Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет: “Информационные технологии и прикладная математика”
Кафедра: 806 “Вычислительная математика и программирование”

Курсовой проект
по курсу “Дискретный анализ”

на тему “Аналог утилиты diff”

Студент	Полей-Добронравова А.В.
Группа	М8О-307Б-18
Преподаватель	А.Н. Ридли
Дата	
Оценка	

Москва, 2021

Описание

diff — утилита Unix-систем сравнения файлов, выводящая разницу между двумя файлами. Эта программа выводит построчно изменения, сделанные в файле.
diff вызывается из командной строки с именами двух файлов в качестве аргументов: diff *original new*. Вывод команды представляет собой изменения, которые нужно произвести в исходном файле original, чтобы получить новый файл new.

В этом традиционном формате вывода:
a - добавлено;
b — удалено**,**
c — изменено.
Перед буквами a, d или c стоят номера строк исходного файла, после них — номера строк конечного файла. Каждая строка, которая была добавлена, удалена или изменена, предваряется угловыми скобками.

По умолчанию, общие для исходного и конечного файлов номера строк не указываются. Строки, которые перемещены, показываются как добавленные на своём новом месте и удалённые из своего прошлого расположения.

Пример работы:

```
(base) MacBook-Pro-Amelia:2lab amelia$ diff test2.txt test.txt
1,12c1,16
< + first 1
< + trin 13
< + three 3
< + nine 9
< + eight 8
< + twelfe 12
< first
< three
< twelfe
< - trin
< twelfe
< trin
\ No newline at end of file
---
> aps
> + aps 125
> + aps 125
> aps
> + heey 15
> + sorry 16
> + privet 13
> aps
> heey
> sorry
> - heey
> aps
> heey
> sorry
> privet
> porp
\ No newline at end of file
```

Идейно утилита показывает, какие операции нужно сделать, чтобы первый файл превратить во второй.

Изучение первоисточника алгоритма

Базовый алгоритм изложен в книгах *An O(ND) Difference Algorithm and its Variations* Юджина В. Майерса и в *A File Comparison Program* Вебба Миллера и Майерса. Рассмотрим оригинал первого документа и переведем его на русский язык.

Асимптотический анализ

Алгоритм имеет сложность O(ND), где N - сумма длин A и B, а D - размер минимального сценария редактирования для A и B. Алгоритм хорошо работает, когда различия невелики (последовательности подобны), и, следовательно, быстр в типичных приложениях. Ожидаемая временная производительность в рамках базовой стохастической модели $O(N + D^2)$. Улучшение алгоритма требует только O(N) памяти, а использование суффиксных деревьев даёт время $O(N*\log N + D^2)$ в худшем случае.

Edit graph

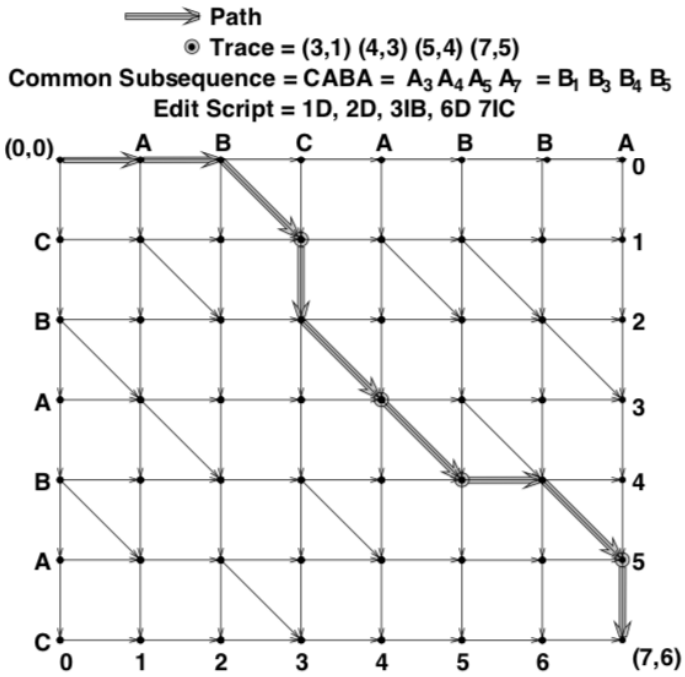
Пусть $A = a_1a_2...a_n$ и $B = b_1b_2...b_m$ - последовательности длиной N и M соответственно. Граф редактирования для A и B имеет вершину в каждой точке сетки (x,y), $x \in [0,N]$ и $y \in [0,M]$. Вершины графа редактирования соединяются горизонтальными, вертикальными и диагональными направленными ребрами, образуя направленный ациклический граф.

Горизонтальные ребра соединяют каждую вершину с ее правым соседом, т. е. $(x-1,y) \rightarrow (x,y)$ для $x \in [1,N]$ и $y \in [0,M]$.

Вертикальные ребра соединяют каждую вершину с соседом ниже нее, т. е. $(x,y-1) \rightarrow (x,y)$ для $x \in [0,N]$ и $y \in [1,M]$.

Если $a_x = b_y$, то существует *диагональное ребро*, соединяющее вершину $(x-1,y-1)$ с вершиной (x,y). Точки (x,y), для которых $a_x = b_y$, называются точками совпадения. Общее количество точек совпадения между A и B - это параметр R, это также число диагональных ребер в графе редактирования, поскольку диагональные ребра находятся в взаимно однозначном соответствии с точками совпадения.

На рисунке ниже показан график редактирования для строк A = abcabba и B = cbabac.



След (trace) длины L - это последовательность L совпадающих точек $(x_1, y_1)(x_2, y_2)...(x_L, y_L)$, таких,что $x_i < x_i+1$ и $y_i < y_i + 1$ для последовательных точек (x_i,y_i) и (x_i+1, y_i+1), $i \in [1, L-1]$. Каждый след находится в точном соответствии с диагональными ребрами пути в графе редактирования от (0,0) до (N,M). Последовательность точек совпадения, посещенных при прохождении пути от начала до конца, легко проверяется как след. Обратите внимание, что L - это число диагональных ребер в соответствующем пути. Чтобы построить путь из следа, возьмите последовательность диагональных ребер, соответствующих точкам совпадения следа, и соедините последовательные диагонали с серией горизонтальных и вертикальных ребер. Это всегда можно сделать как $x_i < x_i + 1$ и $y_i < y_i+1$ для последовательных точек совпадения. Заметим, что заданному следу может соответствовать несколько путей, отличающихся только своими не диагональными ребрами. Рисунок выше иллюстрирует эту связь между путями и трассами.

Подпоследовательность строки - это любая строка, полученная путем удаления нуля или более символов из данной строки. Общая подпоследовательность двух строк, A и B, является подпоследовательностью обеих. Каждый след порождает общую подпоследовательность A и B и наоборот. В частности, $a_{x1}a_{x2}...a_{xL} = b_{y1}b_{y2}...b_{yL}$ является общей подпоследовательностью A и B, если (x1, y1)(x2,y2)...(xL,yL) - это след A и B.

Скрипт редактирования для A и B - это набор команд вставки и удаления, которые преобразуют A в B. Команды скрипта ссылаются на позиции символов внутри A до того, как были выполнены какие-либо команды. Нужно думать, что набор команд в сценарии выполняется одновременно. *Длина скрипта* - это количество добавленных и удаленных символов.

Каждый путь однозначно соответствует скрипту редактирования. Пусть $(x_1, y_1)(x_2, y_2)...(x_L, y_L)$ - след. Пусть $y_0 = 0$ и $y_{L+1} = M+1$. Связанный скрипт состоит из команд: 'xD' для $x \in /\{x_1, x_2,\ldots, x_L\}$ и 'x_k I $b_{y(k)+1},..., b_{y(k)-1}$' для k таких ,что $y_k+1 < y_{k+1}$. Скрипт удаляет N−L символов и вставляет M−L символов. Таким образом, для каждого следа длины L существует соответствующий скрипт длины D = N+M−2L. Чтобы сопоставить сценарий редактирования с путем, просто выполните все команды удаления на A. Результатом является общая подпоследовательность A и B, затем сопоставим подпоследовательность с ее уникальным путем. Важно, что инвертирование действия команд insert дает набор команд delete, которые сопоставляют B с одной и той же общей подпоследовательностью.

Задача нахождения самой длинной общей подпоследовательности (LCS) эквивалентна нахождению пути от (0,0) до (N,M) с максимальным числом диагональных ребер. Задача нахождения кратчайшего сценария(скрипта) редактирования (SES) эквивалентна нахождению пути от (0,0) до (N,M) с минимальным числом не диагональных ребер. Это одинаковые задачи, т.к. путь с максимальным числом диагональных ребер имеет минимальное число не диагональных ребер (D+2L = M+N). Добавим вес или стоимость к каждой вершине. Дадим диагональным ребрам вес 0, а не диагональным ребрам вес 1. Задача LCS/SES эквивалентна нахождению пути минимальной стоимости от (0,0) до (N,M) в взвешенном графе редактирования и, таким образом, является частным случаем задачи кратчайшего пути с одним источником.

Жадный алгоритм O((M+N)D)

Задача поиска кратчайшего сценария редактирования сводится к нахождению пути от (0,0) до (N,M) с наименьшим числом горизонтальных и вертикальных ребер. Пусть D-путь - это путь, начинающийся в точке (0,0) и имеющий ровно D недиагональных ребер. 0-путь должен состоять исключительно из диагональных ребер. Из простой индукции следует, что D-путь должен состоять из (D − 1)-пути, за которым следует недиагональное ребро, а затем, возможно, пустая последовательность диагональных ребер, называемая *змейкой*.

Пронумеруем диагонали в сетке вершин edit graph так,чтобы диагональ k состояла из точек (x, y), для которых x−y=k. Линия, начинающаяся в точке (0, 0), определяется как k=0. K увеличивается в

линиях справа и уменьшается вниз. Таким образом, линия k через (1, 0) имеет k=1, а линия k через (0, 1) имеет k=-1. При таком определении диагонали нумеруются от −M до N. Заметим, что вертикальное (горизонтальное) ребро с начальной точкой на диагонали k имеет конечную точку на диагонали k−1 (k+1), а змейка остается на диагонали, в которой она начинается.

- D-путь должен заканчиваться на диагонали $k \in \{-D, -D+2,...,D-2,D\}$
- Самый дальний достигающий 0-путь заканчивается в точке (x,x), где $x = \min(z-1 \mid\mid a_z \neq b_z$ или $z > M$ или $z > N)$. Наиболее далеко идущий D-путь по диагонали k может без потери целостности быть разложен на наиболее далеко идущий (D − 1)-путь по диагонали k − 1, за которым следует горизонтальный край, за которым следует самая длинная возможная змейка, или он может быть разложен на наиболее далеко идущий (D − 1)-путь по диагонали k+1, за которым следует вертикальное ребро, за которым следует самая длинная возможная змейка.

При написании подробного алгоритма ниже используется ряд простых оптимизаций. Массив V содержит конечные точки наиболее далеко идущих D-путей в элементах V[−D],V[−D+2],...,V[D-2],V[D]. Набор элементов не пересекается от тех, где конечные точки (D+1)-путей будут сохранены в следующей итерации внешнего цикла. Массив V может одновременно содержать конечные точки D-путей, в то время как конечные точки (D+1)-пути вычисляются из них. Для записи конечной точки (x,y) в диагонали k достаточно сохранить только x, поскольку известно, что y=x-k. Следовательно, V[k] - массив содержащий индекс строки конечной точки наиболее далеко идущего пути по диагонали k.

```
     Constant MAX ∈ [0,M+N]

     Var V: Array [–MAX .. MAX] of Integer

1.   V[1] ← 0
2.   For D ← 0 to MAX Do
3.       For k ← –D to D in steps of 2 Do
4.           If k = –D or k ≠ D and V[k – 1] < V[k + 1] Then
5.               x ← V[k + 1]
6.           Else
7.               x ← V[k – 1]+1
8.           y ← x – k
9.           While x < N and y < M and a(x+1) = b(y+1) Do (x,y) ← (x+1,y+1)
10.          V[k] ← x
11.          If x ≥ N and y ≥ M Then
12.              Length of an SES is D
13.              Stop
14.  Length of an SES is greater than MAX
```

FIGURE 2: The Greedy LCS/SES Algorithm

На практике алгоритм ищет D-пути, где D ≤ MAX, и если такой путь не достигает (N, M), то он сообщает, что любой сценарий редактирования для A и B должен быть длиннее MAX в строке 14. Установив константу MAX на M+N, как показано выше, алгоритм гарантированно найдет длину LCS/SES. На рис. 3 ниже показаны D-пути, искомые при применении алгоритма к примеру на рис.1.
Фиктивная конечная точка (0,−1), установленная в строке 1 алгоритма, используется для поиска конечной точки самого дальнего достижения 0-пути. Также обратите внимание, что D-пути расширяются от левой и нижней границ собственно графа редактирования по мере продвижения алгоритма. Эта граничная ситуация правильно обрабатывается, предполагая, что в этой области нет диагональных ребер.

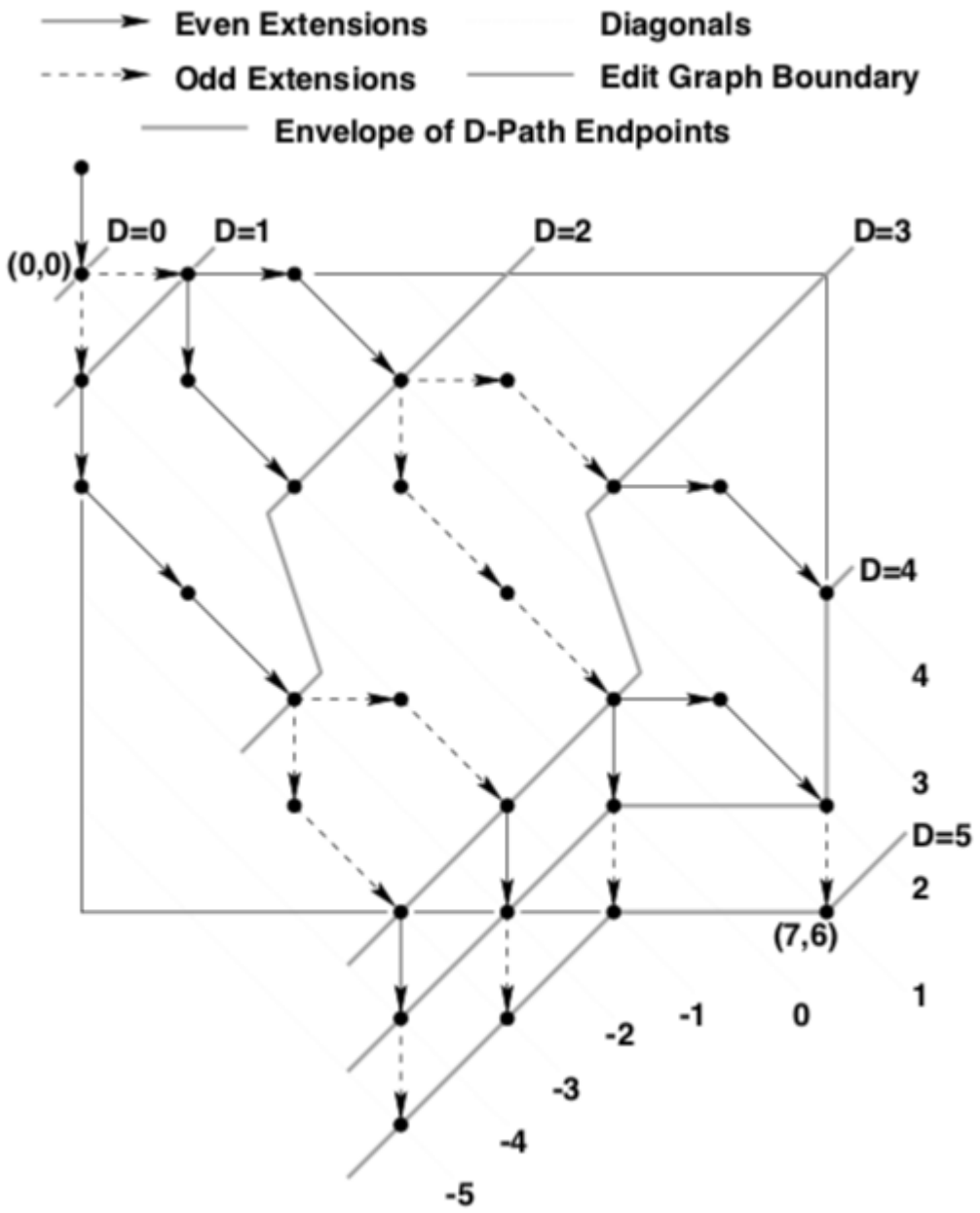


Fig. 3. Furthest reaching paths.

Жадный алгоритм занимает не более O((M+N)D) времени. Строки 1 и 14 потребляют O(1) времени. Внутренний цикл For (Строка 3) повторяется не более (D+1)(D+2)/2 раза, поскольку внешний цикл For (Строка 3) повторяется D+1 раз, а на k - й итерации внутренний цикл повторяется не более k раз. Все линии внутри этого внутреннего цикла занимают постоянное время, за исключением цикла While (Строка 9). Таким образом, время $O(D^2)$ тратится на выполнение строк 2-8 и 10-13.

Цикл While выполняется один раз для каждой диагонали, пройденной в продолжении наиболее далеко идущих путей. Но не более

O((M+N)D) диагоналей пересекаются, т.к. все D-пути лежат между диагоналями -D и D и в этой полосе есть не более (2D+1)min(N, M) точек. Таким образом, алгоритм требует в общей сложности O((M+N)D) времени. Обратите внимание, что только линия 9, обход змеек, является ограничивающим шагом. Остальная часть алгоритма $O(D^2)$. Кроме того, алгоритм никогда не занимает больше времени O((M+N)MAX) в практическом случае, когда пороговое значение MAX устанавливается на значение гораздо меньшее, чем M+N.

Поиск жадного алгоритма прослеживает оптимальные D-пути среди других. Но в V сохраняется только текущий набор наиболее далеко идущих конечных точек. Следовательно, в строке 12 можно указать только длину SES/LCS. Чтобы явно сгенерировать путь решения, память $O(D^2)$ используется для хранения копии V после каждой итерации внешнего цикла. Пусть V_d - копия V, сохраненная после d-й итерации. Чтобы перечислить оптимальный путь от (0,0) до точки $V_d[k]$, сначала определите, находится ли он в конце максимальной змейки, следующей за вертикальным ребром от $V_{d-1}[k+1]$ или горизонтальным ребром от $V_{d-1}[k-1]$. Чтобы быть конкретным, предположим, что это $V_{d-1}[k-1]$. Рекурсивно запишем оптимальный путь от (0,0) до этой точки, а затем запишем вертикальное ребро и максимальную змейку до $V_d[k]$. Рекурсия останавливается при d = 0, и в этом случае змея от (0,0) до ($V_0[0]$, $V_0[0]$) записывается. Таким образом, с O(M+N) дополнительным временем и $O(D^2)$ памятью оптимальный путь можно записать, заменив строку 12 вызовом этой рекурсивной процедуры с $V_D[N-M]$ в качестве начальной точки.

Реализация

```
#include <iostream>
#include <tuple>
#include <string>
#include <cmath>
#include <fstream>

class V {
public:
    V(int b, int e) {
        begin = b;
        end = e;
        data = new int[end - begin + 1];
    }

    ~V() {
        delete[](data);
    }

    int &operator[](int index) {
        return data[index - begin];
    }

private:
    int* data;
    int begin;
    int end;
};

int MyersDiff(std::string* a, int N, std::string* b, int M) {
    int MAX = M + N;
    V V(-MAX, MAX);
    V[1] = 0;
    int x, y;
    for (int D = 0; D <= MAX; D++) {
        for (int k = -D; k <= D; k += 2) {
            if (k == -D || (k != D && V[k - 1] < V[k + 1])) {
                x = V[k + 1];
            } else {
                x = V[k - 1] + 1;
            }
            y = x - k;
            while (x < N && y < M && a[x] == b[y]) {
                x += 1;
                y += 1;
            }
            V[k] = x;
            if (x >= N && y >= M) {
                return D;
            }
        }
    }
    return -1;
}

std::tuple<int, int, int, int, int> MiddleSnake(std::string* a, int N, std::string* b, int M) {
    int delta = N - M;
```

```
    int max = M + N;
    static V fv(-max, max);
    static V rv(-max, max);
    int x, y;
    fv[1] = 0;
    rv[delta + 1] = N + 1;
    for (int D = 0; D <= std::ceil((M + N) / 2.0); D++) {
        for (int k = -D; k <= D; k += 2) {
            if (k == -D || (k != D && fv[k - 1] < fv[k + 1])) {
                x = fv[k + 1];
            } else {
                x = fv[k - 1] + 1;
            }
            y = x - k;
            while (x < N && y < M && a[x] == b[y]) {
                x += 1;
                y += 1;
            }
            fv[k] = x;
            if (delta % 2 != 0 && k >= delta - (D - 1) && k <= delta + D - 1) {
                if (fv[k] >= rv[k]) {
                    return std::make_tuple(rv[k], rv[k] - k, x, y, 2 * D - 1);
                }
            }
        }
        for (int k = -D + delta; k <= D + delta; k += 2) {
            if (k == -D + delta || (k != D + delta && rv[k - 1] >= rv[k + 1])) {
                x = rv[k + 1] - 1;
            } else {
                x = rv[k - 1];
            }
            y = x - k;
            while (x > 0 && y > 0 && a[x - 1] == b[y - 1]) {
                x -= 1;
                y -= 1;
            }
            rv[k] = x;
            if (delta % 2 == 0 && k >= -D && k <= D) {
                if (fv[k] >= rv[k]) {
                    return std::make_tuple(x, y, fv[k], fv[k] - k, 2 * D);
                }
            }
        }
    }
    return {};
}

void SES(std::string* a, int N, std::string* b, int M, int pos, int pos1) {
    std::string *startA = nullptr;
    if (startA == nullptr) {
        startA = a;
    }
    while (*a == *b && N > 0 && M > 0) {
        ++a;
        ++pos;
        ++pos1;
        ++b;
        --N;
```

```
        --M;
    }
    while (*(a + N - 1) == *(b + M - 1) && N > 0 && M > 0) {
        --N;
        --M;
    }
    if (N > 0 && M > 0) {
        int x, y, u, v, D;
        std::tie(x, y, u, v, D) = MiddleSnake(a, N, b, M);
        SES(a, x, b, y, pos, pos1);
        SES(a + u, N - u, b + v, M - v, pos+u, pos1+v);
    } else if (N > 0) {
        std::cout << pos +1 << "," << N + pos << "-"  << ":\n";
        for (int i = 0; i < N; i++) {
            std::cout << "- " <<"\x1b[41m" << a[a + i - startA] << "\x1b[40m" << "\n";
        }
        std::cout << "----------------\n";
    } else if (M > 0) {
        std::cout << "+" << pos1+1 << "," << M + pos1 << ":\n";
        for (int i = 0; i < M; i++) {
            std::cout << "+ " <<"\x1b[42m" << b[i] << "\x1b[40m" <<"\n";
        }
        std::cout << "----------------\n";
    }
}

int main(int argc, const char * argv[]) {
    const char* name1 = argv[1];
    if (!strcmp(name1,"-help")) {
        std::cout << "MYERS diff algorythm \n+[start position],[end position] shows which lines are added on new
positions\n[start position],[end position]- shows which lines are removed from their positions\nEnjoy!\n";
    }
    else {
        const char* name2 = argv[2];
        std::ifstream file1(name1);
        if (!(file1.is_open())){
            std::cout << "No file " << name1 << " found\n";
            return -1;
        }
        std::ifstream file2(name2);
        if (!(file2.is_open())){
            std::cout << "No file " << name2 << " found\n";
            return -1;
        }
        int N_a_max = 20;
        int N_a_size = 0;
        int N_b_max = 20;
        int N_b_size = 0;
        std::string* a = new std::string[N_a_max];
        std::string* b = new std::string[N_b_max];
        std::string buf;
        while(getline(file1, buf)){
            if (N_a_size >= N_a_max) {
                N_a_max = N_a_max * 2;
                std::string* aa = new std::string[N_a_max];
                for (int i = 0; i < N_a_size; i++) {
                    aa[i] = a[i];
                }
```

```
            delete [] a;
            a = aa;
        }
        a[N_a_size] = buf;
        N_a_size++;
    }
    while(getline(file2, buf)){
        if (N_b_size >= N_b_max) {
            N_b_max = N_b_max * 2;
            std::string* bb = new std::string[N_b_max];
            for (int i = 0; i < N_b_size; i++) {
                bb[i] = b[i];
            }
            delete [] b;
            b = bb;
        }
        b[N_b_size] = buf;
        N_b_size++;
    }
    file1.close();
    file2.close();

    std::cout << "\nNumber of changes required: " << MyersDiff(a, N_a_size, b, N_b_size) << "\n" << std::endl;
    SES(a, N_a_size, b, N_b_size, 0, 0);
  }
}
```

Пример работы и тестирование

Пусть имеется два файла:

file1.txt

“Эта часть документа
оставалась неизменной
от версии к версии. Если
в ней нет изменений, она
не должна отображаться.
Иначе это не способствует
выводу оптимального
объёма произведённых
изменений.

Этот абзац содержит
устаревший текст.
Он будет удалён
в ближайшем будущем.

В этом документе
необходима провести
проверку правописания.
С другой стороны, ошибка
в слове - не конец света.
Остальная часть абзаца
не требует изменений.

Новый текст можно
добавлять в конец документа.”

file2.txt

“Это важное замечание!
Поэтому оно должно
быть расположено
в начале этого
документа!

Эта часть документа
оставалась неизменной
от версии к версии. Если
в ней нет изменений, она
не должна отображаться.
Иначе это не способствует
выводу оптимального
объёма информации.

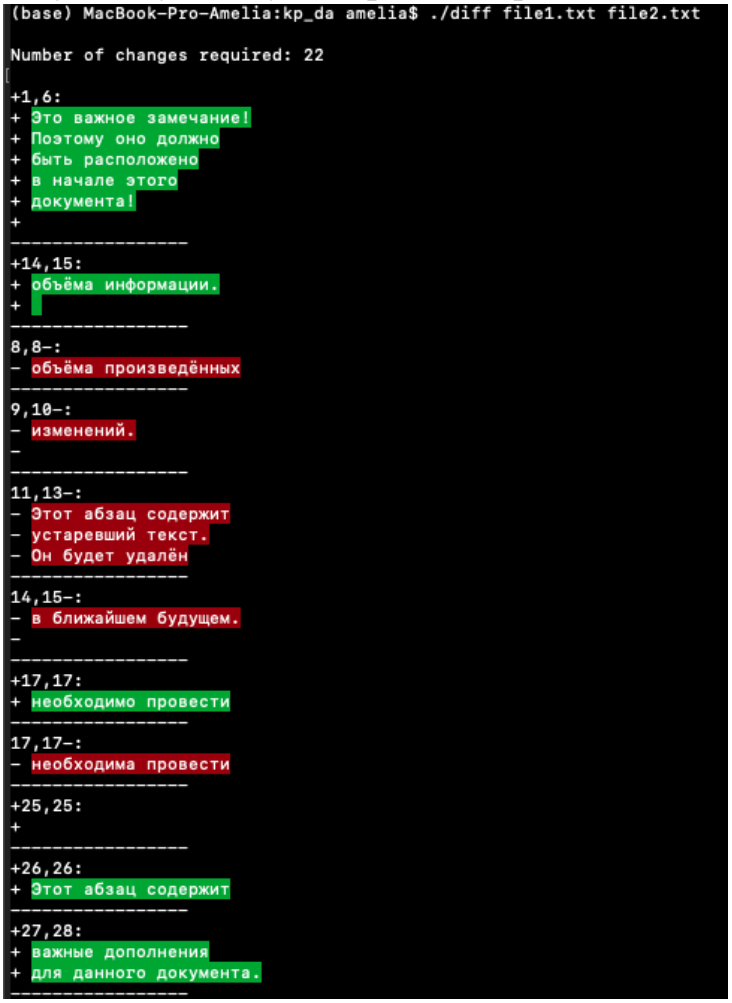
В этом документе
необходимо провести
проверку правописания.
С другой стороны, ошибка
в слове - не конец света.
Остальная часть абзаца
не требует изменений.
Новый текст можно
добавлять в конец документа.

Этот абзац содержит
важные дополнения
для данного документа.”

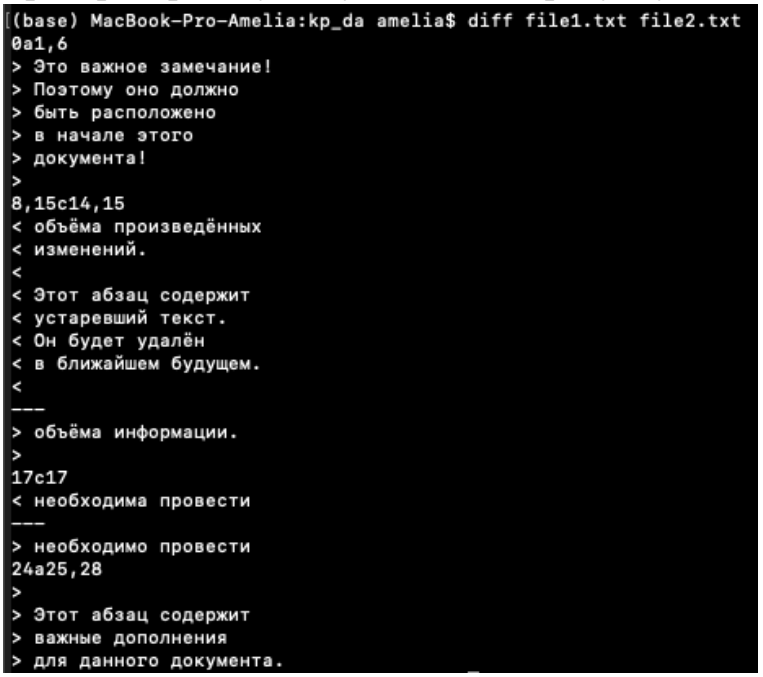
- Реализован флаг -help для прочтения информации об утилите:

```
(base) MacBook-Pro-Amelia:kp_da amelia$ ./diff -help
MYERS diff algorythm
+[start position],[end position] shows which lines are added on new positions
[start position],[end position]- shows which lines are removed from their positions
Enjoy!
```

Запустим утилиту для файлов, представленных выше.



Проверим работу, запустив стандартную утилиту diff.



Как мы видим, результат работы совпадает.

Исследование времени выполнения

Добавив в код программы библиотеку ctime выведем подсчитанное время работы:

```
Time of working 0.686s
```

Теперь с помощью утилиты time выясним, сколько будет работать стандартная утилита diff:

```
MacBook-Pro-Amelia:kp_da amelia$ time diff file1.txt file2.txt

real    0m0.023s
user    0m0.001s
sys     0m0.007s
```

Чем может быть объяснена такая разница в работе? Во-первых моё образование динамических массивов для хранения строк файлов довольно медленно работает из-за множества if операторов, проверяющих, хватает ли памяти. Во-вторых, быстрее было бы возможно использовать какой-то библиотечный контейнер, или даже как говорится в оригинале статьи *суффиксные деревья,* которые дают $O(NlgN + D^2)$, где N - суммарное число строк файлов, D - длина SES.

Например, в данном тесте, используемый мной алгоритм за O(ND) имеет время и память (24+28)*22 = 1144. Если бы я использовала суффиксные деревья, то время было бы равно (52*ln(52)+22*22)=689, что уже в 1,6 раз быстрее.

Вывод

Как было показано выше, время и память, затрачиваемые алгоритмом, очень зависят от структуры данных, которую мы используем для реализации. В предыдущем пункте и в введении приведены конкретные формулы.

Существует еще один известный алгоритм для поставленной задачи: Hunt и McIlroy. Подход Hunt и McIlroy вычисляет совпадения во всем файле и индексирует их в k-кандидаты. k-длина LCS. LCS постепенно расширяется путем нахождения совпадений, которые попадают в правильные ординаты. При этом каждый путь запоминается. Этот подход *выполняет больше работы*, чем необходимо: он запоминает все пути, O(mn) память в худшем случае и O(mn log m) времени.

Алгоритм Майерса в основном выигрывает у него, потому что он не запоминает пути во время работы и ему не нужно смотреть, куда идти, поэтому он может сосредоточиться в любое время только на самых дальних позициях, которых он может достичь с помощью сценария редактирования наименьшей сложности.

На мой взгляд, если целью ставится именно список команд по модификации файла для его превращения в конечный файл, можно гораздо улучшить работу программы, если работать не сравнивая строки целиком, а работать посимвольно. Если у двух строк одинакового номера по счету из старого и нового файла есть длинная общая подпоследовательность, то гораздо эффективнее запустить тот же алгоритм отдельно для этих двух строк.

Если же наша цель - контроль версий, как например в git, то построчный анализ чаще наиболее приятен и понятен человеческому взгляду для отслеживания изменений, если не сделать удобное представление вывода изменений строк.