Санкт-Петербургский Государственный Университет
Математико-механический факультет
Кафедра системного программирования

Библиотека алгоритмов поиска подстрок в тексте с препроцессингом в применении к биоинформатике

Курсовая работа студентки 345 группы Чижовой
Надежды Александровны

Научный руководитель: Вяххи Н.И.

Оглавление

1. Введение

Одна из основополагающих задач в биоинформатике – поиск подстроки в строке. При поиске в небольшом тексте можно использовать практически любой самый простой алгоритм, не заботясь о временных затратах. Однако, так как в этой науке чаще необходимо обрабатывать большие объемы информации, временные затраты алгоритма поиска становятся ощутимы .

На сегодняшний день существует много различных алгоритмов поиска подстроки в строке. Выбор подходящего алгоритма зависит от следующих особенностей задачи:

1. Класс подстрок, которые нам нужно найти. Оптимальный алгоритм будет сильно зависеть от того, нужно ли нам искать короткие или длинные строчки, будет ли там преобладать какие-то отдельные символы или нет, и т.д.
2. Размер алфавита.
3. Возможность проиндексировать строку, в которой ведется поиск. Поиск значительно ускорится, если такая возможность есть.
4. Требуется ли возможность поиска нескольких строк одновременно.

1.1 Постановка и актуальность задачи

В рамках данной курсовой работы рассматривается сравнение эффективности алгоритмов поиска подстроки в строке с помощью построения суфиксного массива и с помощью разбиения строки на блоки одинаковой динны. Данные алгоритмы характеризуются небольшими временными затратами и используются в биоинформатике для поиска подстроки в хромосоме (хромосома хранится, как строка символов объемом 10-250 мб). При обработке таких больших объемов данных очень важно знать, в каких случаях, какой алгоритм будет работать быстрее.

1.2 Средства разработки (Язык Java)

В качестве операционной системы был выбран Windows 7. В качестве среды разработки - IntelliJ IDEA 10. Данная среда предоставляет множество различных средств для работы с языком Java.

Java — объектно-ориентированный язык программирования, разработанный в 1995 году фирмой Sun Microsystems. Изначально язык назывался Oak (Дуб) и задумывался для управления бытовыми устройствами, например, телефонами. Но вскоре язык переименовали и, благодаря активному развитию интернета, Java стал использоваться для написания серверного программного обеспечения и клиентских приложений.

Его популярность связана с тем, что Java - интерпретируемый, а не компилируемый язык. Программы на Java транслируется в промежуточный байт-код (файлы с расширением .class). Для того, чтобы этот код мог запуститься на компьютере, на него нужно установить виртуальную Java-машину - jvm (программа, интерпретирующая байт-коды Java). Благодаря почти полной совместимости виртуальных машин с различными платформами язык Java является платформо-независимым.

Основой синтаксиса Java является синтаксис C++, однако убраны многие потенциально опасные конструкции (Java не имеет препроцессора, запрещен прямой доступ к памяти, не поддерживает множественного наследования).

С момента появления язык много раз обновлялся и его текущая версия – Java 1.6.

1.3 Сравнение с существующими исследованиями

Существует множество исследований, в которых сравниваются эффективности различных алгоритмов поиска подстроки в строке. Однако, большинство исследований проведены для более известных алгоритмов, чем таблица **k**-меров, таких как алгоритмы Бойера-Мура-Хорспула, Кнута-Морриса-Пратта, Рабина-Карпа и Ахо-Корасика.

2. Алгоритмы

Строка, в которой производиться поиск — это строка, кодирующая хромосому.

Хромосома имеет размер порядка 10^6 символов. Алфавит будет состоять из четырех символов-нуклеотидов — {A, C, G, T}. Для одной хромосомы производится поиск по большому набору подстрочек, длины примерно от 10 до 100 символов.

Исходя из таких условий было решено использовать следующие алгоритмы поиска:

- Суффиксный массив
- Суффиксное дерево
- Таблица **k**-меров
- Алгоритм Кнута-Морриса-Пратта

Здесь приведена таблица асимптотических времен работы (время поиска подстроки в строке) для этих алгоритмов в сравнении с наивным алгоритмом (Наивный алгоритм действует достаточно прямолинейно – прикладывает подстроку к строке с каждого символа, пока не получит совпадение)

Суфф. массив	Таблица **k**-меров	Суфф. дерево	КМП	Naive
O(m·logN)	**O(m·N/4^k)**	**O(m)**	**O(N)**	**O(m·N)**

m – длина искомой подстроки.

N – длина исходной строки.

Суффиксное дерево имеет наилучшую асимптотику, но требует приблизительно в 10 раз больше памяти, чем суффиксный массив, и поэтому практически не применим.

В рамках данной работы будут сравниваться первые два алгоритма. Будем рассматривать два этапа работы этих алгоритмов — препроцессинг и поиск подстроки в строке (то есть, поиск первого вхождения)

Теперь рассмотрим каждый из них более детально.

2.1 Суффиксный массив

Дана строка **T** длины **N**. Тогда i-ым суффиксом строки st будет называться ее подстрока с i-ого символа до последнего (N-ого).

Суффиксным массивом **F** строки **T** будет называться массив, элементы которого – все суффиксы строки st, отсортированные в лексикографическом порядке. Суффиксный

массив был разработан, как более экономная структура данных, с точки зрения необходимой памяти, чем суффиксное дерево.

В рамках данной курсовой рассмотрен наиболее популярный алгоритм построения суффиксного массива, требующий **O(N·logN)** времени и **O(1)** памяти, где **N** – длина исходной строки. Алгоритм построен на основе сортировки циклических сдвигов строки и содержит **O(logN)** итераций. На i-ой итерации сортируются подстроки длины 2^i. Поставив в конце строки символ, меньший любого символа в строке, сортировка циклических сдвигов превратиться в сортировку суффиксов.

Если учитывать размер алфавита – **K**, то данный алгоритм требует **O((N+K)·logN)** времени и **O(N+K)** памяти.

Так как каждая подстрока исходной строки является префиксом какого-либо суффикса, а суффиксы лесикографически упорядочены, то подстроку s исходной строки st можно искать бинарным поиском по суффиксам строки. Если само сравнение текущего суффикса и подстроки производить посимвольно, то общая асимптотика поиска подстроки в строке получается **O(|F|·logN).**

Таким образом препроцессинг суффиксного массива требует **O(N·logN)** времени, а поиск подстроки в построенном суффиксном массиве **O(m·logN)**.

2.2 Таблица k-меров

k-мер – последовательность символов, которой идентифицируется часть закодированной молекулы, такой как хромосома. **k**-меры могут быть использованы для поиска интересных участков молекул или нахождения вероятностного распределения комбинаций последовательностей символов в молекуле.

Данный алгоритм написан специально для применения в области биоинформатики и основан на том, что алфавит строки, которая кодирует хромосому, очень мал и состоит всего из четырех букв: **A**, **C**, **G**, **T**. (существуют всего четыре вида нуклеотидов, которые и обозначаются данными буквами).

Исходная строка **T** длины **N** разбивается на блоки одинаковой длины **k** (**k** фиксировано). Всевозможных блоков длины **k** - **4^k**. Затем в массив списков для каждого блока записываются все индексы вхождений в большую строчку.

Обозначим каждую букву числом: **A** = 00, **C** = 01, **G** = 10, **T** = 11. Тогда каждому блоку сопоставляется число и достаточно создать массив из **4^k** массивов, в каждом из которых хранятся начала блоков в строке, и адресовываться к нему по числу, которое сопоставляется блоку. Длины массивов считаются заранее. Первым проходом считаем количество вхождений каждого блока, а вторым записываем в массивы позиции блоков.

Алгоритм поиска подстроки **P** для этой структуры таков.

Для поиска подстроки рассматривается блок, состоящий из первых **k** символов, и для каждого вхождения данного блока в **T** подстрока **P** прикладывается к ней, начиная с этой позиции. Так как все буквы кодируются числами, то за одно сравнение двух **int** чисел сравниваются сразу 16 символов.

Асимптотика алгоритма зависит от размера **k**. Препроцессинг алгоритма требует **O(N+4^k)** времени, и **O(N+4^k)** памяти. А поиск подстроки в построенной структуре **O(m·N/4^k)**, где **m** - длина искомой строки.

В данном случае мы очень сильно ограничены в размерах затрачиваемой памяти, так как рост необходимой памяти экспоненциальный. Таким образом, число символов в одном блоке **k≤20**.

3. Решение поставленных задач

3.1 Единый интерфейс и трудности его создания

Для упрощения анализа алгоритмов и их тестирования был написан единый интерфейс для трех алгоритмов. К сожалению, решение о написании этого интерфейса было принято уже после полугода существования проекта, поэтому возникли вполне предсказуемые трудности.

Интерфейс предоставляет программисту полную функциональность работы с геномом – в нем есть следующие методы:

- **preprocess(String T) –** делает необходимый препроцессинг. На вход подается строка **T.** У алгоритма КМП этот метод не делает ничего, так как собственно препроцессинга в этом алгоритме нет.
- **MatchIterator find(String pattern, int begin, int end) –** в тексте, начиная с символа с номером begin до символа с номером end, находит все вхождения подстроки pattern, если такая существует в данном интервале, и возвращает итератор по позициям вхождения, с которых данные вхождения начинаются. Иначе возвращает -1.
- **int count(String pattern, int begin, int end) –** возвращает количество вхождений строки **pattern** в тексте, начиная с символа с номером begin до символа с номером end.

3.2 Библиотека алгоритмов

Для упрощения дальнейшей работы с алгоритмами был написан ant build script, собирающий данные алгоритмы в библиотеку. Это позволяет непосвященному пользователю пользоваться алгоритмами, не вдаваясь в их реализацию.

3.3 Система тестирования

Для анализа эффективности алгоритмов суффиксного массива и алгоритма индексации строки была написана система тестирования. Система включает в себя тесты для различных объемов исходной строки **T**, искомой строки **P**. Важно наличие строк и шаблонов с разным содержанием, поскольку несмотря на то, что суффмассив работает примерно за одно и то же время для разных строк, время работы таблицы **k**-меров сильно зависит от того насколько часто может встречаться конкретный шаблон. Например, в строке из одних символов **a** алгоритм таблицы **k**-меров будет искать строчку **aaaaa…ab** практически за такое же время, как и наивный алгоритм.

4. Результаты сравнения

4.1 Результаты тестирования

Для сравнения эффективности алгоритмов важно знать количество вхождений к исходную строку и длину подстроки, которую мы ищем. Таблица **k**-меров требуют гораздо больше памяти, чем суффиксный массив, поэтому их эффективность ограничивается размером выделенной памяти (**~4^k**). Поэтому для очень больших исходных строк, в которых символов много больше чем **420**, время работы алгоритма таблицы **k**-меров будет асимтотически таким же, как и у наивного алгоритма **~O(Nm)**.

Мы будем считать **k=12**. Это число будет константой .

Итак, было проведено несколько тестов.

Тест 1

Первый тест состоял в сравнении времени препроцессинга для строчек разной длины этих двух алгоритмов.

файл	**N**	**суфф. массив**	**таблица k-меров**
random.txt	1000000	0.106	0.143
chrY.txt	10192446	1.366	0.224
chr22.txt	34553758	5.362	0.427

Теоретическая оценка времени работы приведен на рис 1.

Тест 2

Второй тест состоял в сравнении времени поиска подстроки из случайного набора латинских букв в строке с разными длинами. Время работы считалось следующим образом: алгоритм запускался на большом количестве (~10^7) одинаковых запросов на

поиск подстроки. Затем проводилось усреднение по времени.

файл	**N**	**суфф. массив**	**таблица k-меров**
random.txt	1000000	1.1·10-3	4.8·10-6
chrY.txt	10192446	1.3·10-3	4.9·10-5
chr22.txt	24562585	1.4·10-3	1.2·10-4

m = 10

файл	**N**	**суфф. массив**	**таблица k-меров**
random.txt	1000000	3.3·10-3	1.4·10-5
chrY.txt	10192446	4.1·10-3	1.6·10-4
chr22.txt	24562585	4.3·10-3	3.6·10-4

m = 30

файл	**N**	**суфф. массив**	**таблица k-меров**
random.txt	1000000	7.8·10-3	3.3·10-5
chrY.txt	10192446	9.1·10-3	3.4·10-4
chr22.txt	24562585	9.7·10-3	8.3·10-4

m = 70

файл	**N**	**суфф. массив**	**таблица k-меров**
random.txt	1000000	1·10-2	4.8·10-4
chrY.txt	10192446	1.3·10-2	4.9·10-4
chr22.txt	24562585	1.4·10-2	1.1·10-3

m = 100

Теоретическая оценка времени работы приведен на рис 2 и рис 3.

4.2 Графики

В этом разделе представлены графики теоретических зависимостей времен работы от **N** и **m**.

График времени препроцессинга в зависимости от длины строки **T**:

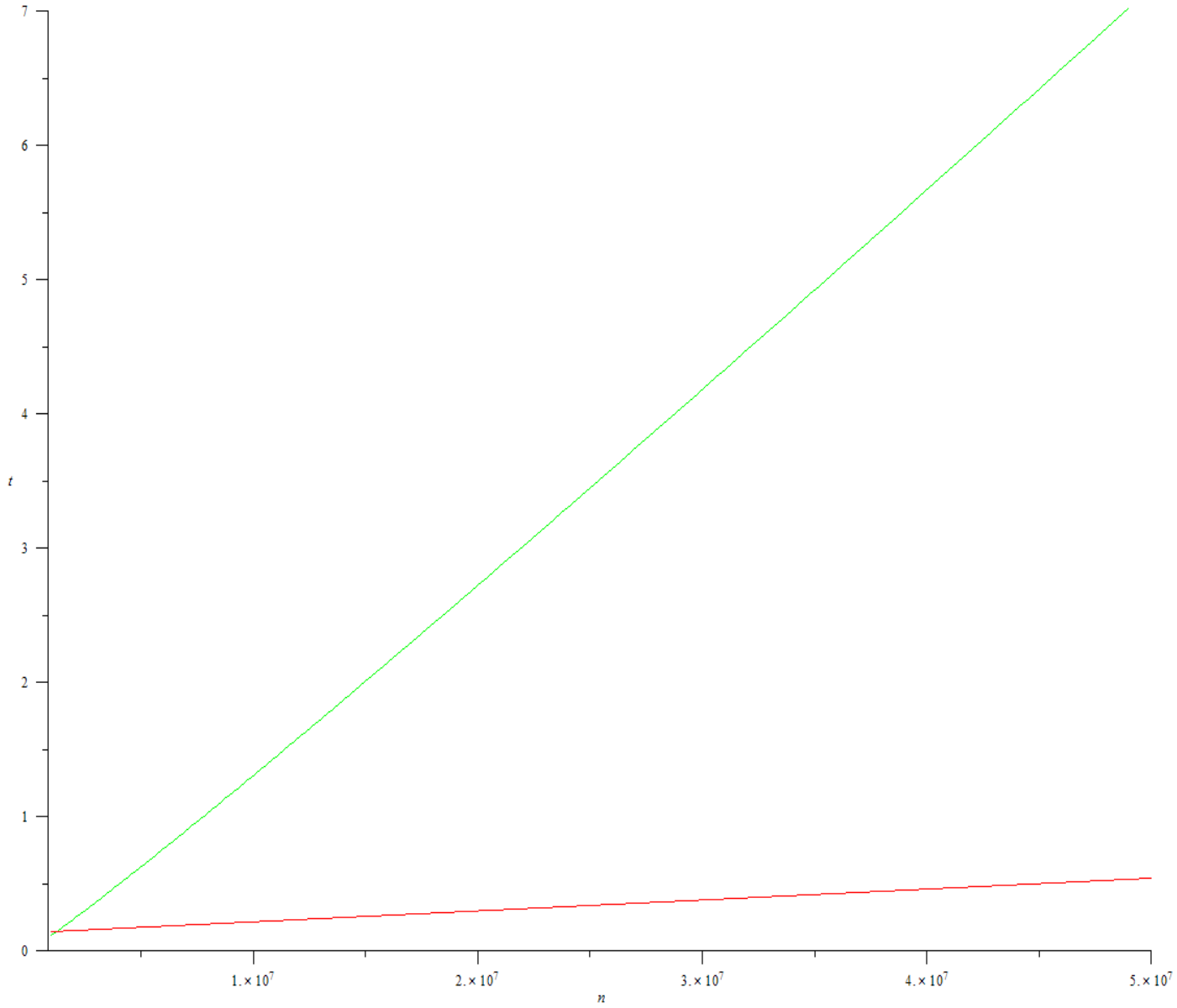

Рис.1 (красный – таблица k-меров, зеленый – суфф. массив)

Время поиска для различных длин строки **T** (время работы от **N**):

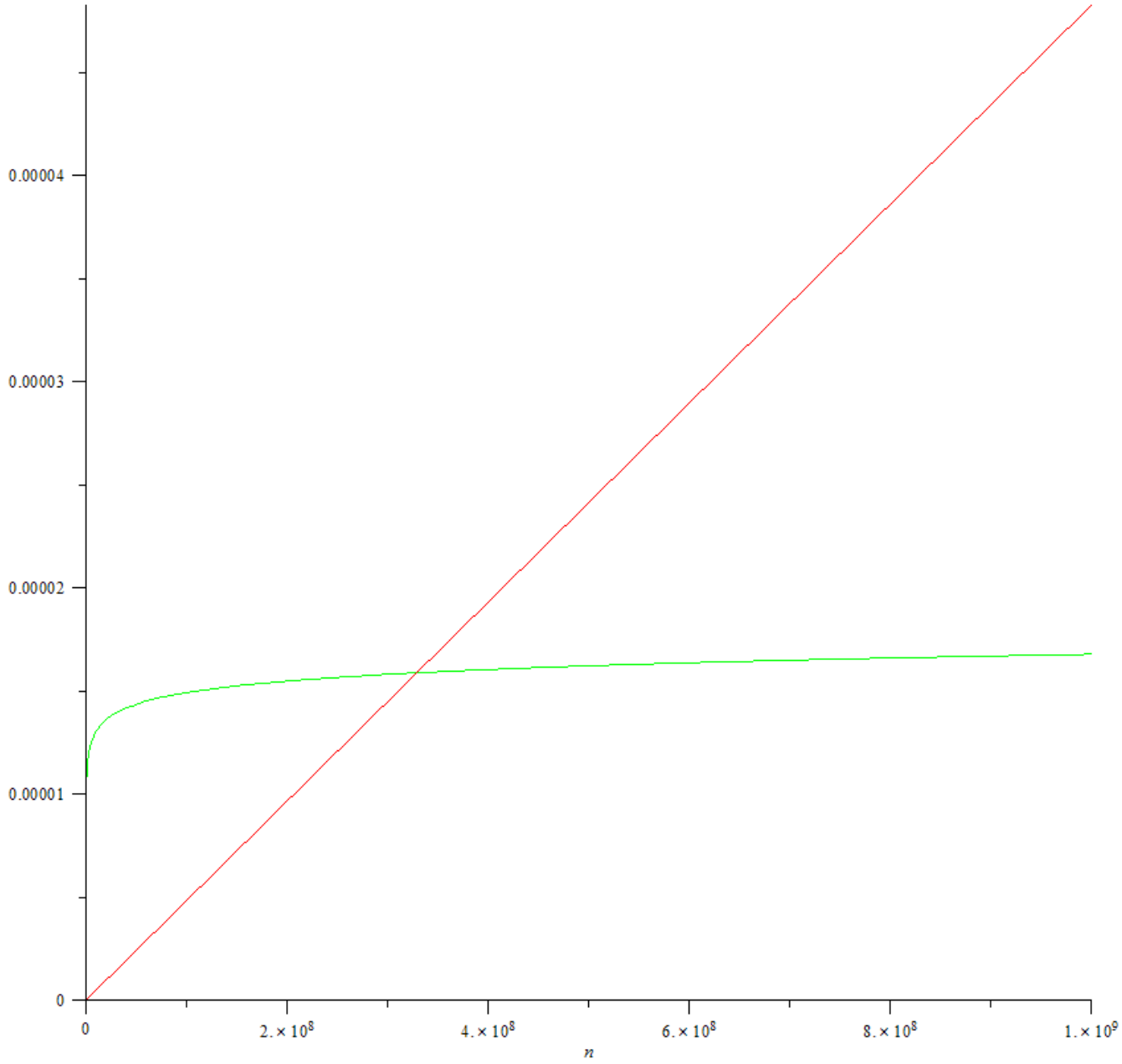

Рис.2 (красный – таблица k-меров, зеленый – суфф. массив)

Время поиска для различных длин искомых подстрок (время работы от **m**):

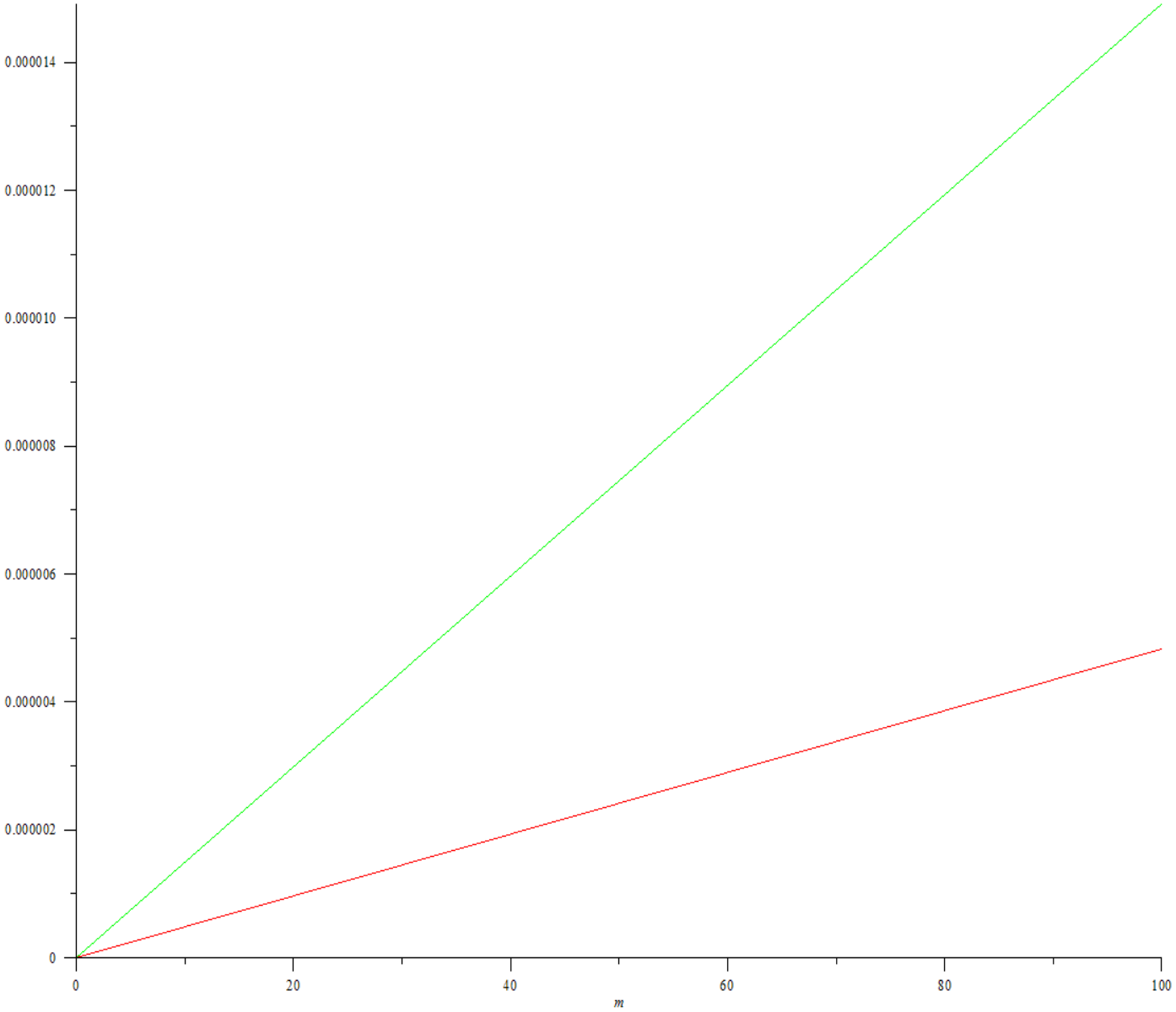

Рис.3 (красный – таблица k-меров, зеленый – суфф. массив)

5.Заключение

Вообще говоря, стоит учитывать, что в рамках курсовой рассматриваются алгоритмы поиска подстроки в строке-хромосоме, в которой могут быть разрывы – не расшифрованные части. При поиске такие части выкидываются. Это процесс требует O(logN) времени, но мы его не учитываем, так как он есть в обоих алгоритмах.

Эффективность алгоритмов зависят от количества ответов на запрос, т.е от частоты, с которой встречается искомая подстрока в тексте. Средняя частота для алгориитма таблицы **k**-меров – **N/4^k**, для суффиксного массива – **logN**. Для **k** = 12 : **N/ 4^k = N/16777216.**

Видно, что **logN> N/16777216** для всех реальных случаев поиска подстроки в хромосоме (хромосома – файл до 250мб).

В результате исследования мы получили, что таблица **k**-меров достаточно быстрый по времени алгоритм. Однако, стоит учитывать, что для больших **k** данная структура требует на **4^k** больше памяти, чем суффиксный массив.

Как алгоритм таблицы **k**-меров, так и суффиксный массив не очень эффективны в случае, когда поиск выдает большой результат, т.е когда искомая строка часто встречается в исходном тексте.

5.1 Возможности для развития

В дальнейшем планируется:

1. Исследование других алгоритмов поиска.
2. Реализация более полной системы тестирования (аудио файлы, видео, графика, exe и т.д).
3. Создание балансировщика.

Список литературы

1. Кормен, Лейзерсон, Ривест, Штайн. **Алгоритмы. Построение и анализ (2-е издание.)**
2. Пападимитриу, Стайглиц. **Комбинаторная оптимизация: алгоритмы и сложность.**
3. Хорстманн, Корнелл. **Java 2. Библиотека профессионала. Том 1 (Основы) (7-е изд.)**
4. Кнут. **Искусство программирования. Том 1**
5. Майкл Т. Гудрич, Роберто Тамассия. **Структуры данных и алгоритмы в Java** .
6. Daniel M. Gusfield. **Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, 1997.**
7. Neil C. Jones, Pavel A. Pevzner. **An Introduction to Bioinformatics Algorithms. MIT Press. 2004**.