Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №4 по курсу «Дискретный анализ»

Студент: Р. Р. Гаптулхаков
Преподаватель: Н. С. Капралов
Группа: М8О-208Б

Дата: 05.01.21
Оценка:
Подпись:

Москва, 2021

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.

Формат входных данных:

Искомый образец задаётся на первой строке входного файла.

В случае, если в задании требуется найти несколько образцов, они задаются по одному на строку вплоть до пустой строки.

Затем следует текст, состоящий из слов или чисел, в котором нужно найти заданные образцы.

Никаких ограничений на длину строк, равно как и на количество слов или чисел в них, не накладывается.

Формат выходных данных:

В выходной файл нужно вывести информацию о всех вхождениях искомых образцов в обрабатываемый текст: по одному вхождению на строку.

Для заданий, в которых требуется найти только один образец, следует вывести два числа через запятую: номер строки и номер слова в строке, с которого начинается найденный образец. В заданиях с большим количеством образцов, на каждое вхождение нужно вывести три числа через запятую: номер строки; номер слова в строке, с которого начинается найденный образец; порядковый номер образца.

Нумерация начинается с единицы. Номер строки в тексте должен отсчитываться от его реального начала (то есть, без учёта строк, занятых образцами).

Порядок следования вхождений образцов несущественен.

1 Описание

Алгоритм Апостолико-Джанкарло является продвинутым алгоритмом алгоритма Боера-Мура. Мы смещаем паттерн слева направо, а сравнение происходит справа налево. При этом, чтобы обеспечить минимальное количество смещений, мы проводим предварительную обработку паттерна.

1 Препроцессинтнг

Препроцессинг - это обработка шаблона для получения определённых данных. Необходимыее данные, полученные в результате препроцессинга, мы будем использовать, чтобы минимизировать колличество сравнинийй.

2 Этапы препроцессинтнга

1. **Nfunction.** Результатом работы функции *Nfunction* будет массив N размером равным длине образа , в i элементе которого, хранится длина суффикса образа совпавшего с длиной суффикса подстроки заканчивающейся в i позиции образа. Для получения данного массива можно использовать z-функцию. Мы ищем z-функцию от реверсированного образа, результататом будет массив, который нужно будет реверсировать.

2. **Lfunction.** *Lfunction* необходим для нахождения позиции, в которой Nj совпадает с суфиксом всей строки. Результатом работы этой функции будет массив, который будет использоваться для нахождения хорошего суффикса.

3. **lfunction.** *lfunction* тоже необходим для нахождения хорошего суффикса, в случае если в правой части от i не находится суффикса образца, мы находим максимальный суффикс совпадающий с префиксом образцп и затем смещаем наш паттерт так, чтобы префикс совпадал с суффиксом.

3 Поиск

Процесс поиска включает в себя движение шаблона по тексту слева направо, а сравнение происходит справа налево. Для реализации алгоритма нужно ввести массив **m**, в котором хранится информация о том, сколько точно символов совпадёт, если приложить к позиции текста конец паттерна. Это число не обязательно максимально. Рассмотрим подробнее, что в каком случае происходит:

1. **Случай 1.** *(m[i] == 0 или m[i] неопределено)* В этом случае мы сравниваем символы паттерна и текста до момента, когда i = 0. Тогда мы находим вхождение и m[k] равно размеру паттерна, где k - позиция в тексте, к которой приложен конец паттерна.

2. **Случай 2.** *(m[h] < n[i])* Здесь мы смещаемся на i -= m[h], пропуская совподающую часть.

3. **Случай 3.** *(m[h] >= n[i] && n[i] == i)* Здесь мы получаем совпадение образца с текстом. Определяем m[k], как разность k и h. Смещаем, используя информацию из *lfunction*.

4. **Случай 4.** *(m[h] > n[i] && n[i] < i)* Здесь мы получаем несовпадение(следствие из опреденения функций). Определяем m[k], как разность k и h. Смещаем, используя информацию из *lfunction*.Смещение бедётся, как max(1, ПХС, ППС).

5. **Случай 5.** *(m[h] > n[i] && n[i] < i)* Пропускаем очевидно совпадающую часть и продолжаем сравнивать с новых позициях паттерна и текста.

2 Исходный код

Проект состоит из 4 файлов: main.cpp, preprocessing.hpp, search.hpp, n_function.hpp, makefile.

1. **main.cpp**: Основной файл, в котором происходи смена регистра, преобразование текста в строку, подсчёт слов, позиции слова в тексте(строка; номер в строке), начальная и конечная позиция слова, создание всех структур данных для хранения результатов препроцессинга;

2. **preprocessing.hpp**: Реализованы все функции препроцессинга;

3. **search.cpp**: Поиск;

Таблица методов и функций

main.cpp	
Функция	Описание
int main()	Главная функция, начальная точка работы программы.
search.hpp	
Тип данных	Описание
void Search()	Поиск.
preprocessing.hpp	
Функция	Описание
void Preprocessing(...)	Обработак паттерна.
void LBigFunction(...)	Получение L-большое.
void LLittleFunction(...)	Получение L-малое.
void RFunction(...)	Получение массива смещения для правила плохого символа.
n_function.hpp	
Функция	Описание
void NFunction(const std::string &s, std::vector<T> &v)	Нахождение массива N.

3 Тест производительности

Для сравнения скорости работы алгоритма я взял наивный алгоритм поиска подстроки в строке, сложность которого O(n*m). Я производил тестирование более, чем на трёх тестах, в каждом из которых мой алгоритм работал медленнее. Для замерки времени использовалась бибилиотека **chrono**.
Наивный алгоритм:

```
std::vector<int> Find(std::string t, std::string p){
    size_t m = p.size() - 1;
    size_t n = t.size();
    std::vector<int> result;

    for(size_t i = 0, j = 0; i < n; ++i)
    {
        while(i < n && j != m && t[i] == p[j])
        {
            ++i;
            ++j;
        }

        if(j == m)
        {
            result.push_back(i-m);
            j = 0;
        }
    }
    return result;
}
```

4 Выводы

Выполнив четвёртую лабораторную работу по курсу «Дискретный анализ», я научился реализовывать один из изученных алгоритмов с последующим применением его «в бою», то есть применение его на практике. Это помогло мне понять плюсы и минусы данной структуры и случаи, когда его стоит использовать. Правильное тестирование помогает на раннем этапе исправлять замеченные ошибки. Также были изучены такие алгоритмы, как: КМП, Боера-Мура, Ахо-Корасика, Рабина — Карпа. Стоит отменить линейную сложность моего алгоритма O(n+m). Он явлвяется достаточно эффективным на обычных текстах.

Список литературы

[1] Ден Гасфилд. *Алгоритмы: Строки деревья и последовательности в алгоритмах.* — Издательский дом «Невский диалект», 2003. Перевод с английского: И. В. Романовский. — 654 с. (ISBN 5-7940-01030-8 (рус.))