Лабораторная работа № 5 по курсу дискретного анализа: Суффиксные деревья

Выполнил студент группы 08-308 Иванов Андрей.

Условие

Необходимо реализовать структуру данных суффиксное дерево и решить задачу по-ставленную во варианте.

Вариант 5: Поиск наибольшей общей подстроки.

Найти самую длинную общую подстроку двух строк с использованием суфф. дерева. В качестве вывода - на первой строке нужно распечатать длину максимальной общей подстроки, затем перечислить все возможные варианты общих подстрок этой длины в порядке лексикографического возрастания без повторов.

Метод решения

Сначала мы считываем две строки. Эти строки объединяются с помощью специальных символов-разделителей @ (разделить 1 и 2 строку между собой) и $ (вспомогатель-ный сентенел для постройки дерева), чтобы можно было однозначно различать части паттерна и текста в процессе построения суффиксного дерева.

Далее мы начинаем построение суффиксного дерева для объединённой строки, ис-пользуя алгоритм Укконена, работающий за O(n), где n - длина строки. В основе лежит процесс построения дерева на каждой итерации путем добавления нового символа из строки, начиная с первого символа и заканчивая последним, используя суффиксные ссылки. Во время добавления символов алгоритм обрабатывает "активную ноду"— те-кущее состояние дерева, и, если необходимо, обновляет суффиксные ссылки между узлами для оптимизации переходов в дерево.

После построения суффиксного дерева, используя метод обхода узлов, мы опреде-ляем тип каждого узла. Узлы классифицируются на три типа: узлы, представляющие первую строку, вторую и узлы, которые содержат подстроки, встречающиеся и в пер-вом, и во втором.

Затем мы ищем самую длинную общую подстроку между двумя строками, прохо-дя по узлам дерева. Мы обновляем максимальную длину совпадения и запоминаем индексы концов всех подстрок максимальной длины. Это делается с использованием информации о глубине текущего узла и его дочерних узлов.

После завершения обхода выводится длина самой длинной общей подстроки, а также сами подстроки, найденные в исходной строке.

Этот алгоритм использует суффиксное дерево, что позволяет эффективно находить общие подстроки и определять их максимальную длину за линейное время относительно длины входных строк. График, почему это работает за линию - будет ниже.

Описание программы

Пограмма состоит из основного файла main.cpp

1. main.cpp (В нем находится реализация суффиксного дерева и его нод, так же в методах класса суффиксного дерева находится вспомогательная программа для определения, для решения основной задачи.

Дневник отладки

Изначально была написана реализация, не проходившая по памяти, потратив время на анализ, был сделан вывод о лишнем расходе памяти и пересмотру реализации алгоритма Укконена. После детального изучения материалов по данной теме, я смог написать более эффективный алгоритм.

Тест производительности

Так как сложность алгоритма зависит от длины строки, нужно протестировать работу нашего алгоритма. Будем создавать строки разной длины так, чтобы линейно увеличи-вать их длины.

DATA LENGTH length	Ukkonen algorithm ms
1.000.000	1943.28
2.000.000	4013.71
3.000.000	6729.92

Табл.1

Выполнив замеры скорости работы программы в зависимости от размера длины строки, сделал вывод, что сложность алгоритма действительно линейна и зависит от длины слова. Увеличивая входные данные на каждом следующем тесте из таблицы вид-но, что время выполнения линейно возрастает (примерно) относительно длины строки.

Исходный код бэнчмарка:

```
#include
<chrono> #include
"SuffTree.h"

std::string RandomString(size_t
size) {
    srand(time(nullp
    tr)); std::string
    str;
```

```
    std::string tmp;
    for (size_t i = 0; i <
        size; ++i) { if
        (rand() % 15 == 0)
        {
           t
        mp
        = ’
        ’;
        }
        els
        e
        {
            tmp = (rand()
        % 26) + ’a’; }

        str.a
    ppend(t
    mp); }

    r
etu
rn
str;
}

int main() {
    std::ios::sync_with_s
    tdio(false);
    std::cin.tie(0);
    auto start =
    std::chrono::high_resolution_clock::now();
    int halfSize = 1500000;
    std::string s = RandomString(halfSize) + SENTINEL_A +
    RandomString(halfSize) + SENTI TSuffTree st(s);
    st.TypeNodeAndFindRes(0, 0, st.InputString);
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> duration =
    end - start; std::cout << "Размер строки - " << 2 *
    halfSize << ’\n’;
    std::cout << "Время работы программы: " <<
duration.count() << " ms"; }
```

Недочёты

Сложно сказать о недочетах в реализации самого алгоритма, ибо лучше и эффективнее такой реализации алгоритма Укконена еще не придумали. Возможно, можно было как-то эффективнее расходовать память, используя битовые операции, но мое мнение, что это уже слишком трудно в реализации.

Выводы

Данная лабораторная работа показала, как можно удобно и быстро искать совпадения в строках. Реализация алгоритма Укконена очень полезное упражнение, ведь такой алгоритм является основой для любого начинающего программиста. Умение работать эффективно со строковыми данными - очень полезный навык для работы. В отличие от прошлой лабораторной работы, здесь мы уже обрабатывали именно текст, а не паттерн.

В этом и есть особенность суффиксного дерева.

Сложность моего алгоритма, как мы удостоверились из тестов - O(n), где n - длина строки, в которой мы ищем совпадения
Написание алгоритма задача намного сложнее, нежели наивный алгоритм простроения суффиксного дерева за кубическую сложность, эффективная работа с алгоритмом -залог успеха любого программиста.