Московский авиационный институт (национальный исследовательский университет)

Факультет компьютерных наук и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №4 по курсу «Дискретный анализ»

Студент: Л. А. Постнов
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б-22

Дата: 05.06.2024
Оценка:
Подпись:

Москва, 2024

Лабораторная работа №4

Задача: Необходимо реализовать один из стандартных алгоритмов поиска образцов для указанного алфавита.

Вариант алгоритма: Поиск одного образца при помощи алгоритма Апостолико-Джанкарло.

Вариант алфавита: Слова не более 16 знаков латинского алфавита (регистронезависимые).

Запрещается реализовывать алгоритмы на алфавитах меньшей размерности, чем указано в задании.

1 Описание

Основная идея алгоритма Апостолико-Джанкарло состоит в том, чтобы модифицировать алгоритм Бойера-Мура таким образом, чтобы опускать заведомо неприводящие к положительным результатам сравнения.

Про сложностьную оценку алгоритма сказано в [1]: Апостолико и Джанкарло предложили вариант алгоритма Бойера-Мура, который допускает замечательно простое доказательство линейной оценки наихудшего времени счета. В этом варианте никакой символ из T не участвует в сравнениях после его первого совпадения с каким-нибудь символом из P. Отсюда немедленно следует, что число сравнений не превзойдет 2m. Каждое сравнение дает либо совпадение, либо несовпадение; последних может быть только T, так как при каждом несовпадении происходит ненулевой сдвиг P а совпадений - не больше T, так как никакой символ T не сравнивается после совпадения с символом из P. Мы покажем, что и остальная вычислительная работа (кроме сравнений) в этом методе линейно зависит от T.

Для этого мы будем держать массив, который равен по размерам тексту, где в каждой ячейке будем хранить число, которое означет, сколько позиций совпадет с суффиксом шаблона начиная с этой позиции при движении в сторону начала текста. Этот массив поможет нам опускать ненужные сравнения.

2 Исходный код

Рассмотрим функции, которые нужны для работы сдвигов по алгоритму Бойера-Мура

```
std::map<std::string, std::vector<int>> bad_symbol_prerocess(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> z_function(const std::vector<std::string> &s) {
    int n = s.size();
    std::vector<int> z(n);
    int left = 0, right = 0;

    for (int i = 1; i < n; ++i) {
        if (i <= right)
            z[i] = std::min(right - i + 1, z[i - left]);
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            ++z[i];
        if (i + z[i] - 1 > right) {
            left = i;
            right = i + z[i] - 1;
        }
    }

    return z;
}

std::vector<int> n_function(std::vector<std::string> s) {
    std::reverse(s.begin(), s.end());
    std::vector<int> z = z_function(s), n(s.size());
    for (int i = 1; i < z.size(); ++i) {
        n[z.size() - i - 1] = z[i];
    }
    return n;
}

std::vector<int> l_fuction(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 good_suffix(const std::vector<int> &l, int i) {
    if (l.size() > i && l[i]) {
        return l.size() - l[i];
    }
    return 0;
}


int bad_symbol(std::map<std::string, std::vector<int>> &table, const std::string &c,
    int pos) {
    if (!table[c].empty()) {
        for (auto elem: table[c]) {
            if (elem < pos) return pos - elem;
        }
    }
    return 1;
}
```

Рассмотрим реализацию самой функции поиска:

```
std::vector<int> Search(std::vector<std::string> &text, std::vector<std::string> &
    pattern) {
    auto N = n_function(pattern);
    auto L = l_fuction(N);
    auto table = bad_symbol_prerocess(pattern);

    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] == 0) && (N[i]) == 0)) {
                if (text[j] == pattern[i]) { // 1st case
                    if (i > 0) {
                        --i;
                        --j;
                    } else {
                        break;
                    }
                } else {
                    mismatched = text[j];
                    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) {
                mismatched = pattern[i - N[i] - 1];
                M[h] = h - j;
                flag = false;
                break;
            } else {
                break;
            }
        } else { // 5th case
            flag = false;
            break;
        }
    }
    if (flag) {
        positions.push_back(h - pattern.size() + 1);
        ++h;
    } else {
        int bad_symbol_res = bad_symbol(table, mismatched, i);
        int good_suffix_res = good_suffix(L, h);
        h += std::max(bad_symbol_res, std::max(1, good_suffix_res));
    }
  }
  return positions;
}
```

3 Консоль

```
wednees@MSI:/mnt/c/Users/leoni/OneDrive/Desktop/study/Discrete_Analysis/lab4$
./solution
cat dog cat dog bird
CAT dog CaT Dog Cat DOG bird CAT
dog cat dog bird
1,3
1,8
```

4 Тест производительности

Тесты производительности представляют из себя следующее: наивный алгоритм поиска сравнивается с алгоритмом Апостолико-Джанкарло. Тестов будет три: на 10^4, 10^5 и 10^6 строк.

```
wednees@MSI:/mnt/c/Users/leoni/OneDrive/Desktop/study/Discrete_Analysis/
lab4$ ./benchmark <test10000.txt
AG: 4ms
Naive: 6ms
wednees@MSI:/mnt/c/Users/leoni/OneDrive/Desktop/study/Discrete_Analysis/
lab4$ ./benchmark <test100000.txt
AG: 39ms
Naive: 78ms
wednees@MSI:/mnt/c/Users/leoni/OneDrive/Desktop/study/Discrete_Analysis/
lab4$ ./benchmark <test1000000.txt
AG: 429ms
Naive: 833ms
```

Как можно заметить, алгоритм Апостолико-Джанкарло работает быстрее, что связано с тем, что сложность наивного алгоритма составляет $O(n * m)$, в то время как сложность алгоритма Апостолико-Джанкарло $O(m + n)$, где n - длина исходного текста, а m - длина шаблона.

5 Выводы

Выполняя данную лабораторную работу по курсу «Дискретный анализ», я познакомился с алгоритмом Апостолико-Джанкарло, разбрался в нем и реализвал на языке программирования. Помимо этого алгортима, потрабовалось еще хорошо разобраться в алгоритме Бойера-Мура, так как алгоритм Апостолико-Джанкарло является его модификацией.

Помимо изучения алгоритмов, я также узнал про работу некоторых функций из стандартной библиотеки C++, предназнаяенных для работы со строчными данными, так как мой вариант представляет в качестве алфавита имеет не просто символы, как в обычном алфавите, а слова, и, как следствие, нужно было подумать, как огранизовать ввод таких данных.

Список литературы

[1] Дэн Гасфилд. *Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология.* Издательский дом «Невский Диалект БХВ-Петербург», Санкт-Петербург, 2003. Перевод с английского И. В. Романовского.