Курсовой проект по курсу дискретного анализа: Утилита diff

Выполнил студент МАИ группы М8О-307Б *Ефимов Александр.*

Постановка задачи

Разработать на языке C или C++ программу, способную выводить наименьшее количество действий, позволяющих преобразовать один файл в другой (см. утилиту **diff**).

Метод решения

Самым простом алгоритмом, выводящим доступный для человека формат, является алгоритм **Юджина Майерса**, поверхностно описать который можно так:

Проблему нахождения наименьшего изменения между двумя файлами A и B размеров N и M одновременно можно рассматривать как проблему нахождения пути на графе размером $N \times M$ между вершинами $(0, 0)$ и (N, M), причем на этом графе:

- Горизонтальная грань $(x, y) \to (x + 1, y)$ означает удаление в файле A строки x (в файлах индексация строк считается с единицы);
- Вертикальная грань $(x, y) \to (x, y + 1)$ означает добавление из файла B строки y;
- Диагональная грань $(x, y) \to (x + 1, y + 1)$ означает, что строки x и y в файлах A и B соответственно равны.

Если считать, что у горизонтальных и вертикальных граней вес равен единице, а у диагональных – нулю, то задача решаема алгоритмом Дейкстры, но он не всегда будет выводить читабельный diff-вывод.

Алгоритм Майерса пользуется тем, что путь в графе всегда идет из верхнего левого угла в нижний правый угол.

Введем понятие уровня и D-пути:

- Уровень k на графе – это номер диагонали по сравнению с диагональю, на которой лежит точка $(0, 0)$. Соответственно диагональ с точкой $(0, 0)$ имеет уровень $k = 0$, диагональ над ней - $k = 1$, диагональ под ней - $k = -1$ и т.д. Уровень можно высчитать по формуле $k = x - y$.
- D-путь – (D - 1)-путь, после которого идет или горизонтальная грань, или вертикальная грань, причем 0-путь – это путь, состоящий только из этой грани. После этой грани существует возможно пустая последовательность диагоналей.

Алгоритм имеет максимум $N+M$ повторений, причем на d-ом повторении идет удлинение всех d-путей, лежащих на уровнях $-d, -d + 2, ..., d - 2, d$. Удлинение на уровне k идет за удлинения пути либо на $k - 1$ уровне или на $k + 1$, в зависимости от того, какой

длиннее. Приоритет будет отдан $k-1$ уровню когда это возможно, т.к. он удлиняется за счет горизонтальной грани (т.е. удаления из A). Первый d-путь, достигший точки (N, M), считается оптимальным.

Описание программы

1. *main.cpp* – принимает из командной строки названия файлов из считывает их построчно в вектор, после чего вызывает функцию строющую diff между этими векторами

2. *diff.h* – содержит шаблонную функцию, строющую diff в соответствии с алгоритмом Юджина Майерса. Фукнция принимает любой тип, имеющий в себе метод *size* и оператор квадратных скобок.

Листинг кода

- **structs.h**

```
#ifndef STRUCTS_H
#define STRUCTS_H

#include <cstdint>

struct TAction {
    enum {ADD, DEL, KEEP} type;
    int64_t x, y;
};

#endif
```

- **diff.h**

```
#ifndef DIFF_H
#define DIFF_H

#include <cstdint>
#include <vector>

#include "structs.h"

std::vector<TAction> build_trace(
        const std::vector< std::vector<int64_t> >& trace,
        int64_t x, int64_t y, size_t total);

template<typename T>
std::vector<TAction>
find_diff(const T& data1, const T& data2) {
    const size_t len1  = data1.size();
    const size_t len2  = data2.size();
    const size_t total = len1 + len2;

    std::vector<int64_t> extensions(2 * total + 1);
    std::vector< std::vector<int64_t> > trace;

    extensions[1 + total] = 0;
    for (int64_t path = 0; path <= total; ++path) {

        trace.push_back(extensions);

        for (int64_t diag = -path; diag <= path; diag += 2) {
            int64_t x, y;
            bool go_down = (diag == -path
                            || (diag != path
```

```
                                && extensions[diag - 1 + total] < extensions[diag
                                ↪  + 1 + total]));

            if (go_down) {
                x = extensions[diag + 1 + total];
            } else {
                x = extensions[diag - 1 + total] + 1;
            }

            y = x - diag;

            while (x < len1 && y < len2 && data1[x] == data2[y]) {
                ++x;
                ++y;
            }

            extensions[diag + total] = x;
            if (x >= len1 && y >= len2) {
                return build_trace(trace, len1, len2, total);
            }
        }
    }

    return std::vector<TAction> ();
}



#endif
```

- **diff.cpp**

```
#include "diff.h"

std::vector<TAction> build_trace(
        const std::vector< std::vector<int64_t> >& trace,
        int64_t x, int64_t y, size_t total) {
    std::vector<TAction> diff_actions;

    for (int64_t d = trace.size() - 1; d >= 0; --d) {
        const std::vector<int64_t>& layer = trace[d];

        int64_t k = x - y;
        int64_t prev_k;

        bool went_down = (k == -d || (k != d && layer[k - 1 + total] < layer[k + 1
        ↪  + total]));
        if (went_down) {
            prev_k = k + 1;
```

```
        } else {
            prev_k = k - 1;
        }

        int64_t prev_x = layer[prev_k + total];
        int64_t prev_y = prev_x - prev_k;

        while (x > prev_x && y > prev_y) {
            --x;
            --y;
            diff_actions.push_back({TAction::KEEP, x, y});
        }

        if (d == 0) {
            continue;
        }

        if (x == prev_x) {
            y = prev_y;
            diff_actions.push_back({TAction::ADD, x, y});
        } else if (y == prev_y) {
            x = prev_x;
            diff_actions.push_back({TAction::DEL, x, y});
        }
    }

    return std::vector<TAction>(diff_actions.rbegin(), diff_actions.rend());
}
```

- **main.cpp**

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>

#include "diff.h"

std::vector<std::string> read_file(char* filename) {
    std::ifstream is(filename);
    std::string line;
    std::vector<std::string> text;

    while(std::getline(is, line)) {
        text.push_back(line);
    }

    return text;
}
```

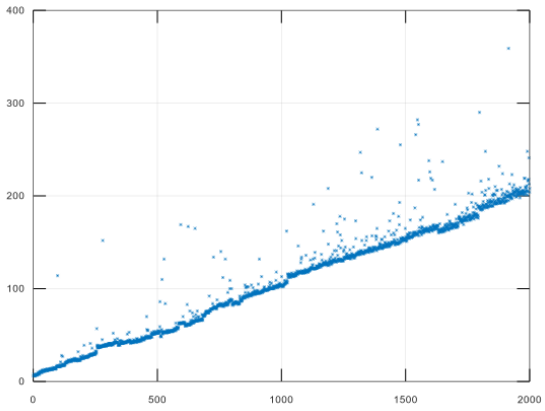
```
19
20  int main(int argc, char* argv[]) {
21      if (argc < 3) {
22          std::cout
23              << "Usage: "
24              << argv[0]
25              << " FILE1 FILE2"
26              << std::endl;
27          return -1;
28      }
29
30      std::vector<std::string> text1 = read_file(argv[1]);
31      std::vector<std::string> text2 = read_file(argv[2]);
32
33      std::vector<TAction> actions( find_diff(text1, text2) );
34
35      for (const auto& act : actions) {
36          switch (act.type) {
37            case TAction::ADD: {
38              std::cout << "+ ";
39              std::cout << text2[act.y] << std::endl;
40              break;
41            }
42            case TAction::DEL: {
43              std::cout << "- ";
44              std::cout << text1[act.x] << std::endl;
45              break;
46            }
47            case TAction::KEEP: {
48              std::cout << "  ";
49              std::cout << text1[act.x] << std::endl;
50              break;
51            }
52          }
53      }
54  }
```

Пример использования

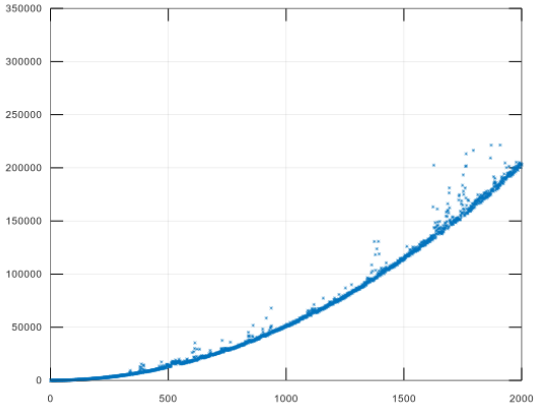
```
src λ pr -m -t test3-1.txt test3-2.txt
The                 The
bright              dark
side                side
of                  of
the                 the
sun                 moon
```

```
src λ ./a.out test3-1.txt test3-2.txt
  The
- bright
+ dark
  side
  of
  the
- sun
+ moon
+
src λ diff -u test3-1.txt test3-2.txt
--- test3-1.txt        2021-03-31 12:21:24.506662618 +0300
+++ test3-2.txt        2021-03-31 12:21:24.506662618 +0300
@@ -1,6 +1,7 @@
 The
-bright
+dark
 side
 of
 the
-sun
+moon
+
```

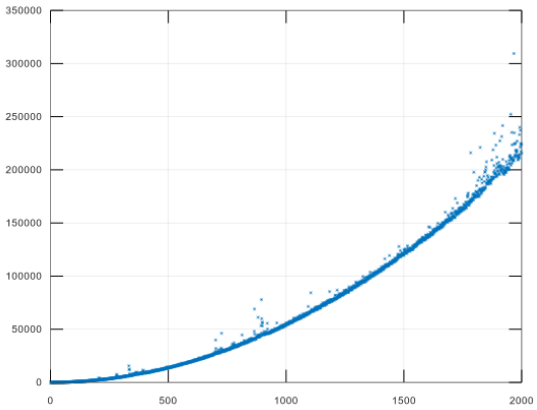
Тест производительности



(a) Построение diff на одинаковых файлах



(b) Построение diff между файлом и его перемешенной версий



(c) Построение diff между файлом и его обратной версий

Построение diff на одинаковых файлах быстрое, так как алгоритм останавливается за один шаг.

Как и объявлено, сложность алгоритма очень схожа с $O((M+N)D)$. Особенно это видно на последнем графе, где, из-за того, что строки файла в обратном порядке, требуется заменить все строки, и глубина $D = N + M$.

Выводы

Кроме своего стандартного использования (сравнения файлов), diff еще можно использовать для:

- Построение patch-файлов. Они могут понадобиться для изменения конкретных частей файла без изменения его всего (в отличие от передачи всего файла);
- Минимальная передача данных при синхронизации бинарных файлов за счет проверки их на одинаковость и отправки только отличий (**rsync**).
- Противу интуитивным понятиям, diff можно обобщить до сравнения всего, что можно редактировать, в том числе двух бинарных файлов.
- Более специфично: обнаружение и сравнение мутации ДНК.

Стоит заметить, для большинства вводов существует более чем одна минимальная разность файлов, но одна разность может иметь раскинутые действия по всему файлу, а другая будет иметь сгруппированные в одном месте действия. Далее рассматриваются четыре алгоритма, встроенные в **git diff**:

1. Алгоритм **Юджина Майерса** в линейном пространстве – стандартный применяемый алгоритм, используемый при вызове утилиты. Чаще всего выводит хорошие результаты за быстрое время и малую память, но на некоторых вводах запинается и выводит сильно смешанный вывод. Другой флаг *–minimal* работает на том же алгоритме, но рассматривает больше промежуточных вариантов для вывода чего-то более читабельного. Можно применять, когда нет причин использовать другие алгоритмы.
2. Алгоритм **patience** – алгоритм делит файл на секции используя общие строки, которые не повторяются ни где в самих документах. Деление файлов на общие секции и поиск изменений в самих секциях выводит сгруппированную разницу чаще, чем у Майерса. Его предпочтительнее применять в файлах, где было элементы больше менялись местами; Пространственная сложность линейная; про алгоритмическую сложность описания нет.
3. Алгоритм **histogram** – является модификацией patience, превосходящий по скорости и результатам и Майерса, и patience. В отличие от patience, ищет не уникальные элементы, а наименее повторяющиеся. Следует применять при сравнении исходного кода.

Один из неупомянутых алгоритмов (который был введен столько раз, что уже неизвестно кто первым его предумал): алгоритм Вагнера-Фишера, который считает расстояния между каждым префиксом обоих массивов, что имеет предположительно пространственную и алгоритмическую сложность $O(NM)$