Лабораторная работа № 5 по курсу дискретного анализа: Суффиксные деревья

Выполнил студент группы М8О-308Б-20 МАИ *Зинин Владислав.*

Условие

1. Необходимо реализовать алгоритм Укконена построения суффиксного дерева за линейное время. Построив такое дерево для некоторых из входных строк, необходимо воспользоваться полученным суффиксным деревом для решения своего варианта задания.

2. Вариант задания: поиск в известном тексте неизвестных заранее образцов

Метод решения

В ходе решения я реализовал 2 класса: класс *Node*, который хранит номер листа, необходимый для вывода позиции, скоторой начинается наш искомый суффикс, левую и правую границу согласно алгоритму Укконена, суффиксную ссылку и ребра, которые являются такими же нодами, а также класс *SuffTree*, который хранит коренную ноду, текущую ноду, на которой мы находимся в ходе построения дерева, позицию в тексте и вспомогательную переменную pos, последнюю добавленную вершину, текст, счетчик суффиксов и листов. Также реализованы следующие функции: *void Add(int inpos)* для добавления символа *void CreateList(int inpos, Node *node)* для создания листа *void CreateSufflink(Node *Node)* для создания суффиксной ссылки *void BreakCreationNode(int inpos)* для случая, когда мы должны создать внутренний узел путем разбиения существуещего ребра на две ноды, также мы соединяем с вершиной новую ноду. *GoToSuffLink()* для перемещения по суффиксной ссылке *bool EdgeFault()* для случая, когда pos длинее "длины"нашего ребра, в таком случае переходим в следующую подходящую нам ноду и заного все перепроверяем

Описание программы

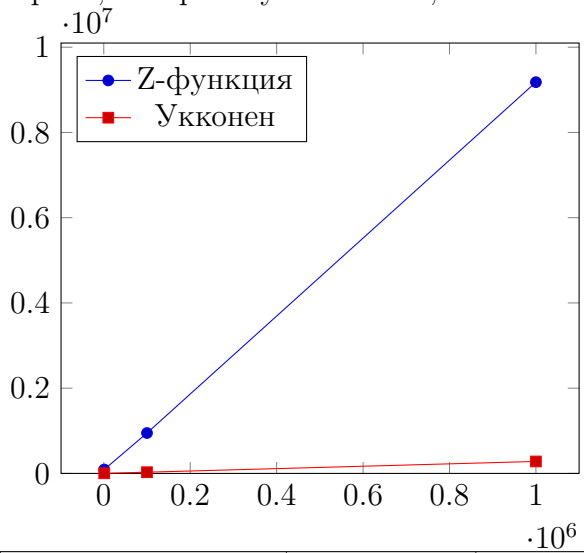
Программа состоит из одного файла *lab5.cpp.*

Дневник отладки

Изначально были проблемы с суффиксными ссылками, а конкретно с реализацией их назначения, в связи с чем было принято решение хранить ссылку на последнюю созданную внутреннюю вершину, чтобы в последующем шаге при добавлении нового суффикса построить суффиксную ссылку. Следующая проблема была с неучотом случая, когда pos длинее "длины"нашего ребра, для этого я создал функцию *bool EdgeFault()*, описанную выше.

Тест производительности

Ниже приведено сравнение времени работы Z-функции и алгоритма Укконена. Поиск паттерна производится в строке, состоящей из 1000 символов. По оси X — количество паттернов, которые нужно найти, по оси Y — время поиска паттерна в милисекундах.



Кол-во паттернов	Z-функция	Укконен
1000	91134	2910
100000	947637	28521
1000000	9179573	280624

Из тестов ясно видно, что Z-функция работает за $O(n^2)$, а алгоритм Укконена — за линейное время.

Недочёты

Вся реализация находится в одном файле, из-за чего падает читабельность кода. Но сделано это из-за того, что как показывает практика, из-за makefile программа может не пройти тестировщик из-за времени либо вовсе не скомпилироваться.

Выводы

В результате проведения лабораторной работы, я познакомился с практическим применением суффиксного дерева, а также с алгоритмом Укконена, благодаря которому суффиксное дерево работает за линейное время. Но тем не менее алгоритм очень сложен в реализации, и поэтому очень трудно искать ошибки.