**Министерство образования и науки Российской Федерации**

**Санкт-Петербургский государственный университет**

**информационных технологий, механики и оптики**

**Факультет** Информационных Технологий и Программирования
**Направление (специальность)** Прикладная математика и информатика
**Квалификация (степень)** Бакалавр прикладной математики и информатики
**Специализация**
**Кафедра** Компьютерных технологий **Группа** 4539

# ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

## К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

Оптимальный поиск в длинных строках
с использованием внешней памяти

**Автор квалификационной работы** Козлов В.А. (подпись)

**Руководитель** Станкевич А.С. (подпись)

Санкт-Петербург
2007

# СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

Часто встречаются задачи поиска подстроки в строке. В основном эта область считается довольно хорошо изученной – имеется множество алгоритмов, разработок и т.п. Однако в основном все это относится к строкам небольшой длины. Допустим, что требуется осуществить поиск в очень длинной строке, настолько длинной, что только она одна помещается в оперативной памяти и больше никакой дополнительной информации, сравнимой с длиной строки, в памяти не уместить. Возможно также, что исходная строка не умещается в оперативной памяти. Для того, чтобы представить откуда может взяться такая строка, не обязательно ориентироваться на базу какой-либо крупной поисковой системы в Интернете. Достаточно посчитать какова будет длина строки, если написать через запятую имена всех файлов вашего компьютера.

Поиск файлов на компьютере обычно осуществляется долго, потому что список всех файлов, как правило, нельзя удержать в оперативной памяти, и операционная система не может построить какую-нибудь классическую структуру данных в этой памяти для быстрого поиска. Практически все алгоритмы такого поиска предполагают построения индекса, длина которого намного больше длины индексируемой строки.

В том случае, когда осуществляется поиск в очень длинных строках, то практически все алгоритмы быстрого поиска становятся неприменимы из-за требования большого количества дополнительной памяти. При этом применимы только те из них, которые не требовательны к памяти, но зато они медленнее. Как правило, в таких случаях используют алгоритм Кнута-Морриса-Пратта [1], так как из всех алгоритмов без требования наличия дополнительной памяти, он считается оптимальным по сложности реализации и теоретической скорости поиска за линейное время.

Современные компьютеры обычно не поддерживают объём оперативной памяти больше 4Гб.

Размер жестких дисков на несколько порядков больше. На момент написания работы обычным пользователям доступны жесткие на 500Гб, к тому же в компьютер можно ставить несколько жестких дисков вместе.

В настоящей работе ставится задача использовать жесткий диск вместо оперативной памяти для хранения индекса, построенного для длинной строки.

При этом в качестве базисного выбран алгоритм быстрого поиска Укконена, у которого скорость выше линейной. Этот алгоритм адаптируется на случай использования жесткого диска, вместо оперативной памяти, при её нехватке.

# ГЛАВА 1. ОБЗОР ИЗВЕСТНЫХ МЕТОДОВ

В описании будем называть строку, в которой осуществляется поиск – текстом $T$, и будем считать её длину равной $n$. Обозначим искомую строку (образец) – $P$, а его длину – $m$.

*Суффикс* – подстрока $S[i..|S|]$, а *префикс* – подстрока $S[1..i]$, $(1 \le i \le |S|)$. Через $S(i)$ обозначается $i$-ый символ строки.

## *1.1. Наивный метод*

### Описание

Это простой метод, который, в первую очередь, приходит на ум любому человеку, даже не знакомому с программированием, когда заходит речь о поиске подстроки.

В методе левый конец образца $P$ ставится вровень с левым концом теста $T$, и все символы образца сравниваются с соответствующими символами текста, пока не встретится несовпадение, либо не исчерпаются все символы образца. В последнем случае констатируется вхождение. Независимо от результата, образец сдвигается на один символ вправо, и сравнение возобновляется. Так действуем до тех пор, пока правый край образца не достигнет правого края текста.

### Оценки

В самом худшем случае число сравнений выполняемых этим методом равно $O(nm)$. На практике указанный метод самый медленный из всех рассматриваемых, но зато ему требуется $O(1)$ дополнительной памяти, и он очень прост в реализации.

### *1.2. Алгоритм Бойера-Мура*

**Описание**

Как и наивный метод, алгоритм Бойера-Мура последовательно прикладывает образец $P$ к тексту $T$ и проверяет совпадение символов. После сравнения образец сдвигается право и сравнение повторяется. Сдвиг происходит пока не закончится текст. Отличия алгоритма Бойера-Мура от наивного, состоит в том, что сравнение осуществляется не слева направо, а справа налево и подстрока может сдвигаться вправо больше чем на один символ.

Сдвиг на несколько символов обычно называют правилом плохого суффикса. Это правило состоит в следующем: если в процессе сравнения символ текста – «$x$» не совпал с символом образца, то требуется сдвинуть образец таким образом, чтобы в этой позиции оказался символ образца «$x$», который самый близкий слева от текущей позиции несовпадения. Если же символ «$x$» в образце левее уже не встречается, то следует сдвинуть образец за эту позицию несовпадения.

Это правило для большого алфавита позволяет делать достаточно большие сдвиги образца и в результате просматривать не весь текст. В случае очень маленького алфавита можно применить ещё одну оптимизацию – правило хорошего суффикса. О ней можно прочитать в книге [1].

**Оценки**

В реальных примерах и достаточно больших алфавитах этот метод работает быстро, и его время быстрее линейного за счет просмотра не всех символов. Однако в самом худшем случае, теоретическое время работы все равно остается $O(nm)$. Дополнительная память требуется для предварительной обработки образца. Правилу плохого суффикса требуется знать для каждого символа, где он встречается левее. Обычно это реализуется матрицей размером $O(n \cdot |\sum|)$, где $\sum$ - алфавит.

### *1.3. Алгоритм Кнута-Морриса-Пратта*

**Описание**

Самый известный алгоритм поиска за линейное время был предложен Кнутом, Моррисом и Праттом. Основа этого алгоритма также заключается в прикладывании образца к тексту с дальнейшим сдвигом. В отличие от наивного метода, алгоритм Кнута-Морриса-Пратта извлекает полезную информацию из тех символов, которые он уже сравнил, и символы текста, которые уже сравнивались с образцом, в дальнейших сравнениях не участвуют. Это обеспечивает просмотр каждого символа текста один раз, что обеспечивает линейное по длине текста время работы.

Для алгоритма требуется следующий препроцессинг образца. Для каждой позиции $i$ образца $P$ определим $sp(i) = sp_i$, как длину наибольшего суффикса $P[1..i]$, который совпадает с префиксом $P$. Иначе говоря, $sp_i$ – это длина наибольшей строки, такой, что с неё начинается подстрока $P[1..i]$, и ей же заканчивается. Функция $sp_1$ считается равным нулю. После этого последовательно вычисляются $sp_i$ ($i$ от 2 до $n$) через уже вычисленные $sp_1$ .. $sp_{i-1}$. Допустим требуется определить $sp_{k+1}$ и $P(k+1) = x$. Сравним символы $P(sp_k+1)$ и $x$. Если они равны, то $sp_{k+1} = sp_k+1$. Иначе сравниваем $P(sp(sp(k))+1)$ и $x$. Если они равны, то $sp_{k+1} = sp(sp(k))+1$. И так по рекурсии. Если дошли до начала образца, то $sp_{k+1} = 0$. Доказательство правильности такого нахождения $sp_i$ можно прочитать в книге [1].

Правило сдвига использует вычисленные значения функции $sp_i$. Допустим, что в процессе сравнения (слева направо) совпали $i$ символов образца, а несовпадение в позиции $i+1$ образца и позиции $k$ текста. Тогда образец необходимо сдвинуть вправо таким образом, чтобы $P[1..sp_i]$ был пристроен к $T[k-sp_i..k-1]$. Другими словами, образец $P$ сдвинется вправо на $i-sp_i$ мест. Тем самым, уже известно, что совпадут первые $sp_i$ символов образца, и их сравнивать уже нет смысла, а необходимо продолжать сравнения с позиции $sp_i+1$ образца и позиции $k$ текста.

**Оценки**

На каждом шаге происходит либо переход к следующему символу текста, либо сдвиг образца. Тем самым, число операций равно $m + s$ ($s$ – число сдвигов, но сдвигов меньше $m$). Следовательно, время работы – $O(m)$, и требуется $O(n)$ дополнительной памяти для хранения $sp_i$.

## *1.4. Обзор суффиксных деревьев*

**Описание**

*Суффиксное дерево T* для $m$-символьной строки $S$ – это ориентированное дерево с корнем, имеющее ровно $m$ листьев, занумерованных от 1 до $m$. Каждая внутренняя вершина, отличная от корня, имеет не менее двух детей, а каждая дуга помечена не пустой подстрокой $S$ (*дуговой меткой*). Никакие две дуги, выходящие из одной вершины, не могут иметь пометок, начинающихся с одного и того же символа. Главная особенность суффиксного дерева заключается в том, что для каждого листа $i$ конкатенация меток дуг на пути от корня к листу $i$ в точности составляет суффикс строки $S$, который начинается в позиции $i$ – подстроку $S$ [$i..m$]. Предполагается, что алфавит конечен.

Например, суффиксное дерево для строки $S$ = *xabxac* изображено на рис. 1.1.

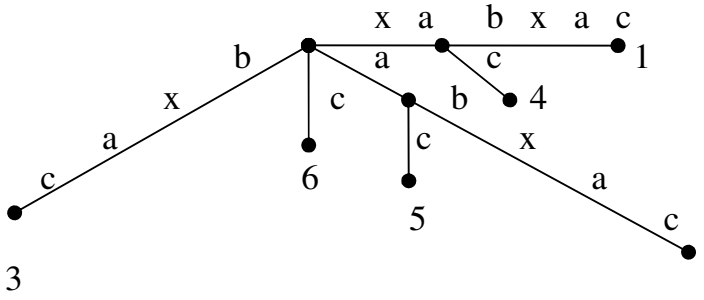


Рис. 1.1. Суффиксное дерево для строки *xabxac*

Определение суффиксного дерева не гарантирует, что оно действительно существует для любой строки $S$. Трудность состоит в том, что если один суффикс совпадает с префиксом другого суффикса, то построить суффиксное дерево,

удовлетворяющее данному выше определению, невозможно, поскольку путь для первого суффикса не сможет закончиться в листе. Обычно во избежание этой трудности предполагается, что последний символ строки $S$ больше нигде не встречается в $S$. Это условие гарантирует возможность построения суффиксного дерева. На практике выполнение указанного условия обеспечивается добавлением в конец строки символа, который не входит в основной алфавит для строки $S$. Далее, по умолчанию, будет предполагаться выполнение этого условия.

Суффиксное дерево – это структура данных, которая глубоко выявляет внутреннее строение текста. Классическим приложением для суффиксных деревьев является задача о поиске подстроки, хотя с помощью суффиксных деревьев эффективно решается много других типов задач: поиск неточного совпадения, общий наименьший предшественник и другие. Описание этих алгоритмов можно прочитать в книге [1].

Пусть дан текст $T$ длиной $m$ символов. Алгоритм поиска, использующий суффиксное дерево, готовится за препроцессионное время $O(m)$ к тому, что, получив неизвестную строку $S$ длины $n$, он должен найти вхождение $S$ в $T$ или сообщить об отсутствии вхождения за время $O(n)$. Это означает, что после препроцессинга, поиск будет осуществляться за время, которое не завит от длины текста.

Алгоритм поиска с использованием суффиксного дерева является самым быстрым алгоритмом поиска для задач, в том случае, когда приходит множество запросов на поиск в заранее известном и неизменяющемся тексте. Однако для этого требуется затратить много препроцессионного времени, хотя оно и линейно по длине текста, но константа велика. Для хранения суффиксного дерева требуется много дополнительной памяти, хотя требуемая память и линейна по длине текста, но опять же достаточно большая константа. Это делает невозможным применение суффиксных деревьев в классических реализациях для очень длинных текстов, так как обычно компьютеры не располагают большим

количеством оперативной памяти. Ниже будет приведено краткое описание одного из таких классических алгоритмов построения.

Допустим, что для текста уже построено суффиксное дерево *T*. Для поиска образца P, будем искать совпадения для символов из *P* вдоль единственного пути в *T* до тех пор, пока либо *P* не исчерпается, либо очередное совпадение не станет невозможным. Во втором случае *P* не входит в текст. В первом случае каждый лист в поддереве, который идет из точки последнего совпадения, имеет своим номером начальную позицию вхождения *P* в текст, а каждая позиция вхождения *P* в текст нумерует такой лист. Этот совпадающий путь единственен, так как никакие две дуги, выходящие из одной вершины, не имеют меток, начинающихся с одного и того же символа. Поскольку было сделано предположение, что алфавит конечен, то работа в каждой вершине занимает константное время, и следовательно, время на проверку совпадения *P* с путем пропорционально длине *P*.

**Оценки**

В итоге, поиск осуществляется за $O(n+k)$, где $k$ – число вхождений подстроки в строку. Ниже будет приведено описание того, как простроить суффиксное дерево за время $O(m)$.

Число листьев – ровно $m$, а число внутренних узлов меньше $m$, так как у каждой внутренней вершины не меньше двух детей. Для хранения каждого листа и каждого внутреннего узла требуется константный объем памяти, следовательно, общая дополнительная память, которая требуется для хранения построенного суффиксного дерева, линейна по длине текста – $O(m)$.

### *1.5. Построение суффиксного дерева методом Укконена*

Впервые метод построения суффиксного дерева за линейное время был предложен Вайнером. Однако этот метод сложнее в реализации и в доказательстве, чем метод Укконена, а также требует больше дополнительной

памяти. Поэтому рассмотрим построение суффиксного дерева за линейное время методом Укконена. Для того чтобы объяснить алгоритм построения этого дерева за линейное время, сначала рассмотрим примитивное построение за $O(n^3)$, а потом применим ряд оптимизаций, которые сведут время построения к линейному.

Будем строить дерево постепенно. Построение будет состоять из $m$ фаз. В каждой *фазе* $i$ происходит добавление к строящемуся дереву всех суффиксов подстроки $S[1..i]$. Таким образом, на $i$-ой фазе к дереву последовательно будут добавлены подстроки $S[1..i]$, $S[2..i]$, $S[3..i]$, … , $S[i\text{-}1..i]$, $S[i]$ – будут выполнены $i$ *продолжений*. При завершении $m$-ой фазы получится искомое суффиксное дерево. Каждая из $m$ фаз добавляет $O(m)$ подстрок, добавление каждой из них происходит «методом из корня» – за $O(m)$. Итого этот метод работает за время $O(n^3)$.

Формально опишем правила продолжения суффикса. Пусть фаза добавления – $i$, а продолжение $j$. При этом алгоритм должен обеспечить существование в дереве подстроки $S[j..i]=\beta S(i)$. Заметим, что $\beta$ уже существует в дереве, так как была добавлена в предыдущей фазе. Когда алгоритм находит конец $\beta$, он действует по следующим правилам:

1. $\beta$ кончается в листе. Тогда следует просто добавить символ $S(i)$ к дуговой метке в конец.
2. Ни один путь из конца $\beta$ не начинается с символа $S(i)$, но, по крайней мере, один начинающийся оттуда путь имеется. Следует добавить новый лист с номером $j$, а его листовая метка будет состоять из одного символа $S(i)$. Если $\beta$ кончалось внутри дуги, то будет создана новая внутренняя вершина.
3. Существует путь из конца $\beta$, который начинается с символа $S(i)$. Ничего делать не требуется.

**Ускорение 1.** Для того чтобы искать конец $\beta$ не за $O(|\beta|)$, а за $O(1)$ вводятся *суффиксные связи*. Суффиксной связью называется указатель из конца строки $x\alpha$ в конец строки $\alpha$, где $x$ – любой символ, $\alpha$ – любая строка, возможно, пустая. При

добавлении каждой новой внутренней вершины, алгоритм должен проставлять суффиксную связь в эту вершину из вершины, добавленной в предыдущем продолжении. Тогда можно доказать, что в любой момент в дереве из каждой вершины (кроме только что добавленной) выходит правильная суффиксная связь.

Таким образом, зная конец в продолжении $j$, можно быстро найти конец для продолжения $j$+1 следующим способом. Из конца $\beta=x\gamma$ поднимаемся к его родителю и запоминаем длину пути между $\beta$ и его родителем. При этом оказываемся в вершине, которая является концом некоторой строки $x\alpha$. Переходим по суффиксной связи в вершину, которая будет являться концом строки $\alpha$. Если родителем оказался корень, то не переходим по суффиксной связи, а уменьшаем длину спуска на единицу.

Спускаемся вниз, для того чтобы попасть в вершину с концом строки $\gamma$. Заметим, что $\gamma$ существует в дереве, так как была добавлена в предыдущей фазе. Следовательно, спуск из $\alpha$ в $\gamma$ следует осуществлять только лишь на длину $|\gamma|-|\alpha|$ равную длине пути, который был пройден при подъеме к родителю. Таким образом, спуск будет осуществлен, пройдя число вершин на пути спуска, а не за длину спуска. Все переходы вниз за одну фазу будут меньше или равны $O(m)$.

В итоге получили, что суффиксные связи обеспечивают время работы $O(n^2)$.

Отметим, что на дугах хранятся не строки, а два числа – начало и конец подстроки в исходном тексте. Эта подстрока и является дуговой меткой.

**Ускорение 2.** Можно заметить две особенности. Во-первых, если на какой-то фазе текущее продолжение было выполнено по правилу 3, то все остальные продолжения до конца фазы будут выполняться по этому правилу. Следовательно, их выполнять не требуется. Во-вторых, если какой-то узел стал листом, то он листом и останется. Следовательно, правый конец всех дуг соединяющих листья с их родителями требуется пометить ссылкой на текущее $i$, а правило 1 не требуется выполнять явно. При этом оно будет выполняться не явно:

при переходе к следующей фазе $i$ будет увеличено на единицу, и, следовательно, увеличатся значения правой границы для всех листьевых дуг.

На основе изложенного, может быть предложен следующий алгоритм:

0. $i = 0, j = 1$.
1. Перейти к следующей фазе (увеличить $i$).
2. Продолжать суффиксы, начиная с текущего продолжения $j$, до тех пор, пока не будет применено правило продолжения 3 или не закончится фаза.
3. Если ещё не конец текста, то перейти к пункту 1.

В итоге на каждой итерации либо увеличивается текущее $j$, либо увеличивается текущее $i$. Всего $O(m)$ итераций. Все продолжения из-за использования суффиксных связей в сумме выполнятся за время $O(m)$.

В результате получили алгоритм построения суффиксного дерева за $O(m)$.

Сделаем замечание относительно хранения суффиксного дерева. Каждая внутренняя вершина хранит два числа – начало и конец подстроки, которой помечена дуга, выходящая к родителю из этой вершины. Вершина хранит также ссылку на родителя, первого ребенка, правого брата и суффиксную связь. Так как листьев ровно $m$, то хранить их будем в массиве $[1..m]$. Номер листа – это его индекс в массиве, а значение в массиве для каждого листа – это ссылка на правого брата (лист или узел). Как же определить дуговую метку, которая идет от листа к родителю? Раз это лист, следовательно, это суффикс, а значит, дуговая метка заканчивается в конце текста (или конце текущей фазы). Начало дуговой метки можно определить, если взять номер листа (это позиция начала суффикса) и прибавить к ней глубину пути от корня до родителя листа.

В итоге каждый узел хранит шесть чисел, а каждый лист – одно число. Внутренних вершин меньше $m$, но в худшем случае $O(m)$. В итоге, например, для случая 32-битной адресации для хранения построенного суффиксного дерева потребуется $28m$ памяти.

# ГЛАВА 2. ПОСТРОЕНИЕ СУФФИКСНОГО ДЕРЕВА С ИСПОЛЬЗОВАНИЕМ ВНЕШНЕЙ ПАМЯТИ

Реализуем классический алгоритм построения суффиксного дерева методом Укконена, который описан в теоретическом введении и постараемся найти все узкие места и оптимизировать построение.

Если просто вместо оперативной памяти использовать внешнюю память, то скорость построения сильно падает. Скорость случайного доступа к оперативной памяти примерно в 100 раз больше, чем скорость случайного доступа к жесткому диску. Поэтому скорость построения суффиксного дерева получается на два порядка ниже, что подтверждается практически.

Как уже говорилось, будем расходовать на хранение каждого узла по четыре байта для ссылки на родителя, правого брата, первого ребенка, суффиксную связь, начало и конец текста на ребре. Это 24 байта на каждый узел. И будем расходовать по одному числу – четыре байта на каждый лист для хранения правого брата, а номер листа – это его номер в массиве.

Ссылки это числа. Поэтому будем пользоваться следующими правилами для адресации узлов и листьев. Листьев всегда ровно $m$ – длина текста. Поэтому ссылки от 1 до $m$ будут ссылками на соответствующие листья. Узлы добавляются постепенно в процессе построения суффиксного дерева. Поэтому их будем нумеровать числами, следующими за $m$. Таким образом, корень дерева будет иметь номер $m$+1, первый добавленный узел – номер $m$+2 и т.д.

На графиках в дальнейшем будет предполагаться, что горизонтальная ось – $X$, вертикальная ось – $F$. На каждом графике указаны два числа: $Fmax$ – максимальная высота функции и $Xmax$ – правая граница диапазона, представленного на графике. Минимальное значение функции и левая граница диапазона равны нулю: $Fmin$=0, $Xmin$=0.

### *2.1. Оптимизация «чтения»*

Заметим, что из исходного текста для алгоритма всегда требуются один следующий символ – исходный текст читается последовательно. Тогда если он первоначально будет находиться в файле на жестком диске, то простая оптимизация по буферизированному чтению пакетами увеличит скорость его считывания до максимальной, так как эта самая быстрая операция для жесткого диска. Это будет *первая оптимизация*.

### *2.2. Оптимизация «назад»*

Подобным образом можно оптимизировать запись новых узлов. Новый появившийся узел будем не сразу записывать во внешнюю память, а предварительно – записывать его в кэш. Все операции с этим узлом будут производиться в кэше до тех пор, пока кэш не будет сброшен на винчестер.

Оказывается, что эта оптимизация дает не только ускорение записи. Построим графики (рис. 2.1, 2.2) зависимости для числа чтений/записей узлов от расстояния до последнего добавленного узла – от возраста узла. Другими словами проанализируем частоту использования узлов в зависимости от того, на сколько давно эти узлы были добавлены. Построим статистику для текста длиной пять мегабайт. Полученное суффиксное дерево будет иметь около 2,7 миллионов узлов.

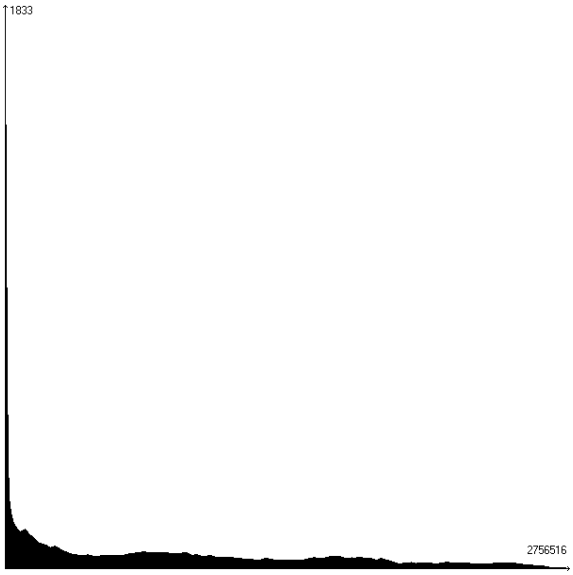


Рис. 2.1.
Зависимость числа чтений от возраста узла

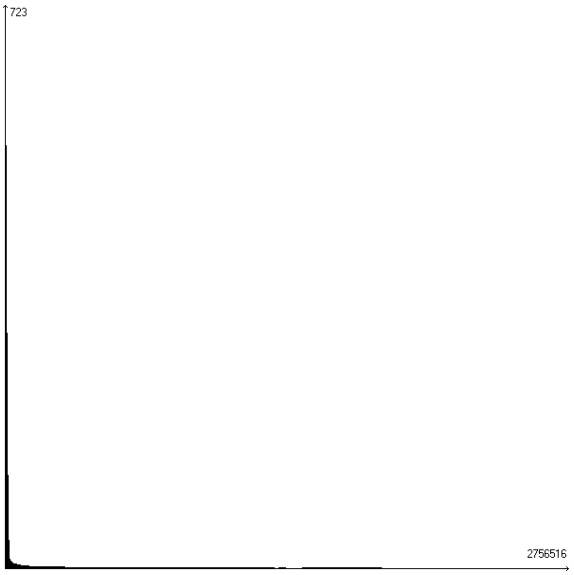


Рис. 2.2.
Зависимость числа записей от возраста узла

Из этих графиков можно сделать вывод, что большое число чтений и записей производится над недавно созданными узлами. При этом, чем старше узел, тем меньше к нему производится обращений на чтении и практически не производится записей. Поэтому применение такого кэша обязательно.

Так как узлы, созданные недавно, самые популярные, то они всегда должны быть доступны в кэше. В том случае, когда кэш заполнится, будем сбрасывать его на винчестер не полностью, а только его более старую половину. Таким образом, доступ к самым популярным узлам будет всегда в оперативной памяти.

Примерно такие же графики получаются при исследовании чтения и записей листьев. Поэтому точно так же, как и с узлами, будем поступать с листьями: перед тем, как записать на диск. Новые листья некоторое время должны пролежать в кэше, а потом должны записываться на диск по половине кэша за один раз. Это будет *вторая оптимизация*.

### *2.3. Оптимизация «вперед»*

Построим графики (рис.2.3 и 2.4) зависимости числа чтений/записей по порядку их создания.

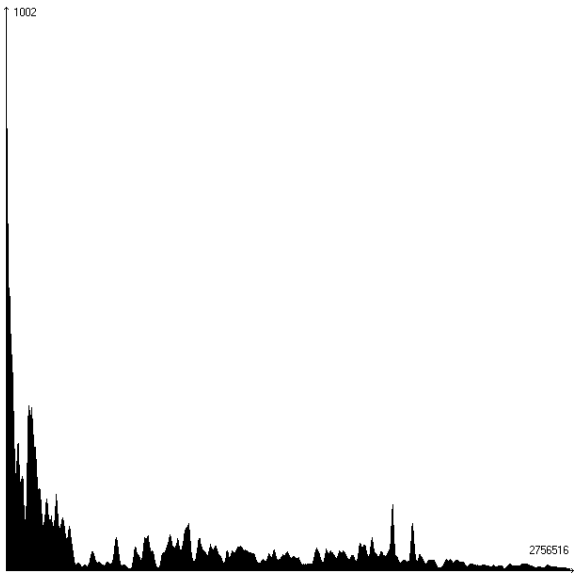


Рис. 2.3.
Зависимость числа чтений от номера узла в прядке добавления

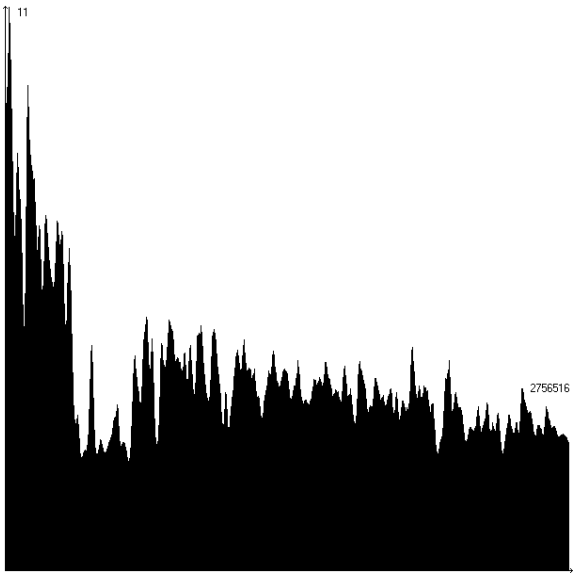


Рис. 2.4.
Зависимость числа записей от номера узла в порядке добавления

Из этих графиков видно, что первые созданные узлы имеют большую популярность, чем узлы созданные далее. Причем для узлов, которые были созданы позднее, распределение частот достаточно равномерно. Введём второй кэш, в котором будем хранить узлы, созданные первыми. Подобные графики получаются при исследовании чтения и записей листьев. Для них применим аналогичный кэш. Все это будет *третья оптимизация*.

Обратим внимание на значение максимальной частоты чтения и записи. Максимальное число чтений в 100 раз больше максимального числа записей. Среднее число записей на узел примерно равно 10, а среднее число чтений для узла на много больше. Такое «превосходство» чтения над записью будет далее везде. Намного важнее оптимизировать чтение, а запись, как правило, распределена почти равномерно. Для всех рассматриваемых зависимостей, если где и наблюдается увеличение частоты записи, то в этом месте также будет наблюдаться повышенная частота чтений. Все дальнейшие зависимости будем рассматривать и оптимизировать только для частоты чтения. При этом число

записей все равно намного меньше числа чтений и, как правило, повторяет характеристики чтения.

### *2.4. Другие оптимизации*

**Оптимизация в глубину.** Рассмотрим зависимость частоты чтения от глубины, на которой окажется узел в конце построения суффиксного дерева. Рассмотрим глубину до 30, так как из графиков будет следовать, что глубже уже точно нет ничего существенного. На рис. 2.5 представлен график числа узлов в зависимости от глубины. Видно, что больше всего узлов на пятом–седьмом уровнях глубины. Суммарное число чтений на каждом уровне глубины показано на рис. 2.6. Если поделим число чтений на число узлов уровня, то получим среднее число чтений для узла. Эта зависимость показана на рис. 2.7.

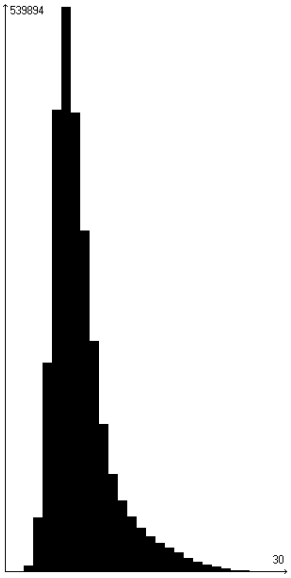


Рис. 2.5.
Зависимость числа узлов от глубины


Рис. 2.6.
Зависимость суммарного числа чтений узлов от глубины

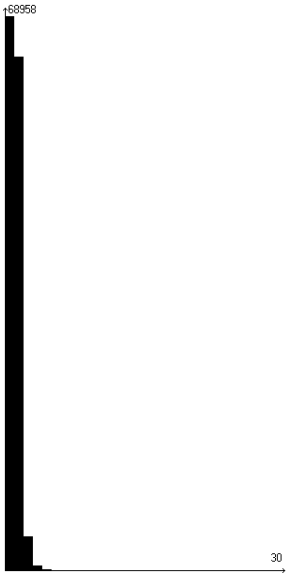


Рис. 2.7.
Зависимость средней частоты чтений одного узла от глубины

Из графика видно, что на первых трех уровнях очень активное обращение к узлам, но, к сожалению, на первых трех уровнях очень мало узлов, а на тех уровнях, где сосредоточена основная часть узлов – средняя частота чтения низкая.

Если построить характеристику по частоте чтения в зависимости от глубины, на которой производится операция чтения, то в результате получаются схожие графики, из которых следует такой же вывод. Для данного параметра оптимизацию применить не удается.

**Оптимизация по числу.** Рассмотрим другой тип зависимости. На рис. 2.8 показана зависимость числа узлов от числа чтений в узле. График имеет форму гиперболы, очень сильно прижатой к осям (на рисунке ее практически не видно). Из графика видно, что подавляющая часть узлов читается не часто, а очень частое обращение производится к не большому числу узлов.

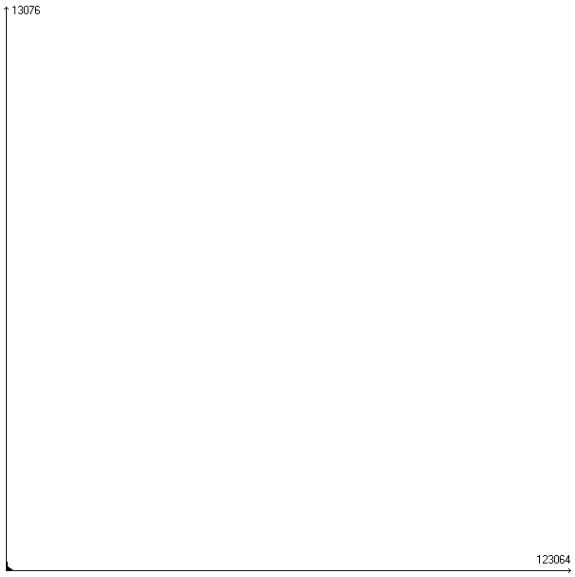


Рис. 2.8.
Зависимость числа узлов от частоты чтения

По этому графику можно найти статистику о том, что 95% чтений производится над 40% вершин. Притом это в самом лучшем случае, если начинать суммирование с часто используемых вершин. Другими словами, основная часть чтений производится над слишком большим числом узлов, и даже если пытаться кэшировать какие-то очень часто использующиеся узлы, то это ничего не даст, так как не удается кэшировать 40% узлов.

Максимальная частота чтения узла для тестируемого текста, как видно из графика, равна 123064. Однако, если найти сумму числа чтений всех узлов,

которые читаются больше 2000 раз, то получится всего два процента от общего числа чтений. Это дает основание предполагать, что вторая и третья оптимизации достаточно хороши.

### *2.5. Сравнение различных типов текстов*

**Сравнение по длине.** Приведенные выше графики построены для текста длиной в пять мегабайт. Построим аналогичные графики с частотой чтения для текста длиной в два раза меньше  и в два раза больше (рис. 2.9).

Отметим интересный факт, что в текстах большой длины, даже в различных типах, как правило, число узлов примерно в двое меньше, чем длина текста.

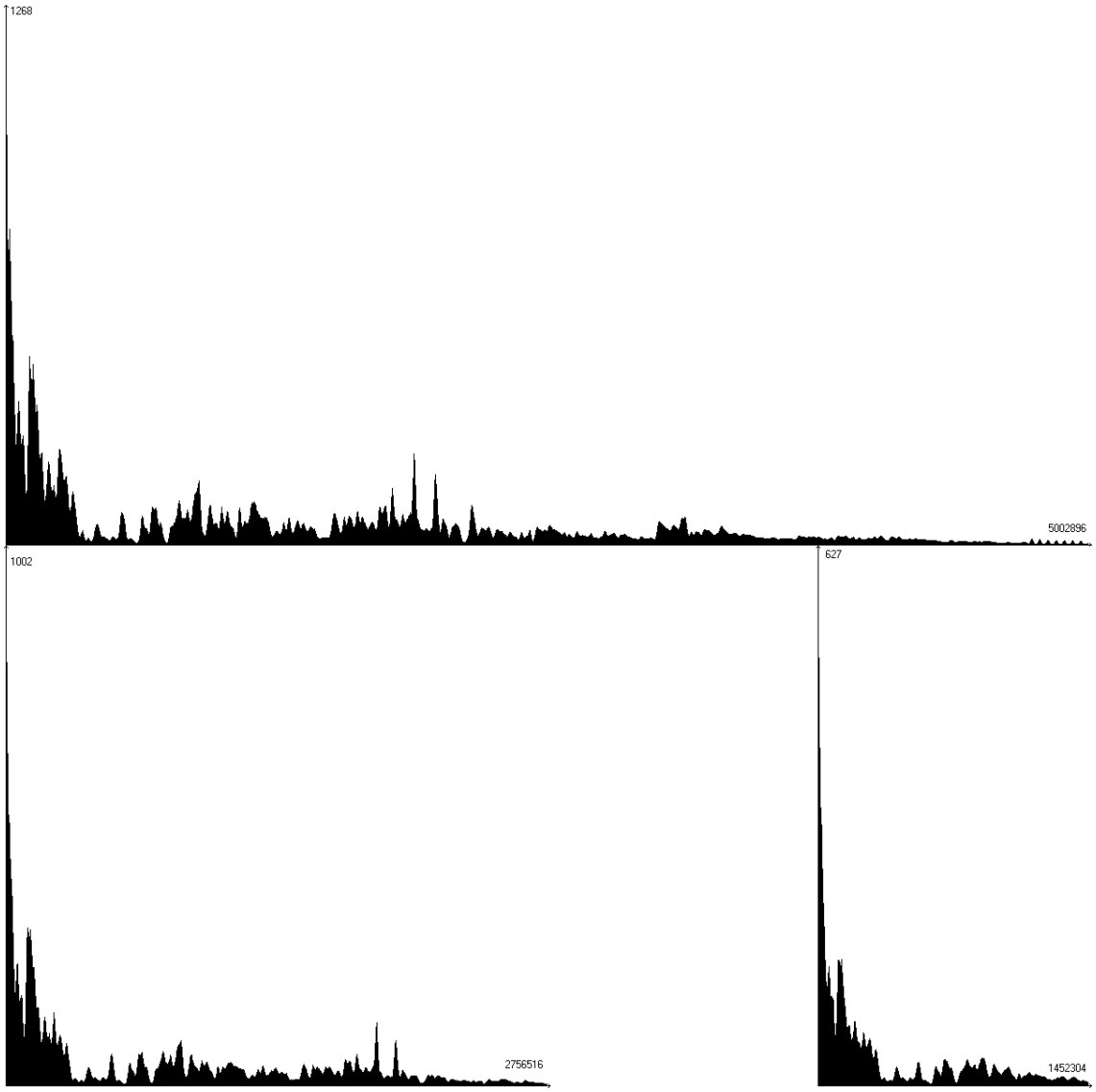


Рис. 2.9.
Графики зависимости частоты чтения от номера узла для одинакового текста урезанного до 10, до 5 и до 2.5 мегабайт

Вспомним про третью оптимизацию – кэшировать сколько-то начальных узлов. Из графиков видно, что ширина зоны до первого спуска – зоны частого обращения не изменяется в зависимости от длины текста. Она равна примерно половине миллиона. При увеличении длины текста высота этой зоны увеличивается (заметим, что у графиков разный масштаб по высоте), но общая картина остается примерно такой же. В итоге кэширование полумиллиона первых узлов позволит ускорить построение суффиксного дерева.

**Сравнение по типу текста.** Рассмотрим, как влияет состав текста на частотные характеристики. Для сравнения рассмотрим три различных типа текстов.

I. Первый тип – текст, состоящий из русских и английских слов – состоящий из больших и маленьких русских и английских букв и плюс некоторые знаки препинания. Для примера взят англо-русский и русско-английский словарь в электронном виде.

II. Второй тип текста – список файлов, разделенный символом, который нельзя употреблять в именах файлов.

III. Третий тип рассматриваемого текста будет случайная строка. Сгенерируем строку требуемой длины из случайных букв английского алфавита.

Построим графики зависимости частоты чтения от порядкового номера добавления узла в суффиксное дерево для каждого типа текста. Во всех случаях взят текст одинаковой длины – пять мегабайт. На рис. 2.10 – 2.12 приведены графики чтения для текстов I, II и III типа соответственно. Все три графика имеют схожие особенности. Во-первых, самые первые добавленные узлы читаются очень часто вне зависимости от текста. Во-вторых, имеется учащение частоты в левой половине графиков и падение частоты чтения в правой половине.

Рис. 2.13, на котором изображена частота записи, приведен, для того чтобы показать какие плавные графики получаются для случайных текстов.

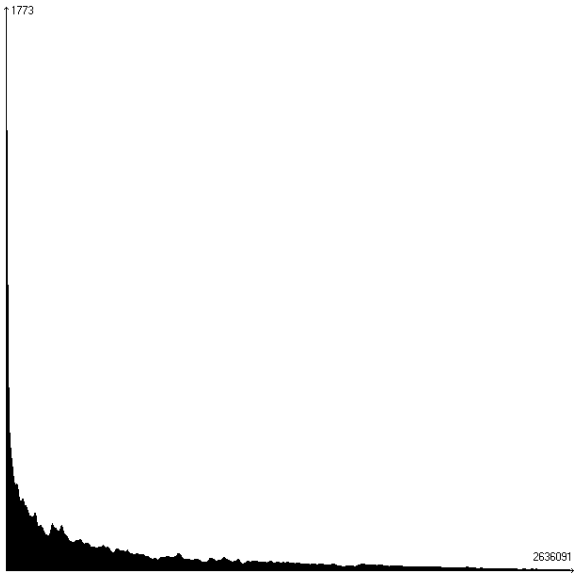


Рис. 2.10.
Зависимость чтения для текста первого типа.
Русские и английские слова

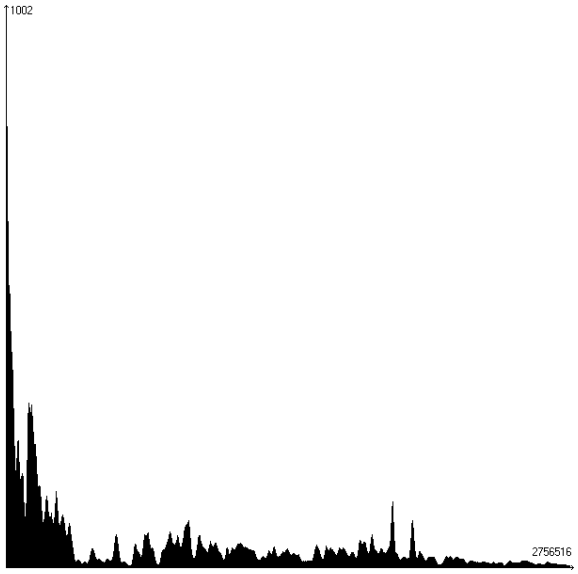


Рис. 2.11.
Зависимость чтения для текста второго типа.
Список файлов

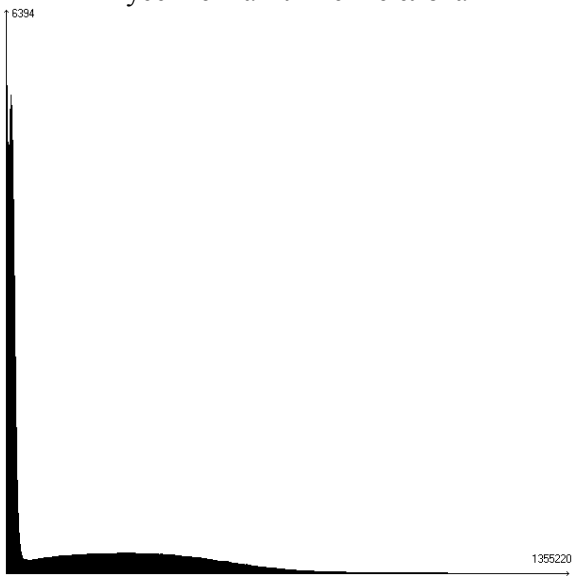


Рис. 2.12.
Зависимость чтения для текста третьего типа.
Случайный текст

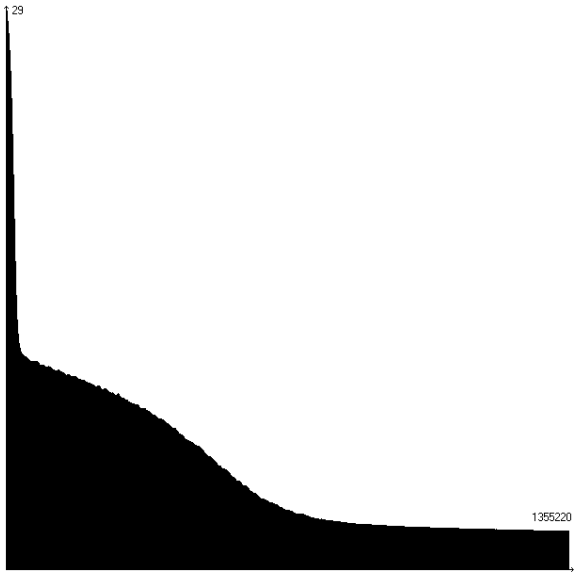


Рис. 2.13.
Зависимость записи для текста третьего типа.
Случайный текст

Зона, которая изображена на графиках, примерно два миллиона узлов, имеет схожий вид всегда. Если строить эти же графики для текста большей длины, то общая структура этой зоны не изменяется. Графики просто продолжаются вправо, сохраняя общую картину в этой зоне, а частота чтения добавленных узлов меньше, и постепенно уменьшаются слева направо. Этот факт

уже рассматривался выше на примере текста второго типа. Для частот записи утверждения, изложенные выше, также верны.

На рис. 2.14 – 2.16 приведены частоты чтения узлов в зависимости от возраста узла для текстов I, II, III типа соответственно. Для них остаются в силе те же самые замечания об увеличении длины текстов.

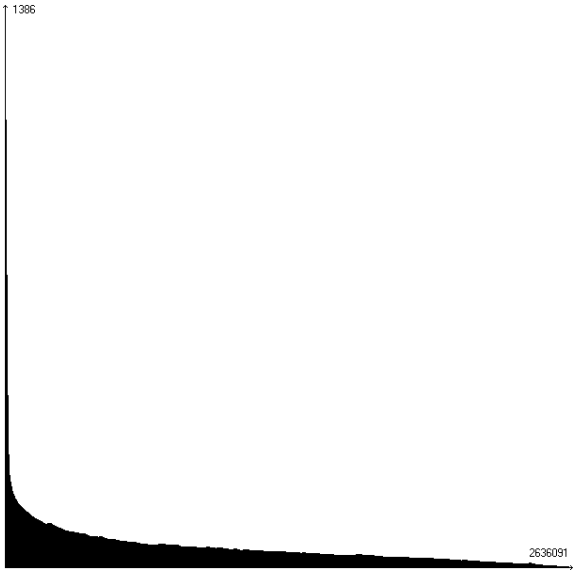


Рис. 2.14.
Зависимость от возраста для текста первого типа.
Русские и английские слова

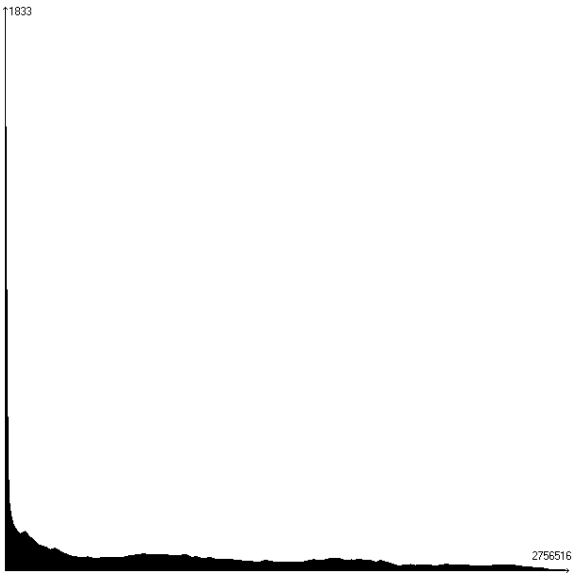


Рис. 2.15.
Зависимость от возраста для текста второго типа.
Список файлов

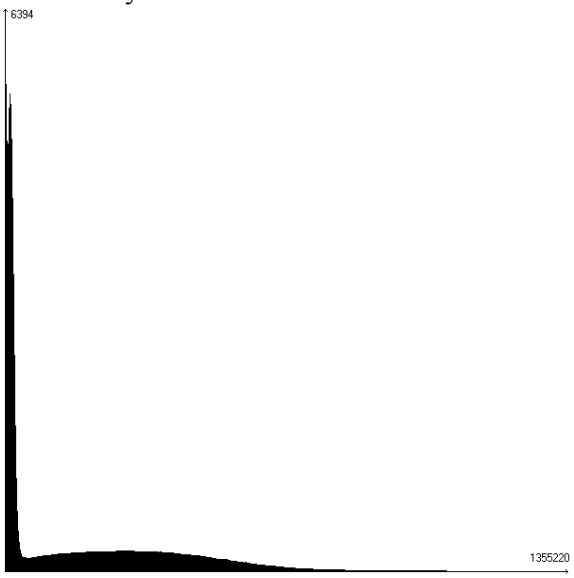


Рис. 2.16.
Зависимость от возраста для текста третьего типа.
Случайный текст

Таким образом, можно отметить, что все предложенные оптимизации справедливы для различных типов текста.

# ГЛАВА 3. ПОИСК ПО СУФФИКСНОМУ ДЕРЕВУ, ПОСТРОЕННОМУ ВО ВНЕШНЕЙ ПАМЯТИ

## *3.1. Оптимизация «вдоль»*

После того, как суффиксное дерево построено, в нем будет выполняться поиск. Постараемся найти всё, что можно сделать, для того чтобы ускорить поиск. Хотя поиск в суффиксном дереве и так быстрый (длина образца плюс число вхождений), но обычно, если для поиска выбирается алгоритм с построением суффиксного дерева, то в системе поиск очень критичен по времени, а следовательно, его ускорение не будет лишним.

Проанализируем частоты чтения узлов для различных образцов поиска в различных зависимостях.

Для построения частот будем производить поиск двух видов образцов. Первый из них, это слова, которые ищут пользователи. Слова взяты из лога одной поисковой системы, так что это как раз именно те образцы, которые запрашивает пользователь. Второй вид образцов для поиска – это случайные наборы английских и русских букв случайной, но ограниченной длины.

Первоначально рассмотрим частоты чтения узлов в том порядке, в котором узлы были созданы (рис. 3.1, 3.2). Поиск производится в тексте третьего типа из предыдущей главы (случайный текст).

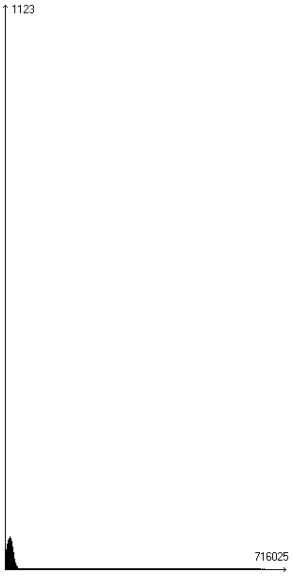


Рис. 3.1.
Зависимость частоты чтения от порядкового номера узла. Случайный образец

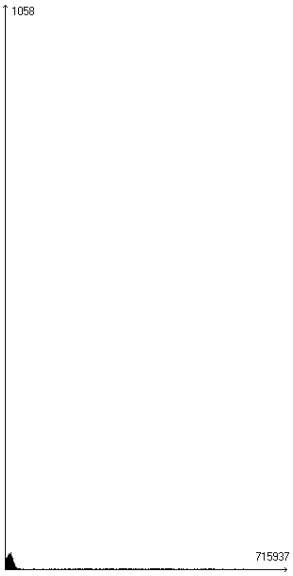


Рис. 3.2.
Зависимость частоты чтения от порядкового номера узла. Слова

Из графиков видно, что очень часто идет обращение только к самой первой группе узлов, а основное сгущение тоже только в начале. Это и понятно, ведь первые два–три уровня используются очень часто, а корень – всегда.

Построим подобные графики для текстов разной длины. На рис. 3.3 изображен тот же график что и на рис. 3.1, но масштаб по вертикали изменен так, чтобы первый бугорок занимал всю высоту, тогда можно увидеть форму остальной части графика. На рис. 3.4 изображен график для таких же параметров, но для текста в два раза длиннее.

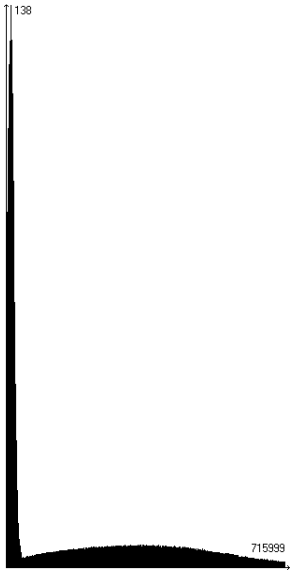


Рис. 3.3.
Изменённый масштаб для рис. 3.1.

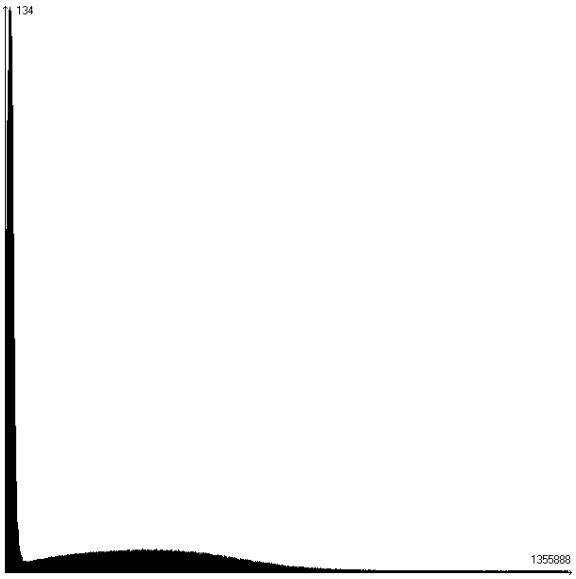


Рис. 3.4.
Тот же график, что и на рис. 3.3, но для длины текста в два раза больше.

Из рассмотрения графиков можно сделать вывод о том, что частота распределения частоты обращений к узлам не зависит от длины текста. При увеличении текста, растет число узлов, но более поздние узлы считываются редко. Площадь первого пика с первым бугорком примерно равна площади второго бугорка, поэтому если кэшировать всё вплоть до начала второго бугорка, то можно уменьшить чтение с диска в два раза.

Подобный вывод делается и при сравнении различных других типов текстов. Всегда наблюдается первый пик и что-то напоминающее первый бугорок. Второй бугорок принимает различные формы и может даже быть более равномерным, но всегда, чем позже был создан узел, тем реже происходит его чтение.

Поиск случайного образца в тексте или поиск образца в случайном тексте плох тем, что число вхождений в таких случаях мало. Поэтому проведённый выше анализ полезен для случая, если требуется найти первое вхождение. На рис. 3.5, 3.6 приведены всё те же графики зависимости. Однако для поиска были использованы образцы – слова, которые вводили пользователи, а поиск

производился в тексте второго типа – список файлов. Такой случай запроса интересен тем, что результат поиска может быть очень большим. Обычно человек ищет то, что должно быть там, где он ищет, и результат может быть также очень большим. На рис. 3.5 построен график частоты чтения узлов при поиске первого вхождения, а на рис. 3.6 – приведены частоты при поиске всех вхождений.

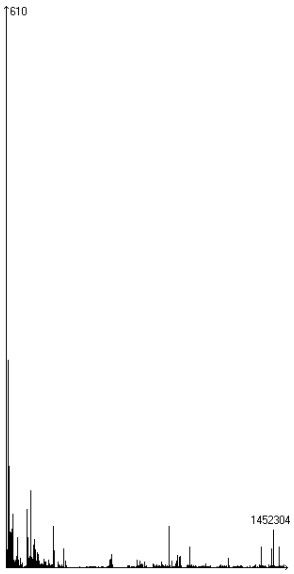


Рис. 3.5.
Зависимость частоты чтения от порядкового номера узла. Поиск первого вхождения

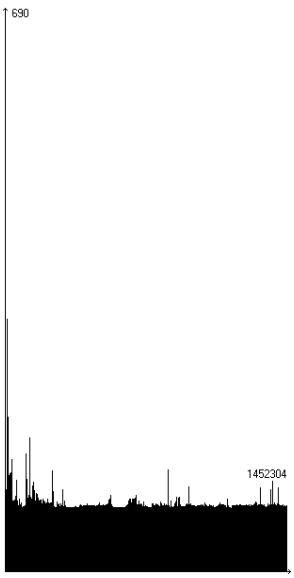


Рис. 3.6.
Зависимость частоты чтения от порядкового номера узла. Поиск всех вхождений

В результате рассмотрения этого вида зависимостей, можно сделать вывод, что для ускорения поиска первого вхождения требуется кэшировать около 100 тысяч первых созданных узлов. Это позволит ускорить поиск примерно в два раза, а для ускорения поиска всех вхождений эта характеристика не дает оптимизаций.

### *3.2. Оптимизация «в глубину»*

Рассмотрим частоты чтения узлов в зависимости от глубины узла. Будем исследовать глубину до 30, так как глубже узлов практически нет. На рис. 3.7 график зависимости числа узлов от глубины. Эта зависимость уже приводилась выше. На рис. 3.8 график зависимости суммарного числа чтений узлов для

каждого уровня глубины. Если поделить суммарное число чтений на число узлов на уровне, то получится график среднего числа чтений узлов от глубины – рис. 3.9.

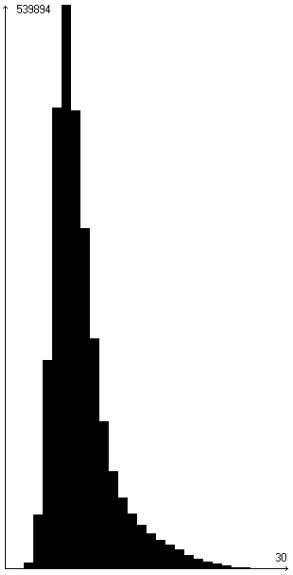


Рис. 3.7.
Зависимость числа узлов от глубины

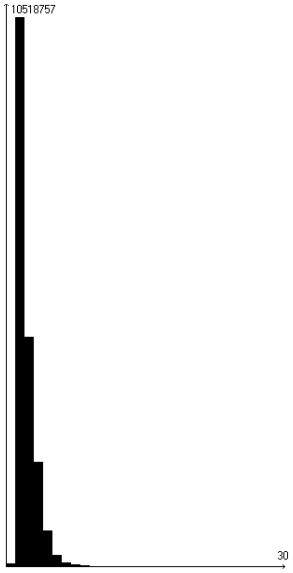


Рис. 3.8.
Зависимость суммарного числа чтений от глубины

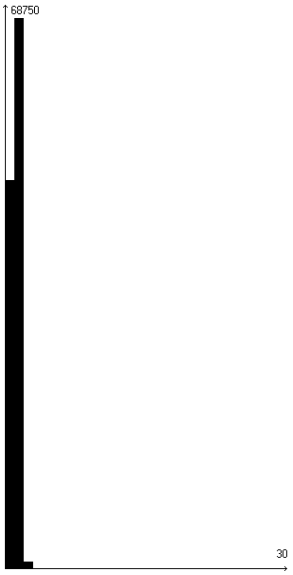


Рис. 3.9.
Зависимость средней частоты чтения одного узла от глубины

Из графиков видно, что очень часто читаются корневой узел и узлы глубины один и два (рис. 3.9). Также видно, что суммарное число чтений глубины от корня до глубины два – больше половины (рис. 3.8). Поэтому необходимо кэширование этих уровней, тем более что на этих уровнях очень небольшое число узлов (рис. 3.7).

Реализуем эту оптимизацию следующим образом. Во время поиска по суффиксному дереву, необходимо спускаться вниз по нему. Поэтому всегда известна текущая глубина. Если глубина меньше, чем заданная граница, то адресация узлов будет одной, а если больше – будет такой же, как и описывалась выше при построении дерева. После построения дерева следует создать новый массив, состоящий только из узлов с первых уровней. Будем хранить в массиве узлы «верхушки» в порядке их обхода. Индексы массива будут формировать их

номера. Если у узла номер $i$, то его адрес будет ($i + m + 1$), где $m$ – длина текста. Можно считать, что число этих узлов константа. Поэтому на оценку скорости построения суффиксного дерева это не повлияет.

Допустим, построено суффиксное дерево, и создан описанный выше массив с новой нумерацией части узлов. Допустим, что поисковый алгоритм, который знает текущую глубину, должен перейти по адресу $X$. Тогда он действует следующим образом:

- Если $X <= m$, то это лист.
- Если $X > m$, то это узел $i = (X – m – 1)$-ый элемент в массиве.
  - Если текущая глубина меньше заданной границы, то следует смотреть в массиве, который размещен в оперативной памяти.
  - Иначе – в массиве, размещенном в файле.

Поиск начинается с корня – узла с порядковым номером 0, его адрес $m + 1$, он находится в оперативной памяти.

При таком подходе, все узлы, которые сохранены в памяти, будут дублироваться в файле, и никогда не будут использоваться. Однако таких узлов, по сравнению с общим числом узлов, очень мало. К тому же перенумерация адресов, оставшихся в файле узлов, слишком сложна, поэтому это будут просто мертвые точки в файле.

Как и в предыдущем случае, графики были построены для поиска в случайном тексте. Точно такие же выводы получаются, если в любом тексте искать случайные образцы. Результатом такого поиска обычно будет очень мало вхождений текста. Если, как и в предыдущем случае, посмотреть на графики поиска пользовательских запросов в списке файлов, и построить такие же зависимости для количества узлов, суммарного количества чтений и среднего количества чтений от глубины, то получаются результаты, которые приведены на рис. 3.10 – 3.12.


Рис. 3.10.
Зависимость количества узлов от глубины


Рис. 3.11.
Зависимость суммарного количества чтений от глубины


Рис. 3.12.
Зависимость средней частоты чтения одного узла от глубины

К сожалению, графики не дают обнадеживающих результатов. По-прежнему средняя частота чтения каждого узла на первых уровнях высока, но суммарное число всех чтений от этих первых уровней не велико по сравнению с общим числом чтений.

В результате исследования данной зависимости получаем примерно такой же вывод, как и в предыдущем случае, когда мы пытались кэшировать сколько-то первых узлов. Если известно, что при поиске число вхождений будет маленьким, например, если требуется искать только первые вхождения, то применение оптимизации с кэшированием верхушки дерева помогает. Иначе, если требуются все вхождения, которых может быть очень много, то оптимизация не помогает, потому что более глубокие части дерева читаются так же часто, как и верхушка.

### *3.3. Поиск с ограничениями*

Поиск всех вхождений требуется не так часто. Любые поисковые системы ограничивают поиск весьма небольшим числом результатов. Например, Яндекс или Google разрешают вывести максимум по 100 результатов поиска на одной

странице. В связи с этим рассмотрим случай поиска пользовательских образцов в списке файлов с ограничением результата поиска – 100 вхождений.

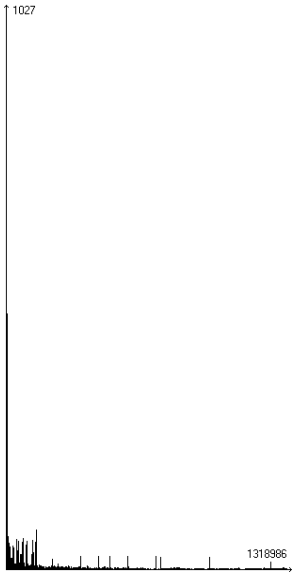


Рис. 3.13.
Зависимость частоты чтения от порядкового номера узла.
Ограничение – 100 вхождений

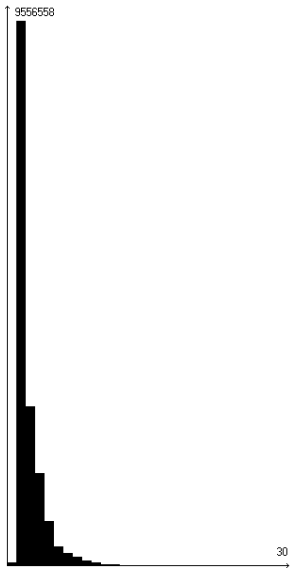


Рис. 3.14.
Зависимость суммарного числа чтений от глубины.
Ограничение – 100 вхождений

Рис. 3.13 не сильно отличается от рис. 3.5 (поиск первого вхождения) и совсем не похож на рис. 3.6 (поиск всех вхождений). То же самое можно сказать и про график на рис. 3.14. Так что можно сделать вывод о том, что обе предыдущие оптимизации будут хорошо работать для случая поиска 100 первых вхождений.

То же самое получается при увеличении максимального числа вхождений: главное чтобы это ограничение было на несколько порядков меньше длины текста.

# ГЛАВА 4. СРАВНИТЕЛЬНЫЙ АНАЛИЗ

После введения всех оптимизаций в главах 2 и 3, определим, какую практическую пользу они приносят в скорости построения суффиксного дерева и поиска по нему.

## *4.1. Построение*

Запустим построение суффиксного дерева алгоритмом без использования всех кэшей и с их использованием. Будем замерять скорости построения. Второй алгоритм будет использовать кэши на один миллион элементов, как это было описано в главе 2. Займет это около 30 мегабайт оперативной памяти. Так же рассмотрим использование кэшей на пять миллионов для сравнения.

Скорость процессора на данном этапе безразлична, всё построение работает со скоростью жесткого диска.

На рис. 4.1 представлен график зависимости текущей скорости в процессе построения суффиксного дерева. По горизонтали отложены мегабайты обработки исходного текста.

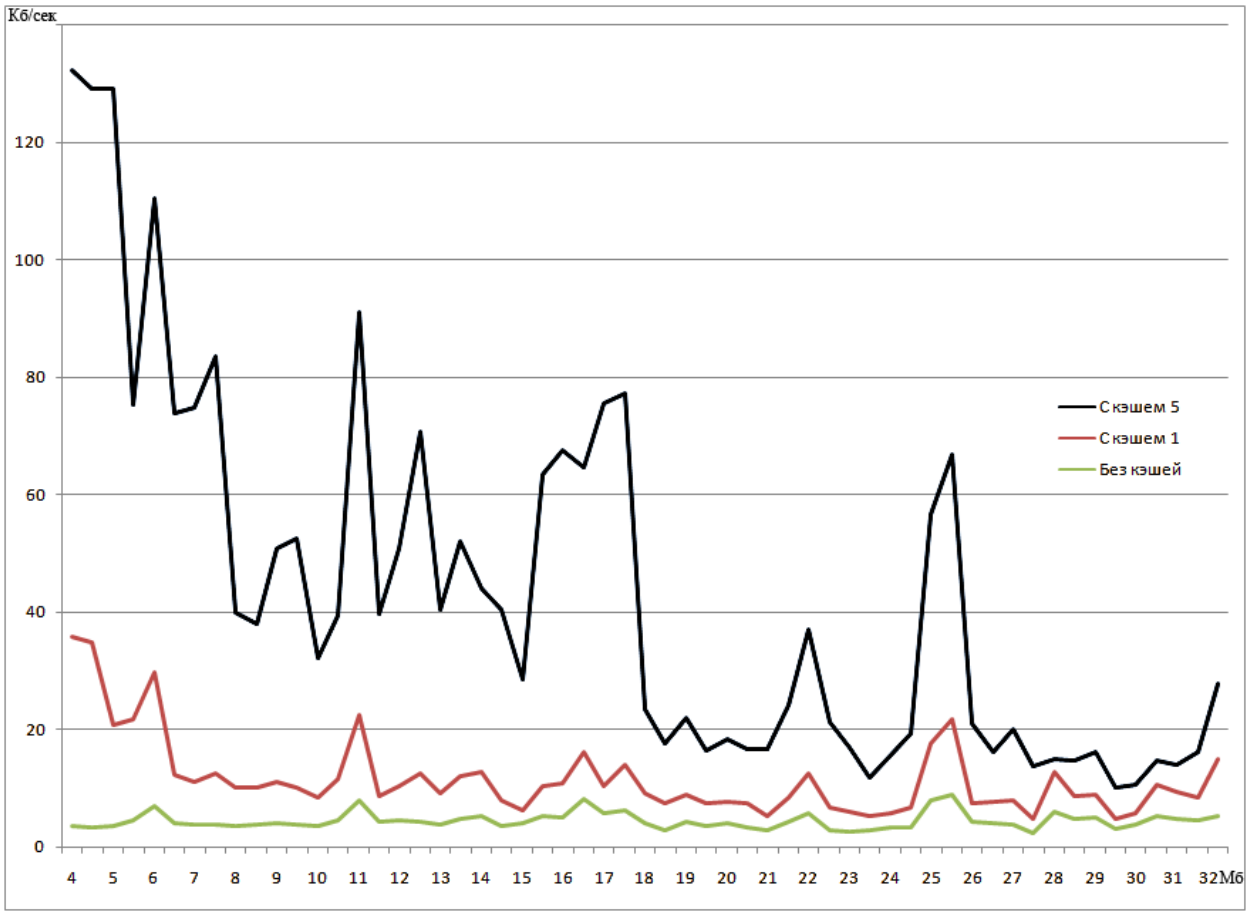


Рис. 4.1.
Зависимость текущей скорости в процессе построения

Из графика видно, что в каждый момент времени текущая скорость алгоритма с использованием оптимизаций, по крайней мере, вдвое больше, чем скорость алгоритма без оптимизаций. При этом можно сделать вывод о том, что чем больше кэш используется, тем больше будет скорость.

Не имеет смысла показывать графики с самого начала построения, когда оно производится только в кэше. В этот момент скорость большая, так как ещё не было никаких обращений к жесткому диску. График начинается с четырех мегабайт обработки исходного текста.

На рис. 4.2 построен подобный график для средней скорости в процессе построения для алгоритма с кэшем в один миллион.

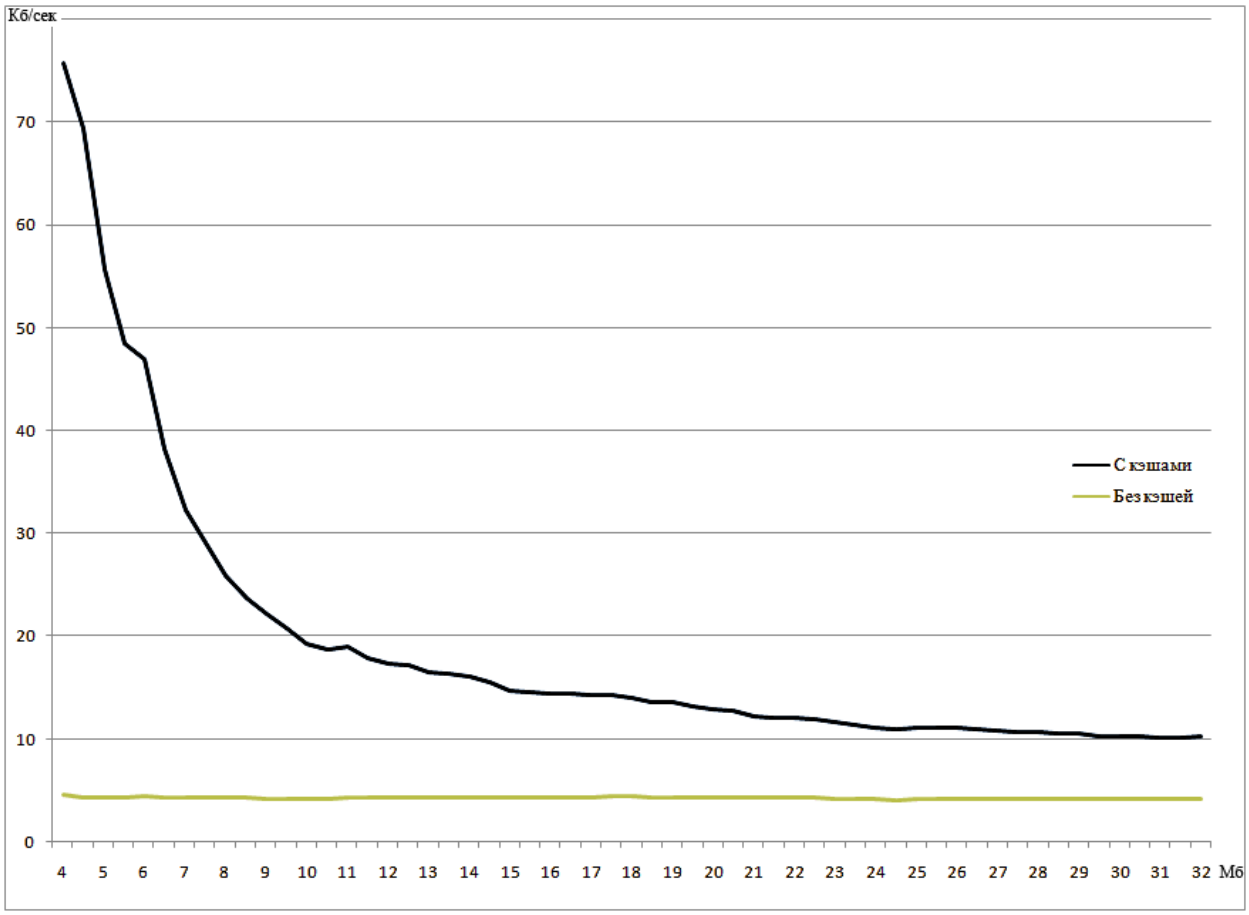


Рис. 4.2.
Зависимость средней скорости в процессе построения

Из этого графика видно, что в каждый момент времени уже средняя скорость так же, как минимум в два раза больше у алгоритма с оптимизациями по сравнению с алгоритмом без оптимизаций.

В итоге, можно сказать, что применение описанных выше оптимизаций увеличивает скорость построения суффиксного дерева.

### *4.2. Поиск*

Установим, как изменяется скорость поиска, при использовании оптимизаций поиска из главы 3, и сравним скорости поиска с другими алгоритмами.

Сделаем много запросов на поиск, и поделим общее время выполнения на число запросов. Получим среднюю скорость выполнения одного запроса. Как и раньше, будем искать случайные образцы и пользовательские образцы. Так же замерим среднюю скорость при поиске первого вхождения, первых 100 вхождений и всех вхождений.

Результаты поиска случайных образцов представлены в табл. 4.1.

Таблица 4.1. Сравнение времени поиска случайных образцов

| | Поиск первого вхождения | Поиск первых 100 вхождений | Поиск всех вхождений |
|---|---|---|---|
| Алгоритм с оптимизациями | 0,54 | 0,55 | 0,55 |
| Алгоритм без оптимизаций | 3,4 | 3,4 | 3,4 |
| Алгоритм «Наивный» | 128 | 129 | 129 |
| Алгоритм Бойера-Мура | 27 | 28 | 28 |
| Алгоритм Кнута-Морриса-Пратта | 240 | 240 | 240 |

Испытания проводились на компьютере с процессором *Pentium4* 3ГГц и жестким диском на 7200 об/мин со средней фрагментацией. Приведенные данные – это среднее время в миллисекундах на выполнение одного поиска в тексте длиной 10 мегабайт.

Как уже отмечалось, при поиске случайного образца результат поиска по сравнению с размером текста обычно не велик. Поэтому время поиска практически не зависит от того, как ограничен результат поиска.

Больший интерес представляют результаты поиска пользовательских образцов, так как в этом случае результат поиска может быть практически любым. Эти результаты представлены в табл. 4.2.

Таблица 4.2. Сравнение времени поиска пользовательских образцов

| | Поиск первого вхождения | Поиск первых 100 вхождений | Поиск всех вхождений |
|---|---|---|---|
| Алгоритм с оптимизациями | 1,1 | **1,6** | 10 |
| Алгоритм без оптимизаций | 3,6 | 4,3 | 12 |
| Алгоритм «Наивный» | 54 | 108 | 133 |
| Алгоритм Бойера-Мура | 11 | 30 | 41 |
| Алгоритм Кнута-Морриса-Пратта | 95 | 200 | 240 |

Из таблицы видно, что применение оптимизаций ускоряет поиск примерно в три раза, если есть ограничение результата, и оптимизации не сильно влияют на скорость поиска, если требуется искать все вхождения.

Задачи о поиске первого вхождения встречаются крайне редко. Однако практический интерес представляет вывод о том, что очень выгодно использовать суффиксные деревья для поиска первых 100 вхождений. Вообще говоря, использовать суффиксные деревья всегда выгодно, если исходная база статическая, и поиск требуется производить достаточно часто. Но если, например, исходный текст динамически изменяется, то может оказаться, что выигрыш во времени при поиске меньше, чем время на построение суффиксного дерева заново. В таком случае следует применять алгоритм Бойера-Мура.

Так же из обеих таблиц видно, что самый медленный из представленных алгоритмов – это алгоритм Кнута-Морриса-Пратта. И это даже, несмотря на то, что теоретическая оценка наивного алгоритма и алгоритма Бойера-Мура – $O(nm)$ – не линейная, а теоретическая оценка алгоритма Кнута-Морриса-Пратта – $O(m+n)$ – линейная. На практике абсолютная простота наивного алгоритма делает его быстрее. Большие скачки по строке при использовании алгоритма Бойера-Мура ставят его на первое место среди алгоритмов без предварительного построения индекса исходного текста, и дают результаты, приближающиеся к алгоритму на суффиксных деревьях.

# ЗАКЛЮЧЕНИЕ

В данной работе было рассмотрено применение суффиксного дерева с использованием внешней памяти.

Были проанализированы частоты чтения и записей узлов и листьев в процессе построения такого дерева. При этом были сделаны выводы о том, что следует производить буферизованное чтение текста, кэширование первых добавленных узлов и кэширование новых появляющихся улов со сбрасыванием половины кэша во внешнюю память, когда кэш заполняется. Эти оптимизации позволили ускорить построение суффиксного дерева в три–пять раз, если брать кэши на один–пять миллионов элементов.

Исследования частот чтения узлов суффиксного дерева при поиске по нему, показали, что если число вхождений не велико, или если ограничивать результаты поиска, то кэширование верхушки дерева дает увеличение скорости поиска в три раза.

В итоге установлено, что если текст не изменяется или изменяется очень редко, то выгодно применять алгоритм поиска с построением суффиксного дерева во внешней памяти. Особенно хорошие результаты дают ситуации, когда требуется искать не все вхождения, а только некоторое число первых вхождений.

Также было показано стандартное заблуждение о том, что для поиска без построения индекса текста следует применять алгоритм Кнута-Морриса-Пратта. На практике, алгоритм Бойера-Мура дает значительно большую скорость поиска. При этом сложность реализации этих алгоритмов примерно одинакова.

## СПИСОК ЛИТЕРАТУРЫ


1. *Гасфилд Д.* Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология. СПб.: Невский Диалект, БХВ, 2003.
2. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 2000.
3. *Уотермен М.* Математические методы для анализа последовательностей ДНК. М.: Мир, 1999.
4. *Manber U.* Finding similar files in a large file system. *USENIX* Conference, 1994.
5. *Лившиц Ю.* Построение суффиксного дерева за линейное время. 2006. http://logic.pdmi.ras.ru/~yura/internet/01ia.pda.
6. Суффиксные деревья. http://algolist.manual.ru/search/lrs/suffix.php.
7. *Ахметов И.* Разработка визуализатора алгоритма Укконена построения суффиксных деревьев на основе технологии *Vizi*. http://is.ifmo.ru/vis/ukkonen/.
8. *РОМИП* – Российский семинар по оценке методов информационного поиска. http://romip.narod.ru.