Санкт-Петербургский Политехнический Университет Петра Великого
Институт кибербезопасности и защиты информации

КУРСОВАЯ РАБОТА

Оптимизация приложения для поиска заимствований в тексте

по дисциплине «Языки программирования»

Выполнили
студент гр. №4851003/00002 Касимов Т.А.

<подпись>

Выполнили
студент гр. №4851003/00002 Калугина А.П.

<подпись>

Преподаватель Малышев Е.В.

<подпись>

«__» _____ 202__г.

Санкт-Петербург
2021

СОДЕРЖАНИЕ

Введение

Оптимизация кода – усовершенствование программы с помощью различных методов преобразования кода ради улучшения его характеристик и повышения эффективности. Среди ее целей можно указать уменьшения объема кода, объема оперативной памяти, используемой программой, ускорение работы программы, уменьшение количества операций ввода вывода (программа становится более автоматизированной). Существуют различные оптимизации: алгоритмические, машинно-независимые, оптимизации переходов, машинно-зависимые, оптимизации на уровне ассемблера и внутри процессора и другие.

Объектом исследования является неоптимизированный программный код раскрашивания карты с помощью минимального количества цветов.

Предметом исследования является код алгоритма раскрашивания карты, реализованный в программе.

Глава 1. Профилирование исходной программы

Перед приступлением к оптимизации программы было выполнено ее профилирование. Время работы исходной программы в зависимости от объемов файлов приведено в таблице 1.

Таблица 1 – Зависимость времени работы программы от объема входных файлов

Объем первого файла	Время работы программы
До 90 Кб	<0,044 сек
90-220 Кб	0,044-0,079 сек
220-330 Кб	0,079-0,096 сек
330-510 Кб	0,096-0,15 сек
510-710 Кб	0,15-0,209 сек
0,71-1,2 Мб	0,209-0,334 сек
1,2-1,56 Мб	0,334-0,402 сек
1,56-2,02 Мб	0,402-0,521 сек
2,02-2,29 Мб	0,521-0,587 сек
2,29-2,86 Мб	0,587-0,724 сек
2,86-3,69 Мб	0,724-0,875 сек
3,69-4,87 Мб	0,875-1,263 сек
4,87-5,28 Мб	1,263-1,318 сек

На рисунке 1 визуализированы данные из таблицы 1.

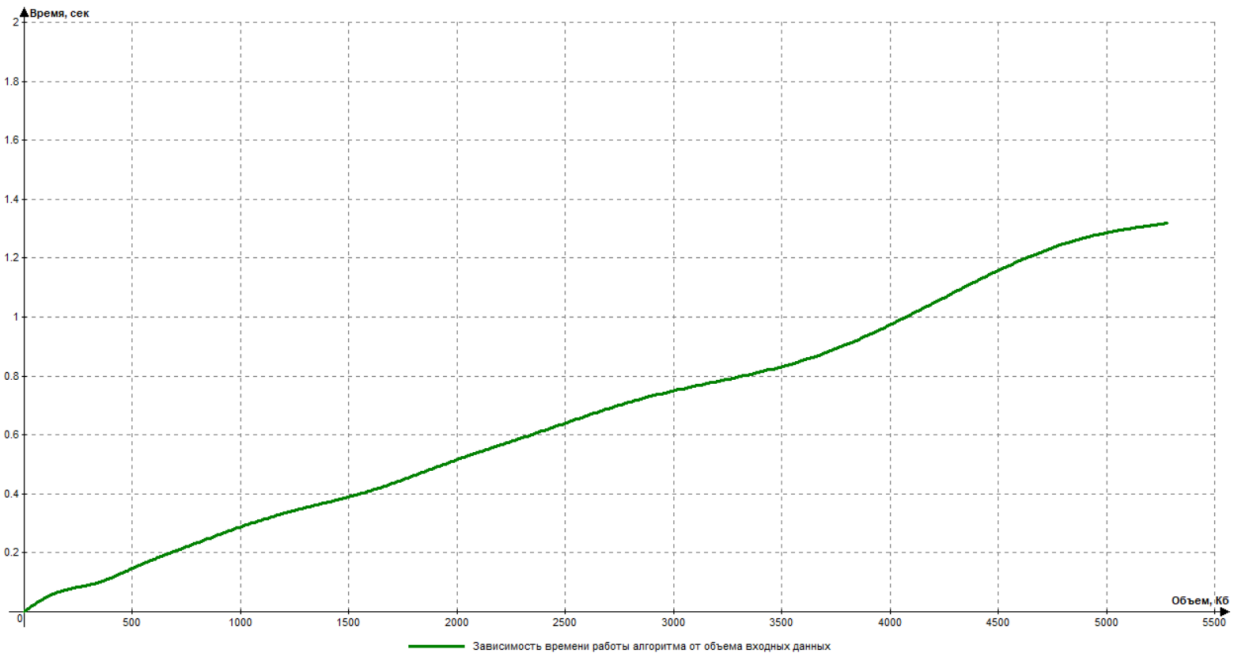

Рисунок 1 – Зависимость времени работы программы от объема входных файлов

Также была оценена сложность алгоритма:

$$O(N + height + const + N + v * 4^{size} + const + const^2 + 2N + cnt + del.cnt + cnt +$$
$$+(1 + cnt)(cnt + cnt + cnt^2 + cnt + cnt + cnt + cnt + cnt) + N + N + const + height) =$$
$$= O(3const + const^2 + 2height + 6N + v * 4^{size} + 7cnt + 8cnt^2 + cnt^3 + del.cnt) =$$
$$= O(2height + 6N + v * 4^{size} + 7cnt + 8cnt^2 + cnt^3 + del.cnt)$$

Разберем слагаемые по порядку:

- N – чтение файла
- Height – создание матрицы файла, нужно выделить память height раз, где height – высота картинки входного файла
- Const (500) – инициализация массива вершин графа, который будет построен на основании входной картинки
- $N + v * 4^{size}$ – обход графа в ширину для закрашивания его областей в разные цвета (из вершины можно пойти в 4 стороны), данный алгоритм будет запущен из каждой вершины, но начнет работать только из незакрашенных областей (их v штук) и в каждой области рекурсия сработает не более size (для каждой области size свой собственный) раз
- Const – выделение памяти под матрицу смежности

- $const^2$ – инициализация матрицы смежности
- 2N – поиск соседей по горизонтали, а затем по вертикали, для этого проходится вся картинка
- Cnt – счет количества вершин
- $del.cnt$ – удаление лишних связей в матрице смежности, в лучшем случае 0, в худшем $(4cnt + cnt^2)cnt$, так как придется удалять связи с четырех сторон для каждой
- Cnt – вывод в .log файл матрицы смежности
- $(1+cnt)(cnt + cnt + cnt^2 + cnt + cnt + cnt + cnt + cnt)$ – от всех вершин происходит раскрашивание дубликата для поиска минимального количества цветов, после чего самый выгодный запускается еще раз для того, чтобы подготовить карту к покраске
- N – раскрашивание карты
- N – запись полученной карты в новый файл
- Const – освобождение памяти от матрицы смежности
- Height – освобождение памяти от матрицы карты

Глава 2. Оптимизация программы

2.1. Оптимизация на уровне алгоритма

Из полученной в предыдущем параграфе формулы можно видеть, что двумя основными факторами времени работы программы являются раскрашивание карты в разные цвета и поиск наилучшего варианта раскраски карты. Также поиск соседей занимает немало времени, так как требует рассмотреть всю картинку дважды побайтого – если размер матрицы Мегабайты, то это большое количество действий.

Раскрашивание карты в разные цвета производится с помощью классического алгоритма обхода графа в ширину, этот алгоритм является рекурсивным, что будет работать довольно долго. Но придумать иную реализацию данного фрагмента программы, которая бы дала выигрыш по времени, не удалось, в связи с чем взялся акцент на алгоритм поиска наилучшего варианта раскраски карты и поиска соседей.

При раскраске, начав с неправильной вершины, может потребоваться больше цветов, чем это реально необходимо. При этом, любую карту можно раскрасить не более, чем 4 цветами. Это значит, что, начав с неправильной вершины и использовав более 4 цветов, можно не продолжать раскрашивать карту данным методом, так как он неправильный. Таким образом в большом графе много путей обрубятся, не завершив свой путь, что сократит время работы данной функции.

Если бы карту просто требовалось раскрасить разными цветами, то можно было бы при нахождении способа, в котором используются не более 4 цветов, сразу останавливать поиск и использовать его, в таком случае лучшим вариантом был бы сразу найденный способ – один раз рассмотреть матрицу смежности.

Но такое решение не соответствовало бы требованиям, поэтому был выбран вариант с отсечением заведомо неверных вариантов.

Для поиска соседей в исходной коде была написана одна функция для обхода в ширину и глубину, из-за чего там постоянно использовались уточнения того, какой вариант сейчас используется – поиск соседей в ширину или глубину. Множество проверок делались в циклах, что сильно увеличивало количество действий. Было принято решение отказаться от всех этих проверок для увеличения быстродействия – исходная функция была заменена на 2 новые – отдельная функция для поиска в ширину и отдельная для поиска в глубину.

Сложность модифицированного кода в целом осталась такая же, за исключением того, что появились лучшие и худшие случаи для поиска раскраски – в худшем все раскраски сделают полное раскрашивание, в лучшем все заходы, кроме одного, раскрасят 4 зоны в разные цвета и потребует еще один цвет, после чего программа закончит окрашивание, так как требуемое количество больше 5. Также в поиске соседей в циклах делалось по 2 сравнения для выбора варианта, что в итоге добавляло примерно 2N действий – модифицированный код не делает эти действия.

Данная оптимизация дала улучшение по времени в 2.5-5%. Наилучший процент ускорения наблюдается на картинках с большим количеством зон – раскрашивание зон при поиске лучшего варианта раскраски часто будет требовать более 4 цветов, поэтому часто будет происходить обрывание функции раскраски на середине работы.

2.2. Машинно-независимая оптимизация

Был оптимизирован код с применением различных ходов. Ниже ходы рассмотрены подробнее.

Была осуществлена оптимизация замена переменной. Исходный и оптимизированный коды представлены ниже.

Исходный:

```
for (int i = 0; i < Height; ++i) {
  for (int j = 0; j < Width; ++j) {
```

```
     if (canvas[i][j] == 0) {//для белых пикселей
        if ((i * j) % (H * W / 40) == 0) {
           printf("\r%d%%", 2 + (int)((40 * i * j) / (H * W)));
        }
```

…

```
for (int i = 0; i < size + 2; ++i) {
        M[i] = (uc*)calloc(sizeof(uc), size + 2);
        if (M[i] == NULL) return error(0);
    }
    for (int i = 0; i < size + 2; ++i) {
        for (int j = 0; j < size + 2; ++j) {
            M[i][j] = 0;
        }
    }
```

…

```
for (int i = H - 1; i >= 0; --i) {
  for (int j = 0; j < W; ++j) {
    if (H * W > 40 && (H - i) * j % (H * W / 40) == 0) {
      printf("\r%d%%", 60 + (int)((40 * (H - i) * j) / H * W));
```

Оптимизированный:

```
int del = H * W / 40;
int del_2 = H * W;
for (int i = 0; i < Height; ++i) {
  for (int j = 0; j < Width; ++j) {
    if (canvas[i][j] == 0) {//для белых пикселей
       if ((i * j) % del == 0) {
         printf("\r%d%%", 2 + (int)((40 * i * j) / del_2));
       }
```

…

```
int new_size = size + 2;
for (int i = 0; i < new_size; ++i) {
    M[i] = (uc*)calloc(sizeof(uc), new_size);
    if (M[i] == NULL) return error(0);
}
for (int i = 0; i < new_size; ++i) {
    for (int j = 0; j < new_size; ++j) {
        M[i][j] = 0;
    }
}
```

…

```
int multy = H * W;
int del = H * W / 40;
for (int i = H - 1; i >= 0; --i) {
  for (int j = 0; j < W; ++j) {
    if (multy > 40 && (H - i) * j % del == 0) {
      printf("\r%d%%", 60 + (int)((40 * (H - i) * j) / multy));
```

В исходном коде много раз высчитывалось значение с помощью действий умножение/вычитание. В оптимизированной версии все это заменено на заранее высчитанное значение, которое используется по ходу кода, на картинках размером 5Мб это экономит более 5 млн действий, так как значение высчитывалось в каждом проходе цикла, а цикл проходил всю картинку.

Также была произведена оптимизация развертка цикла. Ниже представлены исходный и оптимизированный коды.

Исходный:

```
for (int i = 0; i < Height; i++) {
        canvas[i] = (uc*)calloc(sizeof(uc), Width);
        if (canvas[i] == NULL) return error(0);
    }
```

…

Оптимизированный:

```
if (H % 2 == 1)
{
    canvas[0] = (uc*)calloc(sizeof(uc), Width);
    if (canvas[0] == NULL) return error(0);
    for (int i = 1; i < Height; i+=2) {
        canvas[i] = (uc*)calloc(sizeof(uc), Width);
        if (canvas[i] == NULL) return error(0);
        canvas[i+1] = (uc*)calloc(sizeof(uc), Width);
        if (canvas[i+1] == NULL) return error(0);
    }
}
else
{
    for (int i = 0; i < Height; i+=2) {
        canvas[i] = (uc*)calloc(sizeof(uc), Width);
        if (canvas[i] == NULL) return error(0);
        canvas[i + 1] = (uc*)calloc(sizeof(uc), Width);
        if (canvas[i + 1] == NULL) return error(0);
    }
 }
```

Основная идея данной оптимизации – увеличение линейного участка кода, что здесь и происходит. Но появляется риск выйти за пределы массива. Эта проблема была решена проверкой кратности высоты 2. Если кратно, то можно спокойно заполнять по 2 ячейки сразу, иначе заполняется сначала первая, а потом остается кратное количество и оно спокойно заполняется.

Далее было произведено объединение циклов. Ниже представлены исходный и оптимизированный коды.

Исходный:

```
int new_size = size + 2;
for (int i = 0; i < new_size; ++i) {
    M[i] = (uc*)calloc(sizeof(uc), new_size);
    if (M[i] == NULL) return error(0);
}
for (int i = 0; i < new_size; ++i) {
    for (int j = 0; j < new_size; ++j) {
        M[i][j] = 0;
    }
}
```

Оптимизированный:

```
for (int i = 0; i < new_size; ++i) {
    M[i] = (uc*)calloc(sizeof(uc), new_size);
    if (M[i] == NULL) return error(0);
    for (int j = 0; j < new_size; ++j) {
        M[i][j] = 0;
    }
 }
```

Данная оптимизация позволяет не запускать циклы с одними и теми же условиями, но с разными блоками дважды, а сделать все за один проход.

После проведения машинно-независимой оптимизации была снята новая зависимость времени выполнения программы от объема входных данных. Результаты представлены в таблице 2.

Таблица 2 – Зависимость времени работы программы от объема входных данных после проведения машинно-независимой оптимизации

Объем первого файла	Время работы программы	Коэффициент ускорения
До 90 Кб	<0,036 сек	18,29%
90-220 Кб	0,036-0,067 сек	15,2%
220-330 Кб	0,067-0,09 сек	6,25%
330-510 Кб	0,09-0,143 сек	4,47%
510-710 Кб	0,143-0,197 сек	5,75%
0,71-1,2 Мб	0,197-0,317 сек	5,19%
1,2-1,56 Мб	0,317-0,401 сек	0,03%

1,56-2,02 Мб	0,401-0,510 сек	2,2%
2,02-2,29 Мб	0,510-0,574 сек	2,23%
2,29-2,86 Мб	0,574-0,693 сек	4,39%
2,86-3,69 Мб	0,693-0,867 сек	1,15%
3,69-4,87 Мб	0,867-1,24 сек	1,93%
4,87-5,28 Мб	1,24-1,293 сек	1,9%

На рисунке 2 визуализированы данные из таблицы 2.

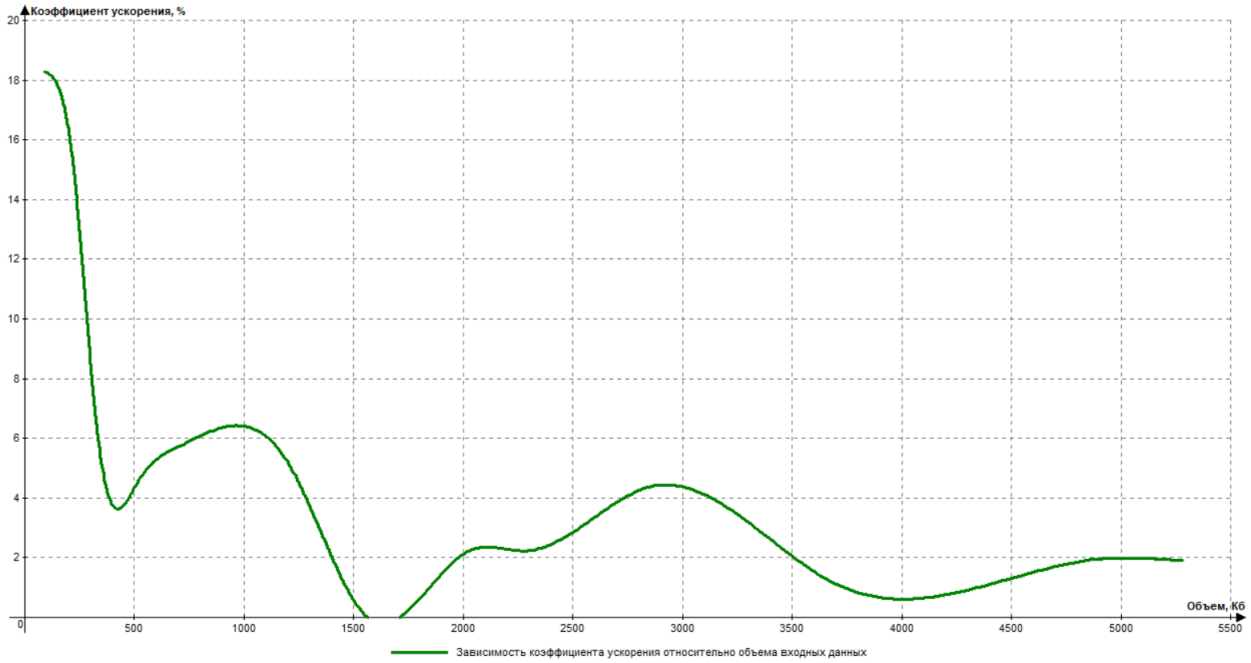

Рисунок 2 – Зависимость коэффициента ускорения от объема входных данных

Как видно из графика, наилучший результат оптимизации дают на небольших объемах входного файла (до 250 Кб) – до 18%. Далее коэффициент ускорения в основном находится в пределах от 2 до 5% – это связано с тем, что обработка файла происходит с помощью нахождения вершин и составления графа, после чего работа идет с графом, а не с огромной картинкой, поэтому, при больших размерах картинки, основные действия – считывание картинки и запись картинки, а обработка будет происходить с очень маленьким объемом данных на фоне размера картинки. Проведенные оптимизации убрали лишние вычисления из считывания и записи картинки, но там осталось много других действий, поэтому на фоне их они теряются.

2.3. Машинно-зависимая оптимизация

В среде разработки Visual Studio в свойствах проекта была включена оптимизация по скорости: «/O2» (рисунок 2). Данная оптимизация будет подбирать более быструю систему команд, учитывать количество регистров, избавляться от лишнего кода, не делать лишних действий (например, переприсваивание), будет проводить счет через регистры, а не через память, где хранятся переменные, собирать несколько циклов в 1, если это возможно.

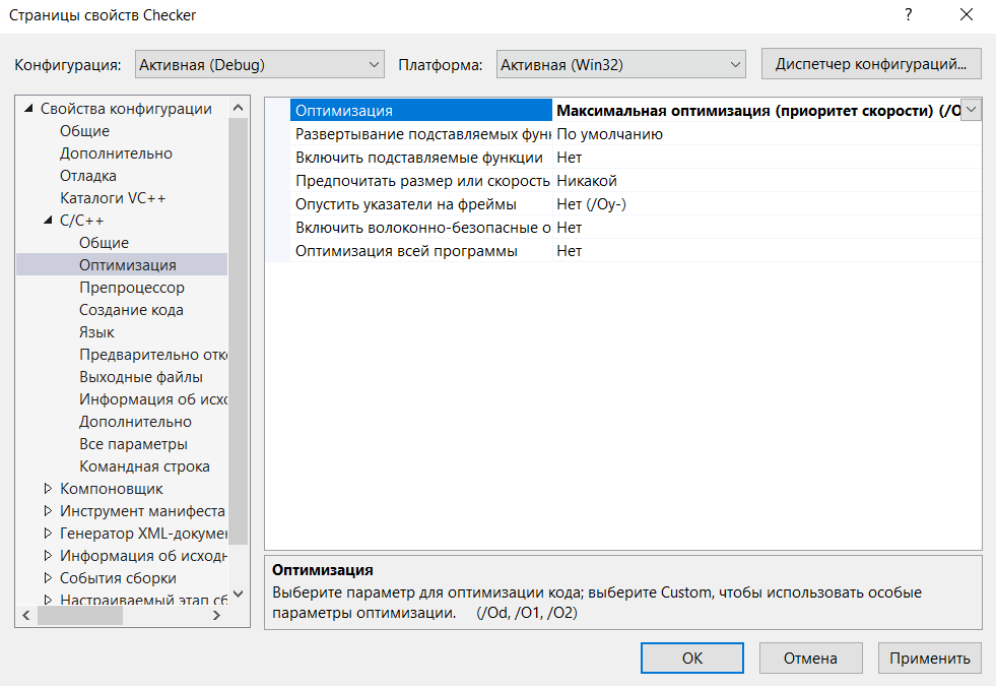

Рисунок 2 – Включение оптимизации в Visual Studio

Также было включено создание параллельного кода (рисунок 3). При включении данной оптимизации, компилятор проанализирует код, выявит циклы, выполнение которых можно ускорить благодаря распараллеливанию (одновременное выполнение разных действий с помощью разных процессорных ядер).

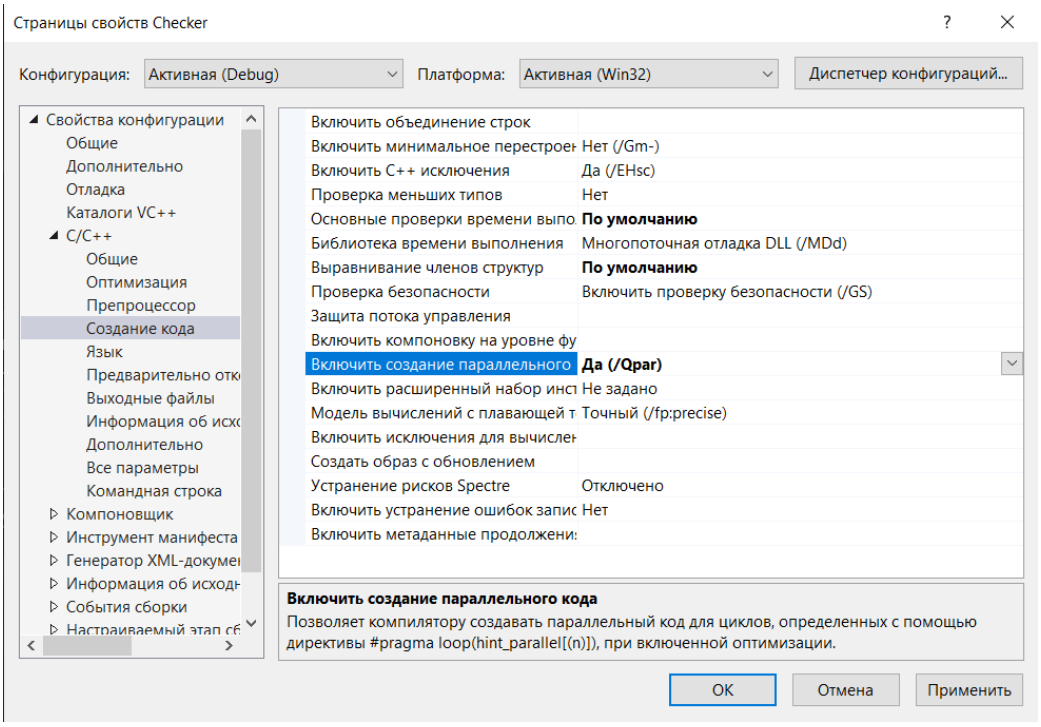

Рисунок 3 – Включение \Qpar в MSVC

После проведения машинно-зависимой оптимизации была снята новая зависимость времени выполнения программы от объема входных данных. Результаты представлены в таблице 3.

Таблица 3 – Зависимость времени работы программы от объема входных данных после проведения машинно-зависимой оптимизации

Объем первого файла	Время работы программы	Коэффициент ускорения
До 90 Кб	<0,033 сек	8,44%
90-220 Кб	0,033-0,065 сек	2,99%
220-330 Кб	0,065-0,082 сек	8,99%
330-510 Кб	0,082-0,127 сек	11,2%
510-710 Кб	0,127-0,187 сек	5,08%
0,71-1,2 Мб	0,187-0,281 сек	11,46%
1,2-1,56 Мб	0,281-0,379 сек	5,49%
1,56-2,02 Мб	0,379-0,472 сек	2,2%
2,02-2,29 Мб	0,472-0,538 сек	7,56%
2,29-2,86 Мб	0,538-0,633 сек	8,87%

2,86-3,69 Мб	0,633-0,817 сек	5,77%
3,69-4,87 Мб	0,817-1,14 сек	8,07%
4,87-5,28 Мб	1,14-1,185 сек	8,36%

На рисунке 3 визуализированы данные из таблицы 3.

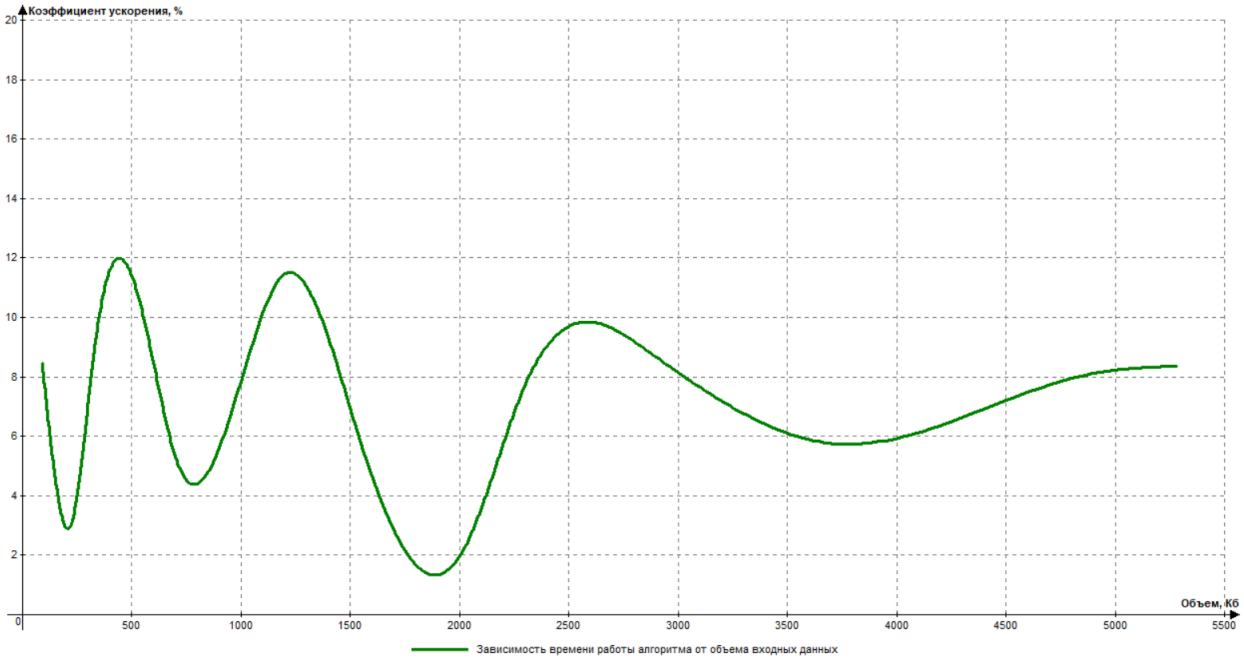

Рисунок 3 – Зависимость коэффициента ускорения от объема входных данных

Как видно из рисунка, коэффициент ускорения после проведения машинно-зависимой оптимизации в среднем составляет 8%.

2.4. Ассемблерная вставка

В коде момент подсчета константных значений в цикле перед входом в него был написан на языке ассемблер. Ассемблерная вставка с подробными комментариями представлена на рисунке 4.

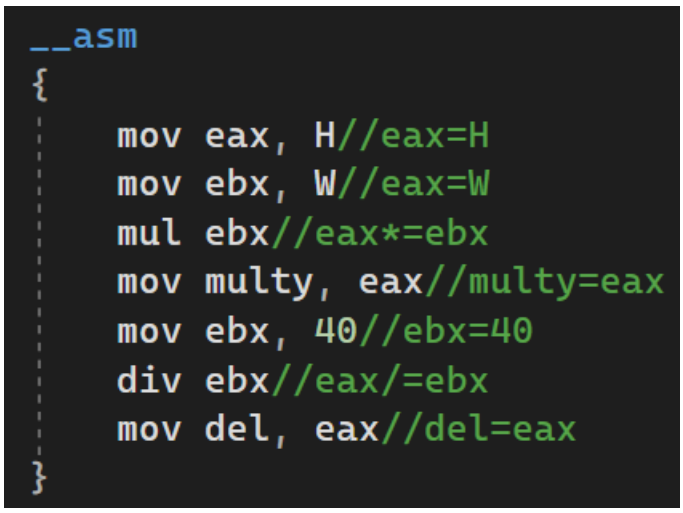

Рисунок 4 – Ассемблерная вставка

Глава 3. Тестирование разработанной программы

В таблице 4 представлено итоговое сравнение начального алгоритма с модифицированным всеми описанными выше оптимизациями.

Таблица 4. Зависимость времени работы программы от размеров обрабатываемых текстов

Размер файла	Время работы программы (сек)		Коэффициент Ускорения, %
	До оптимизации	После оптимизации	
До 90 Кб	<0,044 сек	<0,033 сек	25
90-220 Кб	0,044-0,079 сек	0,033-0,065 сек	18,2
220-330 Кб	0,079-0,096 сек	0,065-0,082 сек	14,6
330-510 Кб	0,096-0,15 сек	0,082-0,127 сек	15,4
510-710 Кб	0,15-0,209 сек	0,127-0,187 сек	10,6
0,71-1,2 Мб	0,209-0,334 сек	0,187-0,281 сек	15,9
1,2-1,56 Мб	0,334-0,402 сек	0,281-0,379 сек	5,8
1,56-2,02 Мб	0,402-0,521 сек	0,379-0,472 сек	9,5
2,02-2,29 Мб	0,521-0,587 сек	0,472-0,538 сек	8,4
2,29-2,86 Мб	0,587-0,724 сек	0,538-0,633 сек	12,6
2,86-3,69 Мб	0,724-0,875 сек	0,633-0,817 сек	6,7
3,69-4,87 Мб	0,875-1,263 сек	0,817-1,14 сек	9,8
4,87-5,28 Мб	1,263-1,318 сек	1,14-1,185 сек	11,1

Данные из таблицы 2 были визуализированы. Результаты приведены на рисунке 5.

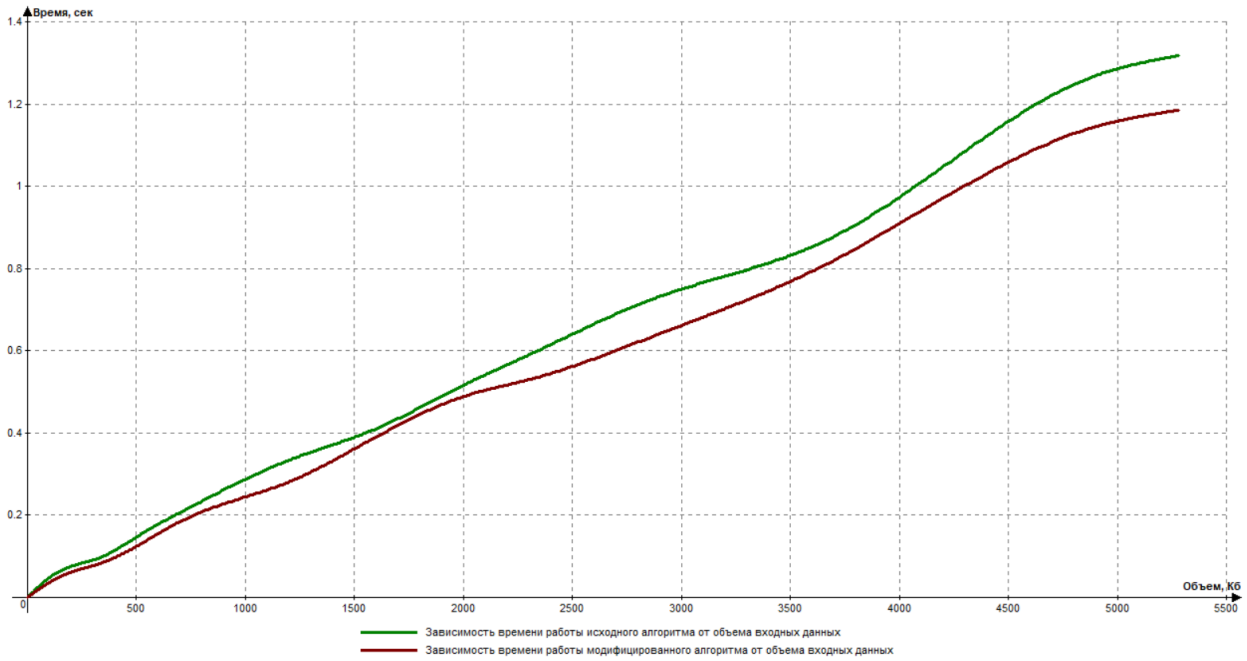

Рисунок 5 – График зависимости времени работы программы от объема входных данных

Работа с Git

Был создан репозиторий одним из участников. Далее владелец репозитория дал доступ на внесение изменений другому участнику, после чего выполнялся проект и по мере его выполнения изменения фиксировались через систему контроля версий Git. В данном репозитории есть всего одна ветвь – main, каждый коммит в которой подписан и сделан после проведения оптимизации или добавления новых файлов, необходимых для проекта. На рисунках 6-7 представлена ветка main.

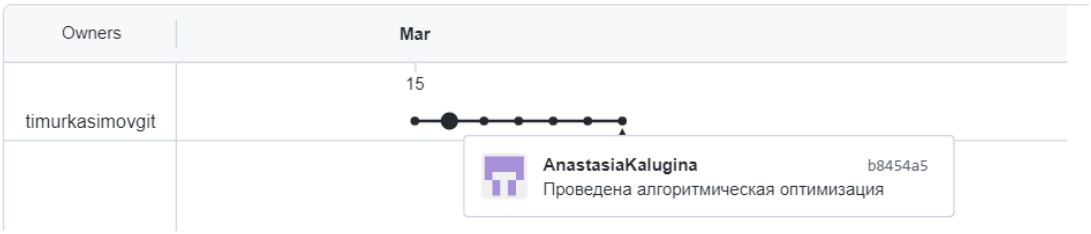

Рисунок 6 – Ветка main репозитория проекта

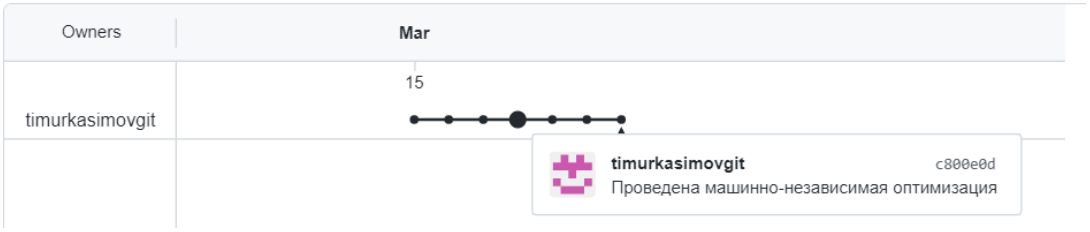

Рисунок 7 – Ветка main репозитория проекта

Как видно из рисунков 6-7, коммиты выполнялись разными участниками проекта и подписывались описанием изменения, которые было внесено.

Для просмотра подробностей об изменении можно нажать на интересующий коммит, GitHub перенаправит на новую страницу, где можно посмотреть более подробный комментарий автора коммита, количество измененных файлов и сами измененные файлы. На рисунке 8 представлен пример.

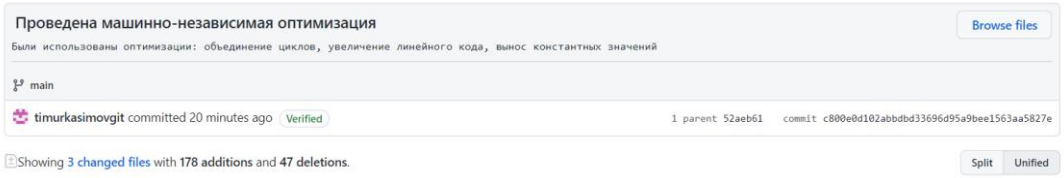

Рисунок 8 – Просмотр подробностей о коммите

Заключение

В ходе выполнения данной работы не удалось понизить сложность алгоритма с помощью оптимизации алгоритма, но удалось создать лучшие случаи, которые смогут сэкономить время – до модификаций алгоритм всегда проверял все раскраски до конца, после модификаций алгоритм не будет проверять раскраску до конца, если понимает, что она заведомо неверная. Для увеличения быстродействия программы были применены также машинно-независимые и машинно-зависимые оптимизации.

После проведения всех оптимизаций, картинки снова были протестированы с замером времени выполнения и сравнены с изначальным временем выполнения. Результаты представлены в таблице 4 (пункт 3).

Как видно из таблицы 4, скорость выполнения программы увеличилась на 25% для картинок малого объема (до 90 Кб), для картинок среднего размера (до 1,2 Мб) скорость увеличилась в среднем на 15%, для картинок большего объема коэффициент ускорения составил в среднем 8%.

Анализируя вклад каждой оптимизации по отдельности, можно сделать вывод, что все они дали примерно равные коэффициенты ускорения. Не смотря на то, что по отдельности вклад получился маленький, если посмотреть на сумму, то для небольших и средних файлов оптимизация получилась довольно успешной.