между скоростями работы программ не значительна, т.к. в этих случаях она ограничивается только производительностью алайнеров *bowtie/bowtie2*. Когда же размер файла превышает 5Гб, наблюдается заметное сокращение времени работы программы на Java за счет изменений в алгоритме. Так, файл размером 24Гб обрабатывается на 120% быстрее, а два файла по 13Гб — на 155%.



*Рис. 4.3.1. Результаты замеров времени оригинальной и портированной программ.*

Аналогичные результаты получены и на твердотельном носителе. Здесь, стоит обратить особое внимание на то, что при использовании опции --**ssd** *Java*-приложение работает на 217 % быстрее.



*Рис. 4.3.2. Результаты замеров времени оригинальной и портированной программ на твердотельном носителе с использование новой опции --ssd.*

Глава 2. Применение метода нечеткого поиска к задаче поиска повторяющихся нуклеотидов последовательности ДНК

В предыдущей главе был рассмотрен метод первичного анализа с целью выявления общей адекватности биологических данных. В настоящей главе мы рассмотрим один из способ дальнейшего анализа и познакомимся с одним из биоинформатических инструментов, предоставляющий широкий спектр возможностей управления, анализа, обработки и визуализации секвенированных данных.

1. *Описание форматов SAM/BAM/CRAM*

Для начала нам потребуется познакомится с одним из самых распространенных форматов представления биологических данных. Об этом формате уже упоминалось в предыдущей главе — это формат SAM (Sequence Alignment/Map). Аналогом ему является формат BAM, который представляет собой бинарный вариант SAM и формат CRAM, имеющий немного другую структуру, но хранящий в себе все ту же информацию. Файл в формате SAM/BAM/CRAM получается в результате выравнивания файла FASTQ на референс FASTA. Мы не будем вдаваться в детали бинарных форматов, хотя на практике именно и используются. Для дальнейшей работы нам понадобится разобраться только в том, какую информацию содержат в себе SAM файлы и как ее интерпретировать.

Файл в формате SAM представляет собой текстовый файл, обладающий следующей структурой: сначала идет секция с заголовком файла, а затем информация о каждом риде. Секция с заголовком, является опциональной, но как правило она присутствует, ибо в ней содержится достаточно много полезной информации о том как происходило выравнивание. Рассмотрим структуру формата подробнее.

Строки заголовка начинаются с символа '@'. Затем следует информация о версии формата, о сортировке файла (coordinate, queryname и unsorted), о референсе и много другой информации, которую полезно иметь.

1. @HD VN:1.5 SO:coordinate
2. @SQ SN:chr1 LN:249250621

*Пример 2.1.1. Заголовка файла в формате SAM.*

В приведенном выше примере, мы наблюдаем, что риды в файле отсортированы по координатам и что выравнивание происходило по первой хромосоме (chr1), длина которой составляет 249250621 нуклеотидов.

Далее следует секция с ридами, а именно информация о их выравнивании. Каждая запись этой секции содержит 11 обязательных полей, разделенных символом табуляции. Спецификация формата предусматривает наличие дополнительных полей, появление которых зависит от настроек алайнера и от цели исследования. Разберемся с каждым из обязательных полей.

1. **QNAME** —имя рида; это поле не уникально: одно и то же имя рида рида может присутствовать в файле неоднократно. Это может быть вызвано например тем, что выравнивание рида на референс произошло несколько раз.

2. **FLAG** —бинарный флаг рида. Благодаря этому полю мы можем получить различные характеристики рида. В данной главе наибольший интерес будут представлять следующие характеристики: является ли рид парным (этому событию соответствует флаг 1), его направление: обратное (16) и обратное парного (32), является ли он оптическим дубликатом, полученным в результате ПЦР (1024). Остановимся на понятии парных ридов. Говорят, что рид является парным, если существует парный ему. Для наглядности, продемонстрируем это на рисунке:

D:\downloads\paired_reads.png

*Рис. 2.1.2. Наглядное представление парных ридов.*

Если мы наложим оба рида на референс, то последовательность нуклеотидов, зажатая между началом первого и концом второго, будет соответствовать выровненному контигу из файла FASTQ, который и послужил образованию этих ридов.

3. **RNAME** —название эталонной последовательности, на которую произошло выравнивание, например название хромосомы.

4. **POS** —самая левая позиция рида, которая выровнялась.

5. **MAPQ** —качество выравнивания. Это значение вычисляется по формуле:

-10 P(нуклеотид, соответствующий **POS** считан неверно)

и округляется до ближайшего целого.

6. **CIGAR** — строка, которая содержит в себе информацию различного рода несоответствиях рида заданному референсу.

7. **RNEXT** —название референсной последовательности для второго в паре рида. Если парный рид выровнялся на ту же референсную последовательность, то этому полю присваивается значение "=". В случае, например, межхромосомного выравнивания, значения полей **RNEXT** и **RNAME** будут различны.

8. **PNEXT** —самая левая позиция парного рида, которая выровнялась.

9. **TLEN** —количество нуклеотидов с самой левой позиции левого рида до самой правой выровненной позиции правого рида.

10. **SEQ** —последовательность нуклеотидов соответствующая риду.

11. **QUAL** —строка, содержащая качество, соответствующее каждому нуклеотиду. Это поле может быть помечено символом "\*", если хранение строки качества алайнером не предусмотрено (как в примере ниже).

1. r001  99 ref  7 30 8M2I4M1D3M = 37  39 TTAGATAAAGGATACTG \*
2. r002  0 ref  9 30 3S6M1P1I4M \*  0  0 AAAAGATAAGGATA  \*
3. r003  0 ref  9 30 5S6M  \*  0  0 GCCTAAGCTAA  \*
4. r004  0 ref 16 30 6M14N5M  \*  0  0 ATAGCTTCAGC  \*
5. r003 2064 ref 29 17 6H5M  \*  0  0 TAGGC  \*
6. r001  147 ref 37 30 9M  =  7 -39 CAGCGGCAT  \* NM:i:1

*Пример 2.1.3. Секция с информацией о выравнивании ридов.*

1. *Описание набора инструментов Picard Tools*

*Picard Tools* — это проект с открытым исходным кодом, представляющий собой набор инструментов командной строки для работы с такими форматами как SAM/BAM/CRAM и VCF. *EstimateLibraryComplexity* — один из инструментов Picard Tools, предназначенный для оценки сложности библиотеки, а именно, оценки количества уникальных молекул в библиотеке. Сложность библиотеки — это количество уникальных фрагментов ДНК, присутствующих в данной библиотеке. Большая сложность библиотеки, вызванная амплификацией во время ПЦР, может повлиять на дальнейший анализ, так как в этом случае мы будем иметь дело с увеличенным количеством повторяющихся ридов. Образование дубликатов во время ПЦР может быть вызвано различными причинами, например, недостаточное количество исходного материала. Повторяющиеся риды так же могут возникнуть из-за наличия оптических дубликатов, появляющиеся в следствии работы оптических датчиков секвенатора.

*EstimateLibraryComplexity* обрабатывает только парные риды. Риды сортируются сначала по первым MIN\_IDENTICAL\_BASES нуклеотидам (как правило MIN\_IDENTICAL\_BASES = 5) рида первого в паре, а затем по первым MIN\_IDENTICAL\_BASES нуклеотидам парного ему. Мы считаем пару ридов дубликатами, если общая частота несоответствий в рида не превосходит заданного значения MAX\_DIFF\_RATE (по умолчанию это число равно 0.03). Кроме того, риды имеющие плохое качество исключаются, для того чтобы получить более аккуратную оценку.

Согласно алгоритму программы, оптические дубликаты рассчитываются отдельно от дубликатов, полученных во время ПЦР, и исключаются при конечном расчете сложности библиотеки. После обработки всех ридов строится гистограмма, которая демонстрирует количество дублирующихся ридов в библиотеке. Для наглядности рассмотрим пример вывода программы:

1. ## HISTOGRAM java.lang.Integer
2. duplication\_group\_count Unknown
3. 1 1807757
4. 2 227281
5. 3 41309
6. 4 8657
7. 5 2031
8. 6 518
9. 7 140
10. 8 53
11. 9 23
12. 10 3
13. 11 3
14. 12 1

*Пример 2.2.1. Вывод работы программы EstimateLibraryComplexity*

Данная гистограмма показывает, что для библиотеки *Unknown* количество ридов имеющих только один дубликат равно 1807757, соответственно два дубликата — 227281 и так далее.

Риды, которые не имеют дубликатов рассматриваются как выбросы и удаляются из оценки сложности библиотеки.

Таким образом, представленный инструмент решает следующую алгоритмическую задачу: на основании входных параметров MAX\_DIFF\_RATE и MIN\_IDENTICAL\_BASES найти количество повторяющихся ридов во входном файле.

1. *Базовый алгоритм*

Перейдем к рассмотрению первоначального алгоритма программы. Как упоминалось ранее, согласно алгоритму работы программы, риды для поиска дубликатов поступают в уже отсортированном порядке. Для хранения пар ридов используется структура *PairedReadSequence* которая хранит в себе следующую информацию о риде и о парном ему:

1. // Последовательность нуклеотидов рида, первого в паре
2. **byte**[] read1;
3. // Последовательность нуклеотидов рида, второго в паре
4. **byte**[] read2;

*Листинг 2.3.1. Инициализация полей класса PairedReadSequence*

В дальнейшем, интерес будут представлять только поля *read1* и *read2.*

Для того чтобы выяснить являются ли пары ридов дубликатами, алгоритм последовательно просматривает все пары. Для более ясного понимания, опишем один шаг алгоритма.

**Шаг n.** На вход поступают два объекта *PairedReadSequence*, **n**-ый и **(n+1)**-ый, предварительно отсортированные по первым MIN\_IDENTICAL\_BASES нуклеотидам. Обозначим первый их как *lhs* и *rhs* соответственно.

1. // Максимально возможная длина рида
2. **final** **int** maxReadLength = (MAX\_READ\_LENGTH <= 0)
3. ? Integer.MAX\_VALUE : MAX\_READ\_LENGTH;
4. // Минимальная длина для первого в паре рида
5. **final** **int** read1Length = Math.min(Math.min(lhs.read1.length,
6. rhs.read1.length), maxReadLength);
7. // Минимальная длина для второго в паре рида
8. **final** **int** read2Length = Math.min(Math.min(lhs.read2.length,
9. rhs.read2.length), maxReadLength);
10. // Максимально возможное количество ошибок в последовательности
11. **final** **int** maxErrors = (**int**) Math.floor(
12. (read1Length + read2Length) \* maxDiffRate);
13. // Счетчик ошибок
14. **int** errors = 0;

*Листинг 2.3.2. Инициализация локальных полей для работы алгоритма.*

Алгоритм посимвольно сравнивает риды: сначала первый в паре, а затем второй. Если частота различий превзошла MAX\_DIFF\_RATE то пары ридов признаются не дубликатами и мы переходим к **n+1** шагу алгоритма, иначе риды признаются дубликатами. Так как мы работаем с предварительно отсортированными по первым MIN\_IDENTICAL\_BASES нуклетидам ридами итерацию алгоритм начинает не с первого элемента, а с элемента стоящего на позиции MIN\_IDENTICAL\_BASES.

1. // Итерация по первому в паре
2. **for** (**int** i = MIN\_IDENTICAL\_BASES; i < read1Length; ++i) {
3. **if** (lhs.read1[i] != rhs.read1[i] && ++errors > maxErrors) {
4. **return** **false**;
5. }
6. }
8. // Итерация по второму в паре
9. **for** (**int** i = MIN\_IDENTICAL\_BASES; i < read2Length; ++i) {
10. **if** (lhs.read2[i] != rhs.read2[i] && ++errors > maxErrors) {
11. **return** **false**;
12. }
13. }

*Листинг 2.3.3. Базовый алгоритм работы программы.*

В худшем случае вычислительная сложность данного алгоритма составляет , где — длина первого в паре рида, а — второго. Это весьма непродуктивно и хотелось бы придумать более эффективный подход.

1. *Метод нечеткого поиска*

Теперь перейдем к оптимизации базового алгоритма. Для этого нам потребуется познакомиться с методом нечеткого поиска. Такое название метод носит в связи с тем, что позволяет осуществлять поиск элементов массива, с допущением некоторого определенного числа несовпадений. Этот метод основан на хэш-функциях и позволяет осуществлять доступ к данным более эффективно.

Для начала определим что такое хэш-функция.

*Определение.* Пусть множество будет множеством массивов длины *k*, а — множество массивов произвольной длины. Хэш-функцией

*h:*

называется преобразование входного массива произвольной длины в массив фиксированной длины. Множество также называют множеством ключей, а множеством значений. Процесс такого преобразования называется хэшированием.

*Замечание 1.* На практике, как правило, выходной массив обладает гораздо меньшей длиной, чем входной.

*Замечание 2.* Обычно, входной массив может содержать в себе объекты какого угодно типа, а выходной массив принято конструировать целочисленным.

*Замечание 3.* Согласно принципу Дирихле, в случае неидеальной хэш-функции, нет взаимно-однозначного соответствия между ключами и значениями. Всегда существует вероятность коллизии, а именно, ситуации, когда двум различным ключам, сопоставляются одинаковые значения. Поэтому важно придумать такую функция, которая выполняла бы свою работу с минимальных количеством коллизий, не теряя при этом производительность.

Более детально познакомиться с представленным алгоритмом и изучить его возможные оптимизации и применения можно здесь [5].

Идея хэширования заключается в том, чтобы разбить входной массив объектов на равные части и сопоставить каждой из них определенной число, полученное по некоторому правилу. Таким образом, возникает два вопроса:

1. по какому правилу преобразовывать части входного массива в числа?
2. на сколько частей разбивать входной массив?

Ответим на эти вопросы в рамках текущей задачи. В нашем случае, множество будет представлять собой множество ридов, а значит в качестве входного массива мы будем рассматривать последовательность символов. Выходной массив будет состоять из целочисленных значений, которые являются результатом применения хэш-функции к входному массиву.

1. **Построение хэш-функции.**

Существует несколько способов построения хэш-функций, например, основываясь на делении с остатком, полиномиальные, комбинированные. Какой вид хэш-функции выбирать зависит от поставленной задачи. В нашем случае достаточно будет построить, пожалуй, самую распространенную.

Пусть имеется строка *read*, которую мы хотим захэшировать, разбитая на *l* подстрок, каждая длиной *k*. Рассмотрим произвольную подстроку , *d*  *[1..l].* Хотим построить такую функцию *h,* которая сопоставляла бы подстроке некоторое целое уникальное число, с минимальной вероятностью коллизий и при этом, не проиграть в быстродействии. Рассмотрим функцию

*h(i) = + ,*

где *i*  *[1..l],* а *p* — простое. Выбор *p* влияет на качество хэш-функции. Например, если *p* будет слишком большим, то вычисление хэша будет занимать много времени, но количество вероятных коллизий будет мало. Однако, если мы возьмем слишком малое *p*, то вероятность коллизий будет велика, за то время выполнения будет меньше. Таким оптимальным числом будет 31. Таким образом, получаем конечную хэш-функцию:

*h(i) = + ,* где *i*  *[1..l].* (1)

Остается только выяснить, как вычисляется *k* и на сколько частей делить входной массив *read*.

1. **Разбиение входной последовательности.**

Перед тем как перейти непосредственно к вычислению длины *k*, вспомним, что перед нами стоит задача поиска дубликатов в множестве последовательностей, причем с допущением некоторого числа отклонений. Обозначим число возможных ошибок как *errorsCount*.

Для наглядности рассмотрим один простой случай. Предположим, что мы сравниваем две последовательности, имеющие одинаковую длину *L*. И пусть эти строки различаются ровно в одном символе. Если мы выберем в качестве значения *k* длину этих последовательностей, то их хэш-значения будут различны, а значит эти строки будут рассматриваться как не дубликаты, хотя по условию задачи такая пара должна считаться дубликатами. Поэтому, оптимальнее всего будет разбить последовательность на *errorsCount* частей, ведь в этом случае хэш-значения не совпадут только для одного блока и нужно будет посимвольно проверять только его.

Длина входной строки, вообще говоря, не кратна величине *errorsCount*, поэтому в подавляющем большинстве случаев деление длины входной последовательности на *errorsCount* будет с остатком. Следовательно, правильнее всего будет разбивать входную последовательность на *errorsCount + 1* блоков, причем первые *errorsCount* блоков будут иметь фиксированную длину *k,* а последний блок будет длиной *L mod errorsCount.*

Очевидно, что выбор *k* зависит от того, на сколько частей мы разбиваем входную строку, а то на сколько частей мы ее делим, зависит от длины этой строки. Кроме того, значение k должно быть одинаковым для всех строк, что мы обрабатываем. Поэтому, при расчете величины *k* мы должны принимать во внимание тот факт, что длины входных последовательностей могут отличаться. Если мы выберем *k* слишком маленьким, то можем не получить ожидаемый выигрыш в производительности, поэтому значение параметра *k* обычно выбирают максимально возможным большим. Следовательно, нужно понять чем ограничивать выбор этого числа.

Вернемся к предыдущему примеру, с тем лишь изменением, что нам заранее известно, что мы работаем со строками, которые имеют максимальную для рассматриваемого множества входных последовательностей длину . Если мы разобьем эти последовательности способом, описанным выше, на *errorsCount + 1* частей, то может случиться так, что полученная длина *k* превзойдет длину минимальной строки , а значит хэширование минимальной строку будет неэффективным, а значит и быстродействие алгоритма в целом будет сомнительным. В таком случае, хорошим ограничением будет служить длина минимальной входной последовательности, и, следовательно, величина *k* может быть вычислена по следующей формуле:

*k =*

1. *Реализация метода нечеткого поиска*

Прежде всего нам потребуется инициализация вспомогательных полей в классе *PairedReadsequence*, а именно массивов, содержащих значения хэш-функций для первого и второго ридов соответственно:

1. **int**[] hash1;
2. **int**[] hash2;

Для дальнейшей работы алгоритма нам потребуется вычислить оптимальную длину *k*. Для этого в цикле по всем ридам в группе выполняем следующие действия:

1. // Вычисляем длину минимального рида
2. **int** minReadLength =
3. Math.min(prs.read1.length, prs.read2.length)
4. - MIN\_IDENTICAL\_BASES);
5. // Вычисляем количество допустимых разбиений
6. **int** l = (**int**) (minReadLength \* MAX\_DIFF\_RATE) + 1;
7. // Вычисляем текущее значение k; если в какой-то момент текущее
8. // значение окажется меньше имеющегося, то присваиваем новое
9. // значение k
10. **int** current\_k = minReadLength / l;
11. **if** (k > current\_k) {
12. k = current\_k;
13. }

*Листинг 2.5.1. Вычисление наиболее оптимального значения k.*

Далее мы для каждого рида *read1* и *read2* заполняем массивы хэш-значений *hash1[]* и *hash2[]* соответственно. Для этого мы в цикле по всем ридам выполняем следующие действия: определим *l* — количество разбиений рида на равные части длинной *k*  и для каждого из этих разбиений вычислим хэш функцию заданную формулой *(1)*.

1. // Мы пропускаем первые MIN\_IDENTICAL\_BASES,
2. // так как в рамках одной группы ридов они одинаковы
3. **int** l = (read.length - MIN\_IDENTICAL\_BASES) / k;
4. // Массив хэш-значений
5. **int**[] hashValues = **new** **int**[l];
6. **for** (**int** i = 0; i < l; i++) {
7. **int** start = MIN\_IDENTICAL\_BASES + i \* k;
8. **int** end = start + k;
9. // Вычисляем i-ое хэш-значение, соответствующее
10. // промежутку (start, end) на риде
11. hashValues[i] = 1;
12. **for** (**int** j = start; j < end; j++) {
13. hashValues[i] = 31 \* hashValues[i] + read[j];
14. }
15. }

*Листинг 2.5.2. Вычисление хэш-функции.*

Таким образом, мы выполнили все необходимые приготовления для перехода к самому поиску дубликатов. Для этого в цикле по парам ридов в рамках одной группы бы будем производить последовательные сравнения по хэш-значениям. Если они различны, то только в этом случае мы переходим к сравнению частей последовательностей. При сравнении коротких подпоследовательностей нам важно учитывать *errorsCount*, ведь если количество отличий превзошло дозволенное заранее заданное число *maxErrors*, то нет смысла продолжать производить сравнения текущих двух ридов и, следовательно, нужно признать их не дубликатами и перейти к следующей паре.

1. // Вычислим наименьшую из длин хэш-массивов
2. // для сравниваемых ридов
3. **int** minHashLength = Math.min(hash1.length, hash2.length);
4. **int** errorsCount = 0;
5. **for** (**int** i = 0; i < minHashLength; i++)
6. **if** (hash1[i] != hash2[i]) {
7. // Начальная и конечная позиции подпоследовательностей
8. // на ридах, соответствующих не совпавшим хэш-значениям
9. **int** start = i \* k + MIN\_IDENTICAL\_BASES;
10. **int** end = (i + 1) \* k + MIN\_IDENTICAL\_BASES;
11. **for** (**int** j = start; j < end; j++)
12. **if** (read1[j] != read2[j]) {
13. errorsCount++;
14. **if** (errorsCount > maxErrors)
15. **return** errorsCount;
16. }
17. }

*Листинг 2.5.3. Сравнение двух ридов.*

Код на Листинге 2.5.3 возвращает *errorsCount* — текущее количество несовпадений у рассматриваемых двух ридов. Если после выполнения этого кода для ридов, которые являются первыми в паре *PairedReadSequence*, *errorsCount* не превосходит *maxErrors*, то мы продолжаем и переходим к сравнению ридов, которые являются вторыми в паре, причем сравнение происходит аналогичным образом.

В итоге, мы построили алгоритм, основанный на хэш-функциях, который позволяет сократить количество посимвольных вычислений при сравнении последовательности строк и, как следствие, уменьшить время работы программы.

Заключение

В настоящей работе были изучены основные положения биоинформатики и был проведен ознакомительный анализ некоторых инструментов этой области: *FastqScreen* и *Picard Tools*.

Кроме того, был изучен алгоритм работы программы *FastqScreen*, переписан с языка *Perl* на *Java* и оптимизирован. Также, был сформулирован и построен критерий принятия решения, основанный на диаграмме размаха, который согласно какому-то наперед заданному уровню значимости, определяет, может ли полученный файл *fastq* быть использован в дальнейшем анализе.

Помимо вышесказанного, в данной работе, был улучшен алгоритм работы одного из инструментов *Picard Tools.* Изначально, утилита *EstimateLibraryComplexity* работала на посимвольном сравнении строк, что не являлось эффективным решением данной задачи. Поэтому, этот алгоритм был изменен: теперь, алгоритм сравнения строк, основывается на построении хэш-таблицы.

Список литературы

1. David W. Mount, *Bioinformatics: Sequence and Genome Analysis*, Cold Spring Harbor Laboratory Press, 2004.

2. Д. В. Ребриков, Д. О. Коростин, Е. С. Шубина, В. В. Ильинский, *NGS: высокопроизводительное секвенирование*, Бином, 2004.

3. Michael Frigge, David C. Hoaglin and Boris Iglewicz, *Opening the Box of a Boxplot*, The American Statistician 42 (4): 257–262, 1988.

4. Ben Langmead, Cole Trapnell, Mihai Pop and Steven L Salzberg, *Ultrafast and memory-efficient alignment of short DNA sequences to the human genome*, Genome Biology, 2009.

5. Alexandr Andoni, *Nearest Neighbor Search: the Old, the New, and the Impossible*, Massachusetts Institute of Technology, 2009.

6. M. Burrows and D. Wheeler*, A block sorting lossless data compression algorithm*, Technical Report 124, Digital Equipment Corporation, 1994.

1. bp (от английского base pair) - стандартное обозначение единицы измерения количества нуклеотидов. [↑](#footnote-ref-1)