

**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: Бачурин П.Д.
Преподаватель: Макаров
Н.К.
Даты:
Оценка:
Подпись:

Москва, 2024

Задача

Реализовать поиск подстрок в тексте с использованием суффиксного дерева. Суффиксное дерево можно построить за $O(n^2)$ наивным методом.

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.

Алгоритм

1. Инициализация данных:
 - Считываем текст, а потом считываем все паттерны.
2. Создание суффиксного дерева:
 - Добавляем к тексту символ \$ и запускаем процесс создания суффиксного дерева.
 - Создаем корень, а далее вставляем в дерево все суффиксы.
 - Процесс вставки суффиксов происходит итеративно. Мы пытаемся пройти по нашему дереву как можно дальше, а потом вставляем оставшуюся часть суффикса.
3. Поиск образцов
 - Для каждого из образцов мы запускаем поиск и ищем все вхождения.
 - Процесс поиска похож на процесс вставки образца. Изначально текущая вершина — корень. Мы смотрим, есть ли из текущей вершины ребро, которое начинается на нашу текущую первую букву. Если есть то мы пытаемся перейти по этому ребру, сравнивая все символы из образца и на ребре.
 - Если образец кончился, а ребро еще нет, то мы заканчиваем поиск. Если образец еще не кончился, то переходим по этому ребру, обновляя левую границу поиска, и запускаем новую итерацию уже из текущего узла.
 - После того, как мы закончили поиск, нам надо найти все вхождения.
 - Процесс поиска вхождений состоит в рекурсивном запуске от всех своих детей, и если дошли до листа, то добавляем в результат номер суффикса.
4. Вывод результата:
 - Для каждого образца выводим его номер и отсортированные позиции, с которых начинается вхождение.

Исходный код

```
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <cstdint>
#include <unordered_map>

class simple_suffix_tree {
private:
    struct vertex {
        int64_t left, right;
        int64_t id;
        std::unordered_map<char, int64_t> edges;

        vertex(): left(-1), right(-1), id(-1) {}
        vertex(const int64_t left, const int64_t right): left(left),
            right(right), id(-1) {}
        vertex(const int64_t left, const int64_t right, const int64_t id):
            left(left), right(right), id(id) {}
    };

    std::string text;
    std::vector<vertex> vertices;

public:
    explicit simple_suffix_tree(const
        std::string &text): text(text)
    { vertices.emplace_back();
      for (int64_t i = 0; i < text.length(); ++i) {
          add_suffix(i, static_cast<int64_t>(text.length() - 1));
      }
    }

private:
    void add_suffix(int64_t left, int64_t right) {
        int64_t curr_vertex = 0;
        int64_t old_left = left;

        while (left <= right) {
            if (curr_vertex == 0) {
                if (vertices[curr_vertex].edges.find(text[left]) !=
                    vertices[curr_vertex].edges.end()) { curr_vertex =
                    vertices[curr_vertex].edges[text[left]];
                }
            }
            if (curr_vertex == 0) {
                vertices.emplace_back(left, right, old_left);
                vertices[curr_vertex].edges[text[left]] = vertices.size() - 1;
                break;
            }
        }
    }
};
```

}
}

```

int64_t start = vertices[curr_vertex].left;
int64_t finish = vertices[curr_vertex].right;
bool cut = false;
for (int64_t i = start; (i <= finish) && (left +
    i - start <= right); ++i) { if (text[i] !=
    text[left + i - start]) {
        vertices[curr_vertex].right = i - 1;
        int64_t old_enter = vertices[curr_vertex].id;
        vertices[curr_vertex].id = -1;
        if (text[finish] == '$') {
            vertices.emplace_back(i, finish, old_enter);
        } else {
            vertices.emplace_back(i, finish);
        }
        vertices[vertices.size() - 1].edges =
            vertices[curr_vertex].edges;
        vertices[curr_vertex].edges.clear();
        vertices[curr_vertex].edges[text[i]] = vertices.size() - 1;
        vertices.emplace_back(left + i - start, right, old_left);
        vertices[curr_vertex].edges[text[left + i - start]] =
            vertices.size() - 1;
        cut = true;
        break;
    }
}
if (cut == false) {
    int64_t new_l = left + finish - start + 1;
    if (vertices[curr_vertex].edges.find(text[new_l]) !=
        vertices[curr_vertex].edges.end()) { curr_vertex =
        vertices[curr_vertex].edges[text[new_l]];
    } else {
        vertices.emplace_back(new_l, right, old_left);
        vertices[curr_vertex].edges[text[new_l]] =
            vertices.size() - 1; break;
    }
    left = new_l;
} else
{ br
eak;
}
}
}

void all_entries(std::vector<int64_t> &res, const
int64_t curr_vertex) { if
(vertices[curr_vertex].edges.empty()) {
    res.push_back(vertices[curr_vertex].id);
    return;
}
}

```

```
    for (const auto &[_ , vertex_id] :  
        vertices[curr_vertex].edges)  
        { all_entries(res, vertex_id);  
    }  
}
```

public:

```

std::vector<int64_t> search(const
    std::string &pattern)
{ std::vector<int64_t> res;
  int64_t curr_vertex = 0, left = 0, right =
  static_cast<int64_t>(pattern.length()-1);
  bool flag = false;
  while (left <= right) {
    if (curr_vertex == 0) {
      if (vertices[curr_vertex].edges.find(pattern[left]) !=
          vertices[curr_vertex].edges.end()) { curr_vertex =
          vertices[curr_vertex].edges[pattern[left]];
      }
      else {
        break;
      }
    }
    int64_t start = vertices[curr_vertex].left;
    int64_t finish = vertices[curr_vertex].right;
    for (int64_t i = 0; (start + i <= finish) && (i
        + left <= right); ++i) { if (pattern[i +
        left] != text[start + i]) {
        flag = true;
        break;
      }
    }
    if (flag == false) {
      left = left + finish - start + 1;
      if (left > right) {
        break;
      }
      if (vertices[curr_vertex].edges.find(pattern[left]) !=
          vertices[curr_vertex].edges.end()) { curr_vertex =
          vertices[curr_vertex].edges[pattern[left]];
      } else {
        break;
      }
    } else
    { br
      eak;
    }
  }
  if ((left > right) && (flag == false) && (!
      pattern.empty())) { all_entries(res,
      curr_vertex);
  }
  return res;
}
};

```



```
std::ostream& operator<<(std::ostream& os, const
    std::vector<int64_t>& v) { for (int64_t idx = 0; idx
    < v.size(); ++idx) {
    os << v[idx] + 1;
    if (idx != v.size() - 1) {
        os << ", ";
    }
}
```

```

    return os;
}

int main() {
    std::string text;
    std::getline(std::cin, text);

    std::vector<std::string> patterns;
    std::string read_pattern;
    while (std::getline(std::cin, read_pattern)) {
        patterns.push_back(read_pattern);
    }

    simple_suffix_tree st(text + "$");
    for (int64_t idx = 0; idx <
        patterns.size(); ++idx) { auto
        entries =
        st.search(patterns[idx]);
        if (entries.empty()) {
            continue;
        }

        std::sort(entries.begin(), entries.end());

        std::cout << idx+1 << ": " << entries << "\n";
    }
}

```

Тесты

1.

ВВОД:
abcdabc
abcd
bcd
bc

ВЫВОД:
1: 1
2: 2
3: 2, 6

2.

ВВОД:
hello world
hello
world
hi

ВЫВОД:
1: 1
2: 7

3.

ВВОД:
baobabbaobab
ba
oba
bab

ВЫВОД:
1: 1, 4,
7, 10
2: 3, 9
3: 4, 10

4.

ВВОД:
baobabbaobabbaobabbaobab
ba
bab
oba
obab
obabba

ВЫВОД:
1: 1, 4, 7, 10, 13, 16, 19, 22
2: 4, 10, 16, 22
3: 3, 9, 15, 21
4: 3, 9, 15, 21
5: 3, 9, 15

Вывод

В ходе выполнения данной работы я вспомнил суффиксные деревья и реализовал наивный алгоритм реализации суффиксного дерева, который работает за $O(n^2)$.

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: алгоритм Кнута-Морриса-Пратта, Бойера-Мура, Рабина-Карпа.