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: О. А. Мезенин
Преподаватель: А. А. Кухтичев
Группа: М8О-306Б-21
Дата:
Оценка:
Подпись:

Москва, 2023

Лабораторная работа №5

Задача: Линеаризовать циклическую строку, то есть найти минимальный в лексикографическом смысле разрез циклической строки.

1 Описание

Первым шагом необходимо удвоить входную строку S, получив строку SS, и дописать терминальный символ \$. Затем построить суффиксное дерево получившийся строки. После чего нужно пройти дерево, двигаясь по дуге, которое начинается с наименьшей в лексикографическом смысле буквы (при этом терминальный символ \$ считается самым старшим символом). Двигаться по дереву нужно до тех пор, пока длина пути от корня не станет больше длины строки S. Получим номер K суффикса листа, в котором мы остановились. Ответом будет подстрока $SS[K : K + |\$|]$.

Для построения суффиксного дерева воспользуемся алгоритмом Укконена [1].

Пусть n – длина входной строки. Дерево строится за $O(n)$ и обходится за $O(n)$ времени. Итого алгоритм линеаризации циклической строки будет работать за $O(n)$ времени и потреблять $O(n)$ дополнительной памяти.

2 Исходный код

Создадим класс `TSuffixTree` – суффиксное дерево.

```
class TSuffixTree {
    struct TNode {
        size_t start;
        int *end;
        int suffixNumber;
        TNode *suffixLink;
        std::unordered_map<char, TNode *> children;

        TNode(size_t start, int *end, int suffixNumber, TNode *suffixLink)
        : start(start), end(end), suffixNumber(suffixNumber), suffixLink(suffixLink) {}
    };
// methods
public:
    TSuffixTree(std::string &text);
    int FindMinimumLineSectionSuffixNumber(int n);
private:
    void Build();
    int Dfs(TNode *node, int n);
// variables
private:
    std::string text;
    TNode *root = nullptr;
};
```

В функции `main` читается строка S, создаётся строка SS, затем создаётся суффиксное дерево и вызывается функция для поиска номера суффикса минимального разреза строки.

При инициализации дерева к полученной строке добавляется терминальный символ \$, инициализируется корень и вызывается функция `Build`. Функция построения дерева с помощью алгоритма Укконена выглядит следующим образом:

```
void TSuffixTree::Build() {
    /* builds a tree using Ukkonen's algorithm */
    TNode *currentNode = root;
    size_t j = 0;
    size_t currentCharIndex = 0;
    size_t currentEdgeLength = 0;
    int *globalEnd = new int(-1);
    for (size_t i = 0; i < text.length(); ++i) {
        /* single phase algorithm (SPA) */
        TNode *lastCreatedNode = nullptr;
        (*globalEnd)++; // "add" a text[phase] to each leaf (rule 1)
        for (; j <= i;) {
            /* single extension algorithm (SEA) */
```

```
14              if (currentEdgeLength == 0) { // starts from root
15                  currentCharIndex = i;
16              }
17
18              if (not currentNode->children.contains(text[currentCharIndex])) { // if
                    edges do not start with currentCharIndex (rules 1, 2)
19                  currentNode->children[text[currentCharIndex]] = new TNode(i, globalEnd,
                        j, root);
20                  if (lastCreatedNode) {
21                      lastCreatedNode->suffixLink = currentNode;
22                      lastCreatedNode = nullptr;
23                  }
24              } else {
25                  TNode *nextNode = currentNode->children[text[currentCharIndex]];
26                  size_t nextEdgeLength = *(nextNode->end) - (nextNode->start) + 1;
27
28                  if (currentEdgeLength >= nextEdgeLength) { // go down
29                      currentCharIndex += nextEdgeLength;
30                      currentEdgeLength -= nextEdgeLength;
31                      currentNode = nextNode;
32                      continue;
33                  }
34
35                  if (text[nextNode->start + currentEdgeLength] == text[i]) { // if edge
                        starts with text[phase] (rule 3)
36                      ++currentEdgeLength;
37                      if (lastCreatedNode and currentNode != root) {
38                          lastCreatedNode->suffixLink = currentNode;
39                      }
40                      break;
41                  }
42
43                  // insert a new node inside the edge (rule 2)
44                  TNode *newNode = new TNode(nextNode->start, new int(nextNode->start +
                        currentEdgeLength - 1), -1, root);
45                  currentNode->children[text[currentCharIndex]] = newNode;
46                  nextNode->start += currentEdgeLength;
47                  newNode->children[text[nextNode->start]] = nextNode;
48                  newNode->children[text[i]] = new TNode(i, globalEnd, j, root);
49                  if (lastCreatedNode) {
50                      lastCreatedNode->suffixLink = newNode;
51                  }
52                  lastCreatedNode = newNode;
53              }
54
55              if (currentNode == root) {
56                  if (currentEdgeLength) {
57                      ++currentCharIndex;
58                      --currentEdgeLength;
```

```
59                  }
60              } else {
61                  currentNode = currentNode->suffixLink;
62              }
63              ++j;
64          }
65      }
66  }
```

В этой функции реализуются все оптимизации и алгоритмы `Single Phase Algorithm (SPA)` и `Single Extension Algorithm (SEA)` [1].

Функции для поиска суффикса минимального разреза строки выглядят следующим образом:

```
1   int TSuffixTree::Dfs(TNode *node, int n) {
2       if (not node) {
3           return -1;
4       }
5       if ((*node->end) > n and node->suffixNumber != -1) {
6           return node->suffixNumber;
7       }
8       if (node->children.empty()) {
9           return -1;
10      }
11      char minChar = 'z'+1;
12      for (auto &key: node->children) {
13          if (key.first != '$' and key.first < minChar) {
14              minChar = key.first;
15          }
16      }
17      if (minChar == 'z'+1) {
18          minChar = '$';
19      }
20      return Dfs(node->children[minChar], n);
21  }
22
23  int TSuffixTree::FindMinimumLineSectionSuffixNumber(int n) {
24      return Dfs(root, n);
25  }
```

3 Консоль

```
aaprold@SAI:~/Documents/GitHub/MAI-DA/lab5$ make lab5
g++ -std=c++2a -pedantic -Wall -Wextra -Werror suffix_tree.cpp main.cpp -o
lab5.out
aprold@SAI:~/Documents/GitHub/MAI-DA/lab5$ ./lab5.out
xabcd
abcdx
aprold@SAI:~/Documents/GitHub/MAI-DA/lab5$ ./lab5.out
aabaacaa
aaaabaac
```

4 Тест производительности

Тесты производительности представляют из себя следующее: алгоритм линеаризации циклической строки с помощью суффиксного дерева будет сравниваться с примитивным алгоритмом: перебором всех строк, полученных циклическим смещением, для поиска наименьшей из таких. Скорость построения дерева также учитывается. Будет три теста со строками, состоящих из 10000, 100000 и 1000000 символов. Все строки случайные.

```
aprold@SAI:~/Documents/GitHub/MAI-DA/lab5$ ./benchmark.out <tests/05.t
Suffix tree search time: 40782us
Primitive search time: 18709us
aprold@SAI:~/Documents/GitHub/MAI-DA/lab5$ ./benchmark.out <tests/06.t
Suffix tree search time: 462362us
Primitive search time: 1252771us
aprold@SAI:~/Documents/GitHub/MAI-DA/lab5$ ./benchmark.out <tests/07.t
Suffix tree search time: 3895521us
Primitive search time: 59833446us
```

Алгоритм линеаризации циклической строки с помощью суффиксного дерева проигрывает только на первом тесте, где длина строки равна 10000 символов. В остальных же тестах этот алгоритм оказался эффективней. Примитивный алгоритм работает за $O(n^3)$ времени, т.к. на каждом из n шагов создаётся новая строка за $O(n)$ и сравнивается с минимумом за $O(n)$ в худшем случае. Алгоритм линеаризации с помощью суффиксного дерева проиграл на первом тесте вероятно из-за большой константы.

5 Выводы

Выполнив пятую лабораторную работу по курсу «Дискретный анализ», разобрался в алгоритме Укконена для построения суффиксного дерева, а также изучил алгоритм линеаризации циклической строки с помощью суффиксного дерева.
Алгоритм линеаризации циклической строки с помощью суффиксного дерева (алгоритм Укконена) имеет линейную сложность и эффективен на строках большой длины.

Список литературы

[1] Гасфилд Дэн
Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология / Пер. с англ. И.В.Романовского. — СПб.: Невский Диалект; БХВ-Петербург (ISBN 0-521-58519-8 (англ.); ISBN 5-7940-0103-8 ("Невский Диалект"); ISBN 5-94157-321-9 (БХВ-Петербург))

[2] *Реализации алгоритмов/Алгоритм Укконена*
URL: `https://ru.wikibooks.org/wiki/Реализации_алгоритмов/Алгоритм_Укконена` (дата обращения: 23.09.2023).