ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

Курсовой проект

по дисциплине «Объектно-ориентированное программирование»

«Алгоритм Рабина-Карпа, алгоритм Бойера-Мура-Хорспула»

Выполнил
студент гр. 3331506/20401 _____ Дубинин Б. В.
(подпись)

Работу принял _____ Ананьевский М.С.
(подпись)

Санкт-Петербург
2025

Введение

Алгоритмы поиска подстроки в тексте относятся к фундаментальным задачам обработки строк и имеют важное значение в компьютерных науках. Среди множества существующих алгоритмов есть алгоритмы Рабина-Карпа и алгоритм Бойера-Мура-Хорспула, каждый из которых предлагает свой подход к решению задачи поиска подстроки.

Предмет исследования: алгоритмы поиска подстроки в тексте.

Задача: изучить два классических алгоритма поиска подстроки: Рабина-Карпа и Бойера-Мура-Хорспула, провести их реализацию на языке программирования C++, исследовать их временную эффективность на различных типах входных данных (различные длины текста), провести сравнительный анализ полученных результатов, сделать выводы о эффективности каждого из алгоритмов.

Актуальность темы. Алгоритмы поиска подстроки находят широкое применение как в обычных пользовательских ПО, так и многочисленных областях компьютерных наук.

1 Алгоритм Рабина-Карпа

1.1 Описание алгоритма

Алгоритм Рабина-Карпа — это алгоритм, предназначенный для оптимального поиска подстроки в тексте. Основная идея алгоритма заключается в том, чтобы не искать заданный паттерн по символам, а использовать хэш-функцию, чтобы отсекать неподходящие участки текста.

Общий принцип работы:

Пусть m – длинна паттерна P, n – длинна текста T. Строка рассматривается как число в системе счисления d (для ASCII d = 256, для цифр d = 10 и т.д.). Считаем хэш паттерна, как

$$hashP = (T[0] * d^{m-1} + T[1] * d^{m-2} + \cdots + T[m-1] * d^{0}) \,\%\, q,$$

где q – модуль (большое простое число), от которого берется остаток от деления на каждой итерации. Это нужно для того, чтобы не выйти за рамки допустимого размера численных переменных, так как при вычислении хэша может получится большое число.

1. Берем первое вхождение текста от 0 до m-1 и считаем хэш этого «окна». Сравниваем с хэшем искомого паттерна.
2. Если хэш совпадает, то проводим сравнение посимвольно (для случая, если хэши совпали при символьной разнице). Если хэши не совпадают, то прокатываем «окно» на один символ и пересчитываем входящий хэш.
3. Новый хэш не считается для всего окна заново, а обновляется по формуле:

$$newHash = \big((oldHash - oldChar * d^{m-1} \,\%\, q) * d + newChar\big) \,\%\, q$$

4. Повторяем пункты 2, 3 до позиции n-(m-1).

1.2 Оценка сложности алгоритма

Временная сложность

1. Начальное вычисление хэша и быстрое возведения в степень с работой в битовом представлении числа степени, где

$$O(m+\log_2 m)$$

2. Основной цикл поиска со сдвигом «окна»:
   - O(n-m) для каждого сдвига
   - O(n-m+1) сравнений хыша окна и паттерна
   - В каждом удачном сравнении O(m) сравнений посимвольных

Получается, что в худшем случае, кода весь текст состоит из правильных вхождений (например «AAAA…A») сложность алгоритма:

$$O((m + log_2 m) + (n - m) * (n - m + 1) * m)$$

2. Алгоритм Бойера-Мура-Хорспула

2.1 Описание алгоритма

Алгоритм Прима — это алгоритм, также предназначенный для поиска подстроки в текстя

Основная идея алгоритма заключается, что алгоритм пытается совершить как можно большие "прыжки" по тексту, чтобы избежать ненужных сравнений. Он делает это, просматривая шаблон справа налево, используя предварительно вычисленную таблицу интервалов прыжков для символов, чтобы определить, на сколько символов можно сдвинуть шаблон вправо при несовпадении

Общий принцип работы:

Перед началом поиска создать таблицу интервалов. Для каждого возможного символа алфавита мы вычисляем, на сколько позиций можно сдвинуть шаблон, если этот символ в тексте не совпал с соответствующим символом в шаблоне. Для всех символов шаблона, кроме последнего, в шаблоне P, сдвиг равен:

$$shift = lengthP - \ i$$

где i – индекс символа в шаблоне P[0..m-2]. Для символов, которых нет в шаблоне, и для последнего символа сдвиг по равен длине шаблона m.

Далее в цикле:

1. Сравниваем первое вхождение текста длинной m справа налево. Произошло совпадение – записываем индекс.
2. Если на каком-то символе сравнения произошло несовпадение, то смещаемся на соответствующий таблице интервал.

2.2 Оценка сложности алгоритма

Временная сложность

1. Начальное построение таблицы сдвигов

O(m)

2. Основной цикл поиска:
    - O(n/m) – если нет вхождений
    - O(n*m) – если весь текст состоит из верных вхождений

Получается, что в худшем случае, кода весь текст состоит из правильных вхождений (например «АААА…А») сложность алгоритма:

$$O(m + n * m)$$

3. Экспериментальное исследование эффективности

Для экспериментального исследования были использованы файлы со 100, 1000, 10000, 1000000, 10000000, 100000000 символами. Синяя ветвь – алгоритм Рабин-Карпа, оранжевая, соотвественно, - алгоритм Бойера-Мура-Хорспула.

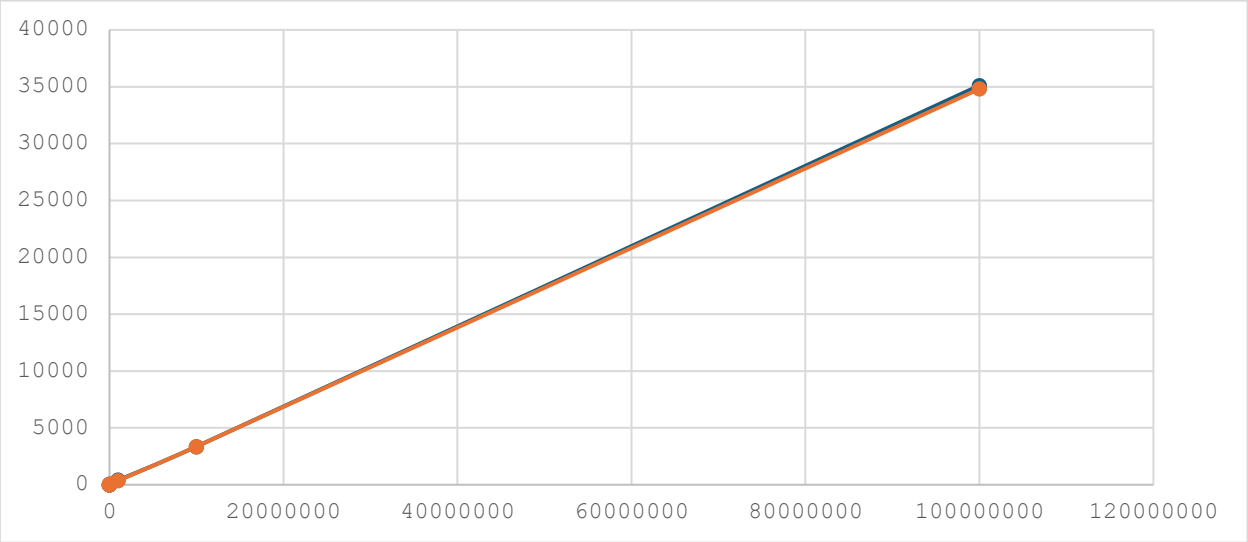

Рисунок 1 – Экспериментальное исследование эффективности

3 Выводы

В курсовой работе были рассмотрены и реализованы два алгоритма поиска подстроки в тексте: Алгоритм Рабина-Карпа, алгоритм Бойера-Мура-Хорспула. Оба алгоритма решают одну и ту же задачу, однако используют различные подходы.

Алгоритм Бойера-Мура-Хорспула обладает временной сложностью $O(m + n * m)$, что делает его предпочтительным для поиска подстроки, тогда как алгоритм Рабина-Карпа имеет сложность $O((m + log_2 m) + (n - m) * (n - m + 1) * m)$, это рассмотрение худших случаев. В ходе экспериментального исследования было выявлено, оба алгоритма при работе с реальным текстом демонстрирует примерно одинаковое время работы.

Список литературы

1. R. N. Horspool (1980) — «Practical fast searching in strings» в журнале Software: Practice and Experience, 10 (6): 501–506.

2. Prim R. C. Shortest connection networks and some generalizations // Bell System Technical Journal. — 1957. — Vol. 36. — P. 1389–1401. — DOI: 10.1002/j.1538-7305.1957.tb01515.x.

Приложение 1. Рабина-Карпа

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

class RabinKarp
{
private:
    const int base = 256;   // для ASCII
    const int module = 101;  // Большое простое число


    int pow(int a, int b) const
    {
        int result = 1;
        a %= module;

        while (b > 0) {
            if (b & 1) {
                result = (result * a) % module;
            }
            a = (a * a) % module;
            b >>= 1;
        }
        return result;
    }



    std::string read_file_content(const std::string& filename)
    {
        std::ifstream file(filename);

        if (!file.is_open()) {
            throw std::runtime_error("Cannot open file");
        }

        std::string content;
        std::string line;
        while (std::getline(file, line)) {
            content += line + "\n";
        }
        file.close();

        if (content.empty()) {
            throw std::runtime_error("File is empty");
        }

        return content;
```

```
    }


    void print_occurrences(const std::vector<int>& occurrences)
    {
        std::cout << "Found " << occurrences.size() << "
occurrences at positions: ";
        for (int pos : occurrences) {
            std::cout << pos << " ";
        }
        std::cout << "\n";
    }



    std::vector<int> search_all(const std::string& text, const
std::string& pattern) const
    {
        std::vector<int> indices;
        int n = text.length();
        int m = pattern.length();

        if (m == 0) {
            throw std::invalid_argument("Pattern is empty");
        }
        if (n == 0) {
            throw std::invalid_argument("Text is empty");
        }
        if (m > n) {
            throw std::invalid_argument("Pattern length longer
than text length");
        }

        int h = pow(base, m - 1);


        int hash_pattern = 0;
        int hash_text = 0;

        for (int i = 0; i < m; i++) {
            hash_pattern = (base * hash_pattern + pattern[i]) %
module;
            hash_text = (base * hash_text + text[i]) % module;
        }


        for (int i = 0; i <= n - m; i++) {
            if (hash_pattern == hash_text) {
                bool match = true;
                for (int j = 0; j < m; j++) {
                    if (text[i + j] != pattern[j]) {
                        match = false;
                        break;
```

```
                    }
                }
                if (match) {
                    indices.push_back(i);
                }
            }


            if (i < n - m) {
                hash_text = (base * (hash_text - text[i] * h) +
text[i + m]) % module;

                if (hash_text < 0) {
                    hash_text += module;
                }
            }
        }

        return indices;
    }

public:
    void search_in_file(const std::string& filename, const
std::string& pattern)
    {
        std::string text = read_file_content(filename);
        print_occurrences(search_all(text, pattern));
    }
};


int main()
{
    RabinKarp rk;
    const std::string pattern = "abc";
    const std::string filename = "text.txt";

    try {
        auto start = std::chrono::high_resolution_clock::now();
        rk.search_in_file(filename, pattern);
        auto end = std::chrono::high_resolution_clock::now();
        auto duration =
std::chrono::duration_cast<std::chrono::microseconds>(end -
start);
        std::cout << "Время выполнения: " << duration.count() /
1000.0 << " миллисекунд" << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << "\n";
    }

    return 0;
}
```

Приложение 2. Алгоритм Бойера-Мура-Хорспула

```
#include <iostream>
#include <string>
#include <vector>
#include <unordered_map>
#include <fstream>
#include <stdexcept>
#include <chrono>

class BoyerMooreHorspool
{
private:
    std::unordered_map<char, int> build_shift_table(const
std::string& pattern)
    {
        std::unordered_map<char, int> shift_table;
        int pattern_length = pattern.length();


        if (pattern_length == 0) {
            throw std::invalid_argument("Pattern cannot be
empty");
        }

        for (int i = 0; i < pattern_length - 1; i++) {
            shift_table[pattern[i]] = pattern_length - 1 - i;
        }

        shift_table[pattern[pattern_length - 1]] =
pattern_length;

        return shift_table;
    }



    std::string read_file_content(const std::string& filename)
    {
        std::ifstream file(filename);

        if (!file.is_open()) {
            throw std::runtime_error("Cannot open file");
        }

        std::string content;
        std::string line;
        while (std::getline(file, line)) {
            content += line + "\n";
        }
        file.close();

        if (content.empty()) {
```

```
            throw std::runtime_error("File is empty");
        }

        return content;
    }



    void print_occurrences(const std::vector<int>& occurrences)
    {
        std::cout << "Found " << occurrences.size() << "
occurrences at positions: ";
        for (int pos : occurrences) {
            std::cout << pos << " ";
        }
        std::cout << "\n";
    }



    std::vector<int> search_all(const std::string& text, const
std::string& pattern)
    {
        std::vector<int> occurrences;
        int text_length = text.length();
        int pattern_length = pattern.length();


        if (pattern_length == 0) {
            throw std::invalid_argument("Pattern is empty");
        }
        if (text_length == 0) {
            throw std::invalid_argument("Text is empty");
        }
        if (pattern_length > text_length) {
            throw std::invalid_argument("Pattern length longer
than text length");
        }

        std::unordered_map<char, int> shift_table =
build_shift_table(pattern);

        int i = 0;
        while (i <= text_length - pattern_length) {
            int j = pattern_length - 1;


            while (j >= 0 && text[i + j] == pattern[j]) {
                j--;
            }

            if (j < 0) {
                occurrences.push_back(i);
```

```
                i++;
            } else {
                char mismatch_char = text[i + pattern_length -
1];

                int shift = pattern_length;
                if (shift_table.find(mismatch_char) !=
shift_table.end()) {
                    shift = shift_table[mismatch_char];
                }

                i += shift;
            }
        }

        return occurrences;
    }

public:
    void search_in_file(const std::string& filename, const
std::string& pattern)
    {
        std::string text = read_file_content(filename);
        print_occurrences(search_all(text, pattern));
    }
};



int main()
{
    BoyerMooreHorspool bmh;
    const std::string pattern = "abc";
    const std::string filename = "text.txt";

    try {
        auto start = std::chrono::high_resolution_clock::now();
        bmh.search_in_file(filename, pattern);
        auto end = std::chrono::high_resolution_clock::now();
        auto duration =
std::chrono::duration_cast<std::chrono::microseconds>(end -
start);
        std::cout << "Время выполнения: " << duration.count() /
1000.0 << " миллисекунд" << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << "\n";
    }

    return 0;
}
```