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. М. Голев
Преподаватель: С. А. Михайлова
Группа: М8О-201Б

Дата: 08.10.25
Оценка:
Подпись:

Москва, 2025

Лабораторная работа №5

Задача: Найти в заранее известном тексте поступающие на вход образцы.

1 Описание

Требуется написать реализацию алгоритма для поиска образцов в заранее известном тексте. Для этого хорошо подходит суффиксное дерево. Для его эффективного построения используется алгоритм Укконена.

Идея алгоритма в том, что он последовательно добавляет в сжатый trie все суффиксы каждого префикса. При этом используются несколько приемов для ускорения:

1. Вместо самих подстрок на ребрах хранятся индексы их начала и конца (в данной реализации - буквы, следующей за концом). При этом для листьев конец определяется, как глобальный указатель. Преимущество от этого будет рассмотрено ниже.

2. Вершина соответствующая строке $x\alpha$ соединяется с вершиной α указателем, называемым суффиксной ссылкой (x обозначает произвольную букву, а α - последовательность букв). Т. к. после суффикса $x\alpha y$ алгоритм вставляет суффикс αy переход по этой ссылке избавляет от необходимости поиска строки α.

3. Если при возвращении к ближайшей вершине, содержащей суффиксную ссылку, была пропущена строка γ, то после перехода по суффиксной алгоритм проходит по строке γ из новой вершины (ее существование следует из порядка добавления суффиксов). Однако в новой вершине строка γ может быть записана на нескольких ребрах. При прохождении этих ребер алгоритм сравнивает только первую букву на каждом ребре и пропускает все остальные. Эта оптимизация называется "прыжками по счетчику".

Вставка очередного суффикса выполняется по следующим правилам: [1]

1. Если для суффикса βx строка β содержится в дереве целиком, но после нее нет x, то x дописывается на последнее ребро (которое будет листом) в пути содержащем данную строку.

2. Если для суффикса $\beta\gamma$ строка β есть в дереве, а γ после нее отсутствует, то после строки бета производится разделение ребра. К созданной вершине присоединяется новый лист со строкой γ.

3. Если суффикс целиком содержится в дереве - не выполняется никаких действий.

Важные замечания:

1. При вставке суффиксов одного префикса правила применяются в порядке возрастания номера.

2. Для применения правило один для всех суффиксов, удовлетворяющих ему, достаточно увеличить глобальный указатель на конец на 1.

3. После первого применения правила 3 для некоторого суффикса текущего префикса можно сразу переходить к следующему префиксу.

Со всеми перечисленными оптимизациями алгоритм Укконена строит настоящее суффиксное дерево за время $O(m)$. [2]

2 Исходный код

Программа определяет макро-подстановки EXTEND SIZE и NOT LEAF. Первая соответсвует размеру увеличения буфера. Вторая помещается в поле $leafNum$ внутреннего узла дерева для обозначения того, что она не является листовой.

Далее определяется структура $Pair$, хранящая пары ключ-значение. Эта структура используется в определении типа Map - хэш-таблице, хранящей пары "первая буква ребра - соответствующая вершина". Следующие за этим типом заголовки функции - операции над ним (рассматриваются ниже).

Затем вводится вспомогательный тип Pos - позиция в тексте.

Наконец за ним следует определение узла дерева. Узел содержит номер листа, позиции начала и конца ребра, хэш-таблицу исходящих ребер и суффиксную ссылку. При этом позиция конца представлена объединением (union) позиции и указателя на позицию. Как объяснялось в описании алгоритма, если ребро является листом - записывается не позиция конца, а указатель на нее. Используется тот факт, что любой указатель в языке C занимает в памяти одинаковое количество байт.

Следом идут заголовки функций, являющихся операциями над узлами дерева.

Затем объявляются функции для считывания текста и паттернов.

После чего определяется структура самого суффиксного дерева. Она содержит указатель на корень и глобальный конец (хранить его в локальной переменной функции построения некорректно, т. к. значение будет выгружена из памяти после завершения работы функции).

Следом объявляются функции построения и удаления суффиксного дерева.

За ними следует объявление функции поиска паттерна по СД (вместе с несколькими вспомогательными).

В конце определяется функция main. В ней считывается текст, после чего по нему сразу строится суффиксное дерево. За ним объявляется буферы для считывания образца и записи номеров суффиксов при обходе (излишняя выделение и освобождение памяти для них избегается).

После этого в цикле считываются паттерны и выполняется их поиск. В конце высвобождается вся выделенная память.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4 #include <inttypes.h>
5
6 #define EXTEND_SIZE 10
7 #define NOT_LEAF UINT32_MAX
8
9 struct S3_Node;
```

```
10
11  typedef struct Pair {
12      char key;
13      struct S3_Node *value;
14  } Pair;
15
16  typedef struct Map {
17      unsigned char size;
18      unsigned char cap;
19      Pair *items;
20  } Map;
21
22  void initMap(Map *map);
23  void setMap(Map *map, char key, struct S3_Node *value);
24  struct S3_Node* getMap(Map *map, char key);
25  void extendMap(Map *map);
26  void deleteMap(Map *map, void (*destructor)(struct S3_Node*));
27
28  typedef char* Pos;
29
30  typedef struct S3_Node {
31      uint32_t leafNum;
32      Pos start;
33
34      union {
35          Pos letter;
36          Pos *ptr;
37      } end;
38
39      Map children;
40      struct S3_Node *suffixLink;
41  } S3_Node, *S3;
42
43  S3 createS3Node(Pos start, Pos end, uint32_t leafNum);
44  char getCharS3(S3 node, long index);
45  long lengthS3(S3 node);
46  S3 splitS3(S3 node, int index);
47  void deleteS3(S3 node);
48  void printS3(S3 node, long offset);
49
50  char* readText(long *length);
51  char* readPattern(char *buffer, long *length, long *capacity);
52
53  typedef struct SuffixTree {
54      Pos end;
55      S3 root;
56  } _SuffixTree, *SuffixTree;
57
58  SuffixTree buildSuffixTree(char *text, long m);
```

```
59 void deleteSuffixTree(SuffixTree tree);
60
61 uint32_t* leavesDFS(S3 root, uint32_t *matches, long *matchCount, long *matchCap);
62 void countingSort (uint32_t *arr, long n);
63 uint32_t* findMatches(long patternNum, SuffixTree tree, char *pattern, long m,
       uint32_t *matches, long *matchCap);
64
65 int main() {
66     long n;
67     char *text = readText(&n);
68
69     SuffixTree tree = buildSuffixTree(text, n);
70
71     long cap = 0;
72     long m = 0;
73     char *pattern = NULL;
74     long patternNum = 0;
75
76     long matchCap = 0;
77     uint32_t *matches = NULL;
78
79     while (1) {
80         m = 0;
81         pattern = readPattern(pattern, &m, &cap);
82         if (feof(stdin) || m == 0) break;
83         patternNum++;
84
85         matches = findMatches(patternNum, tree, pattern, m, matches, &matchCap);
86     }
87
88     free(matches);
89     free(pattern);
90     deleteSuffixTree(tree);
91     free(text);
92
93     return 0;
94 }
```

main.c	
void initMap(Map *map);	Конструктор типа Map.
void setMap(Map *map, char key, struct S3_Node *value);	Операция записи значения по ключу в Map.
struct S3_Node* getMap(Map *map, char key);	Операция чтения значения по ключу в Map.
void extendMap(Map *map);	desc
void deleteMap(Map *map, void (*destructor)(struct S3_Node*));	Деструктор типа Map. Принимает деструктор, вызываемый для потомков.
S3 createS3Node(Pos start, Pos end, uint32_t leafNum);	Конструктор узла дерева
char getCharS3(S3 node, long index);	Операция получения буквы на ребре по индексу.
long lengthS3(S3 node);	Операция получения длины ребра.
void deleteS3(S3 node);	Деструктор суффиксного поддерева.
void printS3(S3 node, long offset);	Вывод суффиксного дерева на экран (для отладки).
char* readText(long *length);	Функция, считывающая текст.
char* readPattern(char *buffer, long *length, long *capacity);	Функция, считывающая паттерн в переданный буфер.
SuffixTree buildSuffixTree(char *text, long m);	Функция построения суффиксного дерева (реализация алгоритма Укконена).
void deleteSuffixTree(SuffixTree tree);	Деструктор суффиксного дерева.
uint32_t* leavesDFS(S3 root, uint32_t *matches, long *matchCount, long *matchCap);	Поиск всех суффиксов, в которые входит данная подстрака. Принимает буфер для записи вхождений.
void countingSort (uint32_t *arr, long n);	Сортировка подсчетом найденных позиций вхождений.
uint32_t* findMatches(long patternNum, SuffixTree tree, char *pattern, long m, uint32_t *matches, long *matchCap);	Функция поиска паттерна в тексте. Использует переданный буфер для вхождений.

3 Консоль

```
gcc main.c -o app.out
./app.out
abacabadaba
aba
1: 1,5,9
bac
2: 2
a
3: 1,3,5,7,9,11
```

4 Тест производительности

Тест производительности представляет из себя следующее: сравнивается время поиска паттерна $ATGC$ в случайное последовательности из 10^6 букв A, T, G и C. методами find (из стандартной библиотеки языка C++) и реализованным алгоритмом поиска через суффиксное дерево. Время на ввод и вывод не учитывается.

```
smoking_elk@DESKTOP-PJPQAEE:~/discran-labs/lab5/benchmark$ make
g++ find.cpp -o find.out
gcc suffix_tree.c -o suffix_tree.out
smoking_elk@DESKTOP-PJPQAEE:~/discran-labs/lab5/benchmark$ make test_find
./find.out <./test.txt | grep "time"
time: 5.208ms
smoking_elk@DESKTOP-PJPQAEE:~/discran-labs/lab5/benchmark$ make test_suffix_tree
./suffix_tree.out <./test.txt | grep "time"
time: 4.169000ms
```

Как видно, из результатов, поиск, использущий метод find, имеет время того же порядка, что и реализованный поиск через суффиксное дерево. Эти результаты согласуются со сложностной оценкой алгоритмов: $O(n + m)$ для find (который, скорее всего, использует алгоритм Бойера-Мура), но требует вызова t раз для поиска всех вхождений и $O(n+m+t)$ для алгоритма поиска, использующего суффиксное дерево.

5 Выводы

Выполнив пятую лабораторную работу по курсу «Дискретный анализ», я научился реализовывать алгоритм Укконена, а также производить поиск паттерна в суффиксном дереве и работать с типом *union* в языке C. Также я попрактиковался в реализации хэш-таблицы на языке C.

Список литературы

[1] *Алгоритм Укконена — Википедия.*
URL: `https://en.wikipedia.org/wiki/Ukkonen's_algorithm` (дата обращения: 03.10.2025).

[2] Дэн Гасфилд *Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология* — Издательский дом «Невский Диалект», 2003. Перевод с английского: И. В. Романовского.