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. Терво
Преподаватель:	А. А. Кухтичев
Группа:	М8О-207Б
Дата:	
Оценка:	
Подпись:	

Москва, 2022

Лабораторная работа №5

Задача: Необходимо реализовать алгоритм Укконена построения суффиксного дерева за линейное время. Построив такое дерево для некоторых из выходных строк, необходимо воспользоваться полученным суффиксным деревом для решения своего варианта задания

Алфавит строк: строчные буквы латинского алфавита (т.е. от a до z).

Вариант задания: найти самую длинную общую подстроку двух строк.

1 Описание

Требуется реализовать алгоритм Укконена построения суффиксного дерева за линейное время.
В простейшей реализации метод построения суффиксного дерева работает за $O(n^3)$. В нём каждый суффикс продлевается независимо от остальных. Алгоритм Укконена является усовершенствованием данного метода, и работает за линейное время — $O(n)$.
В работе алгоритм использует суффиксные ссылки. Говоря простым языком, суффиксная ссылка — это ссылка из вершины с путевой меткой $x\alpha$ в вершину с меткой α, то есть в вершину с такой же меткой, но без первого символа. Более корректное определение приведено в [1]. Суффиксные ссылки нужны для того, чтобы быстро перейти в следующий суффикс и выполнить в нём необходимые операции.
Для ускорения продления суффиксов и уменьшения расхода памяти в каждом узле хранятся индекс начала и конца подстроки, причём конец подстроки хранится в виде указателя на переменную, в которой он записан. Это сделано для того, чтобы не продлевать каждый суффикс в отдельности, а добавлять по одному символу сразу ко всем.
Для нахождения максимальной общей подстроки двух строк A и B построим дерево для строки $A\$B*$, где \$ и * — знаки, не входящие в алфавиты обеих строк. Затем найдём самую глубокую вершину, в одном поддереве которой заканчивается строка A, а в другом — строка B. Путевая метка этой вершины и будет максимальной общей подстрокой.

2 Исходный код

Опишем класс дерева и вспомогательную структуру TEdge для хранения информации о ребре. Нам понадобятся конструктор от строки и методы нахождения наиболее длинной общей подстроки.

```
class TSuffixTree {
  private:
    struct TEdge {
      int Left;
      std::shared_ptr<int> Right;
      int IdTo;

      TEdge(int l, std::shared_ptr<int> r, int id) : Left(l), Right(r), IdTo(id) {}
    };

    std::string DataString;
    std::vector< std::vector< TSuffixTree::TEdge > > Data;
    std::vector<int> SuffixPtr;
    std::vector<int> PathSize;
  public:
    TSuffixTree(const std::string & s);
    void Print();
    void Find(const char, const char);
    int RecMarker(const char, const char, const int, const int, int &, std::vector<
        int> &);
    void RecCatchStrings(const int, const int, const int, const std::string, std::
        vector<int> &, std::vector<std::string> &);
  };
}
```

В таком случае main.cpp получается достаточно небольшой.

```
#include "suffix_tree.hpp"

const char SENTINEL1 = '$';
const char SENTINEL2 = '*';

int main() {
  std::string s;
  std::cin >> s;
  std::string t = s + SENTINEL1;
  std::cin >> s;
  t = t + s + SENTINEL2;
  TSuffixTree st(t);
  st.Find(SENTINEL1, SENTINEL2);
  return 0;
}
```

Самым сложным является реализация конструктора дерева от строки, являющаяся собственно реализацией алгоритма Укконена. Дерево имеет максимум $2 * n$ вершин и $2 * n - 1$ рёбер, поэтому его удобно хранить списками смежности.

```
TSuffixTree::TSuffixTree(const std::string & s) : DataString(s), Data(s.size() * 2),
    SuffixPtr(s.size() * 2), PathSize(s.size() * 2) {
  bool newVertex = false;
  std::shared_ptr<int> end(new int);
  *end = 0;
  int curVertex = 0;
  int l = 0;
  int vertexId = 0;
  int curEq = 0;
  int passEq = 0;
  int curEdgeId = -1;
  for (size_t i = 0; i < s.size(); ++i) {
    ++*end;
    int lastCreatedVertexId = 0;
    int nextCreatedVertexId = -1;
    while (l < *end) {
      if (curEdgeId == -1) {
        int nextEdgeId = -1;
        for (size_t j = 0; j < Data[curVertex].size(); ++j) {
          if (DataString[Data[curVertex][j].Left] == DataString[l + passEq + curEq])
              {
            nextEdgeId = j;
            curEq = 1;
            break;
          }
        }
        if (nextEdgeId == -1) {
          newVertex = true;
        }
        curEdgeId = nextEdgeId;
      } else {
        if (DataString[Data[curVertex][curEdgeId].Left + curEq] == DataString[l +
            curEq + passEq]) {
          ++curEq;
        } else {
          newVertex = true;
        }
      }
      if (curEq > 0 and Data[curVertex][curEdgeId].Left + curEq == *Data[curVertex][
          curEdgeId].Right) {
        curVertex = Data[curVertex][curEdgeId].IdTo;
        curEdgeId = -1;
        passEq = passEq + curEq;
        curEq = 0;
      }
      if (newVertex) {
        ++vertexId;
```

```
      if (curEq == 0 and curEdgeId == -1) {
        Data[curVertex].push_back(TSuffixTree::TEdge(l + passEq, end, vertexId));
      } else {
        nextCreatedVertexId = vertexId;
        TEdge curEdge = Data[curVertex][curEdgeId];
        std::shared_ptr<int> newRightBorder(new int);
        *newRightBorder = Data[curVertex][curEdgeId].Left + curEq;
        Data[curVertex][curEdgeId].Right = newRightBorder;
        Data[curVertex][curEdgeId].IdTo = vertexId;
        curEdge.Left = *newRightBorder;

        Data[vertexId].push_back(curEdge);
        ++vertexId;
        Data[vertexId - 1].push_back(TSuffixTree::TEdge(l + curEq + passEq, end, vertexId));
        PathSize[vertexId - 1] = PathSize[curVertex] + *Data[curVertex][curEdgeId].Right - Data[curVertex][curEdgeId].Left;

        if (lastCreatedVertexId > 0) {
          SuffixPtr[lastCreatedVertexId] = nextCreatedVertexId;
        }
        lastCreatedVertexId = nextCreatedVertexId;
      }
      int nextCurVertex = SuffixPtr[curVertex];
      int nextPassEq = PathSize[nextCurVertex];
      int nextCurEq = curEq + passEq - nextPassEq - 1;
      int nextEdgeId = -1;
      while (nextCurEq > 0) {
        for (size_t j = 0; j < Data[nextCurVertex].size(); ++j) {
          if (DataString[Data[nextCurVertex][j].Left] == DataString[l + 1 + nextPassEq]) {
            nextEdgeId = j;
            break;
          }
        }
        int curRight = *Data[nextCurVertex][nextEdgeId].Right;
        int curLeft = Data[nextCurVertex][nextEdgeId].Left;
        if (nextEdgeId != -1 and curRight - curLeft <= nextCurEq) {
          nextPassEq = nextPassEq + curRight - curLeft;
          nextCurEq = nextCurEq - curRight + curLeft;
          nextCurVertex = Data[nextCurVertex][nextEdgeId].IdTo;
          nextEdgeId = -1;
        } else {
          break;
        }
      }
      if (nextEdgeId != -1) {
        curEq = nextCurEq;
      } else {
```

```
          curEq = 0;
        }
        curEdgeId = nextEdgeId;
        curVertex = nextCurVertex;
        passEq = nextPassEq;
        ++l;
        newVertex = false;
      } else {
        if (i < s.size() - 1) {
          break;
        }
      }
    }
  }
}
```

3 Консоль

```
[alext@alext-pc solution]$ make
g++ -std=c++17 -pedantic -g -Wall -Wextra -Wno-unused-variable main.cpp -o
solution
[alext@alext-pc solution]$ ./solution
abcabcabc
abcax
4
abca
[alext@alext-pc solution]$ ./solution
abracadabra
xyzabredabuey
3
abr
dab
[alext@alext-pc solution]$ ./solution
aaaaabaabaaa
aa
2
aa
[alext@alext-pc solution]$ ./solution
abcdefg
abcdefg
7
abcdefg
```

4 Тест производительности

Сравним наивный алгоритм поиска наибольшей общей подстроки с поиском подстроки с помощью суффиксного дерева.

```
[alext@alext-pc solution]$ ./naive <tests/test10.txt
Naive algorithm time: 2us
[alext@alext-pc solution]$ ./naive <tests/test100.txt
Naive algorithm time: 82us
[alext@alext-pc solution]$ ./naive <tests/test1k.txt
Naive algorithm time: 6346us
[alext@alext-pc solution]$ ./naive <tests/test10k.txt
Naive algorithm time: 635613us
[alext@alext-pc solution]$ ./solution <tests/test10.txt
Ukkonen algorithm time: 33us
[alext@alext-pc solution]$ ./solution <tests/test100.txt
Ukkonen algorithm time: 46us
[alext@alext-pc solution]$ ./solution <tests/test1k.txt
Ukkonen algorithm time: 303us
[alext@alext-pc solution]$ ./solution <tests/test10k.txt
Ukkonen algorithm time: 2772us
```

Видно, что на маленьких строках наивный алгоритм выигрывает за счёт простоты построения небольшой матрицы и вычисления на ней. Но уже на строках длиной 100 видно, что он начинает проигрывать по времени алгоритму с использованием суффиксного дерева.
Также видно, что сложность наивного алгоритма больше, чем у алгоритма с суффиксным деревом: при увеличении размера входных данных на порядок время наивного алгоритма растёт на два порядка, а время суффиксного — только на один.

5 Выводы

Во время выполнения этой лабораторной работы я вспомнил определение суффиксного дерева. Изучил наивный алгоритм его построения и алгоритм Укконена.
Этот алгоритм работает достаточно быстро и эффективен для поиска множества шаблонов в строке. Но он имеет и минусы, в числе которых его сложность для понимания.
Для поиска одного шаблона в строке удобнее использовать другие алгоритмы, такие как КМП, алгоритм Бойера-Мура или Рабина-Карпа. Для решения же моей задачи — поиска наибольшей общей подстроки проще использовать метод динамического программирования.

Список литературы

[1] *Алгоритм Укконена — Викиконспекты*
URL: https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Укконена (дата обращения: 16.12.2021).