Лабораторная работа № 4 по курсу дискрeтного анализа: Строковые алгоритмы

Выполнил студент группы М8О-208Б-20 МАИ Фаттяхетдинов Сильвестр.

# Условие

Кратко описывается задача:

1. Необходимо реализовать один из стандартных алгоритмов поиска образцов для указанного алфавита.
2. Алгоритм Апостолико-Джанкарло, над алфавитом слов, состоящих не более, чем из 16 латинских букв.

# Метод решения

# Ввод в данной задаче не самый тривиальный, т.к. алфавитом являются не буквы, а строки. Информация о паттерне хранится в одном векторе строк, информация текста в векторе строк (его содержимое) и в векторе чисел (длины строк, они понадобятся позже для корректного вывода).

# Перед началом поиска алгоритм Апостолико-Джанкарло, как и алгоритм Бойера-Мура, модификацией которого он и является, необходимо провести некоторые подготовительные операции. Вычисляются N-функция, L-функция, а также подготавливается таблица вхождений символов в паттерн. После этого запускается сам поиск. Сравнение происходит начиная с позиции h = pattern.size + 1, производится оно справа налево. В отличие от обычного Бойера-Мура, где сравнение происходит со смещением на 1 символ, для этого используются данные из вектора M, длина которого совпадает с длиной текста, заполняется он прямо во время работы поиска, а не препроцессинга. Если выявлено совпадение – добавляем индекс в вектор positions, прикладываем паттерн на 1 символ правее. Если нет – считаем сдвиг по правилу плохого символа, по правилу плохого суффикса. Выбираем из них максимальное значение и сдвигаем паттерн. После работы поиска, который возвращает вектор индексов, где было найдено совпадение, при помощи ранее сформированного вектора длин строк считается, в какой строке и на какой позиции от начала этой строки было найдено совпадение, после чего выводится на экран.

# Описание программы

Вся программа содержится в одном файле main.cpp

#include <iostream>

#include <string>

#include <vector>

#include <map>

#include <algorithm>

#include <sstream>

std::pair<int, int> GetStringNumber(int number, const std::vector<int> &string\_lengths) {

    int string\_number = 0;

    while (number - string\_lengths[string\_number] > 0) {

        number -= string\_lengths[string\_number];

        ++string\_number;

    }

    ++string\_number;

    return {string\_number, number};

}

std::map<std::string, std::vector<int>> Preprocess(std::vector<std::string> &pattern) {

    std::map<std::string, std::vector<int>> result;

    for (int i = pattern.size() - 1; i >= 0; --i) {

        result[pattern[i]].push\_back(i);

    }

    return result;

}

std::vector<int> ZFunction(const std::vector<std::string> &s) {

    std::vector<int> z(s.size());

    int idx\_begin = 0, idx\_end = 0;

    for (int i = 1; i < s.size(); ++i) {

        if (i <= idx\_end)

            z[i] = std::min(idx\_end - i + 1, z[i - idx\_begin]);

        while (i + z[i] < s.size() && s[z[i]] == s[i + z[i]])

            ++z[i];

        if (i + z[i] - 1 > idx\_end) {

            idx\_begin = i;

            idx\_end = i + z[i] - 1;

        }

    }

    return z;

}

std::vector<int> NFunction(std::vector<std::string> s) {

    std::reverse(s.begin(), s.end());

    std::vector<int> z = ZFunction(s), n(s.size());

    for (int i = 1; i < z.size(); ++i) {

        n[z.size() - i - 1] = z[i];

    }

    return n;

}

std::vector<int> LFuctionStrong(const std::vector<int> &n) {

    std::vector<int> l(n.size());

    for (int i = 0; i < n.size(); ++i) {

        if (n[i]) {

            l[n.size() - n[i]] = i;

        }

    }

    return l;

}

int GoodSuffixRule(const std::vector<int> &l, int i) {

    if (l.size() > i && l[i]) {

        return l.size() - l[i];

    }

    return 0;

}

int BadSymbolRule(std::map<std::string, std::vector<int>> &table, const std::string &c, int pos, const int &size) {

    if (!table[c].empty()) {

        for (auto elem: table[c]) {

            if (elem < pos) return pos - elem;

        }

    }

    return 1;

}

std::vector<int>

Search(const std::vector<std::string> &text, const std::vector<std::string> &pattern,

       std::map<std::string, std::vector<int>> &table,

       const std::vector<int> &N, const std::vector<int> &L, const std::vector<int> &string\_lengths) {

    int h = pattern.size() - 1;

    std::vector<int> M(text.size(), -1);

    std::vector<int> positions;

    while (h < text.size()) {

        bool flag = true;

        int position\_to\_stop = h - pattern.size();

        std::string mismatched = "?";

        int i = pattern.size() - 1;

        for (int j = h; j > position\_to\_stop;) {

            if (M[j] == -1 || (!M[j] && !N[i])) { // 1st case

                if (text[j] == pattern[i]) {

                    if (i > 0) {

                        --i;

                        --j;

                    } else {

                        break;

                    }

                } else {

                    M[h] = h - j;

                    flag = false;

                    break;

                }

            } else if (M[j] < N[i] && M[j]) {   // 2nd case

                j -= M[j];

                i -= M[j];

            } else if (M[j] == N[i] && M[j]) {  // 3rd case

                if (i == N[i]) {

                    M[h] = h - j;

                    break;

                }

                i -= M[j];

                j -= M[j];

            } else if (M[j] > N[i]) {           // 4th case

                if (i == N[i]) {

                    M[h] = h - j;

                    break;

                } else if (N[i] < i) {

                    M[h] = h - j + N[i];

                    flag = false;

                    break;

                } else {

                    if (text[j] == pattern[i]) {

                        if (i > 0) {

                            --i;

                            --j;

                        } else {

                            break;

                        }

                    } else {

                        M[h] = h - j;

                        flag = false;

                        break;

                    }

                }

            } else { //  5th case

                flag = false;

                break;

            }

        }

        if (flag) {

            positions.push\_back(h - pattern.size() + 1);

            ++h;

        } else {

            int bad\_symbol\_res = BadSymbolRule(table, mismatched, i, pattern.size());

            int good\_suffix\_res = GoodSuffixRule(L, h);

            h += std::max(bad\_symbol\_res, std::max(1, good\_suffix\_res));

        }

    }

    return positions;

}

int main() {

    std::ios::sync\_with\_stdio(false);

    std::cin.tie(nullptr);

    std::cout.tie(nullptr);

    std::string string, pstring;

    std::vector<std::string> pattern, text;

    std::vector<int> string\_lengths;

    std::getline(std::cin, pstring);

    std::stringstream pattern\_stream(pstring);

    while (pattern\_stream >> pstring) {

        std::transform(pstring.begin(), pstring.end(),

                       pstring.begin(), tolower);

        pattern.push\_back(pstring);

    }

    while (std::getline(std::cin, string)) {

        std::stringstream text\_stream(string);

        int string\_len = 0;

        while (text\_stream >> string) {

            ++string\_len;

            std::transform(string.begin(), string.end(),

                           string.begin(), tolower);

            text.push\_back(string);

        }

        string\_lengths.push\_back(string\_len);

    }

    auto N = NFunction(pattern);

    auto L = LFuctionStrong(N);

    auto table = Preprocess(pattern);

    std::vector<int> res = Search(text, pattern, table, N, L, string\_lengths);

    for (auto &r: res) {

        auto p = GetStringNumber(r + 1, string\_lengths);

        std::cout << p.first << ", " << p.second << "\n";

    }

    return 0;

}

# Дневник отладки

Во время реализации данного алгоритма мне пришлось столкнуться с двумя ошибками.

1. Мелкие ошибки в индексах. Где-то лишний «+1», где-то отсутствующий, из-за этого алгоритм ломался и выдавал неправильный ответ.
2. Из-за неоптимального подсчёта позиций символов в строках программа превышала лимит по времени.

Обе они были исправлены.

# Тест производительности

1) m = 100, n = 10000

Апостолико-Джанкарло – 6 ms

Наивный – 7 ms

2) m = 1000, n = 100000

Апостолико-Джанкарло – 46 ms

Наивный – 75 ms

3) m = 10000, n = 1000000

Апостолико-Джанкарло – 439 ms

Наивный – 830 ms

В самом деле, видно, что сложность алгоритма Апостолико-Джанкарло – линейная.

# Выводы

Благодаря данной лабораторной работе я узнал о существовании довольно интересной модификации алгоритма Бойера-Мура и вспомнил, как работает и реализовывается Z-функция. Также убедился в том, что сложность алгоритма – линейная, и он, в самом деле, быстрее наивного.