Лабораторная работа №4 по курсу дискретного анализа: поиск образца в строке.

Выполнил студент группы М80-208Б-20 Морозов Артем Борисович.

Условие

Вариант алгоритма:

1. Поиск одного образца-маски: в образце может встречаться «джокер», равный любому другому символу. При реализации следует разбить образец на несколько, не содержащих «джокеров», найти все вхождения при помощи алгоритма Ахо-Корасик и проверить их относительное месторасположение.

Вариант алфавита:

2. Числа в диапазоне от 0 до 2^32 - 1.

Метод решения

Алгоритм Ахо-Корасик хорош тем, что позволяет искать несколько паттернов в тексте за время O(n + m + k), где n – длина текста, m – суммарная длина всех паттернов, k – количество вхождений паттернов в текст. В этом его принципиальное отличие от пройденных ранее в курсе алгоритмов Кнута-Морриса-Пратта, Бойера-Мура и Апостолико-Джанкарло. Да, они тоже могут справиться с задачей поиска нескольких паттернов по тексту, однако сложность у них будет значительно выше, нежели у алгоритма Ахо-Корасик. У них будет сложность O(nh + m), где n – длина текста, h – количество всех паттернов, m – длина всех паттернов, что уступает нашему алгоритму.

Метод же решения данной задачи мало чем отличается от алгоритма Ахо-Корасик в привычном его представлении: строку из паттернов мы разбиваем на несколько подстрок, не содержащих джокеры (например, для строки qwe?rt?y у нас будет 3 подстроки, то есть 3 паттерна – qwe, rt, y). При этом при разбивании изначально паттерна на подпаттерны мы запоминаем индекс вхождения каждого паттерна (например, у паттерна qwe индекс вхождения 0, rt - индекс вхождения 4, y – 7). Далее, как и в привычном алгоритме Ахо-Корасик, мы строим структуру данных Trie из наших паттернов. В нашем Trie будет так же присутствовать две сущности – связь неудач и связь выхода, которые заметно ускоряют сам алгоритм Ахо-Корасик. Связи неудач нам нужны для того, чтобы не проходить весь Trie заново при каждом несовпадении, а в случае несовпадения текущего зачения паттерна в тексте переходить по специальной ссылке на ту ноду, которая вместе со своими предками образует наибольший префикс нашего уже проверенного суффикса (например, в тексте xxxpotattooxx, при несовпадении после t в паттерне potato мы сразу переходим к префиксу tat слова tattoo). Связи же выхода нужны нам для того, чтобы не забывать, что какие-то паттерны могут быть подпаттернами других, поэтому их вхождения также необходимо учитывать. Если пользоваться лишь одними связями неудач, то информация о таких паттернах теряется. После, мы заводим специальный массив счетчиков, равный длине текста, и считаем вхождения нашего паттерна в наш текст только тогда, когда в какой-то ячейке нашего массива счетчиков значение будет равно количеству паттернов в нашей структуре данных Trie. Таким образом, задача сводится к построению дерева из наших паттернов, созданию связей неудач и связей выхода и единственным проходом по нашему тексту.

Описание программы

Программу условно можно разбить на 2 подраздела: функции для обработки поступающего паттерна и поступающего текста, и функции, отвечающие за работу с

Trie. Нода нашего Trie у меня представлена структурой, в которой хранится 7 полей: значение текущей вершины, список потомков текущей вершины, указатель на родителя, вектор позиций, нужный нам для запоминания начала нашего паттерна в строке с джокерами, булевская переменная, отвечающая за конец нашего паттерна, вязи выхода и неудач. В самом Trie хранится указатель на корень и количество паттернов в Trie. Функции для Trie находятся в специальном пространстве имен TrieAhoKorasik, чтобы отделить их от функций другого предназначения.

Дневник отладки

В основном проблемы касались обработки вводимого текста и паттерна, однако на одном тесте так же было выявлено, что программа не обрабатывала подходящий нам случай нахождения совпадения в функции Search, но вскоре недочеты были устранены.

Тест производительности

В своем бенчмарк-тесте я решил сравнить алгоритм Ахо-Корасик с наивным алгоритмом. Я провожу 5 тестов:

1 тест – длина паттерна с джокерами равна трем, суммарное количество строк текста равно 100:

Результат работы алгоритма Ахо-Корасик:

```
DA > lab4 > benchmark > ≡ naive.txt
  1    432 microseconds
```

Результат работы наивного алгоритма:

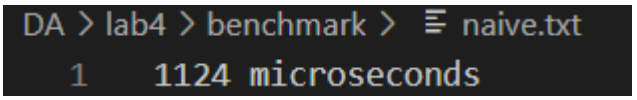
```
DA > lab4 > benchmark > ≡ naive.txt
  1    674 microseconds
```

2 тест – длина паттерна с джокерами равна пяти, суммарное количество строк текста равно 1000:

Результат работы алгоритма Ахо-Корасик:

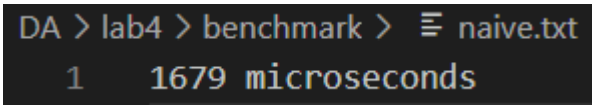
```
DA > lab4 > benchmark > ≡ ahokorasik.txt
  1    910 microseconds
```

Результат работы наивного алгоритма:

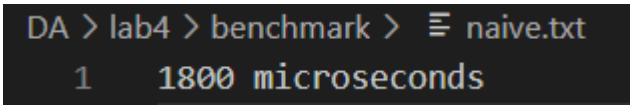


3 тест – длина паттерна с джокерами равна десяти, суммарное количество строк текста равно 5000:

Результат работы алгоритма Ахо-Корасик:



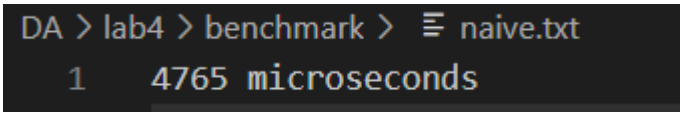
Результат работы наивного алгоритма:



4 тест – длина паттерна с джокерами равна двадцати, суммарное количество строк текста равно 10000:

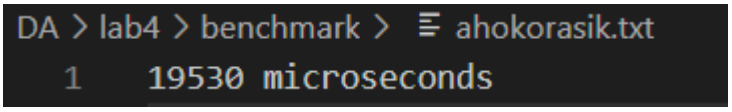
Результат работы алгоритма Ахо-Корасик:

Результат работы наивного алгоритма:

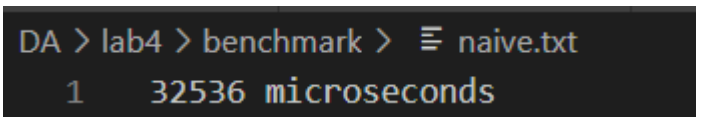


5 тест – длина паттерна с джокерами равна пятидесяти, суммарное количество строк текста равно 50000:

Результат работы алгоритма Ахо-Корасик:



Результат работы наивного алгоритма:



Как мы видим, на первых трех тестах наивный алгоритм мало чем отличался от нашего алгоритма, однако на двух последних больших тестах стало заметно преимущество Ахо-Корасик. Я думаю, что связано это с тем, что наивный алгоритм работает за сложность O(n*m), где n – длина текста, m – длина паттерна. При относительно больших данных это очень невыгодно.

Недочёты

Недочетов в программе обнаружено не было, однако стоит упомянуть, что программа работает только при условии корректного ввода, так как была разработана исключительно в учебных целях. Любой неправильный ввод может убить работоспособность моей программы.

Выводы

Данная лабораторная работа помогла мне лучше осознать такую структуру данных, как Trie, его устройство и функционал. С его помощью я реализовал алгоритм Ахо-Корасик для решения задачи поиска образца с джокерами. Что же касается алгоритмов на строках, то у них очень большая область применения, так как любая программа, хоть как-нибудь занимающаяся поиском какого-то паттерна в строке, использует внутри себя алгоритмы на строках.