ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»
Факультет компьютерных наук
Образовательная программа бакалавриата «Программная инженерия»

СОГЛАСОВАНО
Научный руководитель,
Заместитель декана по учебно-методической работе факультета компьютерных наук
доцент департамента больших данных и информационного поиска, канд. социологических. наук

_____ И.Ю. Самоненко
«___» _____ 2022 г.

УТВЕРЖДАЮ
Академический руководитель образовательной программы «Программная инженерия»
профессор департамента программной инженерии, канд. техн. наук

_____ В.В. Шилов
«___» _____ 2022 г.

Подп. и дата	
Инв. № дубл.	
Взам. инв. №	
Подп. и дата	
Инв. № подл	

Приложение для визуализации алгоритма Фараха

Пояснительная записка

ЛИСТ УТВЕРЖДЕНИЯ

RU.17701729.10.03-01 81 01-1-ЛУ

Исполнитель:
студент группы БПИ206

_____ / Г. В. Вавилов /
«____» _____ 2022 г.

Москва 2022

УТВЕРЖДЕН
RU.17701729.10.03-01 81 01-1-ЛУ

Приложение для визуализации алгоритма Фараха

Пояснительная записка

RU.17701729.10.03-01 81 01-1

Листов 18

Подп. и дата	
Инв. № дубл.	
Взам. инв. №	
Подп. и дата	
Инв. № подл	

Москва 2022

Содержание

1. Введение

1.1. Наименование:

1.1.1. Название темы разработки на русском языке:

Приложение для визуализации алгоритма Фараха.

1.1.2. Название темы разработки на английском языке:

Farach Algorithm Visualization Application

1.2. Документ, на основании которого ведется разработка

Программа выполнена в рамках учебного плана подготовки бакалавров по направлению 09.03.04 «Программная инженерия» и утвержденная академическим руководителем программы тема курсового проекта «Приложение для визуализации алгоритма Фараха».

2. Назначение и область применения

2.1. Назначение

2.1.1. Функциональное назначение

Приложение наглядно показывает, как должно строиться сжатое суффиксное дерево для введенной пользователем произвольной строки символов при помощи алгоритма Фараха. Приложение визуализирует каждый этап алгоритма и объясняет выполненные шаги.

2.1.2. Эксплуатационное назначение

Текстовое описание алгоритма не всегда бывает удобным для восприятия. Приложение позволяет детально и наглядно рассмотреть каждый этап работы алгоритма на различных входных данных. Визуализация должна помочь непосредственно при реализации алгоритма.

Пользователь сможет посмотреть на построение сжатого суффиксного дерева для произвольной строки символов.

2.2. Характеристика и область назначения:

Алгоритм Фараха [1] используется для построения сжатого суффиксного дерева [9]. Алгоритм интересен тем, что имеет линейную временную сложность и может работать с бесконечными алфавитами.

Суффиксное дерево имеет множество применений. При помощи него можно осуществлять следующие операции:

1) За линейное время искать подстроки в строке.
2) За линейное время искать количество различных подстрок в строке.
3) За линейное время строить суффиксный массив. [10]
4) Сжатие данных. [2]

Алгоритм очень объемный и сложный для понимания. Приложение для визуализации должно помочь пользователю с формированием представления о работе алгоритма и об используемых в нем способах хранения данных.

Основная область применения – сфера образования.

3. Технические характеристики

3.1. Постановка задачи на разработку программы

Программа должна решать задачу визуализации алгоритма Фараха для построения сжатого суффиксного дерева. Алгоритм разбивается на шаги. Каждый шаг должен быть графически проиллюстрирован. По возможности, переходы между шагами должны сопровождаться анимацией. Каждый шаг алгоритма должен сопровождаться емким текстовым объяснением.

Программа должна предоставлять пользователю возможность выполнения следующих действий:

- Ввести символьную строку для дальнейшей обработки.
- Запустить пошаговую визуализацию алгоритма.
- Приостановить визуализацию алгоритма.
- Возобновить визуализацию алгоритма, если она была приостановлена.
- Перейти к предыдущему шагу визуализации.
- Перейти к следующему шагу визуализации.
- Запустить визуализацию алгоритма для новой строки без перезапуска приложения.

3.2. Описание применяемых математических методов

Алгоритм Фараха является рекурсивным. В совокупности с возможностью возвращаться на предыдущий шаг алгоритма, это обстоятельство затрудняет реализацию визуализации.

Разобьем алгоритм на шаги. Шаг – небольшая часть элементарных операций алгоритма. Примеры шагов: сортировка массива, удаление одинаковых элементов в массиве. В зависимости от входных данных, количество шагов в работе алгоритма будет меняться. При этом, для каждых входных данных существует определенная конечная последовательность шагов, при выполнении которой, входные данные будут преобразованы в выходные данные.

Такой подход решает проблему с переходами в рекурсивном алгоритме, так как, запустив алгоритм, можно вычислить шаги, которые необходимо затем визуализировать. Визуализацию можно представить в виде конечного автомата, в котором переходы определяются шагами

алгоритма, а состояния – представление данных между шагами алгоритма. Начальное состояние – входные данные. Описанный автомат будет выглядеть следующим образом (рис. 1).

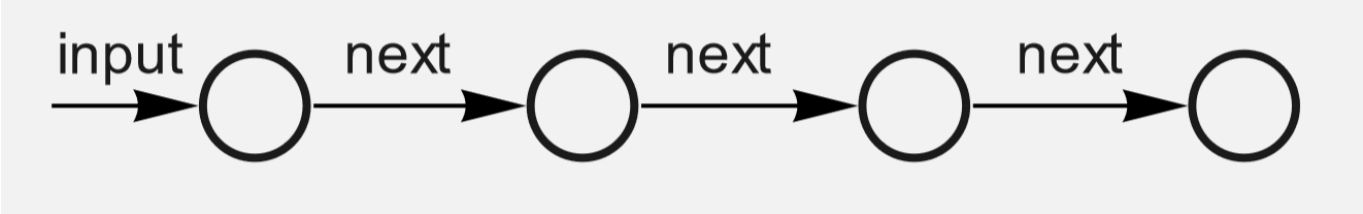

Рисунок 1. Конечный автомат визуализации.

В текущей модели нет возможности перейти в предыдущему состоянию визуализации. Для каждого перехода *next* определим переход *prev*, который будет полностью отменять действие *next*. Другими словами, *prev* будет совершать переход в предыдущее состояние визуализации.

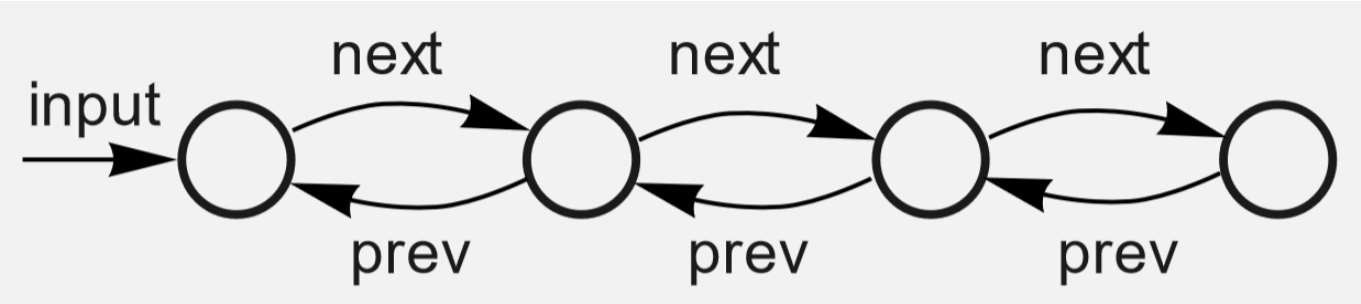

Рисунок 2. Итоговый конечный автомат визуализации.

3.3. Описание алгоритма и функционирования программы

3.3.1. Описание алгоритма Фараха для построения сжатого суффиксного дерева

Алгоритм описан в приложении 2.

3.3.2. Описание реализации программы

Конечный автомат, полученный в результате работы алгоритма, удобно представить в виде двусвязного списка. Переход *next* является шагом алгоритма, и реализуется в соответствии с описанием алгоритма. Переход *prev* отменяет действие соответствующего перехода *next*. Для реализации описанного поведения будет удобно запоминать данные, которые были изменены в результате применения перехода *next*, чтобы затем восстановить их в переходе *prev*.

3.4. Описание метода организации входных и выходных данных

3.4.1. Входные данные

На вход алгоритма подается символьная строка с максимальной длиной 20 символов. Ограничение обусловлено тем, что суффиксные деревья для строк большей длины не будут умещаться на экране пользователя.

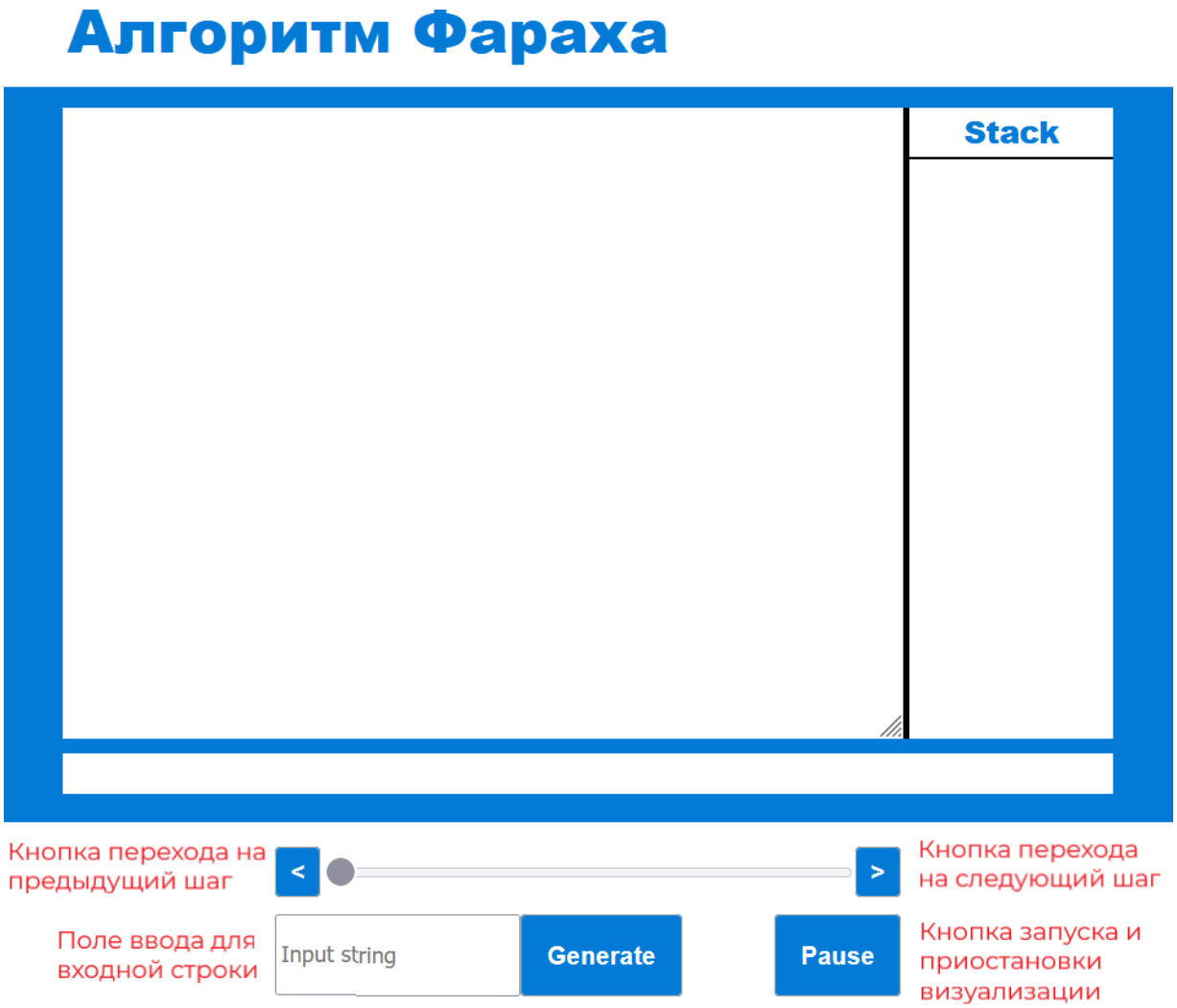

Рисунок 3. Пользовательский интерфейс.

Входная строка вводится в поле “Поле ввода для входной строки”.

Управление приложением происходит при помощи кнопок (рис. 3):

- “Generate” – По введенной входной строке генерирует конечный автомат визуализации.
- “Pause” – Приостанавливает автоматическое воспроизведение визуализации.
- “Play” (появляется, когда визуализация приостановлена) – Возобновляет автоматическое воспроизведение визуализации.
- Кнопки “<” и “>” – Совершают переход в предыдущему и следующему шагу визуализации соответственно.

3.4.2. **Выходные данные**

Результат работы программы – визуализация, которая иллюстрирует состояние данных в процессе работы алгоритма. Переходы между состояниями по возможности сопровождаются анимацией. Финальное состояние визуализации – сжатое суффиксное дерево, построенное для строки, введенной пользователем.

3.5. Описание состава технических и программных средств

3.5.1. **Программные средства**

Исходный код программы выполнен с использованием языков программирования TypeScript, HTML, CSS. Исполняемый файл получается в результате транспиляции и упаковки кода, написанного на TypeScript, в один JavaScript-файл, что позволяет запускать программу в браузере.

Программа использует следующие открытые библиотеки из реестра пакетов **npm** (Node package manager):

- D3.js – Библиотека для управления отображением данных на веб-странице.

Разработка велась на JavaScript платформе Node.js. Библиотеки из реестра пакетов **npm**, которые были использованы только для разработки или тестирования:

- **babel** – Транспилятор JavaScript. Использовался для преобразования TypeScript в JavaScript.
- **typescript -** Язык программирования, предоставляющий дополнительные возможности для языка JavaScript. Например, типы и интерфейсы.
- **@types/d3 –** TypeScript-типы для библиотеки D3.js
- **@types/jest –** TypeScript-типы для фреймворка jest.
- **jest –** JavaScript фреймворк для тестирования кода.
- **babel-jest –** Надстройка для jest, позволяющая транспилировать код, прежде чем будут запущены тесты.
- **webpack –** Сборщик кода для JavaScript. Позволяет поддерживать модульную структуру проекта.
- **webpack-dev-server –** Инструмент для создания локального сервера для разработки, поддерживающего сборку через webpack.
- **babel-loader –** Webpack-модуль, позволяющий транспилировать код при помощи babel во время сборки.

- **css-loader –** Webpack-модуль, позволяющий преобразовать CSS в JavaScript модуль во время сборки.
- **style-loader –** Webpack-модуль, размещающий CSS-код на веб-страницу во время сборки.
- **html-webpack-plugin** – Надстройка для webpack, позволяющая организовать работу с html-файлами во время сборки.

3.5.2. **Технические средства**

Для работы программы необходимы следующие компоненты:

- Процессор: Intel Pentium 4 / Athlon 64 или более поздней версии с поддержкой SSE2
- Видеокарта GeForce GTX 470/Radeon R7 260X или лучше.
- Операционная система: Windows 7 / Windows 8 / Windows 10 / Windows 11 / MacOS X 11 и выше / Linux
- Свободное место на жёстком диске: 256 МБ.
- Оперативная память: 4ГБ.
- Доступ в интернет.
- Клавиатура.
- Мышь или заменяющее устройство ввода.

Требования обусловлены необходимостью использования веб-браузера для работы программы.

4. Ожидаемые технико-экономические показатели

4.1. Предполагаемая потребность

Программа будет использоваться преподавателями и учениками технических ВУЗов в образовательных целях. Также программа будет актуальна для людей, занимающихся спортивным программированием.

4.2. Ориентировочная экономическая эффективность

Программа существенно упрощает восприятие и понимание алгоритма, что будет экономить время, затрачиваемое на обучение.

5. Список используемых источников

1. Алгоритм Фараха [Электронный ресурс] : Викиконспекты / Университет ИТМО – Электрон. текст. дан. – Москва: ИТМО, 2010 – URL: http://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Фараха, свободный (дата обращения 23.12.2021)
2. Алгоритмы LZ77 и LZ78 [Электронный ресурс] : Викиконспекты / Университет ИТМО – Электрон. текст. дан. – Москва: ИТМО, 2010 – URL: https://neerc.ifmo.ru/wiki/index.php?title=Алгоритмы_LZ77_и_LZ78, свободный (дата обращения 30.01.2022)
3. ГОСТ 19.101-77 Виды программ и программных документов. Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
4. ГОСТ 19.201-78 Техническое задание. Требования к содержанию и оформлению. Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
5. ГОСТ 19.301-79 Программа и методика испытаний. Требования к содержанию и оформлению. Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
6. ГОСТ 19.401-78 Текст программы. Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
7. ГОСТ 19.404-79 Пояснительная записка. Требования к содержанию и оформлению. Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
8. ГОСТ 19.505-79 Руководство оператора. Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
9. Сжатое суффиксное дерево [Электронный ресурс] : Викиконспекты / Университет ИТМО – Электрон. текст. дан. – Москва: ИТМО, 2010 – URL: https://neerc.ifmo.ru/wiki/index.php?title=Сжатое_суффиксное_дерево, свободный (дата обращения 30.01.2022)
10. Суффиксный массив [Электронный ресурс] : Викиконспекты / Университет ИТМО – Электрон. текст. дан. – Москва: ИТМО, 2010 – URL: https://neerc.ifmo.ru/wiki/index.php?title=Суффиксный_массив, свободный (дата обращения 30.01.2022)

Приложение 1

6. СПИСОК ТЕРМИНОВ

6.1. Суффикс строки

Подстрока данной строки, в которую входит последний символ строки.

6.2. Суффиксное дерево

Дерево, которое содержит все суффиксы данной строки. Каждое ребро дерева помечено одним символом. Каждый узел не может содержать два одинаковых исходящих ребра.

6.3. Сжатое суффиксное дерево

Суффиксное дерево, в котором ребра могут быть помечены подстрокой данной строки. Каждая вершина не может содержать исходящие ребра с одинаковыми первыми символами.

6.4. Четное суффиксное дерево

Сжатое суффиксное дерево, которое содержит суффиксы строки, находящиеся на четных позициях.

6.5. Нечетное суффиксное дерево

Сжатое суффиксное дерево, которое содержит суффиксы строки, находящиеся на нечетных позициях.

6.6. Временная сложность

Функция, которая зависит от размера входных данных. Представляет время работы алгоритма.

6.7. Линейная временная сложность

Время работы алгоритма пропорционально функции вида $y = kx+b$, где x – размер входных данных.

6.8. Алфавит

Множество символов, из которых составлена строка.

6.9. Транспиляция

Эквивалентное преобразование кода программы из одного языка программирования в другой с полным сохранением работоспособности.

Приложение 2

7. ОПИСАНИЕ АЛГОРИТМА ФАРАХА ДЛЯ ПОСТРОЕНИЯ СЖАТОГО СУФФИКСНОГО ДЕРЕВА

Алгоритм выполняется рекурсивно и состоит из пяти этапов.

1) Сжатие алфавита.
2) Построение четного суффиксного дерева.
3) Построение нечетного суффиксного дерева.
4) Слияние четного и нечетного деревьев.
5) Удаление двойных ребер.

7.1. Этап 1. Сжатие алфавита

На вход принимается строка.
Возвращается построенное суффиксное дерево.

Если длина строки равна 1, то суффиксное дерево для нее тривиально (рис. 4). Дерево возвращается, шаг 1 завершается.

Рисунок 4 - Тривиальное дерево

Если длина строки нечетна, в конец строки добавляется специальный символ, который не равен ни одному из символов текущего алфавита.
Строка разбивается на пары подряд идущих символов.
Пары символов сортируются лексикографически при помощи поразрядной сортировки.
Удаляются копии пар.
В исходной строке пары символов заменяются на индексы этих пар в массиве, полученном на предыдущем шаге.
Получилась строка длиной в два раза меньше, чем была изначально.
Выполнить этап 1 для сжатой строки.
Выполнить этап 2. Построить четное дерево.
Выполнить этап 3. Построить нечетное дерево.
Выполнить этап 4. Произвести слияние четного и нечетного деревьев.
Выполнить этап 5. Удалить двойные ребра в полученном дереве.

7.2. Этап 2. Построение четного дерева

На вход принимается суффиксное дерево для сжатой строки, полученное на предыдущем этапе.

Возвращается четное суффиксное дерево, построенное для текущей строки.

Каждый символ из ребер суффиксного дерева заменяется обратно на его пару символов. Массив с парами был получен на предыдущем этапе.

После раскрытия пар могут возникнуть ребра с одинаковыми первыми символами, принадлежащие одной вершине. Создаем новую вершину между такой вершиной и ребрами с одинаковым первым символом (рис. 5).

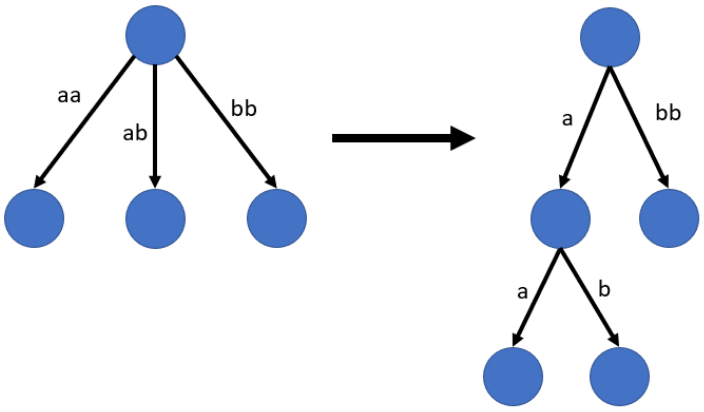

Рисунок 5 - Исправление ребер с одинаковыми первыми символами

Четное суффиксное дерево построено.

7.3. Этап 3. Построение нечетного дерева

На вход принимается четное суффиксное дерево для сжатой строки, полученное на предыдущем этапе.

Возвращается нечетное суффиксное дерево, построенное для сжатой строки.

Строим суффиксный массив по нечетному дереву.

Дописываем к каждому суффиксу в массиве символ, предшествующий этому суффиксу в изначальной строке.

Сортируем суффиксы по дописанному символу.

Получили суффиксный массив для нечетного дерева.

Строим нечетное суффиксное дерево по суффиксному массиву.

7.4. Этап 4. Слияние деревьев

На вход принимается четное и нечетное дерево, полученные на двух предыдущих этапах.

Возвращается новое дерево – результат слияния.

Рекурсивно выполняется следующий алгоритм:

Имеем два указателя на корни четного и нечетного поддерева и указатель на корень нового поддерева.

Просматриваем ребра по одному в каждой вершине.

- Если первые символы ребер различны, копируем ребро с меньшим первым символом в новое дерево. Переходим к следующему ребру в вершине, ребро которой было скопировано.
- Если первые символы ребер одинаковы и длины строк в ребрах равны, копируем оба ребра в новое дерево. Переходим к следующему ребру в обоих вершинах.
- Если первые символы ребер одинаковы и длины строк в ребрах различаются, в новое дерево добавляются оба ребра. Создается дополнительная вершина, которая разбивает большее ребро на два ребра. Первое ребро содержит префикс длины строки в меньшем ребре.

Рекурсивно спускаемся в следующую вершину после ее добавления в новое дерево.

7.5. Этап 5. Удаление двойных ребер

На вход принимается дерево, в котором могут быть двойные ребра, полученное на предыдущем этапе.

Возвращается дерево без двойных ребер.

Рекурсивно проходится по всем вершинам дерева, и выполняем следующие действия для каждого двойного ребра:

Если два ребра имеют полностью одинаковые строки – удаляем одно из ребер.

Иначе находим общий префикс и создаем новую вершину и ребро с этим префиксом, ведущее в эту вершину.

Из новой вершины выпускаем два ребра, к которых будут находиться строки двойного ребра без общего префикса.

Разделяем ребра, выходящие из двойного ребра на четные и нечетные и присоединяем их к соответствующему четному/нечетному новому ребру. (рис. 6)

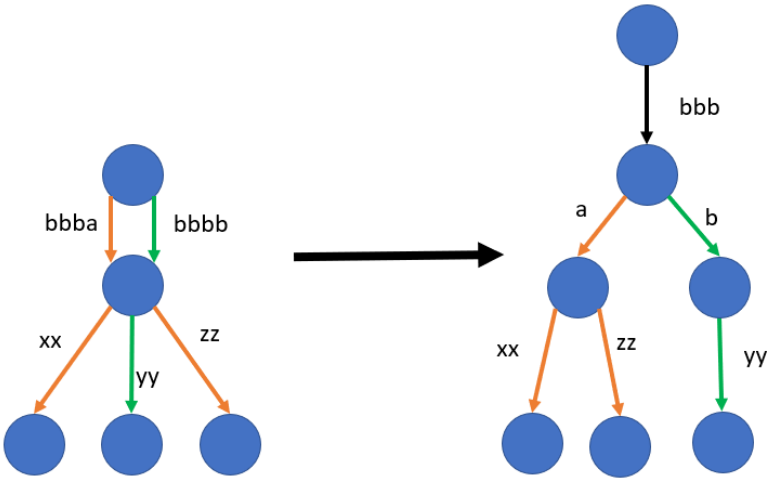

Рисунок 6 - Удаление двойного ребра

Лист регистрации изменений

Лист регистрации изменений									
Номера листов (страниц)					Всего листов (страниц в докум.)	№ документа	Входящий № сопроводительного докум. и дата	Подп.	Дата
Изм.	Измененных	Замененных	Новых	Аннулированных					