ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

Отчет о программном проекте

на тему: Токенизированный алгоритм Майерса (diff)

Выполнили:

Студент группы БПМИ215	Подпись	П.А. Шишихин И.О.Фамилия
Студент группы БПМИ215	Подпись	Н.А. Голубев И.О.Фамилия
Студент группы БПМИ215	Подпись	К.Н. Ниневский И.О.Фамилия
Студент группы БПМИ215	Подпись	Я.В. Аникиев И.О.Фамилия

25.05.2023
Дата

Принял:

Руководитель проекта — Грищенко Виктор Сергеевич
Имя, Отчество, Фамилия

к.ф.-м.н.
Должность, ученое звание

Институт Математики и Механики им Н.Н.Красовского
Место работы (Компания или подразделение НИУ ВШЭ)

Дата проверки 25.05 2023 — 9 (Оценка (по 10-ти бальной шкале)) — Подпись

Москва 2023

Содержание

Аннотация

Существуют разные алгоритмы и подходы для решения задачи сравнения содержимого файлов. В данной работе рассматривается и реализуется алгоритм Myers Diff algorithm с токенизацией для достижения максимальной эффективности и гибкости популярной утилиты diff.

Ключевые слова: алгоритм, Майерс, токенизация, сравнение.

1 Основные термины, определения и сокращения

Токенизация (tokenization) — процесс разбиения текста, файла или, более обобщённо, последовательности байтов, используя определенное правило, на логически неделимые в рамках этого правила единицы, так называемые *токены* (tokens). Токенами могут быть, например, ASCII символ, Unicode символ в UTF-8 кодировке, строка текста, название переменной в коде и прочее. То, что представляют из себя токены, задается правилом токенизации.

Строка — в контексте работы под строкой может подразумеваться не только последовательность текстовых символов, но и произвольных токенов, полученных в результате токенизации.

Подпоследовательность строки (subsequence) — строка, полученная удалением одного или нескольких токенов из данной строки.

Общая подпоследовательность двух строк (common subsequence) — подпоследовательность каждой из двух данных строк.

Скрипт (edit script) — для строк A и B скриптом является такая последовательность удалений и вставок токенов в A, в результате которой из A получается B. Заметим, что алгоритм Майерса используется для нахождения такого скрипта минимальной возможной длины (количество примененных удалений и вставок).

Diff — различие двух строк. Это неформальное определение, потому что различие в зависимости от контекста целесообразно выражать по-разному: в виде скрипта или наибольшей общей подпоследовательности. Следует понимать, что эти задачи равносильны.

Владелец токена — файл, которому принадлежит токен с точки зрения Diff. Для токена может быть ровно три вида владения: токеном владеет исключительно старый файл (то есть в Diff нужно показать, что токен был в старом файле, но в новом его нет), токеном владеет исключительно новый файл (то есть в DIff нужно показать, что токена не было в старом файле, но в новом он есть), токеном владеют оба файла (то есть в Diff нужно показать, что токен есть в обоих файлах).

Замена (replacement) — замена нескольких подряд идущих токенов одного файла на последовательность подряд идущих токенов другого файла. Нетрудно видеть, что весь Diff представляется в виде нескольких подобных замен.

Обозначения для оценки сложности алгоритма

N — суммарная длина (в токенах) входных данных.

D — длина минимального edit script для двух входных строк из токенов.

2 Введение

При работе с цифровыми данными регулярно встает задача поиска различий между файлами, которую можно решать, к примеру, утилитой *git diff* для поиска различий между старой версией кода и новой. При использовании такого рода утилит возникает ряд проблем: программа должна быть эффективна и по памяти, и по времени работы, а также должна быть гибкой в плане того, как следует интерпретировать информацию. Так, например, не всегда однозначно, стоит искать разницу построчно (например, искать только новые, или измененные, или удаленные строки кода, как единое целое) или посимвольно (например, если в строке изменили только один символ, то целесообразнее показать, что изменился ровно один символ, а не вывести измененную строку целиком).

В основе работы лежит несколько ключевых идей: сам алгоритм Майерса [10] для поиска различий в информации, а также токенизация. Например, за счет токенизации можно, не меняя алгоритм, искать различие как и в смысле строк, так и в смысле символов, с минимальными затратами на разработку. При этом сложность токенизаторов ничем не ограничивается. Более того, можно использовать токенизаторы для языков программирования, чтобы находить различия в типах переменных, сигнатурах функций и т.д. Таким образом, используя знание о структуре информации в файле, мы можем давать более осмысленную и наглядную информацию пользователю программы.

Также стоит отметить, что токенизация входных данных сокращает время работы основной части алгоритма, так как уменьшается размер данных, для которых нужно искать скрипт. В частности, учитывая, что средняя длина слова в русском языке примерно равна 5, то, разбив текст по словам, время самого алгоритма Майерса для русских текстов снижается как минимум в 5 раз.

Задачи разделены между участниками проекта следующим образом:

1. Реализация алгоритма Майерса - Голубев Никита
2. Реализация токенизаторов - Ниневский Камиль, Шишихин Павел, Голубев Никита, Аникиев Ян
3. Внедрение Language Server - Ниневский Камиль
4. Визуализация диффа - Аникиев Ян, Шишихин Павел
5. Тестирование - Шишихин Павел, Голубев Никита
6. Интеграция - Аникиев Ян

3 Обзор литературы

Идея алгоритма Майерса описана самим Юджином Майерсом в его статье [10] 1986 года. Подробнее о самом алгоритме можно прочитать в разделе 5.5.

На основе алгоритма Майерса есть несколько реализаций утилиты diff. Одна из них – `GNU diff` [9], авторы: Paul Eggert, Mike Haertel, David Hayes, Richard Stallman and Len Tower. Эта программа является стандартной в операционной системе `GNU/Linux`. Другая известная реализация – инструмент `Git diff` [3] для работы в системе контроля версий `Git`, автор: Linus Torvalds. Обе программы имеют открытый исходный код, написаны на `C`, проверены временем и существенно оптимизированы.

Наш вариант `diff` отличается тем, что написан на `C++` и умеет использовать токенизаторы любой сложности. Это позволяет пользователю видеть более правильную, с точки зрения смысла, разницу между двумя файлами.

Больше всего нам интересно внедрить токенизатор, который эффективно работает с файлами, содержащими программный код, т.к. именно программисты чаще всего пользуются утилитой diff. Существует множество различных реализаций таких токенизаторов, например TextMate [7], Antlr [1], Language Server [4]. Мы остановились на выборе Language Server, потому что его легче всего было подключить к нашему проекту. Подробнее об этом в разделе 5.4.

Для того, чтобы можно было сосредоточиться на добавлении ценного и нового функционала в наш проект, мы выбрали готовое решение для парсинга аргументов командной строки в виде единственного `header`-файла: argparse [2]. Оно позволяет добавлять новые параметры практически любого формата с минимальными усилиями разработчика, легко интегрируется в проект, обладает богатым функционалом и удобен для использования пользователем.

4 Описание функциональных и нефункциональных требований к программному проекту

Функциональные требования

Результатом работы должна быть комплириуемая программа, написанная на языке C++. Программа должна принимать на вход в качестве параметров командной строки пути к файлам, между которыми необходимо найти различие, а также параметры токенизации. В результате работы программы на стандартный поток вывода должно быть напечатано различие в переданных на вход файлах. Программа должна быть эффективной и по времени исполнения, и по потребляемой памяти. Для выполнения этих требований программа должна использовать алгоритм Майерса [10] с худшим временем работы $O(ND)$, $O(N)$ используемой памяти и $O(N + D^2)$ ожидаемым временем работы.

Нефункциональные требования

Программа должна быть легко модифицируемой, модульной, чтобы на каждом глобальном этапе её модернизации можно было внести изменения в способ обработки данных. Программа должна предоставлять интерфейс для написания пользовательских токенизаторов без внесения изменений в сам алгоритм.

5 Содержательная часть

Содержательная часть разделена на несколько подчастей. Сначала мы взглянем на программу сверху, не углубляясь в детали, потом планомерно пойдем по каждому этапу работы программы: токенизации, алгоритму нахождения diff и визуализации результата работы программы (он же edit script).

5.1 Интерфейс командной строки

Наша программа является консольным приложением. Она имеет следующую сигнатуру:

```
diff [parameters] old_file new_file
```

Обязательные параметры:

`old_file`

Относительный или абсолютный путь к первому файлу, от которого требуется получить diff.

`new_file`

Относительный или абсолютный путь ко второму файлу, от которого требуется получить diff.

Необязательный параметры:

`-t, --tokenizer TOKENIZER_MODE [default: "line"]`

Выбор режима работы токенизатора:

`symbol`

Посимвольное разбиение файлов на токены

`word`

Пословное разбиение файлов на токены (пробельные символы: ' ', \t, \n, \v, \f, \r)

`line`

Построчное разбиение файлов на токены

`ignore-all-space`

Построчное разбиение файлов на токены, при этом строки считаются равными, если при удалении всех пробелов (' ', \t) они будут равны

`ignore-space-change`

Построчное разбиение файлов на токены, при этом строки считаются равными, если при замены всех непустых последовательностей пробелов (' ', \t) на один пробельный символ они будут равны

`semantic-code`

Режим токенизации, учитывающий семантику языка при токенизации программных файлов, основанный на Language Server(подробнее в разделе 5.4)

`-p, --parser PARSER_MODE [default: "utf-8"]`

Выбор режима работы парсера:

`bytes`

Файлы рассматриваются как последовательности байт.

`utf-8`

Файлы рассматриваются как последовательности символов в кодировке UTF-8

`-r, --recursive`

Рекурсивный diff двух переданных директорий. diff вызывается для файлов, чьи относительные пути, включая их имена, относительно переданных директорий совпадают.

`-b, --benchmark`

Измерение времени работы программы, с отдельным замером для `mmap`, токенизации, работы самого алгоритма и вывода

`--common-prefix PREFIX [default ’’]`

Перед выводом общей части diff так же выводится заданный `PREFIX`

`--old-prefix PREFIX [default "+"]`

Перед выводом части diff, содержащийся только в первом файле, так же выводится заданный `PREFIX`

`--new-prefix PREFIX [default "+"]`

Перед выводом части diff, содержащийся только во втором файле, так же выводится заданный `PREFIX`

`-c, --common`

При выводе программы общая часть файлов так же печатается.

`--raw`

При выводе программы данные не окрашиваются в цвет (зеленый для добавления, красный для удаления)

`--show-common-newlines`

При выводе программы общая часть файлов так же печатается.

`--show-pos`

При выводе программы печатается так же номер выводимой стоки в исходном файле.

5.2 Общий взгляд на работу программы

Здесь опишем, что происходит внутри main.cpp во время работы программы.

5.2.1 Работа для одиночного файла

На вход main.cpp(на графе обозначен как *Main*) пользователь передает два пути до файла, которые он хочет сравнить, и собственные опции. Дальше происходит следующая цепочка действий:[1]

1. *Main* в первую очередь при запуске парсит аргументы командной строки сторонней библиотекой `argparse` [2]. С её помощью извлекаются все необходимые параметры, такие как тип токенизатора и пути к файлам.
2. *Main* вызывает функцию `const char* MmapFile(const char* file_name, size_t file_size)`, отображая содержимое двух переданных файла на память и получая указатель на его начало.
3. *Main* создает `std::string_view` на полученных указателях `const char*` на начала переданных файлов.
4. *Main* получает `Tokenizer`, соотстветствующий заданным пользователем параметрам, посредствам функции
   `std::unique_ptr<Tokenizer> GetTokenizer(TokenizerMode tokenizer, ParserMode parser, std::string_view old_file, std::string_view new_file)` из библиотеки *Tokenizer*
5. *Main*, используя методы полученного базового класса `Tokenizer` :
   `std::vector<TokenInfo> GetOldTokensInfo()` и `std::vector<TokenInfo> GetNewTokensInfo()`, получает закодированное токенизированное представление каждого из двух файлов
6. *Main* передает полученные представления функции
   `Script ShortestEditScript(vector<TokenInfo> from, vector<TokenInfo> to)` для получение edit script (см. раздел 1)
7. *Main* создает класс `DiffPrinter`, соотвествующий заданным пользовательским параметрам.
8. *Main* вызывает метод `DiffPrinter::Print(std::ostream &out, Script script, Tokenizer tokenizer, std::vector<TokenInfo> old_codes, std::vector<TokenInfo> new_codes)` из библиотеки *PrintDiff* с полученным скриптом, для печати отформатированной разницы файлов пользователю (в данном случае `output = std::cout`)
9. *DiffPrint*, используя методы класса `Tokenizer`:
   `TokenType DecodeOld(TokenInfo code)` и `TokenType DecodeNew(TokenInfo code)` из библиотеки *Tokenizer*, расшифровывает edit script в исходные токены и выводит разницу в переданных файлах пользователю

[1] Для простоты понимания некоторые опции у функций были опущены

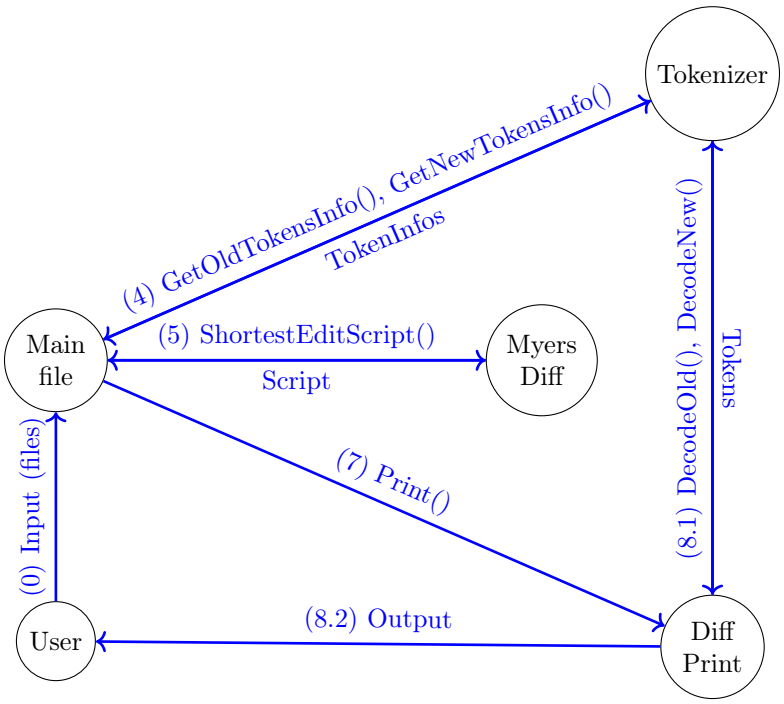


Диаграмма 0. Визуализация архитектуры программы
Цифры в скобках обозначают этапы работы программы в соответсвии с планом выше, под стрелками обозначается возвращаемое значение

5.2.2 Работа для дирекорий

В случае рекурсивного алгоритма для директорий пользватель передает два пути до директорий, которые он хочет сравнить, и собственные опции. Далее происходит следующая цепочка действий:

1. Обойдем все содержимое первой директории посредством `recursive_directory_iterator` из стандартной библиотеки `std::filesystem`.
2. Для очередного файла проверяем, есть ли файл с таким же относительным путем во второй директории, если его нет, то сообщаем, что этот файл есть лишь в первой директории и берем следующий файл.
3. Если файл есть и во второй директории, то проверяем, совпадают ли их типы. Если не совпадают, то сообщаем об этом и переходим к следующему файлу.
4. Для файлов с одинаковым типом проверяем, являются ли они регулярными файлами (не сокетами, директориями и т.п.) если являются, то выводим их diff (см. 5.2.1).
5. После проверки всех файлов первой директории остается лишь обойти содержимое второй и вывести, какие файлы есть только в ней (все общие файлы уже обработаны).

5.3 Токенизация

Токенизация — первый этап обработки входных данных. Для нее используется библиотека *Tokenizer*. Рассмотрим ее подробнее.

5.3.1 Предоставленная функциональность

Для взаимодействия с библиотекой предоставлен виртуальный класс `Tokenizer`. Для пользователя этот класс предоставляет интерфейс из четырех публичных виртуальных методов:

```
public:
    std::vector<TokenInfo> GetOldTokensInfo();
    std::vector<TokenInfo> GetNewTokensInfo();
```

```
TokenType DecodeOld(const TokenInfo& code);
TokenType DecodeNew(const TokenInfo& code);
```

`GetOldTokensInfo` и `GetNewTokensInfo` преобразует отображение старого и нового файла на `std::string_view` в набор токенов `std::vector<TokenType>`. Этот набор токена в последующем отдается алгоритму Майерса (о нем речь пойдет в разделе 5.5).

`TokenInfo` является классом, у которого есть два поля: `uint32_t code, begin`, соответствующий коду токена и его позиции его начала в исходном файле, методы, позволяющие получить эти поля (поля являются приватными): `uint32_t GetCode(), GetBegin()` и определены операторы сравнения `operator ==` и `operator !=`. Это требуется для корректной работы основного алгоритма.

`DecodeOld` и `DecodeNew` по коду токена `TokenInfo` возвращает сам токен для вывода пользователю.

5.3.2 Архитектура и принцип работы

Опишем более подробно внутренее устройство библиотеки. Внутри базового класса `Tokenizer` определены так же два приватных виртуальных метода:

```
private:
    std::optional<TokenType> GetToken(std::string_view input, size_t pos)
    std::optional<TokenType> GetSymbol(std::string_view input, size_t pos)
```

Первый метод возвращает токен, который начинается в `input` с позиции `pos`. Второй же считывает символ, аналогично первому, `input` с позиции `pos`. Второй метод определяется сразу внутри базового класса, как метод, считывающий символ в кодировке UTF-8, если выбран такой режим парсера, либо просто символ с данной позиции, если парсер побайтовый. Оба метода в случае какой-либо ошибки при считывании (преждевременный конец `input`, или символ, не соотвествтующий заданному парсеру) возвращают `std::nullopt`.

Благодаря данным методам можно сразу реализовать методы `Decode`, как просто вызов `GetToken` с позиции `TokenInfo::GetBegin()`. Это сделано именно так, из-за того, что не для всех токенизаторов равенство в смысле токена обозначает равенство в смысле строк. Поэтому восстановление по присвоенному коду токена перестало быть однозначным, из-за чего и приходится хранить начало токена и просто брать его из `input`.

Рассмотрим, какие классы от него наследуются.

`class MapUsingTokenizers: public Tokenizer`. Данный класс так же являются промежуточным для других классов, так как в нем не определен метод `GetToken`. Он переопределяет метод `GetTokenInfos` и так же внедряет свои виртуальные методы:

```
private:
    bool IsEqual(TokenType lhs, TokenType rhs)
    size_t GetHash(TokenType token)
```

Дефолтно `IsEqual` просто вызывает `operator==`, а `GetHash` — функцию стандартной библиотеки `std::hash`. Данный класс нужен только для того, чтобы задать логику кодирования токенизатора. А состоит она в следующих шагах:

1. Достаем токен из файла, если он пустой - завершаем работу.
2. Получаем хэш токена посредством вызова метода `GetHash`.
3. Посредством хранящейся `std::unordered_map` проверяем, был ли уже токен с таким хэшом, если не было — добавляем наш токен в хэшмапу и задаем ему код, равный значению хэша, после чего переходим к шагу 1. Если не было — переходим к шагу 4
4. Проверяем, является ли токен с нашим значением хэша в хэшмапе равным нашему посредством метода `IsEqual`. Если является — говорим, что код токена равен хэшу и переходим к шагу 1. Иначе увеличиваем значение хэша на 1 и переходим к шагу 3.

`class SymbolTokenizer: public MapUsingTokenizers`. Данный токенизатор разбивает все данные на символы, в соответствии с парсером. Единственная функция, которую для него необходимо доопределить — это `GetToken`, но в данном случае это будет просто являться вызовом функции `GetSymbol`.

`class SymbolSplitTokenize<IsSplitter>: public MapUsingTokenizers`. Данный токенизатор разбивает все данные на токены, в соответствии с разделителями (сам разделитель является часть токена, после которого он находится). `IsSplitter` — структура, имеющая `bool operator()(TokenType symbol)`. Разделителем считается символ, удовлетворяющий данному оператору структуры `IsSplitter`.

Опредяляя структуры `struct IsWordSplitter` и `struct IsLineSplitter` мы получаем `WordTokenizer` и `LineTokenizer`, как `SymbolSplitTokenizer<IsWordSplitter>` и `SymbolSplitTokenizer<IsLineSplitter>` соответственно.

Для `WordTokenizer` разделителем являются все пробельные символы, а именно: `' '`, `\t`, `\n`, `\v`, `\f`, `\r`. Для `LineTokenizer` разделителем же является символ `\n`.

`class IgnoreAllSpaceTokenizer: public LineTokenizer`. Данный токенизатор разбивает все данные на строки. Причем считает, что две строки являются равными, если при удалении пробелов(`' '`, `\t`) из обеих строк они будут равны.

Для работы данного токенизатора достаточно лишь перегрузить у `LineTokenizer` методы `GetHash` и `IsEqual`. Тогда просто будем в `GetHash` удалять из токена все пробелы, после чего вызывать `std::hash`, а в `IsEqual` сравниванием, соотвествуя токенизатору.

`class IgnoreSpaceChangeTokenizer: public LineTokenizer`. Данный токенизатор разбивает все данные на строки. Причем считает, что две строки являются равными, если при замене всех непустых последовательностей пробелов(`' '`, `\t`) из обеих строк на один пробельный символ они будут равны.

Для работы данного токенизатора достаточно лишь перегрузить у `LineTokenizer` методы `GetHash` и `IsEqual`. Тогда просто будем в `GetHash` заменять в токене все пробельные последовательности на один пробел, после чего вызывать `std::hash`, а в `IsEqual` сравниванием, соотвествуя сказанному выше.

5.3.3 Обоснование и визуализация архитектурного решения

Несложно заметить, что в этой библиотеке активно используются концепции ООП, в частности, полиморфизм и наследование. Полиморфизм помогает отделить логику программы от интерфейса Tokenizer, в результате чего для использования нужных функций не нужно вникать в подробности реализации, а достаточно изучить пару строчек кода. А наследование помогает избавляться от дублирования кода, а также помогает гибко и быстро добавлять новый токенизатор. Как мы видели, для многих токенизаторов достаточно было определить лишь 1-2 метода, чтобы получить новый режим обработки. В качестве примера, рассмотрим новый токенизатор для csv(comma separated values). Для его добавление необходимо определить одну структуру `IsComma`. Продемонстрируем легкость добавления токенизатора, вместе с этим изобразив наглядно архитектуру библиотеки и процесс работы функций.

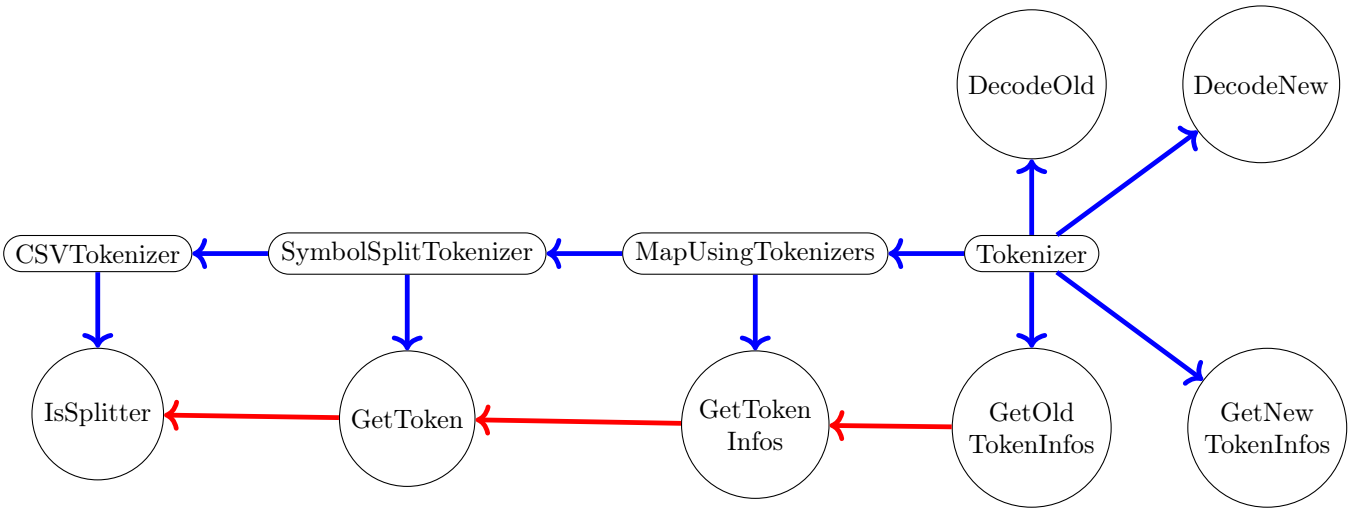


Диаграмма 1. Визуализация архитектуры токенизатора

Здесь функции(и структуры из одной функции) обведены кругами, а структуры — овалами. Синие стрелки показывают отношение наследования или владениями объектами. Красные стрелки обозначают использование функцией другой функции на примере вызова функции *GetNewTokenInfos*

5.4 Language Server

Одним из самых распространненых сценариев использования утилиты *diff* является сравнение 2 файлов, содержащих программный код. Обычно это сравнение происходит между старым и новым вариантом кода, различающимися незначительно. В этом случае стандартные алгоритмы *diff* могут выдавать очень непонятный для человека результат. Например, если заменить в строке `int` на `short`, то стандартные утилиты выведут всю строку, и пользователю придется тратить время на то, чтобы понять, что именно изменилось в этой строке. Использование пословного токенизатора отчасти решает эту проблему, но не является идеальным.
Рассмотрим следующий пример:
old file: `std::vector<int> a;`
new file: `std::set<long long> a;`
Нам, легко увидеть, что изменилось только две вещи - контейнер и тип хранимого значения. Пословный токенизатор, не вдаваясь в семантику языка, решит, что `std::set<long` и `long>` являются отдельными токенами и выведет всю строку в качестве различия. Это не тот результат, который мы хотим видеть. Для решения этой проблемы нужно уметь токенизировать программный код.

Существует много решений этой задачи. Два самых частоиспользуемых это *tmLanguage*[8] и *Language Server*[4]. Расскажем о них поподробнее.

5.4.1 tmLanguage

Стандартное решение для токенизации программного кода. Это часть текстового редактора *textmate*[7] с открытым исходном кодом. *tmLanguage* - набор конфигурационных файлов, основаннных на регулярных выражениях. Самая большая ценность этого решения - количество конфигурационных файлов - они есть практически для каждого языка программирования. Однако использовать эти файлы сложно, так как исходный код *textmate* сложен своими зависимостями, а сами конфигурационные файлы сложно устроены, что усложняет разработку токенизатора с нуля на основе этого решения.

5.4.2 Language Server

Language Server - это протокол взаимодействия между средой разработки и сервером, представленный Microsoft. Большинство актуальных на текущий момент сред разработки используют именно его для работы с файлами. Сервер является отдельной программой, которая работает с определенным язык программирования. IDE на основе протокола взаимодействует с сервером, отправляя запросы и получая результаты обработки файла. Мы решили выбрать этот вариант, поскольку он является более семантическим - сервер смотрит на файл точно также, как это делал бы компилятор, а также наиболее дружелюбным к внедрению. Недостатком *Language Server* перед *tmLanguages* является необходимость установки сервера под конкретный язык программирования. Впрочем, вполне вероятно, что у пользователя он уже установлен, если он пишет и сравнивает файлы на конкретном языке программирования.

5.4.3 Внедрение Language Server

Для взаимодействия с сервером мы написали отдельную библиотеку `LSCommunicator`. Так как сервер является отдельным процессом, для запуска, отправки и получения сообщения мы использовали библиотеку `Qt::Core`[6]. Обработку *json* файлов, которые используются для коммуникации с сервером, мы сделали с помощью библиотеки `nlohmann/json`[5].

Самым важным запросом к серверу для нас является *textDocument/semanticTokens/full*, ответ на который являет собой токенизированный файл в сжатом формате. Библиотека инкапсулирует в себе инициализацию сервера, технические уведомления, запрос, получение результата и его интерпретцию(декодирование).

Для пользовтеля библиотека предоставляет интерфейс в виде функции
`std::unordered_set<size_t> GetParseResult(const std::string& file_path)`
которая принимает путь до файла, а возвращает множество границ токенов. В это множество входят как начало токенов, так и их концы. Это связано с нашей реализацией токенизаторов - нам важны лишь границы токенов, а также с особенностью работы Language Server - он вовзвращает только те токены, которые ему удалось успешно обработать.

5.4.4 Демонстрация полученного результата

Демоснтрацию работы будем приводить на следующих 2 файлах:

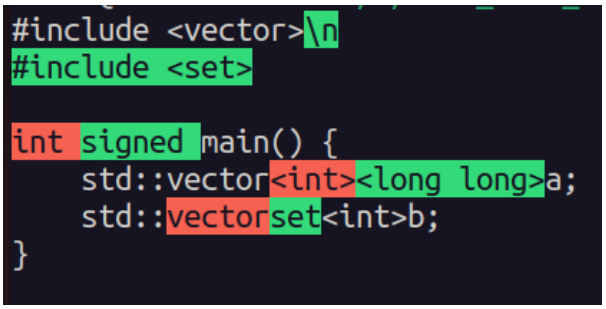
```cpp
#include <vector>

int main() {
    std::vector<int>a;
    std::vector<int>b;
}
```

old.cpp

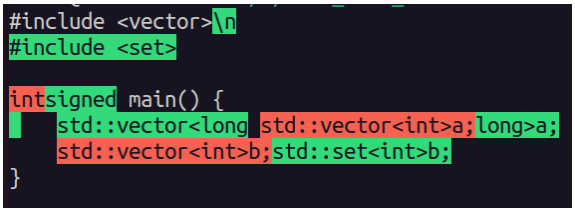
```cpp
#include <vector>
#include <set>

signed main() {
    std::vector<long long>a;
    std::set<int>b;
}
```

new.cpp

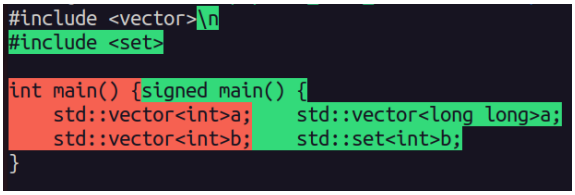
Приложим вывод трех токенизаторов для сравнения:



Semantic-code tokenizer



Word tokenizer



Line tokenizer

В первом случае хватает одного взгляда для того, чтобы понять, что изменилось в коде. Второй вариант превращает вывод в кашу из-за пробела в типе `long long`, третий же вариант вообще не учитывает семантику языка, из-за чего получается неинформативный вывод.

5.5 Алгоритм Майерса

Алгоритм Майерса выполняет задачу нахождения наибольшей общей подпоследовательности двух последовательностей токенов. В данном разделе будет приведено только общее описание алгоритма Майерса без доказательств оценок на время работы алгоритма и потребляемую им память, т.к. они приведены в цитируемой литературе [10].

5.5.1 Стандартный алгоритмп поиска НОП

Прежде, чем перейти к описанию алгоритма Майерса, стоит уделить внимание классическому алгоритму поиска наибольшей общей подпоследовательности (далее НОП) с квадратичной паматью от размера входных данных. В нём используется прием, называемый динамическим программированием с использованием двумермного массива `dp` размера $n \times m$, где значение в ячейке с индексами i, j равно длине НОП для префиксов переданных массивов длины i и j соотвественно. Здесь n и m — размеры входных последовательностей токенов A и B соответственно.

Тогда будем считать значения массива динамического программирования в порядке увеличения индексов i, j, чтобы узнать значение i-го и j-го элемента, в начале проверим, равны ли эти элементы в исходных массивах, и если это так, то возмьем максимум среди элементов `dp[i - 1][j - 1] + 1`, `dp[i - 1][j]` и `dp[i][j - 1]`. Если же элементы не равны, то будем брать максимум среди `dp[i - 1][j], dp[i][j - 1]`. Таким образом,

$$\texttt{dp[i][j]} = \begin{cases} \texttt{max(dp[i - 1][j - 1] + 1, dp[i - 1][j], dp[i][j - 1])}, & A_i = B_j \\ \texttt{max(dp[i - 1][j], dp[i][j - 1])}, & A_i \neq B_j \end{cases}$$

Рассмотрим то, каким образом получили значение элемента $dp_{i,j}$. Из формулы выше легко видеть, что значение $dp_{i,j}$ получено либо из элемента над ним с индексами $i-1$, j, либо из элемента слева от него с индексами i, $j-1$, либо из элемента по диагонали слева сверху с индексами $i-1$, $j-1$. Тогда, зная это, мы можем восстановить путь (в данном контексте это будет сама НОП) из элемента с индексами $0, 0$ в элемент n, m. При этом, элементы, в которые мы перешли по диагонали являются элементами из НОП. То есть, чтобы восстановить НОП, достаточно найти путь, при этом не обязательно поддерживать все значения массива dp с квадратичным использованием памяти. Будем называть такой путь оптимальным путем. Явно обозначим, что практический интерес рассмотрения только оптимального пути заключается в том, что это позволяет сократить использование памяти до линейного, что является одной из целей работы.

5.5.2 Поиск оптимального пути

Напоминание определения: обозначим за D то, сколько изменений вида удалить/вставить элемент нужно применить к первому массиву, чтобы получить второй. Другими словами, D — длина edit script. Тогда если L — длина НОП, то $D = n + m - 2L$.

Заметим, что на оптимальном пути количество не диагональных переходов равняется числу D. Предложение: давайте научимся находить для фиксированного k, возможно ли перейти из левого верхного угла (ячейки с индексами 0, 0) в правый нижний (ячейки с индексами n, m), если разрешено совершить ровно k не диагональных переходов.

Чтобы научиться такое считать, научимся считать элемент с наибольшой координатой i (он же будет элементом с наибольшой координатой j), который мы можем достичь на каждой диагонали, если было совершено равно k не диагональных переходов. Для каждой диагонали выполнен инвариант, что разность между координатами i и j на всех элементах этой диагонали сохраняется, тогда занумеруем все диагонали по разности координат i и j. Искомый оптимальный путь без затруднений считается для $k = 0$, так как у нас в таком случае запрещены какие-либо изменения (не диагональные переходы), а, значит, мы можем лишь ходить по диагонали, что означает, что рассматриваются только подряд идущие равные элементы на префиксе каждого из двух массивов. Таким образом, вычисление оптимального пути для $k = 0$ сводится к следующему: будем увеличивать i, исходно равный 0, пока i-ые элементы у двух массивов совпадают. Как только найдется такое i, что элементы двух массивов на i-й позиции не совпадают, то это означает, что оптимальный путь для $k = 0$ найдет, и все остальные диагонали не достижимы.

Научимся узнавать ответ для $k+1$, если ответ для k известен. Пусть мы посчитали ответ для всех диагоналей при k изменениях (не диагональных переходах), тогда чтобы посчитать ответ для какой-либо диагонали для $k + 1$. Для этого возмьем максимум из ответа (ответ — ячейка, в которую приводит оптимальный путь) для соседней диагонали с меньшим номером (нумерация по разности координат элементов на диагонали) при k изменениях, увеличив его первую координату на 1, и ответа соседней диагонали с большим индексом при k изменениях, увеличив его вторую координату на 1, после чего будем совершать диагональные переходы до тех пор, пока элементы равны. Данный шаг работает за $O((n + m)D)$, однако, чтобы восстановить путь нам потребуется порядка $O((n+m)D)$ памяти, так как нам надо будет для каждой диагонали хранить, какой был переход, и это будет необходимо делать на каждом шаге.

Однако заметим, что для того, чтобы посчитать ответ для следующего количества изменений, нужно знать только ответ для значения перед ним, это означает, к примеру, что для того, чтобы найти значение D нам нужно лишь $O(n)$ памяти. Тем самым, зная, как был получены значения в последнем слое, мы сможем узнать, сколькими диагональными переходами заканчивается оптмиальный путь.

5.5.3 Оптимизация по памяти

Применим метод разделяй и властвуй, и найдем среднюю диагональную часть пути (возможно пустую) то есть такие значения i, j, k, что значения на отрезке $[i, i + k]$ первого массива и на отрезке $[j, j + k]$ второго массива равны, при этом, чтобы получить из префикса длины i первого массива префикс длины j второго массива понадобится $\left\lfloor \frac{D}{2} \right\rfloor$ операций изменения (удалений или вставок), и, аналогично, чтобы получить из суффикса длины $n - i - k$ первого массива суффикс длины $m - i - j$ второго массива потребуется $\left\lceil \frac{D}{2} \right\rceil$ операций. Чтобы найти эту часть, будем поддерживать те же величины (пути с наибольшими координатами) по диагоналям из точки $(0, 0)$ при фиксированном количестве изменений в сторону увеличения координат, а также из точки (n, m) в сторону уменьшения координат. Будем пересчитывать данные значения по мере увеличения количества изменений, пока не найдется такая диагональ, у которой максимальные пути в прямую и в обратную сторону пересекаются. Найдя эту диагональ запомним последнюю часть этого пути по диагонали. Пусть она начинается в точке (i, j) и имеет длину k, тогда отрезок первого массива $[i, i + k]$ будет входить в НОП, при этом элементы, входящие в НОП перед ним, будут НОП для подотрезков $[0, i-1]$, $[0, j-1]$ первого и второго массивов соответственно, а элементы после него будут НОП для подотрезков $[i + k + 1, n]$, $[j + k + 1, m]$ первого и второго массивов. Тогда найдем ответы для этих подотрезков рекурсивно.

Тогда мы получили рекурсию, время работы которой вычисляется как:

$$T(n + m, D) = \begin{cases} T\left(k_1, \frac{D}{2}\right) + T\left(k_2, \frac{D}{2}\right) + O(nD), & D > 1 \\ O(n + m), & D \leqslant 1 \end{cases}$$

,

где $k_1 + k_2 \leqslant n + m$, тогда по мастер теореме получаем, что $T(n + m, D) = O((n + m)D)$, тем самым мы получили алгоритм, который за время $O(nD)$ и $O(n)$ памяти находит НОП.

5.5.4 Теоретические оптимизации

Время работы данного алгоритма можно улучшить, используя суффиксное дерево. Построив его на двух исходных массивах токенов, мы сможем находить наибольшую диагональ из фиксированных i, j, то есть наибольшее k, что подмножества элементов на отрезке $[i,\ i+k]$ первого массива и на отрезке $[j,\ j+k]$ второго массива элементов равны за $O(1)$. Мы это сможем сделать, применив алгоритм поиска наибольшего общего предка в дереве (LCA). Тогда итоговое время работы будет $O((n + m) \log(n + m) + D^2)$, так как рекурсия

теперь работает за $O(D^2)$, а время на построение суффиксного дерева и структуры LCA составляет $O((n + m)\log(n + m))$.

Однако такой подход нет смысла применять на практике, так как он имеет достаточно большую константу в силу особенностей реализации суффиксного дерева. При этом ожидаемое время работы алгоритма состовляет $O(n+m+D^2)$, что уже является лучшей ассимптотикой, по сравнению с описанной выше версией алгоритма Майерса.

5.6 Визуализация

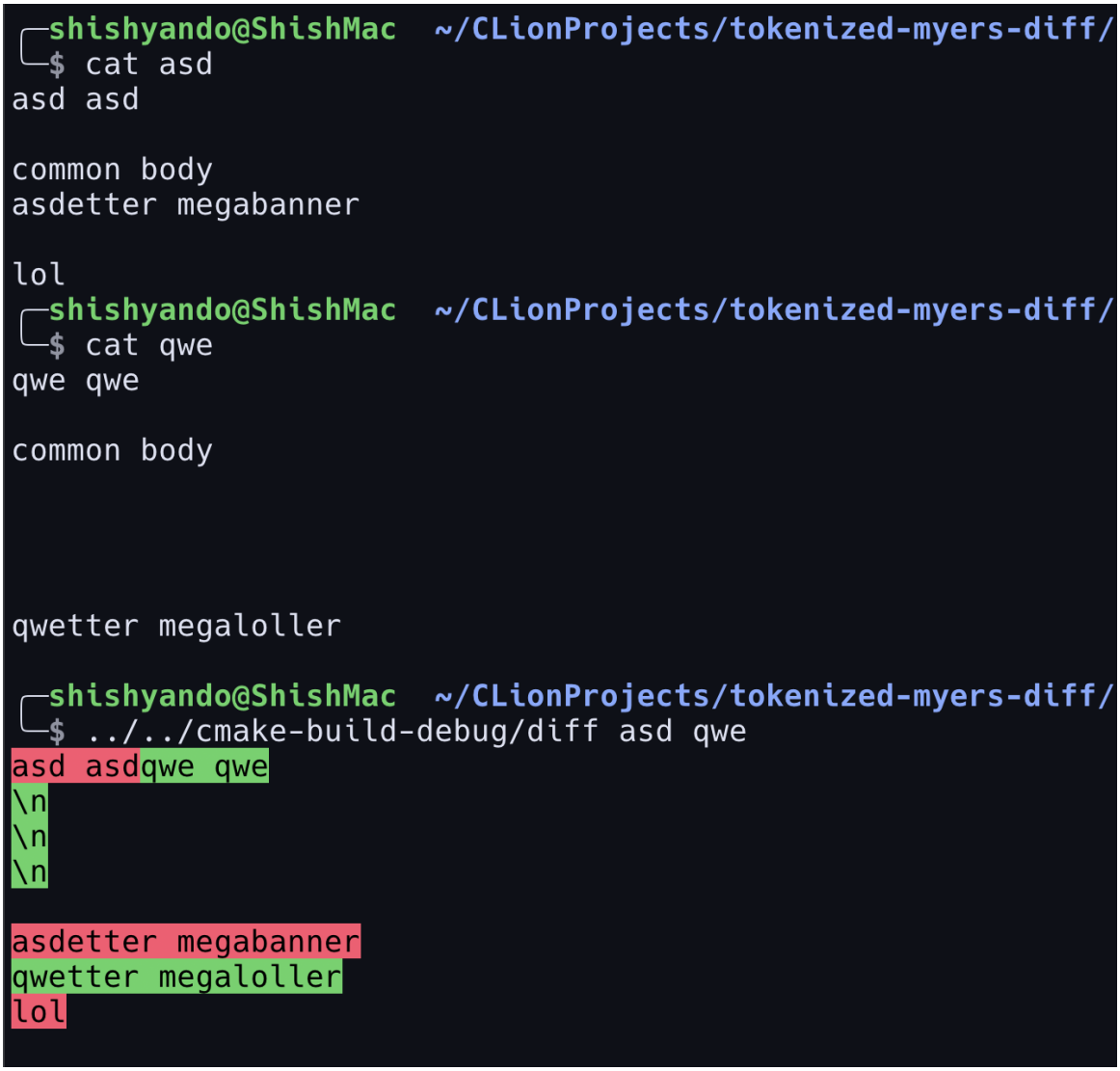
После того, как в программе отрабатывает токенизация и сам алгоритм, мы получаем кратчайший Edit Script. Следует понимать, что на этапе вывода разницы двух файлов на стандартный поток вывода у нашей программы есть только отображенные в память файлы и внутренняя информация токенизаторов. Возникает следующая задача: по Edit Script восстанавливать diff. Это задача разбивается на две подзадачи:

1. Найти множество токенов для отображения с разметкой владельца токена. Напомним, что для каждого токена может быть только один из трех сценариев: токен есть только в старом файле (то есть нужно отобразить его удаление), токен есть только в новом файле (то есть нужно отобразить его добавление), токен есть в обоих файлах (то есть этот токен не входит в diff)

2. Наиболее наглядно отобразить каждый токен в зависимости от параметров вывода и владельца токена.

Сначала научимся решать первую подзадачу. Edit Script содержит в себе последовательный список всех необходимых замен, причем каждая замена содержит в себе ровно по одному подотрезку последовательных токенов на каждый из двух файлов, для которых мы хотим построить Diff. Заметим, что внутри каждой замены порядковые номера токенов строго возрастают, это же условие верно для любых замен. Тогда будем поддерживать два указателя: один указатель идет по токенам из старого файла, а другой указатель идет по токенам из нового файла. Заметим, что условие на порядковые номера токенов внутри замен гарантируют монотонность движения каждого из двух указателей. Тогда нам достаточно просто последовательно рассмотреть все токены этими двумя указателями и за одну итерацию определять владельца токена. Для этого заметим следующий факт: если хотя бы один указатель соответствует началу замены, то второй указатель тоже соответствует началу замену в втором файле. Тогда мы можем однозначно определить владельца каждого токена в рамках этой замены: все токены в подотрезке замены старого файла имеют в качестве своего владельца старый файл, а все токены в подотрезке замены нового файла имеют в качестве своего владельца новый файл. После обработки замены мы приходим либо в состояние конца обоих файлов, либо в состояние общей части, где у каждого токена владельцем будут оба файла. В таком случае мы увеличиваем сразу оба указателя, потому что они указывают на общие токены двух файлов, до тех пор, пока не наткнемся на очередную замену. В итоге имеем, что для каждого токена в каждом из двух файлов у нас определен владелец, и можно вывести информацию об этом пользователю.

Вывод информации про владельцев токенов — вторая подзадача. На вывод влияет конфигурация, заданная параметрами командной строки при запуске программы. Самый человекочитаемый вариант вывода — цветной, однако он не всегда доступен, а также иногда пользователь хочет более гибко подстроить под себя вывод на примере конкретного текста. Рассмотрим на примерах, какие режимы работы вывода есть и для чего они могут быть полезны.



На этом примере видно самый простой вариант работы утилиты, построчный дифф без вывода общей части файлов. В таком формате не выводится общая часть файлов, то есть токены, у которых владельцем являются оба файла, а также вывод в цветном режиме. В таком режиме мы выбрали отображение цветного фона через контрольные ASCII символы для терминала.

На данном скриншоте изображен режим с "сырым"отображением диффа, то есть без использования цветов. Этот режим может быть полезен, например, когда дифф записывается внутри какого-то скрипта в файл и не отображается напрямую в терминале пользователя. В этом режиме по умолчанию `[+]` означает новый текст, а `[-]` означает удаленный старый.

```
shishyando@ShishMac  ~/CLionProjects/tokenized-myers-diff/
$ ../../cmake-build-debug/diff --raw asd qwe -c
[-]asd asd[+]qwe qwe

common body[+]
[+]
[+]

[-]asdetter megabanner
[+]qwetter megaloller
[-]lol
```

```
shishyando@ShishMac  ~/CLionProjects/tokenized-myers-diff/
$ ../../cmake-build-debug/diff asd qwe -c
asd asdqwe qwe

common body\n
\n
\n

asdetter megabanner
qwetter megaloller
lol
```

```
shishyando@ShishMac  ~/CLionProjects/tokenized-myers-diff/tests/samples ‹node-›  ‹› (change-tokenizers*)
$ ../../cmake-build-debug/diff --raw --new-prefix=" <add>: " --old-prefix=" <del>: " asd qwe
 <del>: asd asd <add>: qwe qwe
 <add>:
 <add>:
 <add>:

 <del>: asdetter megabanner
 <add>: qwetter megaloller
 <del>: lol
```

В данном режиме дополнительно отображается общая часть файлов. По умолчанию она отображается без особого префикса и без особого цветового выделения в цветном режиме. Для режима без цветов через параметры командной строки можно также передать новый префикс для общей части файлов, который будет выводиться по тому же принципу, что и `[+]` и `[-]`. Последние два тоже можно менять соответствующими параметрами.

```
shishyando@ShishMac  ~/CLionProjects/tokenized-myers-diff/
$ ../../cmake-build-debug/diff asd qwe --show-pos
0:0
asd asdqwe qwe

20:20
\n
\n
\n


21:24
asdetter megabanner

41:25
qwetter megaloller

42:44
lol
```

Для вывода диффа поддерживается режим с выводом позиций в файлах, на которых найдено различие. Этот режим основан на том, что каждая замена означает вырезание подотрезка текста старого файла с последующей заменой на подотрезок текста нового файла. На этапе вывода мы знаем позицию начала каждого выводимого токена в его файле, поэтому перед выводом замены мы можем вывести то место, в котором она происходит. Местом здесь является просто порядковый номер символа, с которого начинается токен. Поскольку начало токена для каждого из двух файлов может лежать в разных местах, через двоеточие выводится сначала позиция в старом файле, потом в новом.

Еще один параметр позволяет более подробно рассмотреть служебные символы не только в области замен, но и в общей части файлов. Это может быть полезно, когда нужно четко понимать расположение символов перевода строки. При этом они будут выделяться более темным блеклым оттенком, чтобы меньше отвлекать пользователя и не сливаться с выводом общей части файлов.

```
shishyando@ShishMac  ~/CLionProjects/tokenized-myers-diff/tests/samples
$ ../../cmake-build-debug/diff -c --show-common-newlines asd qwe
asd asdqwe qwe\n
\n
common body\n
\n
\n
\n
asdetter megabanner\n
qwetter megalolller\n
lol\n
```

Режимы вывода можно всячески комбинировать. Под вывод создан отдельный класс со своими методами и настройками. Все настройки вывода вынесены в структуру, через которую происходит конфигурация. Эта структура заполняется в `main` из аргументов командной строки. Это гибкий подход к настройке вывода и через него можно легко добавлять новый функционал.

5.7 Интеграция

Интегрировать нашу программу можно в любую другую программу, где в том или ином виде пользователь хочет сравнивать тексты. Также можно полностью заменить другую программу, если она предоставляет те же основные функции, что и наша. Рассмотрим две наиболее популярные утилиты, которыми пользуются миллионы людей каждый день.

1. Стандартная программа `diff` [9] в операционной системе GNU/Linux. Пользователь может в любой директории запустить сравнение файлов, набрав команду `diff file1 file2`. Не будем заменять или модифицировать стандартную программу, вместо этого позволим пользоваться нашей программой с теми же удобствами. Для этого сначала соберём бинарный файл нашего проекта, назовём его `myers_diff` и положим в удобную директорию. Теперь добавим выбранную директорию `<dir>` в переменную окружения `$PATH` командой `export PATH="<dir>:$PATH"`. чтобы это изменение осталось навсегда, добавим эту команду в конец файла `.bashrc`. Теперь пользователь может вызывать нашу программу из любого места системы, написав `myers_diff file1 file2`.

2. Инструмент `git diff` [3] в системе контроля версий `Git`. С помощью данной утилиты можно сравнивать файлы из разных веток и коммитов, но нас интересует только базовая возможность вывода разницы между файлами в рабочей директории (опция `--no-index`). Система `Git` написана на языке `C`, наша же программа – на языке `C++`. Из-за сложных зависимостей и устаревшего кода в `Git`, скомпилировать её вместе с нашей программой не представляется возможным. Пойдём другим путём. Отдельно скомпилируем наш проект и положим бинарник в папку с исходниками Гит. Далее, внимательно изучив код Гита, находим место, где парсится аргумент `--no-index`, и добавляем туда парсинг нового аргумента `--myers`, который будет запускать бинарник нашего проекта посредством вызова функции `execvp`. После этого собираем Гит уже с нашей вставкой кода. Теперь пользователю доступна возможность использовать нашу утилиту: `git diff --myers file1 file2`.

```
440
441     /* Check for using myers-diff */
442     if (!strcmp(argv[1], "--myers")) {
443         execvp("./myers-diff", argv + 1);
444     }
445
```

5.8 Тестирование

Тестирование представляет большой интерес, потому что это лучший способ проверить эффективность реализованного алгоритма на практике. Для end to end тестирования был выбран текст первых двух томов романа "Война и Мир"Л.Н. Толстого. Это очень показательный тест, потому что он достаточно большой (в файле 6700 строк и 1466823 символов в кодировке UTF-8, а размер файла 2609729 байт, что примерно равно 2.5МБ), чтобы проверить как потребление памяти запущенной программой, так и время её работы. Кроме того, текст произведения использует символы кодировки UTF-8. Таким образом, проверяются все компоненты программы: посимвольное чтение UTF-8 символов, разбиение на токены (и слова, и символы, и строки), работа самого алгоритма и edit script наименьшей длины в человекочитаемом формате. Для тестирования были использованы два файла с одинаковым содержимым (два тома романа), но в одном из них все точки заменены на символ `#`. Всего заменённых символов 13338. Заметим, что это число в точности равно D для токенизатора по символам и для токенизатора по словам, потому что для каждой замененной точки есть слово, которое предшествует этой точке, то есть нужно будет заменить ровно 13338 слов. Если же рассматривать строки, то строк для замены будет меньше, чем общее число строк, потому что в одной строке может встречаться несколько точек. В итоге строк для замены будет 5325. Время исполнения программы замерялось с помощью утилиты `hyperfine`, в которой проводилось 10 запусков для прогрева и 10 запусков для замера среднего времени работы:

Тип токенизатора	Суммарное количество токенов	Минимальный edit script	Время работы (сек.)
line	25288	5326	0.142
word	956150	13320	1.055
symbol	2933646	13328	1.217
ignore-all-space	25288	5326	0.156
ignore-space-change	25288	5326	0.158

Таблица 1: Результаты тестирования myers-diff на Войне и Мире

При анализе результатов тестирования становится понятно, что выбор типа токенизатора имеет огромное значение при вычислении diff. Заметим, что для построчного токенизатора наибольшая константа появляется при использовании хеш-таблицы с большими строками в качестве ключей. Таким образом, хоть токенов и мало, алгоритм всё равно работает почти секунду. В случае со словами достигается приемлемый результат по скорости при достаточно высокой точности. За счет эффективности алгоритма, его можно применять на таком большом тексте и в посимвольном режиме для максимальной точности, и это всё ещё занимает не так много времени.

Для сравнения времени работы запустим стандартную shell утилиту `diff`. Заметим, что наш алгоритм гарантирует минимальность полученного диффа (то есть минимальность размера edit script), в то время как стандартная утилита по умолчанию этого не делает. Чтобы заставить его вести себя так, как мы хотим, пропишем ему в опциях флаг `--minimal`. Так же мы не сможем сравнить его со всеми нашеми токенизаторами, так как в нем нет пословного и посимвольного режима (что говорит о большей гибкости нашей реализации), поэтому сравним лишь с теми, которые у него есть. Для замера времени работы вновь будем пользоваться утилитой `hyperfine` с 10 запусками прогрева и 10 запусками замера:

Тип токенизатора	Время работы diff (сек.)	Время работы myers-diff (сек.)
line	0.368	0.142
ignore-all-space	0.369	0.156
ignore-space-change	0.369	0.158

Таблица 2: Результаты тестирования myers-diff и diff –minimal на Войне и Мире

Как мы видим, наш алгоритм выигрывает у стандартной утилиты почти в 2.5 раза. Однако давайте еще больше усложним задачу. Сделаем два файла, состоящих из 100'000 строк. Первый файл будет состоять только из строк `'a'`, а второй будет случайно заполнен строками `'a'` и `'b'`. Тем самым размер итогового

edit-script будет примерно равен 50'000. Заодно повторим то же самое для 0 строк, дабы проверить, как алгоритм справляется с маленькими файлами. Вновь воспользуемся `hyperfine` и замеряем время:

Время работы diff (сек.)	Время работы myers-diff (сек.)
51.323	11.536

Таблица 3: Результаты тестирования myers-diff и diff –minimal на файле размера 100'000 строк

Со столь большим файлом наш алгоритм стал показывать время почти в 5 раз, что является значительным выигрышом. Однако мы все еще не проверили время для сравнительно маленьких файлов, например 100-200 строк. Здесь уже время является трудно замеримым, так как оно составляет меньше, чем 5мс, из-за чего утилита по измерению времени дает большую погрешность. Однако это означает, что на маленьких файлах обе версии справляются за очень быстрое, практически неразличимое время. Из этого следует, что алгоритм непроигрывает никогда, а с ростом файла, еще и начинает сильно выигрывать.

6 Заключение

Подведём итоги. Почти все поставленные цели курсовой работы были выполнены: мы реализовали алгоритм Майерса с токенизацией на языке `C++`; добавили широкий спектр параметров запуска программы, которые может выставлять пользователь; добавили 5 базовых токенизаторов, а также внедрили токенизатор программного кода, основанный на *Language Server*; написали легко расширяемый код токенизации; реализовали красивую и многофункциональную визуализацию результата работы программы.

Нам также удалось при некоторых входных данных сделать программу более эффективной по времени, чем у аналогичных утилит.

Выбранная нами тема очень актуальна, т.к. IT-сектор продолжает активно развиваться во всём мире. Люди, и в особенности программисты, постоянно сталкиваются с потребностью в сравнении файлов посредством утилит, удобном визуализаторе разницы. Наш проект можно развивать дальше. Например, добавить больше различных токенизаторов на основе анализа кода программ различных языков программирования. Как мы показали выше, процесс внедрения новой функциональности является легким и быстрым.

Таким образом, мы успешно завершили свой программный проект и узнали много нового в процессе его реализации.

Список литературы

[1] *Antlr*. URL: https://www.antlr.org/.

[2] *argparse single-header library*. URL: https://github.com/p-ranav/argparse.

[3] *Git diff Manual*. URL: https://git.github.io/htmldocs/git-diff.html.

[4] *Language Server Protocol*. URL: https://microsoft.github.io/language-server-protocol/.

[5] *nlohmann/json library*. URL: https://json.nlohmann.me/.

[6] *Qt::Core library*. URL: https://doc.qt.io/qt-6/qtcore-index.html.

[7] *TextMate source code*. URL: https://github.com/textmate/textmate.

[8] *tmLanguages description*. URL: https://macromates.com/manual/en/language_grammars.

[9] Paul Eggert David MacKenzie and Richard Stallman. *Comparing and Merging Files*. for Diffutils 3.10 and patch 2.5.4. Free Software Foundation, 2023. URL: https://www.gnu.org/software/diffutils/manual/diffutils.pdf.

[10] E. Myers. *An O(ND) difference algorithm and its variations*. Algorithmica 1, 251–266. Department of Computer Science, University of Arizona, Tucson, AZ 85721, U.S.A., 1986. URL: https://doi.org/10.1007/BF01840446.