Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №5 по курсу «Дискретный анализ»

Студент: А.Д. Волков
Преподаватель: А. А. Кухтичев
Группа: М8О-306Б

Дата: 01.10.2024
Оценка:
Подпись:

Москва, 2024

Лабораторная работа №5

Задача: Найти в заранее известном тексте поступающие на вход образцы с использованием суффиксного массива.

Вариант алгоритма: Поиск с использованием суффиксного массива. Можно делать как через суффиксное дерево, так и через сортировку.

1 Описание

Тебуется реализовать алгоритм поиска паттерна в тексте с помощью суффиксного массива. Данную задачу можно разбить на две подзадачи: алгоритм построения суффиксного массива, так как изначально он нам не дан, и сам алгоритм поиска паттерна в тексте. Начнем с алгоритма построения.

Существует два способа реализации алгоритма построения суффиксного массива: с помощью суффиксного массива и с помощью сортировки подсчетом. Алгоритм построения с помощью суффиксного дерева работает за линейное время, в случае если суффиксное дерево строится с помощью алгоритма Укконена. Алгоритм построения с помощью сортировки подсчетом работает за линейно-логарифмическое время, но его преимущество перед алгоритмом построения с помощью суффиксного дерева заключается в том, что он легче в понимании и реализации. Поэтому в своей лабораторной работе я реализовал именно этот алгоритм построения суффиксного массива. [1]

Также существует два способа реализации поиска паттерна в суффиксном массиве. Все они так или иначе базируются на алгоритме бинарного поиска. В случае если бы у нас была гарантия, что в тексте паттерн встречается не более одного раза, то мы могли бы использовать алгоритм бинарного поиска, модифицированный с помощью LCP-функции (Longest Common Prefix) - функции, которая ищет длину наибольшего общего префикса у двух строк. Такой алгоритм бы работал за время $O(m + logn)$, где m - количество символов в паттерне, а n - количество символов в тексте. Но в нашем случае, где нет никаких гарантий того, что паттерн войдет в текст не более одного раза хорошо подойдет немного модифицированный алгоритм левостороннего бинарного поиска. Такой алгоритм будет искать самое первое удовлетворяющее нас вхождение в массив. Модификация заключается в том, что как мы нашли самое левое вхождение, мы идем по массиву дальше, чтобы найти все вхождения паттерна в текст. Мы можем позволить себе так сделать из-за особенностей суффиксного массива. Такой алгоритм будет работать за время $O(mlogn + k)$, где n - количество символов в тексте, m - количество символов в паттерне, k - количество вхождений паттерна в текст. Таким образом мы найдем абсолютно все вхождения паттерна в текст.

2 Исходный код

Для начала опишем классы, которые присутствуют в программе. class TItem - класс суффикса в тексте хранящий в себе три переменные типа int: int idx - индекс начала суффикса в тексте, int old_equvalence_class - прошлый класс эквивалентности суффикса, соответствующий текущему индексу начала суффикса и int current_equivalence_class - текущий класс эквивалентности суффикса. У данного класса есть конструктор TItem(int idx, int old_equvalence_class, int current_equivalence_class) - который присваивает соответствующим полям соответствующие значения.

Опишем функции. int Modulo(int a, int b) - функция для правильного целочисленного деления для реализации индексации по циклическим строкам. int NextPowerOf2(size_t size) - функция, вычисляющую следующую ближайшую к числу степень двойки. Это функция нужна для подготовки строки к построению суффиксного массива, так как перед построением суффискного массива, для удобства реализации, необходимо добавить в конец незачащих символов столько, чтобы длина строки равнялась какой-либо степени двойки. std::vector<int> CountingSortWithIdxs(std::vector<char> array) - модифицированная функция для сортировки подсчетом. Данная функция нужна для сортировки букв в лексикографическом порядке. Так как каждая буква является символом в таблице ASCII и у каждой буквы есть свой код, то мы можем отсортировать эти буквы по их кодам. Также, в этой функции сортировки подсчетом, помимо того что по вектору перемещаются буквы, образую отсортированную последоветльность букв, в соответствующем буквам порядке передвигаются и их индексы в исходном тексте. Функция сортировки возвращает полученную перестановку индексов. Это необходимо для дальнейшей корректной работы алгоритма построения суффиксного массива. Сам алгоритм сортировки описывать смысла нет, так как он уже описывался мной в ЛР №1. void CountingSort(std::vector<TItem> array) - еще один модифицированный алгоритм сортировки подсчетом. Данная функция уже рассчитана на сортировку суффиксов по их текущим классам эквивалентности. Модификация алгоритма заключается в том, что он оставляет на исходных позициях старые классы эквивалентности и также выполняет перемещение индексов начала суффиксов. Это также необходимо для дальнейшей корректной работы алгоритма построения суффиксного массива. Сам алгоритм сортировки описывать смысла нет, так как он уже описывался мной в ЛР №1. std::vector<int> SuffixArrayBuilder(std::string text, int added_sentinels) - функция, реализующая алгоритм построения суффиксного массива. Для объяснения работы этой функции будет удобнее разделить эту функцию на две части: выполнение первого шага построения суффиксного массива и выполнение остальных шагов построения суффиксного массива. Опишем первый шаг алгоритма. На вход подается строка с текстом, длина которого равна степени двойки, так как до этого мы дополнили эту строку до такой длины. Также на вход подается числа, равное количеству дополнительных символов, которые мы добавили в текст. Создаем вектора для текста и для классов эквивалентности. Далее с помо-

щью функции для сортировки подсчетом для букв мы получаем вектор с индексами начала отсортированных суффиксов длины 1. Далее мы заполняем вектор с классами эквивалентности. Первый элемент в таком векторе всегда будет равен 0. Далее он заполняется так: если буква на текущем индексе отличается от буквы, стоящей на прошлом индексе, то класс эквивалентности в текущей позиции равен классу эквивалентности на прошлой позиции + 1, иначе текущий класс эквивалентности равен прошлому классу эквивалентности. Далее мы создаем и заполняем вектор с классами эквивалентности, соответствующие индексу начала суффикса. Этот вектор нам понадобится для пересчета классов эквивалентностей на следующем этапе. Далее мы создаем вектор с классами TItem для суффиксов и заполнаяем его. Для поля idx мы пока оставляем те же значения. Для поля old_equivalence_class мы присваеваем значения вектора с классами эквивалентности соответствующим индексу начала суффикса (позже станет понятно почему). Для поля current_equivalence_class мы присваеваем значения из вектора с обычными классами эквивалентности. На этом первый шаг алгоритма закончен. Теперь рассмотрим выполнение остальных шагов алгоритма. Они находятся отдельно в цикле, так как остальные шаги немного отличаются от первого шага. Цикл работает по степени двойки, и заканчивается когда степень двойки достигает значения в два раза меньшего длины строки. Далее для всех суффиксов мы вычисляем новые индексы начала этих суффиксов длины в два раза больше. Затем пересчитываем новые классы эквивалентности для суффиксов длиной в два раза больше предыдущих с помощью поля old_equivalence_class. Затем мы выполняем сортировку подсчетом уже для новых классов эквивалентности и теперь получаем отсортированные суффиксы длиной в два раза предыдущих. Далее нам нужно пересчитать текущие классы эквивалентности. Это делается чере пары значений: класс эквивалентности для первой части суффикса-класс эквивалентности для второй части суффикса. Для второй части суффикса классы эквивалентности уже посчитаны, а для первой части суффикса мы берем класс эквивалентности уже из прошлого шага, как раз-таки с помощью поля old_eqivalence_class. Затем, по обычному принципу вычисления классов эквивалентности мы уже сравниваем полученные пары. Затем мы пересчитываем поля old_equivalence_class и переходи к следующей итерации цикла. Таким образом мы получаем суффиксный массив для данной строки. Теперь мы возвращаем его, но отрезаем первые n значений, где n - количество добавленных к тексту символов, потому что в итоговом суффиксном массиве их не должно быть. int StringsComparator(std::string text, std::string pattern, int idx) - компаратор для строк. Возвращает значения 0 - паттерн и текст равны, -1 - паттерн меньше, 1 - паттерн больше. int main() - точка входа в программу. Здесь реализовано считывание текста, паттернов, поиск паттернов и вывод итоговых значений. Алгоритм поиска уже описан в главе 1.

```
1 #include <iostream>
2 #include <vector>
3 #include <string>
```

```cpp
#include <algorithm>
#include <limits.h>

const size_t ALPHABET_SIZE = 256;

class TItem
{
public:
    int idx, old_equivalence_class, current_equivalence_class;
    TItem(int idx, int old_equvalence_class, int current_equivalence_class)
    {
        this->idx = idx;
        this->old_equivalence_class = old_equvalence_class;
        this->current_equivalence_class = current_equivalence_class;
    }
};

int Modulo(int a, int b)
{
    return (a >= 0 ? a % b : (b + a) % b);
}

int NextPowerOf2(size_t size)
{
    int power = 0;
    while (size > (1 << power))
    {
        power++;
    }
    return (1 << power);
}

std::vector<int> CountingSortWithIdxs(std::vector<char>& array)
{
    int tmp[ALPHABET_SIZE] = {0};
    std::vector<char> result = std::vector<char>(array.size(), char(0));
    std::vector<int> result_idxs = std::vector<int>(array.size(), 0);
    for (size_t i = 0; i < array.size(); ++i)
    {
        tmp[int(array[i])]++;
    }
    for (size_t i = 1; i < ALPHABET_SIZE; ++i)
    {
        tmp[i] += tmp[i - 1];
    }
    for (size_t i = array.size(); i > 0; --i)
    {
        --tmp[int(array[i - 1])];
        result[tmp[int(array[i - 1])]] = array[i - 1];
```

```
53          result_idxs[tmp[int(array[i - 1])]] = i - 1;
54      }
55      for (size_t i = 0; i < array.size(); ++i)
56      {
57          array[i] = result[i];
58      }
59      return result_idxs;
60  }
61
62  void CountingSort(std::vector<TItem>& array)
63  {
64      int tmp[array.size()] = {0};
65      std::vector<TItem> result = std::vector<TItem>(array.size(), {0, 0, 0});
66      for (size_t i = 0; i < array.size(); ++i)
67      {
68          result[i].old_equivalence_class = array[i].old_equivalence_class; // old
                equivalence classes are in need order for index requests
69      }
70      for (size_t i = 0; i < array.size(); ++i)
71      {
72          tmp[array[i].current_equivalence_class]++;
73      }
74      for (size_t i = 1; i < array.size(); ++i)
75      {
76          tmp[i] += tmp[i - 1];
77      }
78      for (size_t i = array.size(); i > 0; --i)
79      {
80          --tmp[array[i - 1].current_equivalence_class];
81          result[tmp[array[i - 1].current_equivalence_class]].current_equivalence_class =
                array[i - 1].current_equivalence_class;
82          result[tmp[array[i - 1].current_equivalence_class]].idx = array[i - 1].idx;
83      }
84      for (size_t i = 0; i < array.size(); ++i)
85      {
86          array[i] = result[i];
87      }
88  }
89
90  std::vector<int> SuffixArrayBuilder(std::string text, int added_sentinels)
91  {
92      std::vector<char> text_vector;
93      std::vector<int> equivalence_classes = std::vector<int>(text.size(), 0);
94      for (size_t i = 0; i < text.size(); ++i)
95      {
96          text_vector.push_back(text[i]);
97      }
98      std::vector<int> suffix_array = CountingSortWithIdxs(text_vector);
99      equivalence_classes[0] = 0;
```

```
100     for (size_t i = 1; i < text.size(); ++i)
101     {
102         equivalence_classes[i] = (text_vector[i] == text_vector[i - 1] ?
                equivalence_classes[i - 1] : equivalence_classes[i - 1] + 1);
103     }
104     std::vector<int> equivalence_classes_specified_index = std::vector<int>(text.size()
            , 0);
105     for (size_t i = 0; i < suffix_array.size(); ++i)
106     {
107         equivalence_classes_specified_index[suffix_array[i]] = equivalence_classes[i];
108     }
109     std::vector<TItem> items_array;
110     for (size_t i = 0; i < text.size(); ++i)
111     {
112         items_array.push_back({suffix_array[i], equivalence_classes_specified_index[i],
                 equivalence_classes[i]}); // not old equivalence classes, but in need
                order for index requests
113     }
114     for (int p = 0; (1 << p) < text.size(); ++p)
115     {
116         for (size_t i = 0; i < text.size(); ++i)
117         {
118             items_array[i].idx = Modulo((items_array[i].idx - (1 << p)), text.size());
119             items_array[i].current_equivalence_class = items_array[items_array[i].idx].
                    old_equivalence_class;
120         }
121         CountingSort(items_array);
122         std::vector<std::pair<int, int>> pair_equivalence_classes;
123         for (size_t i = 0; i < items_array.size(); ++i)
124         {
125             pair_equivalence_classes.push_back({items_array[i].
                    current_equivalence_class, items_array[Modulo((items_array[i].idx + (1
                    << p)), text.size())].old_equivalence_class});
126         }
127         items_array[0].current_equivalence_class = 0;
128         for (size_t i = 1; i < items_array.size(); ++i)
129         {
130             items_array[i].current_equivalence_class = (pair_equivalence_classes[i] ==
                    pair_equivalence_classes[i - 1] ? items_array[i - 1].
                    current_equivalence_class : items_array[i - 1].current_equivalence_class
                     + 1);
131             items_array[items_array[i].idx].old_equivalence_class = items_array[i].
                    current_equivalence_class;
132         }
133     }
134     std::vector<int> result_suffix_array;
135     for (size_t i = added_sentinels; i < items_array.size(); ++i)
136     {
137         result_suffix_array.push_back(items_array[i].idx);
```

```
138     }
139     return result_suffix_array;
140 }
141 
142 int StringsComparator(std::string text, std::string pattern, int idx) // 0 - , -1 -  ,
        1 -
143 {
144     int flag = 0;
145     for (size_t i = 0; i < pattern.size(); ++i)
146     {
147         if (idx + i + 1 > text.size())
148         {
149             flag = 1;
150             break;
151         }
152         if (text[idx + i] != pattern[i])
153         {
154             flag = (text[idx + i] > pattern[i] ? -1 : 1);
155             break;
156         }
157     }
158     return flag;
159 }
160 
161 int main()
162 {
163     std::string input_text;
164     getline(std::cin, input_text);
165     int added_sentinels = NextPowerOf2(input_text.size() + 1) - input_text.size(); //
        +1 - $  , -1 $  , +1   $
166     std::string input_text_with_sentinels = input_text + std::string(added_sentinels, '
        $');
167     std::vector<int> suffix_array = SuffixArrayBuilder(input_text_with_sentinels,
        added_sentinels);
168     std::string pattern;
169     int patterns_counter = 0;
170     while (getline(std::cin, pattern))
171     {
172         std::vector<int> result;
173         patterns_counter++;
174         if ((pattern.size() > input_text.size()) || (pattern.size() == 0))
175         {
176             continue;
177         }
178         int l = 0;
179         int r = suffix_array.size() - 1;
180         while (l + 1 < r)
181         {
182             int m = (l + r) / 2;
```

```
183             int compare_result = StringsComparator(input_text, pattern, suffix_array[m
                    ]);
184             if (compare_result == 1)
185             {
186                 l = m;
187             }
188             else
189             {
190                 r = m;
191             }
192         }
193         if (StringsComparator(input_text, pattern, suffix_array[l]) == 0)
194         {
195             result.push_back(suffix_array[l]);
196             for (size_t i = l + 1; i < suffix_array.size(); ++i)
197             {
198                 if (StringsComparator(input_text, pattern, suffix_array[i]) == 0)
199                 {
200                     result.push_back(suffix_array[i]);
201                 }
202                 else
203                 {
204                     break;
205                 }
206             }
207         }
208         else if (StringsComparator(input_text, pattern, suffix_array[r]) == 0)
209         {
210             result.push_back(suffix_array[r]);
211             for (size_t i = r + 1; i < suffix_array.size(); ++i)
212             {
213                 if (StringsComparator(input_text, pattern, suffix_array[i]) == 0)
214                 {
215                     result.push_back(suffix_array[i]);
216                 }
217                 else
218                 {
219                     break;
220                 }
221             }
222         }
223         if (result.size() != 0)
224         {
225             std::cout << patterns_counter << ": ";
226             std::sort(result.begin(), result.end());
227             std::cout << result[0] + 1;
228             for (size_t i = 1; i < result.size(); ++i)
229             {
230                 std::cout << ", " << result[i] + 1;
```

```
231             }
232             std::cout << "\n";
233         }
234     }
235 }
```

main.cpp	
int Modulo()	Функция целочисленного деления по модулю
int NextPowerOf2()	Функция подсчета следующей ближайшей степени двойки
std::vector<int> CountingSortWithIdxs()	Функция сортировки подсчетом для букв
void CountingSort()	Функция сортировки подсчетом для чисел
std::vector<int> SuffixArrayBuilder()	Функция построения суффиксного массива
int StringsComparator()	Функция компаратора строк
int main()	Точка входа программы

```
1  class TItem
2  {
3  public:
4      int idx, old_equivalence_class, current_equivalence_class;
5      TItem(int idx, int old_equvalence_class, int current_equivalence_class)
6      {
7          this->idx = idx;
8          this->old_equivalence_class = old_equvalence_class;
9          this->current_equivalence_class = current_equivalence_class;
10     }
11 };
```

3 Консоль

```
lexasy@lexasy$ cat tests/01.t
cacaacaca
ac
caca
aca
dddaa
aadbbbada
cdddccabbd
```

```
abb
acac
a
dabc
lexasy@lexasy$ make
g++ main.cpp -o solution
lexasy@lexasy$ ./solution <tests/01.t
1: 2,5,7
2: 1,6
3: 2,5,7
8: 5
9: 2,4,5,7,9
```

4 Тест производительности

Тест производительности представляет из себя следующее: сравенение поиска вхождений паттерна в текст с помощью стандартного инструмента std::search и поиска вхождений паттерна в текст с помощью суффиксного массива. Данные для теста представляют собой текст, длинной до 1000 символов и 10000 паттернов, которые как могут входить в текст так и нет.

```
lexasy@lexasy$ make
g++ main.cpp -o solution
lexasy@lexasy$ g++ benchmark.cpp
lexasy@lexasy$ ./solution <tests/01.t | grep time
time: 20418ms
lexasy@lexasy$ ./a.out <tests/01.t | grep time
time: 41051ms
```

Как видно, наш поиск с помощью суффиксного массива работает в два раза быстрее чем поиск с помощью std::search. Это связано с тем, что std::search использует наивный алгоритм поиска, который работает по сути за квадратичное время, а именно за $O(n * m)$, где n - длина текста, m - длина паттерна. А ко всему перечесленному можно добавить то, что у нас может быть несколько вхождений, и std::search будет применен заново, то есть итоговая сложность получается $O(m * n * k)$, где k - количество вхождений паттерна в текст. Это очень много. Наш же алгоритм никогда не превышает даже квадратичную сложность. Очевидно что он будет работать намного быстрее.

5 Выводы

Выполнив 5 лабораторную работу по курсу Дискретный Анализ, я научился реализовывать алгоритм для построения суффиксного массива. Также я вспомнил алгоритм для сортировки подсчетом и алгоритм левостороннего бинарного поиска, который не применял с первого курса. Также я научился сортировать строки в лексикографическом порядке, не сравнивая строки.

Список литературы

[1] http://e-maxx.ru/algo/suffix_array: Суффиксный массив - 2008