Лабораторная работа № 4 по курсу дискретного анализа: Сбаласированные деревья

Выполнил студент группы М8О-208Б-20 МАИ *Каширин Кирилл.*

Условие

Необходимо реализовать один из стандартных алгоритмов поиска образцов для указанного алфавита.

Вариант алгоритма: Поиск большого количества образцов при помощи алгоритма Ахо-Корасик.

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

Запрещается реализовывать алгоритмы на алфавитах меньшей размерности, чем указано в задании.

Метод решения

Написание лабораторной состояло из трех частей.

1. Первая часть - это создание структуры Trie, которая хранит в себе не ключ, а метки, а ключом является путь от корня до этой метки.

2. Вторая часть - проход по префиксному дереву и проставление связей неудач и связей выхода. Связи неудач проставлялись следующим образом: от узла, в котором находимся поднимаемся уровнем выше и переходим по связи неудач этого узла, если после перехода можно перейти в узел с таким же значением, в котором мы были первоначально - мы переходим и у исходного узла ставим ссылку на этот узел. Узлы которые расположены у корня ссылаются на корень. Связь выхода строится в том случае, если один или несколько связей неудач указывает на конец паттерна.

3. Третья часть - это построения автомата для реализации алгоритма Ахо-Корасика. Мы проходимся по тексту и идем по Trie, сравнивая элементы. Если мы находимся в корне Trie и элементы не равны, мы сдвигаемся по тексту, если равны мы двигаемся по Trie. Если у нас произошли расхождения элементов в узле, который не является корнем, то мы переходим по связям неудач, сравнивая элементы, после каждого перехода. При успешном продвижении, мы должны также обращать внимание на связи выхода. Если они существуют или же мы оказались в узле, который является концом какого-либо паттерна, мы выводим ответ (номер строки, номер позиции в тексте и номер паттерна).

Описание программы

Программа состоит из одного файла, но разделена на три части: описание структуры TrieNode и структуры Text, описание класса Trie, в главной функции описан интерфейс взаимодействия пользователя с алгоритмом Ахо-Корасика согласно условию задания.

Методы класса Trie:

- *void Add(vector<string> s, int numberOfPattern)* - добавление элемента паттерна в Trie.
- *void BuildConnection(TrieNode* root)* - построение связей неудач и выхода.
- *void Search(TrieNode* cur, vector<Text*> text, int pos, vector<int> sizeString)* - поиск образца в тексте по алгоритму Ахо-Корасика.

Дневник отладки

1. Когда первый раз была загружена программа на чекер, она получила Wrong answer на 8 тесте. Ошибка заключалось в неправильном подсчёте элементов паттерна, поскольку я считал количество пробелов, а лишних пробелов было намного больше. Исправил я это тем, что начал считать количество считываний элементов.
2. Далее я получил Time Limit на 12 тесте. Из-за того, что в конструктре я забыл инициализировать булевскую переменную, которая отвечала за то, является ли узел Trie концом паттерна или нет.
3. После этого, программа дошла до 13 теста, где получила Memory Limit из-за того, что аргументы функции я не брал по адресу

Тест производительности

Для анализа производительности была написала программа поиска образца в текстке наивным способом. Для сравнения производительности я подготовил 3 файла в который 100000, 1000000, 10000000 строк.
Получились следующие результаты (Ахо-Корасик || Наивный) :
100000 входных данных - 331 ms || 285 ms
1000000 входных данных - 944 ms || 1804 ms
10000000 входных данных - 1942 ms || 9878 ms
Таким образом, можно заметить, что при $10\hat{5}$ входных данных Ахо-Корасик уступает наивному алгоритму, но при больших данных алгоритм становится быстрее в несколько раз.

Выводы

В результате этой лабораторной работы я не только изучил новый алгоритм поиска подстроки в строке Ахо-Корасик, но и познакомился и реализовал новую для меня структуру - префиксное дерево. Алгоритм Ахо-Корасика обеспечивает линейный рост сложности от длины текста во время поиска одного образца, благодаря данному методу можно например искать определенный текст в электронной библиотеке.Также я сравнил свой алгоритм, с наивным, который имеет сложность $O(n\hat{2})$, и убедился, что наивный алгоритм медленнее.