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: М. А. Волков
Преподаватель: А. А. Кухтичев
Группа: М8О-207Б-19
Дата: 6 апреля 2021 г.
Оценка:
Подпись:

Москва, 2021

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 суффисным деревом для решения своего варианта задания.

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

Вариант: Найти самую длинную общую подстроку двух строк.

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 $O(n^2)$. Подробное доказательство изложено в [1].

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

Для того, чтобы написать дерево, использовался класс под названием *TBoris*. Главные методы в этом классе тут *Build*, *SetText* и *Colorize*, которые строят суфф дерево, вставляется текст для анализа и поиск ответа соответственно. Метод *Colorize* был так назван, потому что я, опираясь на свой же придуманный метод поиска ответа, красил вершины цветом: красный цвет – принадлежность вершины к 1 слову, синий – ко 2-ому слову.

```
#ifndef SUFF_TREE_LAB5_MAIN_H
#define SUFF_TREE_LAB5_MAIN_H
const char FANTOM_CHAR1 = '%';
const char FANTOM_CHAR2 = '$';

class TBoris {
  //---variables
public:
  std::shared_ptr<int> end;
private:
  std::string texts;
  int globID = -1;
  int one2two = -1;


  //-----------------------simply nodes--------------------
  struct TBorNode {...};

protected:
  TBorNode* root;

//----functions
public:
  TBoris() {
    end = std::make_shared<int>(0);
    root = new TBorNode (root, globID, -1, end);
    root->url = root;
  };

  ~TBoris() {
    delete root;
  };

  void SetText(const std::string &str1, const std::string &str2) {
    texts = str1 + FANTOM_CHAR1 + str2 + FANTOM_CHAR2;
    one2two = str1.size();
  }

//--------------------------------ITERATOR------------------------------
```

```
40  class TIterator {...};
41
42  void Build() {
43    int activeLen = 0;
44    int edge = 0;
45    TIterator Node(root);
46    bool splitFlag = true;
47
48    for (*end = 0; *end < texts.size(); ++*end) {
49      Node.SetPrevNull();
50      while (globID <= *end) {
51        TBorNode* checkNode = (Node.Next(texts[edge]));
52        while (checkNode != nullptr && activeLen > *checkNode->last - checkNode->begin)
              {
53          activeLen = activeLen - 1 - (*checkNode->last - checkNode->begin);
54          Node = checkNode;
55          edge = *end - activeLen;
56          checkNode = Node.Next(texts[edge]);
57        }
58
59        if (Node.IsPrevNotNull() && activeLen == 0) {
60          Node.GetPrevNode()->url = Node.GetActiveNode();
61          Node.SetPrevNull();
62        }
63
64        if (activeLen <= 0 && !Node.FindPath(texts[*end])) {
65          splitFlag = true;
66
67          Node.Insert(std::pair<char, TBorNode*>(texts[*end], new TBorNode(root, globID
                , *end, end)));
68          if (Node.IsPrevNotNull()) {
69            Node.GetPrevNode()->url = Node.GetActiveNode();
70            Node.SetPrevNull();
71          }
72          Node.GoThrowURL();
73          Node.SetPrevNull();
74          edge = 0;
75        }
76
77
78        else if (Node.Next(texts[edge]) != nullptr &&
79                 activeLen > 0 &&
80                 texts[*end] != texts[Node.Next(texts[edge])->begin + activeLen]) {
81          splitFlag = true;
82          {
83            TIterator tmp(Node.Next(texts[edge]));
84            tmp.Split(root, globID, texts, end, *end, Node.Next(texts[edge])->begin +
                  activeLen, Node.GetActiveNode());
85          }
```

```
        {
          TIterator tmp(Node.Next(texts[edge]));
          if (Node.IsPrevNotNull()) {
            Node.GetPrevNode()->url = tmp.GetActiveNode();
            Node.SetPrevNull();
          }

          if (Node.GetActiveNode() == root) {
            Node.SetPrev(tmp.GetActiveNode());
            ++edge;
            --activeLen;
          }
          else {
            Node.GoThrowURL();
            Node.SetPrev(tmp.GetActiveNode());
          }
        }
      }

      else {
        if (splitFlag) {
          splitFlag = false;
          edge = *end;
        }
        ++activeLen;
        break;
      }
    }
  }
  --*end;
 }

private:
 void ColorizeHelp(TIterator &node, std::set<std::string> &ans, int &ansCount, int
     len) {
  len = len + (*node.GetActiveNode()->last - node.GetActiveNode()->begin) + 1;
  if (node.GetActiveNode()->leaf) {
    if (node.GetActiveNode()->begin <= one2two) {
      node.ColorRed();
      return;
    }
    node.ColorBlue();
    return;
  }

  for (const auto &item : node.GetActiveNode()->next) {
    TIterator nextNode(item.second);
    ColorizeHelp(nextNode, ans, ansCount, len);

```

```
134       if (nextNode.GetBlue()) {
135         node.ColorBlue();
136       }
137       if (nextNode.GetRed()) {
138         node.ColorRed();
139       }
140     }
141 
142     if (node.BothColored()) {
143       if (len > ansCount) {
144         ans.clear();
145         ansCount = len;
146         ans.insert(node.GetString(texts, *node.GetActiveNode()->last - len + 1, *node.
                GetActiveNode()->last));
147       }
148       else if (len == ansCount) {
149         ans.insert(node.GetString(texts, *node.GetActiveNode()->last - len + 1, *node.
                GetActiveNode()->last));
150       }
151     }
152   }
153 
154 public:
155   void Colorize() {
156     std::set<std::string> ans;
157     int ansCount = 0;
158     TIterator node(root);
159     for (const auto &item : node.GetActiveNode()->next) {
160       TIterator nextNode(item.second);
161       ColorizeHelp(nextNode, ans, ansCount, 0);
162     }
163 
164     std::cout << ansCount << std::endl;
165     if (!ans.empty()) {
166       for (const auto &item : ans) {
167         std::cout << item << std::endl;
168       }
169     }
170   }
171 
172 
173 };
174 #endif //SUFF_TREE_LAB5_MAIN_H
```

Для удобства передвижения по суффиксному дереву, мной был придуман итератор с нужными для меремещения методами такими, как *Insert* и *Split*. Данная структура данных мне помогла локально изменять свой код, при этом изменяя глобальную логику моей работы. Например я обнаружил ошибку в разделении вершины. Мне нужно было поменять метод *Split* и все остальное работает хорошо.

```
class TIterator {
private:
  TBorNode* prevNode;
  TBorNode* activeNode;

public:
  TIterator() {
    activeNode = nullptr;
    prevNode = nullptr;
  }

  explicit TIterator(TBorNode* node) {
    activeNode = node;
    prevNode = nullptr;
  }

  ~TIterator() {
    activeNode = nullptr;
    prevNode = nullptr;
  };

  TBorNode* Next(const char &c) {
    if (activeNode == nullptr) {
      return nullptr;
    }

    auto tmp = activeNode->GetNodeElem(c);
    if (tmp != nullptr) {
      return tmp;
    }
    return nullptr;
  }

  bool FindPath(const char &c) {
    if (activeNode == nullptr) {
      return false;
    }

    auto tmp = activeNode->GetNodeElem(c);
    if (tmp != nullptr) {
      return true;
    }
    return false;
```

```
44      }
45
46      void GoThrowURL() {
47        if (activeNode->url == nullptr) {
48          throw std::logic_error("Trying to call nullptr in URL");
49        }
50        prevNode = activeNode;
51        activeNode = activeNode->url;
52      }
53
54      void Insert(const std::pair<char, TBorNode*> &tmp_pair) {
55        activeNode->Insert(tmp_pair);
56      }
57
58      void Split(TBorNode* _root, int &_globID, const std::string &_texts,
59                 const std::shared_ptr<int> &_end,
60                 const int &begin, const int &splitter, TBorNode* Node) {
61        Node->next.erase(_texts[activeNode->begin]);
62        Node->Insert(std::pair<char, TBorNode*>(_texts[activeNode->begin],
63                                                new TBorNode (_root, _globID,
64                                                              activeNode
                                                                  ->
                                                                  begin
                                                                  ,
65                                                              splitter
                                                                   -
                                                                  1)
                                                                  ))
                                                                  ;
66        TBorNode* tmp = Node->GetNodeElem(_texts[activeNode->begin]);
67        --_globID;
68
69        tmp->Insert(std::pair<char, TBorNode*>(_texts[begin],
70                                               new TBorNode(_root, _globID,
                                                       begin,
71                                                            _end)))
                                                           ;
72        activeNode->begin = splitter;
73        tmp->Insert(std::pair<char, TBorNode*>(_texts[activeNode->begin],
74                                               activeNode));
75      }
76    };
```

Также была сделана структурка небольшая, в которую я помещаю всю информацию о вершине.

```
struct TBorNode {
  //variables
  int begin;
  std::shared_ptr<int> last;
  bool leaf = true;
  TBorNode* url;
  std::unordered_map<char, TBorNode*> next;
  int id;
  bool red = false;
  bool blue = false;

  //constructors
  TBorNode(TBorNode* root, int &globID, const int &_begin,
           const std::shared_ptr<int> &end) {
    id = globID++;
    last = end;
    begin = _begin;
    url = root;
  }

  TBorNode(TBorNode* root, int &globID, const int &_begin, const int &end) {
    id = globID++;
    last.reset(new int(end));
    begin = _begin;
    url = root;
    leaf = false;
  }

  TBorNode() = delete;

  //destructors
  ~TBorNode(){
    url = nullptr;
    for (auto item : next){
      delete item.second;
    }
  };

  void PrintNode(const std::string &text) {
    for (int i = begin; i <= *last; ++i) {
      std::cout << text[i];
    }
  }

  void Insert(const std::pair<char, TBorNode*> &tmp_pair) {
    next.insert(tmp_pair);
    leaf = false;
```

```
48      }
49
50      TBorNode* GetNodeElem(const char &c) {
51        if (!next.empty()) {
52          if (next.find(c) == next.end()) {
53            return nullptr;
54          }
55          return next.find(c)->second;
56        }
57        return nullptr;
58      }
59
60    };
```

3 Тест производительности

Сравним наиный алгоритм, который сортирует все циклические разрезы строки, с алгоритмом, использующим суффиксное дерево.

Тесты состоят из строк длины 100, 1000, 10000.

```
test1.txt
Naive time 0.1 ms
My time 0.2 ms

test2.txt
Naive time 0.54 ms
My time 0.20 ms

test3.txt
Naive time 3.484 ms
My time 0.171 ms
```

Видно, что на маленькой строке константа в сложности алгоритма Укконена даёт о себе знать, а на больших строках квадратичная сложность не может соревноваться с линейной.

4 Выводы

Во время выподнения данной лабораторной работы, я, проверяя свою программу на похожем задании на сайте *codeforses*, понял, что алгоритм Укконена работает очень быстро. Данный алгоритм даже не занимает много места в оперативной памяти при непосредственном выполнении.

К сожалению написание данного алгоритма достаточно сложное и поэтому большенство программистов выбирают алгоритм КМП. Или же, решая мою задачу, используют динамический метод программирования.

Список литературы

[1] *Алгоритм Укконена — Викиконспекты*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Укконена` (дата обращения: 19.03.2021).

[2] *Visualization of Ukkonen's Algorithm*
URL: `http://brenden.github.io/ukkonen-animation/` (дата обращения: 18.03.2021).