Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт № 8 информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №5 по курсу**

**«Дискретный анализ»**

**алгоритм Укконена**

Студент: Пермяков Никита Александрович

Группа: М80 – 208Б-19

Вариант: 2

Преподаватель: [*Кухтичев Антон Алексеевич*](https://mai.ru/education/schedule/ppc.php?guid=a3f854e1-f771-11e7-ae95-485b3919ee6d)

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2020

**Содержание**

1. Постановка задачи
2. Метод и алгоритм решения
3. Описание программы
4. Дневник отладки
5. Тестирование производительности
6. Вывод

**Постановка задачи**

Необходимо реализовать алгоритм Укконена построения суффиксного дерева за линейное время.

**Алфавит строк**: строчные буквы латинского алфавита (т.е. от a до z).

**Вариант 2**: Поиск с использованием суффиксного массива. Найти в заранее известном тексте поступающие на вход образцы с использованием суффиксного массива.

**Входные данные**: текст располагается на первой строке, затем, до конца файла, следуют строки с образцами.

**Выходные данные**: для каждого образца, найденного в тексте, нужно распечатать строку, начинающуюся с последовательного номера этого образца и двоеточия, за которым, через запятую, нужно перечислить номера позиций, где встречается образец в порядке возрастания.

**Метод и алгоритм решения**

Каждый узел содержит итераторы, указывающие на начало и конец этой подстроки в тексте и суффиксную ссылку, указывающею на вершину с таким же суффиксом, только без первого символ. При отсутствии такой вершины – на корень. Также есть словарь с ребрами, выходящими из данной вершины. В дереве храним текст с терминальным символом в конце, по которому ищем, указатель на корень, переменную remainder, которая показывает, сколько суффиксов еще надо вставить. Указатель fake\_node указывает на вершину, из которой необходимо создать суффиксную ссылку, если в данной фазе уже была вставлена вершина по правилу *```Если ребро разделяется ~~и~~ вставляется новая вершина, и если это не первая вершина, созданная на текущем шаге, ранее вставленная вершина и новая вершина соединяются через специальный указатель, суффиксную ссылку```*, и сейчас оно используется вновь. Указатель current\_node указывает на вершину, которое имеет ребро current\_point, в котором мы сейчас находимся. current\_lenght показывает на каком расстоянии от этой вершины мы находимся. Итеративно проходим по тексту для создания дерева. На каждой итерации начинается новая фаза и remainder увеличивается на 1. Далее пока все не вставленные суффиксы не вставлены в дерево выполняем цикл. Если в той вершине, в которой мы остановились еще нет ребра, начинающегося с первой буквы обрабатываемого суффикса, то по правилу продолжений *```current\_node остается корнем. текущее ребро становится первым символом нового суффикса, который нужно вставить, т.е. b current\_lenght уменьшается на 1```* создаем новую вершину, которая будет листом.

Если это необходимо, создаем суффиксную ссылку.

Если в той вершине, в которой мы остановились, уже есть такое ребро, то нужно пройти вниз по ребрам на current\_lenght и обновить current\_node.

Если некоторый путь на этом ребре начинается со вставляемого символа, значит по правилу ```*В любой фазе, если правило продления применяется в продолжении суффикса, начинающего в позиции j, оно же и будет применяться во всех дальнейших продолжениях (от j+1 по i) до конца фазы.*``` нам ничего делать не надо, заканчиваем фазу, оставшиеся суффиксы будут добавлены неявно. Увеличиваем current\_lenght на 1 (т.к. учитываем, что этот символ уже есть на данном пути), по необходимости строим суффисную ссылку. Если никакой путь не начинается со вставляемого символа, то нужно разделить ребро в этом месте, вставив 2 новых вершины – одну листовую и одну разделяющую ребро. Далее по необходимости добавляем суффиксную ссылку. Уменьшаем remainder на 1, если вставили суффикс в цикле. Если после всех этих действий current\_node указывает на корень и current\_lenght больше 0, то уменьшаем current\_lenght на 1, а current\_point устанавливаем на первый символ нового суффикса, который нужно вставить. Если current\_node не корень, то переходим по суффиксной ссылке.

После конструирования дерева, строим суффиксный массив. В нем расположен вектор, в котором находятся начальные позиции суффиксов, и все эти суффиксы лексикографически упорядочены. Массив строим из дерева, выполняя обход в глубину.

Т.к. словарь, который находится в каждой вершине, это упорядоченный контейнер, то номера позиции после обхода в глубину будут также лексикографически упорядочены.

Поиск вхождений в массиве осуществляется с помощью бинарного поиска. В зависимости от того, лексикографически меньше или больше буква в паттерне и буква в тексте, границы поиска в массиве сужаются наполовину. В конце возвращается диапазон начальных позиций, в которых найдены вхождения.

**Описание программы**

main.cpp – файл с реализацией

**Дневник отладки**

1 – 2) Синтаксические ошибки компиляции

3 – 5) Неверное сложение, ошибки в перегрузке оператора, взятие копии числа, ошибки в логике вычислений

6-9) Переполнение представления числа

10 – 14) Ошибки в логике вычислений

15) Успешная попытка

16 - 17) Рефакторинг и оптимизация

**Тестирование производительности**

Тесты проводились 5 раз для каждой конфигурации, рассчитывалось среднее значение времени исполнения

(2, 3 колонка для кастомной и библиотечной версии соответственно).

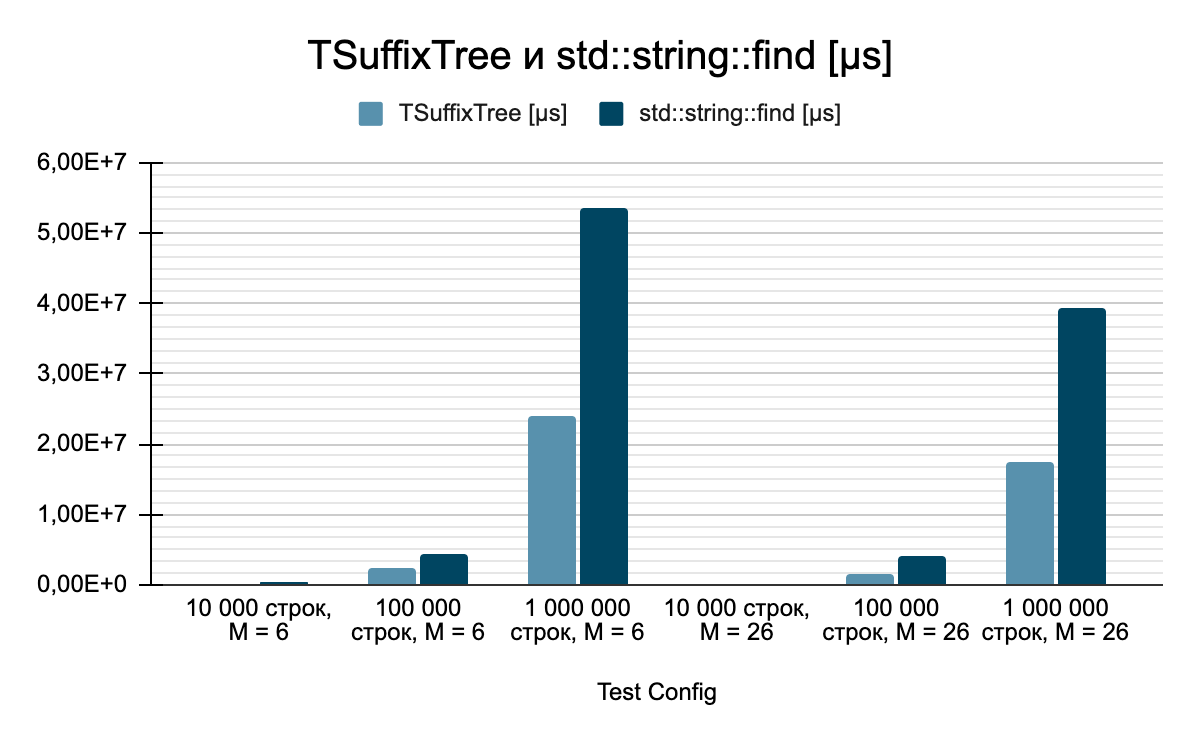
М - мощность алфавита

1000  - длина случайного текста

[1,50) -  диапазон возможной длины случайного паттерна

 Таблица 1. Результаты тестов производительности

|  |  |  |
| --- | --- | --- |
| **Test Config** | **TSuffixTree [µs]** | **std::string::find [µs]** |
| **10 000 строк, М = 6** | **260746** | **508278** |
| **100 000 строк, М = 6** | **2389398** | **4404435** |
| **1 000 000 строк, М = 6** | **23955876** | **53444390** |
| **10 000 строк, М = 26** | **139460** | **204579** |
| **100 000 строк, М = 26** | **1650844** | **4175808** |
| **1 000 000 строк, М = 26** | **17571014** | **39477540** |

****

**Вывод**

В результате работы был реализован алгоритм Укконена и поиск с использованием суффиксного массива. Было проведено сравнение суффиксного дерева за линейное время с методом find типа данных string из Стандартной библиотеки С++.

Результаты тестов производительности доказали эффективность алгоритма Укконена по сравнению с методом find - состоящий из алгоритмом поиска Кнута - Морриса - Пратта.

Можно отметить следующие недостатки алгоритма Укконена:

* Размер суффиксного дерева сильно превосходит входные данные, поэтому при очень больших входных данных алгоритм Укконена сталкивается с проблемой memory bottleneck problem (когда процессор в ограниченный промежуток времени требует больше памяти, чем доступно на это время).
* Для ответа на запрос о существовании перехода по текущему символу за O(1) необходимо хранить линейное количество информации от размера алфавита в каждой вершине. Поэтому, с увеличением мощности алфавита растет объем необходимой памяти. Можно сэкономить на памяти, храня в каждой вершине только те символы, по которым из неё есть переходы.
* Существуют кэш-эффективные алгоритмы, превосходящие алгоритм Укконена на современных процессорах. Cache-oblivious алгоритмы используют метод “Разделяй-и-властвуй”, в котором задача разбивается на маленькие задачи и подзадачи. В процессе разделения, у нас получаются небольшие подзадачи. В какой-то момент, эти подзадачи начинают помещаться в размер кэша
* Также алгоритм предполагает, что дерево полностью должно быть загружено в оперативную память.