Московский авиационный институт (национальный исследовательский университет)

Факультет компьютерных наук и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №5 по курсу «Дискретный анализ»

Студент: С. Ю. Свиридов
Преподаватель: А. А. Кухтичев
Группа: М8О-306Б-22

Дата:
Оценка:
Подпись:

Москва 2025

Лабораторная работа №5

Задача: Суффиксные деревья

Вариант алгоритма: Поиск образца с использованием статистики совпадений

Формат ввода

На первой строке располагается образец, на второй — текст.

Формат вывода

Последовательность строк содержащих в себе номера позиций, начиная с которых встретился образец. Строки должны быть отсортированы в порядке возрастания номеров.

1 Описание

Суффиксные деревья представляют собой сжатый trie, на ребрах которого хранятся всевозможные суффиксы текста, из которого дерево состоит. Однако структура суффиксного дерева может немного видоизменяться в зависимости от способа его построения. В моем варианте суффиксное дерево строится с помощью алгоритма Укконена за `O(n)`. После чего по дереву строится статистика совпадений. Сама статистика представляет собой некий массив, по длине совпадающий с текстом, в котором я буду искать вхождения паттерна, который на соответствующих индексах хранит длину наибольшей общей подстроки текста и образца. Чтобы получить номер вхождения, нужно просто получить индекс статистики и совпадений, который соответствует ячейке массива, хранящей в себе число, равное длине паттерна.

Реализация

Для начала строим суффиксное дерево по паттерну. Чтобы получить значение в соответствующем индексе статистики, нужно выполнить поиск в дереве. Выполняем поиск суффиксов текста в дереве до тех пор, пока не будет найдено несовпадение. Длину совпавшего суффикса записываем в нужный индекс массива ms (ms - matching statistic). Для перемещения по дереву мы будем использовать суффиксные ссылки и простые сравнения.

2 Исходный код

Для начала стоит рассказать о том, как описывается суффиксное дерево и как оно строится.

```
1  class TSuffTree {
2     public:
3      TSuffTree(const std::string &s) : str(std::move(s + SENTINEL)) {
4          root = new TNode(0, 0, INTERNAL, nullptr);
5
6          activeNode = root;
7          activeEdge = 0;
8          activeLength = 0;
9
10         remain = 0;
11
12         end = -1;
13         suffNum = 0;
14
15         size_t len = str.length();
```

```
16
17          for (size_t i = 0; i < len; i++) {
18              Phase(i);
19          }
20      }
21      ~TSuffTree() {
22          DeleteTree(root);
23      }
24
25      void Phase(size_t phaseNum) {
26          remain++;
27          end++;
28
29          TNode *lastAddedInternalNode = nullptr;
30
31          while (remain > 0) {
32              if (activeLength == 0) {
33                  activeEdge = phaseNum;
34              }
35              if (activeNode->getEdge(str, activeEdge) == nullptr) {
36                  activeNode->getEdge(str, activeEdge) = new TNode(phaseNum, &end, suffNum
                        , nullptr);
37
38                  suffNum++;
39                  remain--;
40
41                  if (lastAddedInternalNode != nullptr) {
42                      lastAddedInternalNode->suffLink = activeNode;
43                      lastAddedInternalNode = nullptr;
44                  }
45              } else {
46                  if (Walkdown())
47                      continue;
48                  if (getNextCharAct() == str[phaseNum]) {
49                      if (lastAddedInternalNode != nullptr && activeNode != root) {
50                          lastAddedInternalNode->suffLink = activeNode;
51                          lastAddedInternalNode = nullptr;
52                      }
53                      activeLength++;
54                      break;
55                  } else {
56                      TNode *toInsert = new TNode(phaseNum, &end, suffNum, nullptr);
57                      suffNum++;
58
59                      TNode *justInserted = Insert(toInsert);
60                      if (lastAddedInternalNode != nullptr) {
61                          lastAddedInternalNode->suffLink = justInserted;
62                      }
63                      lastAddedInternalNode = justInserted;
```

```
                remain--;
            }
        }
        if (activeNode == root) {
            if (activeLength > 0) {
                activeLength--;
                activeEdge = phaseNum - remain + 1;
            }
        } else {
            activeNode = activeNode->suffLink;
        }
    }
}


std::vector<int> getMatchStatistic(const std::string &text);
```

Итак, у нашего суффиксного дерева в модификаторе доступа `public` есть всего 2 метода, не считая конструктора и деструктора: `Phase` и `getMatchStatistic`. В конструкторе данного класса мы передаем текст, по которому будет построено дерево, этот текст разбивается на префиксы и обрабатывается с помощью функции `Phase`. Так как дерево строится алгоритмом Укконена, то на ребре хранятся не подстроки текста, а два числа, которые является границами соответствующей подстроки в тексте. Такой способ построения дерева имеет 3 основых эвристики:

1. На каждой итерации инкрементировать значение конца диапазона для каждого листа дерева.

2. Если окончили поиск в середине ребра или во внутренней вершине, то должны создать новый лист. При этом если закончили в середине ребра, то нужно поделить существующее ребро на два: первое ребро будет иметь диапазон от своего начала, до номера символа, после которого случилось несовпадение, а второе ребро - от символа, на котором случилось несовпадение и до конца. Если закончили во внутренней вершине, то просто создаем лист и соединяем его ребром с текущей внутренней вершиной.

3. Если на какой то итерации текущий символ существует в каком-то из ребер, то ничего не делаем, идем дальше.

Так же нужно добавить суффиксные ссылки, дабы избежать ненужных и долгих поисков от корня. Суффиксная ссылка будет у каждой внутренней вершины и указывать она будет на наибольший из возможных суффиксов подстроки, конец которой является вершиной, из которой мы ведем эту суффиксную ссылку. Условно, суффиксная ссылка из конца подстроки "abcd"будет указывать на конец подстроки "bcd".

Теперь рассмотрим основную функцию, которая будет строить статистику совпадений основываясь на дереве.

```
1  std::vector<int> TSuffTree::getMatchStatistic(const std::string &text) {
2      size_t len = text.length();
3      std::vector<int> result(len);
4
5      TNode *lastNode = root;
6      TNode *curNode;
7      size_t curEdge = 0;
8      size_t curLen = 0;
9
10     size_t ind = 0;
11
12     for (int i = 0; i < len; i++) {
13         if (curLen == 0) {
14             curNode = lastNode->getEdge(text, ind);
15         } else {
16             curNode = lastNode->getEdge(str, curEdge);
17         }
18
19         if (curNode != nullptr) {
20             size_t curEdgeLen = curNode->getLength();
21             while (curLen > curEdgeLen) {
22                 curEdge += curEdgeLen;
23                 curLen -= curEdgeLen;
24                 lastNode = curNode;
25                 curNode = curNode->getEdge(str, curEdge);
26                 curEdgeLen = curNode->getLength();
27             }
28             assert(curNode != nullptr);
29             while (1) {
30                 if (curNode->getLength() == curLen) {
31                     curLen = 0;
32                     lastNode = curNode;
33                     curNode = curNode->getEdge(text, ind);
34                 }
35                 if (curNode == nullptr || str[curNode->getLeft() + curLen] != text[ind])
                        {
36                     break;
37                 }
38                 curLen++;
39                 ind++;
40                 if (ind >= len) {
41                     for (int k = i; k < len; k++) {
42                         result[k] = ind - k;
43                     }
44                     return result;
45                 }
46             }
```

```
47              if (curNode != nullptr) {
48                  curEdge = curNode->getLeft();
49              }
50          }
51
52          result[i] = ind - i;
53
54          if (lastNode == root) {
55              if (curLen > 0) {
56                  curLen--;
57                  curEdge++;
58              } else {
59                  ind++;
60              }
61          } else {
62              lastNode = lastNode->suffLink;
63          }
64      }
65
66      return result;
67  }
```

Идея состоит в том, чтобы для каждой позиции текста искать наибольшую общую строку с паттерном, начинающуюся с этой позиции. Для этого будем посимвольно сравнивать подстроки паттерна и текста. Начинаем с корня дерева. Перемещаемся по нему вниз, пока не достигнем листа или несовпадения символов. Чтобы ускорить процесс, введем ограничение - если конец текущей совпавшей подстроки совпадает с концом текста, то можно дозаполнить результирующий массив ms длинами ее префиксов, это позволит не делать лишних проверок. Так же используем суффиксные ссылки: если при несовпадении остановились в середине ребра, то переходим к ближайшей внутренней вершине, ближе к корню, и переходим по суффиксной ссылке; если остановились во внутренней вершине, переходим сразу по суффиксной ссылке.

3 Консоль

```
potatogrill24@DESKTOP-7CM71EV:~/progs/Diskran/laba5$ ./a.out
aba
qabababababz
2
4
6
potatogrill24@DESKTOP-7CM71EV:~/progs/Diskran/laba5$ ./a.out
a
bbbbbbbbbbbbbbbbbbbabbbbaaaaaaaaaaaaaaaaaaaaaaaaaaaa
21
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
```

4 Тест производительности

В тестах я решил посмотреть на время построения дерева по образцу, длиной 3426, 6852, 10278 символов. Так же сравню время поиска вхождений этого паттерна в текст, длиной 30873 символа:

```
potatogrill24@DESKTOP-7CM71EV:~/progs/Diskran/laba5$ g++ bench.cpp
potatogrill24@DESKTOP-7CM71EV:~/progs/Diskran/laba5$ ./a.out
Pattern length: 3426
Tree building: 1134ms
potatogrill24@DESKTOP-7CM71EV:~/progs/Diskran/laba5$ ./a.out
Pattern length: 6852
Tree building: 1729ms
potatogrill24@DESKTOP-7CM71EV:~/progs/Diskran/laba5$ ./a.out
Pattern length: 10278
Tree building: 2944ms
```

Как видно из тестов, дерево действительно строится примерно за `O(n)`, потому что в три раза увеличив размер текста, время построения дерева увеличилось примерно в три раза, что говорит о линейной зависимости между временем построения и длиной текста.

```
potatogrill24@DESKTOP-7CM71EV:~/progs/Diskran/laba5$ g++ bench.cpp
potatogrill24@DESKTOP-7CM71EV:~/progs/Diskran/laba5$ ./a.out
Pattern length 3426; Text length 30873
Matching statistic: 627ms
Naive search: 967ms
```

По тестам видно, что статистика совпадений выигрывает по времени у наивного алгоритма, однако так происходит не всегда. Например при менее длинном паттерне и тексте наивный алгоритм выиграет у статистики совпадений по времени. Так же стоит учесть, что замер времени происходил без учета построения дерева.

Выводы по производительности

Я считаю, что искать вхождения паттерна в строку с помощью статистики совпадений и суффиксного дереве не особо правильно, потому что, во-первых, довольно сложно реализовать эффективный алгоритм построения суффиксного дерева, к тому же оно достаточно затратно по памяти; а во-вторых, есть куда более эффективные алгоритмы для таких задач, например алгоритм КМП или алгоритм Бойера-Мура.

5 Выводы

В ходе выполнения данной лабораторной работы, я попробывал реализовать две структуры - суффиксное дерево и статистику совпадений. Было достаточно проблематично сделать эффективный алгоритм, но дело сделано. В целом, я считаю, поиск вхождений образца в строку с помощью суффиксного дерева и статистики совпадений не имеет особого смысла, поскольку поиск вхождений с помощью самого суффиксного дерева достаточно эффективен, и также есть более быстрые алгоритмы для решения подобного рода задач, например алгоритм Кнута-Морриса-Пратта или алгоритм Бойера-Мура. Однако реализация суффиксного дерева (алгоритма Укконена) и статистики совпадений очень неслабо тренирует логику и умение писать код.

Список литературы

[1] *Алгоритм Укконена*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Укконена`

[2] *Книга Гасфилда*
Гасфилд, Д. (2003). Строки, деревья и последовательности в алгоритмах (с. 171)