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. А. Кабанов
Преподаватель: С. А. Михайлова
Группа: М8О-301Б-22

Дата:
Оценка:
Подпись:

Москва, 2025

Лабораторная работа №5

Задача: Поиск в известном тексте неизвестных заранее образцов.

Найти в заранее известном тексте поступающие на вход образцы.

Формат ввода: Текст располагается на первой строке, затем, до конца файла, следуют строки с образцами.

Формат вывода: Для каждого образца, найденного в тексте, нужно распечатать строчку, начинающуюся с последовательного номера этого образца и двоеточия, за которым, через запятую, нужно перечислить номера позиций, где встречается образец в порядке возрастания.

1 Описание

Требуется реализовать алгоритм Укконена построения суффиксного дерева за линейное время.

Алгоритм Укконена в самой простоей реализации имеет сложность $O(n^3)$, так как добавляет каждый суффикс каждого префикса строки в отличии от добавления всех суффиксов строки. Основная идея в том, чтобы оптимизировать его и получить сложность $O(n)$.

Заметим, что проще будет продлевать суффиксы на один символ. В некоторых случаях при продлении можно идти не по всем символам, а сразу по ребру дерева, что значительно уменьшает число шагов.

Пусть в суффиксном дереве есть строка $x\alpha$ (x - первый символ строки, α - оставшаяся строка), тогда α тоже будет в суффиксном дереве, потому что α является суффиксом $x\alpha$. Если для строки $x\alpha$ существует некоторая вершина u, то существует и вершина u для α. Ссылка из u в v называется суффиксной ссылкой.

Суффиксные ссылки позволяют не проходить каждый раз по дереву из корня. Для построения суффиксных ссылок достаточно хранить номер последней созданной вершины при продлении. Если на этой же фазе мы создаём ещё одну новую вершину, то нужно построить суффиксную ссылку из предыдущей в текущую.

Сложность алгоритма с использованием продления суффиксов и суффиксных ссылок $O(n^2)$.

Для ускорения до $O(n)$ нужно уменьшить объём потребляемой памяти. Будем в каждом ребре дерева хранить не подстроку, а только индекс начала и конца подстроки.

У исходной строки n суффиксов и будет создано не более n внутренних вершин, в среднем продление суффиксов работает за $O(1)$.

При использовании всех вышеописанных эвристик получим временную и пространственную сложность $O(n)$.

Для нахождения минимального лексикографического разреза строки s построим суффиксное дерево от удвоенной строки и найдём лексикографически минимальный путь длины $|s|$ в дереве. Сложность $O(n)$.

2 Исходный код

Реализуем суффиксное дерево и алгоритм, описанные в предыдущем пункте. Были произведены некоторые изменения в коде, связанные с реализацией на C++: к примеру, вместо хранения индексов строки хранятся итераторы.

```
#include <bits/stdc++.h>

struct Node;
struct STree;

struct Node {
  std::map<char, Node *> to;
  std::string::iterator begin;
  std::string::iterator end;
  Node *suffixLink;

  Node(std::string::iterator begin, std::string::iterator end)
  : begin(begin), end(end), suffixLink(nullptr) {}

  ~Node() {};
};

struct STree {
  std::string text;
  Node *root;
  Node *activeNode;
  Node *lastAdded;
  std::string::iterator activeEdge;
  int32_t activeLength;
  int32_t remainder;

  STree() = default;
  STree(std::string &str)
  : text(str), activeLength(0), activeEdge(text.begin()), remainder(0) {
    root = new Node(text.end(), text.end());
    lastAdded = root;
    activeNode = root;
    root->suffixLink = root;

    for (std::string::iterator it = text.begin(); it != text.end(); ++it) {
      addLetter(it);
    }
  }

  ~STree() { destroy(root); }

  void addLetter(std::string::iterator i) {
    lastAdded = root;
```

```
    ++remainder;
    while (remainder > 0) {
      activeEdge = (activeLength == 0) ? i : activeEdge;
      std::map<char, Node *>::iterator it =
      activeNode->to.find(*activeEdge);
      Node *next;
      if (it == activeNode->to.end()) {
        Node *leaf = new Node(i, text.end());
        activeNode->to[*activeEdge] = leaf;
        lastAdded->suffixLink =
        (lastAdded != root) ? activeNode : lastAdded->suffixLink;
        lastAdded = activeNode;
      } else {
        next = it->second;
        if (checkEdge(i, next)) {
          continue;
        }
        if (*(next->begin + activeLength) == *i) {
          ++activeLength;
          lastAdded->suffixLink =
          (lastAdded != root) ? activeNode : lastAdded->suffixLink;
          lastAdded = activeNode;
          break;
        }
        Node *split = new Node(next->begin, next->begin + activeLength);
        Node *leaf = new Node(i, text.end());
        activeNode->to[*activeEdge] = split;
        split->to[*i] = leaf;
        next->begin += activeLength;
        split->to[*next->begin] = next;
        lastAdded->suffixLink =
        (lastAdded != root) ? split : lastAdded->suffixLink;
        lastAdded = split;
      }
      --remainder;
      if (activeNode == root && activeLength > 0) {
        --activeLength;
        activeEdge = i - remainder + 1;
      } else {
        activeNode = activeNode->suffixLink;
      }
    }
  }

  void searchLeaves(Node *node, std::vector<int32_t> &answer, int32_t loc) {
    if (node->end == text.end()) {
      answer.push_back(text.size() - loc + 1);
    } else {
      Node *child;
```

```
93      for (auto it = node->to.begin(); it != node->to.end(); ++it) {
94        child = it->second;
95        searchLeaves(child, answer, loc + child->end - child->begin);
96      }
97    }
98  }
99
100 std::vector<int32_t> search(std::string &str) {
101   std::vector<int32_t> answer;
102   int32_t loc = 0;
103   Node *current = root;
104   if (str.length() > text.length()) {
105     return answer;
106   }
107   for (std::string::iterator pos = str.begin(); pos != str.end(); ++pos) {
108     auto path = current->to.find(*pos);
109     if (path == current->to.end()) {
110       return answer;
111     }
112     current = path->second;
113     loc += current->end - current->begin;
114     for (std::string::iterator p = current->begin;
115     p != current->end && pos != str.end(); ++p, pos++) {
116       if (*p != *pos) {
117         return answer;
118       }
119     }
120     if (pos == str.end()) {
121       break;
122     }
123     --pos;
124   }
125   searchLeaves(current, answer, loc);
126   sort(answer.begin(), answer.end());
127   return answer;
128 }
129
130 void destroy(Node *node) {
131   for (auto it = node->to.begin(); it != node->to.end(); ++it) {
132     destroy(it->second);
133   }
134   delete node;
135 }
136
137 bool checkEdge(std::string::iterator pos, Node *node) {
138   int32_t edgeLength = (pos + 1 < node->end) ? (pos + 1 - node->begin)
139   : (node->end - node->begin);
140   if (activeLength >= edgeLength) {
141     activeEdge += edgeLength;
```

```
      activeLength -= edgeLength;
      activeNode = node;
      return true;
    }
    return false;
  }
};

int main(int argc, char *argv[]) {
  std::ios_base::sync_with_stdio(false);
  std::cin.tie(nullptr);

  std::string text;
  std::cin >> text;
  text += "$";

  STree tree(text);
  int32_t k{0};
  std::vector<int32_t> answer;
  std::string pattern;
  while (std::cin >> pattern) {
    if (pattern.empty()) {
      break;
    }
    ++k;
    answer = tree.search(pattern);
    if (!answer.empty()) {
      std::cout << k << ": ";
      for (std::size_t i = 0; i < answer.size(); ++i) {
        std::cout << answer[i] << ((i == answer.size() - 1) ? "\n" : ", ");
      }
    }
  }
  return 0;
}
```

3 Консоль

```
[anton@home lab5-suffix-tree]$ make
g++ -O2 -lm -fno-stack-limit -std=c++20 -x c++ solution.cpp -o executable
[anton@home lab5-suffix-tree]$ ./executable
abcdabc
abcd
bcd
bc
1: 1
2: 2
3: 2,6
```

4 Тест производительности

Тест производительности представляет из себя следующее: сравнение написанного мной алгоритма за $O(n)$ и наивного алгоритма за $O(n^3)$.

```
[anton@home lab7-dp]$ make
g++ main.cpp -o main
g++ naive.cpp -o naive

...

./main <tests/1.in
Suffix tree time 0.178 ms
./naive <tests/1.in
Naive time 0.116 ms
./main <tests/2.in
Suffix tree time 1.680 ms
./naive <tests/2.in
Naive time 3.180 ms
./main <tests/3.in
Suffix tree time 17.795 ms
./naive <tests/3.in
Naive time 81.510 ms
./main <tests/4.in
Suffix tree time 18.455 ms
./naive <tests/4.in
Naive time 1334.365 ms
```

Видно, что построение суффиксного дерева по алгоритму Укконена и дальнейший поиск по нему выигрывают у наивного алгоритма.

5 Выводы

Выполнив седьмую лабораторную работу по курсу «Дискретный анализ», я вспомнил структуры данных, связанные со строками, и реализовал алгоритм Укконена, ознакомился с приложениями суффиксного дерева.

Суффиксное дерево позволяет быстро искать множество шаблонов в тексте, чего не могут другие алгоритмы. Но в повседневных задачах чаще требуется найти один шаблон в тексте, где лучше использовать более простые алгоритмы.

Список литературы

[1] *Алгоритм Укконена - Викиконспекты*
URL: https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Укконена (дата обращения: 10.12.2024).

[2] *Visualization of Ukkonen's Algorithm*
URL: http://brenden.github.io/ukkonen-animation/ (дата обращения: 10.12.2024).