Лабораторная работа № 5 по курсу дискретного анализа: Суффиксные деревья

Выполнил студент группы М8О-308Б-21 МАИ *Белоносов Кирилл.*

Условие

1. Общая постановка задачи: Реализовать алгоритм построения суффиксного массива и поиска всех вхождений паттерна в текст с помощью него.

2. Вариант задания(2):

   Алгоритм: Найти в заранее известном тексте поступающие на вход образцы с использованием суффиксного массива.
   Формат ввода: Текст располагается на первой строке, затем, до конца файла, следуют строки с образцами.
   Формат вывода: Для каждого образца, найденного в тексте, нужно распечатать строчку, начинающуюся с последовательного номера этого образца и двоеточия, за которым, через запятую, нужно перечислить номера позиций, где встречается образец в порядке возрастания.

Метод решения

Для нахождения всех вхождений паттерна в текст необходимо создать суффиксный массив. Суффиксный массив - структура данных содержащая индексы суффиксов текста, осортированные в лексикографическом порядке. Это можно сделать двумя способами: С помощью существующего суффиксного дерева за линейное время или за O(nlogn) из исходного текста. Выберем второй метод. Данный алгоритм работает на принципе цифровой сортировки циклических подстрок. Для начала допишем к нашему тексту терминирующий символ, который не может встречаться в тексте. Далее отсортируем наш элементы нашего текста. Благо, зная алфавит, мы можем сделать это за линейную сложность сортировкой подсчетом (необходимо использовать её стабильную версию). Далее занесем каждый элемент в класс эквивалентности, созданный по позрастанию.

Предположим мы выполнили действие на i шаге, выполним действие на i + 1 шаге. Мы отсортирвали элементы размером 2^i, и дописав циклически еще элемент, мы можем их отсортировать за линейное время. Для этого нам достаточно сдвинуть циклически все индексы на текущую степень двойки, тогда мы получим все элементы отсортированные по второму индексу (Как раз здесь и используется принцип цифровой сортировки). Теперь нам достаточно отсортирвать все элементы с помощью предыдущих классов эквивалентности. Чтобы поставить новые классы эквивалентности, достаточно сравнить пары предыдущих элементов и в случае различия добавить еще один класс эквивалентности.

Для поиска же достаточно написать бин поиск по суффиксному массиву. В случае вхождения паттерна, достаточно проверить его окрестность, чтобы найти все вхождения

Описание программы

Сначала опишем функцию создания суффиксного массива. Она будет принимать константный указатель на текст и размер алфавита (Я использовал 256 для работы со всей таблицей ASCII)

```
std::vector<int> CreateSuffixArray(const std::string& s, size_t
    ALPHABET) {
     int n = s.size();
     std::vector<int> p(n), cnt(ALPHABET), c(n);
     for(int i = 0; i != n; ++i)
         ++cnt[s[i]];
     for(int i = 1; i != ALPHABET; ++i)
         cnt[i] += cnt[i - 1];
     for(int i = n - 1; i >= 0; --i)
         p[--cnt[s[i]]] = i;
     c[p[0]] = 0;
     int classes = 1;
     for(int i = 1; i != n; ++i) {
         if(s[p[i]] != s[p[i - 1]])
             ++classes;
         c[p[i]] = classes - 1;
     }
     std::vector<int> pn(n), cn(n);
     for(int h = 0; (1 << h) < n; ++h) {
         std::vector<int> cnt_loc(classes);
         for(int i = 0; i != n; ++i) {
             pn[i] = p[i] - (1 << h);
             if (pn[i] < 0)
                 pn[i] += n;
         }
         for(int i = 0; i != n; ++i)
             ++cnt_loc[c[pn[i]]];
         for(int i = 1; i != classes; ++i)
             cnt_loc[i] += cnt_loc[i - 1];
         for (int i = n - 1; i >= 0; --i)
             p[--cnt_loc[c[pn[i]]]] = pn[i];
         cn[p[0]] = 0;
         classes = 1;
         for (int i = 1; i != n; ++i) {
             int mid1 = (p[i] + (1 << h)) % n,  mid2 = (p[i - 1] + (1
                << h)) % n;
```

```
            if (c[p[i]] != c[p[i-1]] || c[mid1] != c[mid2])
                ++classes;
            cn[p[i]] = classes-1;
        }
        c = cn;
    }
    return p;
}
```

Теперь напишем функцию для поиска. Возвращать все вхождения будет удобно в виде вектора индексов. Также в задании просят выводить их в отсортированном виде, для этого используем функцию sort. Сам алгоритм представляет собой бинарный поиск по суффиксному массиву. Дополнительно напишем функции для сравнения подстрок текста с паттерном - stringsEqual и stringsLess

```
bool stringsEqual(const std::string& s1, const std::string& s2, int
   index) {
    if(s1.size() - index < s2.size()) {
        return false;
    } else {
        for(int i = 0; i < s2.size(); ++i) {
            if(s1[i + index] != s2[i]) {
                return false;
            }
        }
        return true;
    }
}

bool stringsLess(const std::string& s1, const std::string& s2, int
   index) {
    for(int i = 0; i < std::min(s2.size(), size_t(s1.size() -
       index)); ++i) {
        if(s1[i + index] < s2[i]) {
            return true;
        }
        if(s1[i + index] > s2[i]) {
            return false;
        }
    }
    if(s1.size() - index < s2.size()) {
        return true;
    } else {
        return false;
    }
}
```

```
std::vector<int> SuffixArraySearch(const std::string& pattern, const
   std::string& text, const std::vector<int>& suffixArray)
{
    int n = text.size();
    int m = pattern.size();
    int l = 0, r = n - 1;
    std::vector<int> indices;
    if(m == 0)
        return indices;
    while (l <= r)
    {
        int mid = l + (r - l)/2;

        if(stringsEqual(text, pattern, suffixArray[mid])) {
            indices.push_back(suffixArray[mid] + 1);
            int tmp = mid - 1;
            while(tmp >= 0 && stringsEqual(text, pattern,
               suffixArray[tmp])) {
                indices.push_back(suffixArray[tmp] + 1);
                --tmp;
            }
            tmp = mid + 1;
            while(tmp < n && stringsEqual(text, pattern,
               suffixArray[tmp])) {
                indices.push_back(suffixArray[tmp] + 1);
                ++tmp;
            }
            return indices;
        }
        if(stringsLess(text, pattern, suffixArray[mid])) {
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    }
    return indices;
}
```

Дневник отладки

Попытка 1-5

Неправильный формат вывода ответа, в случае если не существует вхождения. В случае отстувия вхождения, ничего не выводить

Попытка 5-42 (ID 90863654)

Данные попытки связаны с возникновением странной ошибки IL. В результате было выяснено, что это связано с проблемами в памяти. Был найден массив с неправльной длинной (массив cn). Проблема была решена

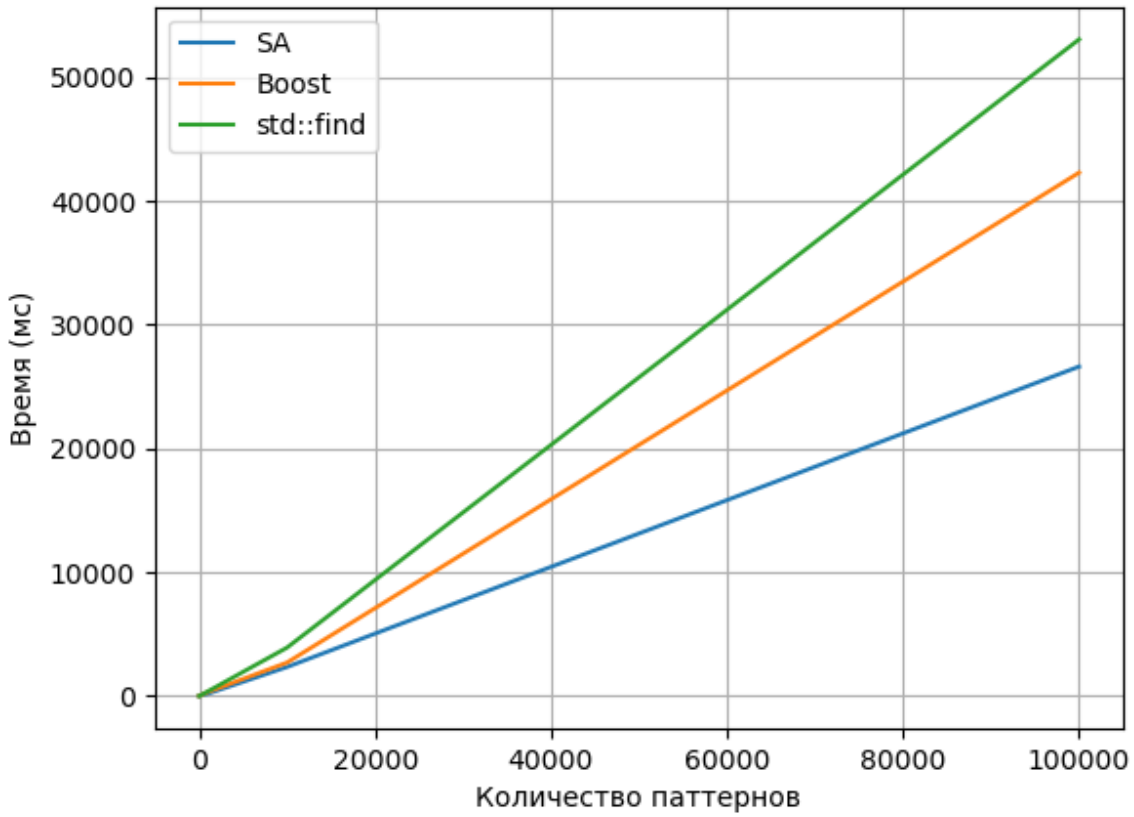
Попытка 43-58 (ID 90863654)

Возникала ошибка TL, были проведены различные оптимизации. Например передача текста по ссылке

0.1 Результаты тестирования

Тест производительности

Тестирование программы производилось на 5 сгенерированных тестах Для сравнения было выбрана функция из библиотеки boost $boost :: find_all$ которая находит все вхождения подстроки в строку:
Также использовался стандартный find из библиотеки std
Вставка:



Получившиеся результаты удивляют особо не удивляют, в ранних лабах можно было наблюдать, что поиск в библиотеках работает не самым лучшим образом

Выводы

При выполнении данной лабораторной работы, я познакомился с такой структурой данных как суффиксный массив. Он хранит отсортированные индексы всех суффиксов подстроки. Данный алгоритм позволяет использовать меньше памяти по сравнению с суффиксным массивом. Сложностное построние за O(nlogn) без использования суффиксного дерева

Данная структура данных используется в bigdata из-за своего быстрого поиска за O(m + logn) и малого хранения памяти.

Несмотря на то, что я реализовал поиск за O(mlogn) он показал себя не плохо по сравнению с функциями из стандартной библиотеки