Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №4 по курсу «Дискретный анализ»

Студент: А. А. Терво
Преподаватель: А. А. Кухтичев
Группа: М8О-207Б

Дата:
Оценка:
Подпись:

Москва, 2021

Лабораторная работа №4

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

Вариант алгоритма: Поиск одного образца-маски: в образце может встречаться «джокер» (представляется символом ? — знак вопроса), 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: Числа в диапазоне от 0 до $2^{32} - 1$

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

Формат входных данных: Искомый образец задаётся на первой строке входного файла.

В случае, если в задании требуется найти несколько образцов, они задаются по одному на строку вплоть до пустой строки.

Затем следует текст, состоящий из слов или чисел, в котором нужно найти заданные образцы.

Никаких ограничений на длину строк, равно как и на количество слов или чисел в них, не накладывается.

Формат результата: В выходной файл нужно вывести информацию о всех вхождениях искомых образцов в обрабатываемый текст: по одному вхождению на строку.

Для заданий, в которых требуется найти только один образец, следует вывести два числа через запятую: номер строки и номер слова в строке, с которого начинается найденный образец. В заданиях с большим количеством образцов, на каждое вхождение нужно вывести три числа через запятую: номер строки; номер слова в строке, с которого начинается найденный образец; порядковый номер образца.

Нумерация начинается с единицы. Номер строки в тексте должен отсчитываться от его реального начала (то есть, без учёта строк, занятых образцами).

Порядок следования вхождений образцов несущественен.

1 Описание

Требуется реализовать алгоритм Ахо-Корасик.

Как сказано в [1], задача алгоритма заключается в следующем: «Дан набор строк в алфавите размера k суммарной длины m. Необходимо найти для каждой строки все ее вхождения в текст».

Суть алгоритма заключается в следующем:

1. Построение бора

Строится бор (нагруженное дерево) из строк образца. Построение выполняется за $O(m)$, где m — суммарная длина строк.

2. Преобразование бора

Алгоритм удобно реализовать в виде конечного автомата. Узлы бора можно понимать, как состояния автомата, а корень как начальное состояние.

Узлы бора, в которых заканчиваются строки, становятся терминальными.

Для переходов по автомату заведём в узлах несколько функций:

$parent(u)$ — возвращает родителя вершины u;

$\pi(u) = \delta(\pi(parent(u)), c)$ — суффиксная ссылка, и существует переход из $parent(u)$ в u по символу c;

$$\delta(u, c) = \begin{cases} v, & if\ v\ is\ son\ by\ symbol\ c\ in\ trie; \\ root, & if\ u\ is\ root\ and\ u\ has\ no\ child\ by\ symbol\ c\ in\ trie; \\ \delta(\pi(u), c), & else. \end{cases}$$

Можно понимать рёбра бора как переходы в автомате по соответствующей букве. Однако если попытаться перейти по букве, для которой нет соответствующего ребра, нас может постигнуть неудача, но такого быть не должно. Для этого существуют суффиксные ссылки.

Суффиксная ссылка $\pi(u) = v$, если $[v]$ — максимальный суффикс $[u]$, $[v] \neq [u]$.
В нашем случае алгоритм претерпит некоторые изменения. Это связано с тем, что в образце встречаются джокеры. В дерево нужно будет добавить не одну строку образца, а все его подстроки, не содержащие джокеров.

Непосредственно при поиске при проходе по тексту нужно для каждой подстроки без джокеров находить начальные позиции их вхождения в текст. Для каждого такого начала j подстроки P_i в T увеличиваем счётчик в ячейке $j - l_i + 1$ вектора C на единицу. После окончания прохода текста просматриваем вектор C в поисках ячеек со значением, равным k – числу подстрок образца без джокеров. Вхождение P в текст, начинающееся с позиции p, имеется только в том случае, если $C(p) = k$.

Основными этапами работы программы являются заполнение вектора образца, составление по нему бора, создание ссылок, заполнение вектора текста и совершение поиска за один проход по тексту.

2 Исходный код

Объявление классов узла бора и самого бора.
`Trie::Create(patterns)` — создаёт дерево из вектора patterns.
`Trie::Search(text, patLen, answer)` — ищет вхождения в текст образцов из дерева, выводит ответ в консоль, patLen — размер вектора образцов.

```
  class TTrie;

  class TTrieNode {
  public:
    friend class TTrie;
    TTrieNode();
    ~TTrieNode() {};
  private:
    std::map<unsigned long, TTrieNode *> to;
    TTrieNode* suffLink;
    std::vector<int> out;
  };

  class TTrie {
  public:
    TTrie();
    void Create(const std::vector<std::string> &);
    void Search(const std::vector<unsigned long> &, const int &, std::vector<std::pair<
        int, int>> &);
    ~TTrie() {};
  private:
    TTrieNode *root;
    std::vector<int> lensPatterns;
    int woJoker;
    void CreateLinks();
  };

}
```

3 Консоль

```
[alext@alext-pc solution]$ ./solution
213 ? 8 5 ?
49869 213 5 5 767 564 6969 5 8 8 5 981 8 69 213 8 8 86 5 938
314 8 65 11781 892 22114 9248 4823 44913 63632 8 5 65 5 74 213 13373 8 5 19
8 213 99642 8 5 5 3748 7734 56653 8 8494 5 58 4835 1 56771 213 8 213 62921
9 8152 6 213 213 23791 213 5 213 91167 5 61 8 5 8 5 213 213 64 5
5 8 8 8726 29669 1163 8 213 64987 93 213 8 9996 1 5 8 7156 8 213 7
56711 8 167 135 96 4289 5 58142 286 8 51827 14 213 3388 8341 8 213 24 5 213
8 14 3222 4833 9 213 8 213 5 12 7362 73 963 7 5 213 634 213 5 17678
8 9136 816 27 5 92947 213 89833 278 54646 5 8 87815 8 11729 213 8 93667 8 7186
5 8 35 5 213 213 51278 5 213 45 213 8 8 6 449 8 92 5 5 8
642 213 3345 83 8 3 6682 5 31332 213 213 8 5 7195 8 7159 145 8 213 8

2,16
3,2
10,10

[alext@alext-pc solution]$ ./solution
1 2 3 4 5 ? ? ? ?
1 2 3 4 5 6 9 12
1 2 3 4 5 7 10 13
1 2 3 4 5 8 11 14

1,1
2,1
```

4 Тест производительности

Тест производительности представляет из себя следующее: поиск образцов с помощью алгоритма Ахо-Корасик и с помощью наивного алгоритма. Образец достаточно мал (5 чисел) по сравнению с текстом (2500, 5000 и 10000 строк). Последний тест представляет собой длинный образец с джокерами, текст объёмом 1000 строк, вхождение образца на каджом числе.

```
[alext@alext-pc tests]$ ./solution <2,5k-normal.t
Aho-Corasick algorithm: 35050us
Naive algorithm: 12067us
[alext@alext-pc tests]$ ./solution <5k-normal.t
Aho-Corasick algorithm: 70024us
Naive algorithm: 24251us
[alext@alext-pc tests]$ ./solution <10k-normal.t
Aho-Corasick algorithm: 139041us
Naive algorithm: 47547us
[alext@alext-pc tests]$ ./solution <1k-same.t
Aho-Corasick algorithm: 219538us
Naive algorithm: 2337851us
```

Видно, что оба алгоритма соответствуют заявленной сложности, при размере образца много меньшем размера текста наивный алгоритм работает почти за линейное время, что и видно из тестов. За счёт простоты реализации и при таком размере образца он выигрывает у алгоритма Ахо-Корасик в абсолютном времени. Если сделать образец больше, а количество вхождений, пусть даже неполных, чаще, наивный алгоритм будет уже не так эффективен. Это видно в последнем тесте, где для наивного алгоритма представлен один из худших случаев работы. В нём алгоритму Ахо-Корасик удалось выиграть даже в абсолютном времени, за счёт того, что его сложность составляет $O(m + n + a)$, в отличие от наивного, чья сложность равна $O(m * (n - m))$, где m — длина образца, n — длина текста, a — количество появлений подстрок шаблона.

5 Выводы

Применение алгоритмов поиска подстрок в строке чрезвычайно широко распространено. Они используются в любых приложениях, выполняющих обработку текстовых данных.

В лабораторной мне было предложено решить задачу поиска подстроки в строке при помощи модификации алгоритма Ахо-Корасик для образца с джокерами.

Алгоритм Ахо-Корасик представляется в среднем достаточно эффективным, но у него есть как свои достоинства, так и недостатки.

К последним прежде всего относится его сложность к восприятию и усвоению принципа его работы. А также то, что он требует O(n) памяти, в отличие от наивного алгоритма, который занимает O(1) памяти.

К достоинствам алгоритма можно отнести стабильную невысокую вычислительную сложность.

Алгоритм Ахо-Корасик выгоден, когда есть список длинных, сложных и постоянных паттернов, и их необходимо искать во множестве разных текстов. В таком случае достаточно один раз построить бор и далее искать паттерны в тексте за один проход.

Когда же паттерны часто изменяются и/или много короче по сравнению с текстом, есть смысл использовать наивный алгоритм ввиду его простоты и приемлемой эффективности.

Список литературы

[1] *Алгоритм Ахо-Корасик — Викиконспекты.*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Ахо-Корасик` (дата обращения: 7.12.2021).

[2] *Алгоритм Ахо-Корасик / Хабр.*
URL: `https://habr.com/ru/post/198682/` (дата обращения: 7.12.2021).

[3] *Aho–Corasick algorithm - Wikipedia.*
URL: `https://en.wikipedia.org/wiki/Aho-Corasick_algorithm` (дата обращения: 7.12.2021).