Лабораторная работа № 4 по курсу дискретного анализа: Строковые алгоритмы

Выполнил студент группы 08-208 МАИ *Былькова Кристина.*

Условие

Общая постановка задачи: Необходимо реализовать один из стандартных алгоритмов поиска образцов для указанного алфавита.

Искомый образец задаётся на первой строке входного файла. 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.

В выходной файл нужно вывести информацию о всех вхождениях искомых образцов в обрабатываемый текст: по одному вхождению на строку. Нумерация начинается с единицы. Номер строки в тексте должен отсчитываться от его реального начала (то есть, без учёта строк, занятых образцами).

Вариант алгоритма: Поиск одного образца при помощи алгоритма Кнута-Морриса-Пратта

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

Метод решения

Моя программа работает следующим образом: изначально я считываю весь паттерн в буфер, затем разделяю его на слова и помещаю их в отдельный вектор pattern. Затем я вызываю префикс-функцию для паттерна. Далее работаю с текстом. Также считываю его в буффер и записываю слова в вектор text. Когда его размер превышает 2*размер pattern, я вызываю свой алгоритм КМП. Затем я очищаю уже проверенную часть текста и продолжаю записывать оставшуюся, благодаря чему более эффективно расходуется память.

Сам алгоритм Кнута-Морриса-Пратта предполагает использование префикс-функции. Для более эффективной работы я писала сильную префикс-функцию, использующую Z-функцию.

Начнем с Z-функции. Вообще Z-функция от строки str определяется как массив z, такой что z_i равно длине максимальной подстроки, начинающейся с i-й позиции, которая равна префиксу s. В своем коде я использовала быстрый подсчет Z-функции: Идем слева направо и храним z-блок — самую правую подстроку, равную префиксу, которую успели обнаружить. Границы блока обозначаются l и r. Далее ищем z_i. Мы знаем, что i-й символ может лежать либо правее z-блока, либо внутри него:

- Если правее, то мы просто наивно перебором найдем z_i (максимальный отрезок, начинающийся с str_i и равный префиксу), и объявим его новым z-блоком;

- Если i-й элемент лежит внутри z-блока, то мы можем посмотреть на значение z_{i-l} и использовать его, чтобы инициализировать z_i, выбирая минимум между z_{i-l} и r-i+1. Так как, если z_{i-l} левее правой границы z-блока, то $z_i = z_{i-l}$ и больше быть не может. Если же упирается в границу, то "обрезаем"его до неё и будем увеличивать на единичку.

Префикс-функция представляет собой массив длины n, i-ый элемент которого определяется следующим образом: это длина наибольшего собственного суффикса подстроки s[0...i], совпадающего с её префиксом (собственный суффикс — значит не совпадающий со всей строкой). Сильная префикс-функция вычисляется на основе Z-функции следующим образом: мы идем по массиву справа налево и задаем значения префикс-функции: sp[i + z[i] - 1] = z[i].

Алгоритм КМП реализован следующим образом: Задаем n − длина текста, а m − длина паттерна. Сначала запускаем основной цикл поиска, выполняется пока i не превысит n - m. В цикле инициализируем индекс для шаблона j. Далее запускаем внутренний цикл проверки совпадения символов, если они совпадают, то увеличиваем индекс j. Если длина паттерна совпадает с j, значит весь паттерн совпал с частью текста − выводим номер строки и слова. Если частичное совпадение произошло и j больше, чем значение в sp[j - 1] + 1, то сдвигаем индекс i с учетом значения из sp: i = i + j - sp[j - 1] - 1. Затем увеличиваем i для продолжения поиска.

Описание программы

Были написаны структура и функции:

- struct TWord − структура слова, предназначенная для хранения самого слова, его размера, хеша (для быстрого сравнения), номера и номера строки, в которой это слово встречается.
- std::vector<int> ZFunction(const std::vector<TWord> str) − Z-функция
- std::vector<int> SPFunction(const std::vector<TWord> str) − сильная префикс-функция
- void KMP(const std::vector<TWord> pattern, const std::vector<TWord> text, const std::vector<int> sp, int start) − функция алгоритма Кнута-Морриса-Пратта

Дневник отладки

Основной проблемой было превышение лимита памяти. Изначально я не использовала буффер, а вызывала алгоритм поиска сразу для всего текста, что крайне негативно сказывалось на затрачиваемой памяти. Именно в этом и была ошибка. После её устранения, программа прошла все тесты.

Тест производительности

Для проверки производительности моего КМР алгоритма я использовала сравнение с наивным алгоритмом поиска. Сравнение производилось на входных данных размеров 10^3, 10^4 и 10^5.

```
kristinab@LAPTOP-SFU9B1F4:~/ubuntu_main/DA_labs/lab4$ g++ lab4.cpp && ./a.out
Test: 10^3:
NaiveSearch: 0.822 ms
KMP: 0.049 ms
----------
Test: 10^4:
NaiveSearch: 5.418 ms
KMP: 0.564 ms
----------
Test: 10^5:
NaiveSearch: 25.715 ms
KMP: 2.412 ms
```

Благодаря замерам видно, что наивный алгоритм поиска сильно уступает алгоритму Кнута-Морриса-Пратта. Это происходит изж-за того, что классический алгоритм поиска допускает лишние сравнения на этапе поиска образца в тексте, а алгоритм с применением префикс-функции – нет. Обработка таких сравнений длится дольше, чем предпроцессинг префикс функции.

Выводы

В результате данной лабораторной работы была написана и отлажена программа на языке C++, реализующая поиск одного образца при помощи алгоритма Кнута-Морриса-Пратта. Я реализовала алгоритм КМП с использованием сильной префикс-функции и Z-функции. Сложность моей реализации алгоритма Кнута-Морриса-Пратта $O(n * m)$, где n – длина текста, а m – длина паттерна.

В общем случае, задачи поиска подстроки в строке часто встречаются в жизни. Один из очевидных примеров это поиск контента в Интернете по ключевому слову или по ключевой фразе.