Министерство образования и науки Российской федерации

Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

«Алтайский государственный технический университет им. И.И. Ползунова»

Факультет информационных технологий

Кафедра прикладной математики

Отчет защищен с оценкой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись)

«\_\_\_\_\_\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2015 г.

Отчет

по лабораторной работе № 4

Суффиксные деревья

(название лабораторной работы)

по дисциплине «Алгоритмы и структуры данных»

**ЛР 09.03.04.11.000 О**

Студент группы ПИ-32 Глушков Г.Г.

(И.О. Фамилия)

Преподаватель преподаватель Н.Д. Бубнова

(должность, ученое звание) (И.О. Фамилия)

Барнаул 2015

*Задание. Вариант 5.*

Формулировка задачи:

Нахождение максимальной повторяющейся подстроки

Для данной строки y, |y| = n>0, найти самую длинную подстроку, встречающуюся в y больше одного раза

1. Алгоритм решения:
2. Считывание строки с файла.
3. Построение дерева:

Алгоритм Укконена строит последовательность неявных суффиксных деревьев, последнее из которых преобразуется в настоящее суффиксное дерево строки S.

Неявное суффиксное дерево префикса S [1..i] строк и S получается аналогично из суффиксного дерева для S[1..i]$ удалением символов $, дуг и вершин, как описано выше.

Обозначим через T неявное суффиксное дерево стро­ки S[1..i], где i изменяется от 1 до m.

Хотя неявное суффиксное дерево может иметь листья не для всех суффиксов, в нем закодированы все суффиксы S – каждый символами какого-либо пути от корня этого неявного суффиксного дерева. Будем использовать неявные деревья указанного ти­па как вспомогательные в алгоритме Укконена для того, чтобы получить настоящее суффиксное дерево для S.

Алгоритм Укконена строит неявное суффиксное дерево Ti для каждого префик­са S[1..i] строки S, начиная с T1 и увеличивая i на единицу до тех пор, пока не будет построено дерево Tm.

Алгоритм Укконена делится на m фаз. В фазе i + 1 дерево Ti+1 строится из Ti. Каждая фаза i+1 в свою очередь делится на i+1 продолжений, по одному на каждый из i+1 суффиксов S[1..i+1]. В продолжении с номером j фазы i+1 алгоритм сначала находит конец пути из корня, помеченного подстрокой S[j..i]. Затем он продолжает эту подстроку, добавляя к ее концу символ S (i + 1) если толь ко S (i + 1) еще не существует. Т.е. в фазе i+1 сначала помещается в дерево строка S [1..i+1], а за ней строки S[2..i + 1], S[3..i + 1], ... (соответственно, в продолжениях 1, 2, 3, ...). Продолжение i +1 фазы i + 1 продолжает пустой суффикс строки S[1..i] — включает в дерево строку из одного символа S(i + 1) (если только ее там не было). Дерево T1 состоит из одной дуги, помеченной символов S(1).

Пусть S[j..i] = b — суффикс S[1..i]. В продолжении с номером j, когда алгоритм находит конец суффикса b в текущем дереве, алгоритм продолжает b для того, чтобы обеспечить присутствие суффикса bS(i+1) в дереве. Алгоритм действует по одному из следующих трех правил.

1) В текущем дереве путь b кончается в листе. Это значит, что путь от корня с меткой b доходит до конца некоторой «листовой» дуги (дуги, входящей в лист). При изменении дерева следует добавить к концу метки этой листовой дуги символ S(i+1)

2) Ни один путь из конца строки b не начинается символом S(i + 1), но, по крайней мере, один начинающийся оттуда путь имеется.

В этом случае должна быть создана новая листовая дуга, начинающаяся в конце b и помеченная символом S(i + 1). При этом, если b кончается внутри дуги, должна быть создана новая вершина. Листу в конце новой листовой дуги сопоставляется j

3) Некоторый путь из конца строки b начинается символом S (i + 1). В этом случае строка bS(i + 1) уже имеется в текущем дереве, и поэтому ничего не требуется делать.

Пусть ха обозначает произвольную строку, где х — ее первый символ, а а — оставшаяся подстрока (возможно, пустая). Ес­ли для внутренней вершины v с путевой меткой ха существует другая вершина s(v) с путевой меткой а, то указатель из v в s(v) называется суффиксной связью. Любая вновь созданная внутренняя вершина будет иметь суффиксную связь с концом следующего продолжения.

Таким образом, суффиксные связи будут исходить из всех внутренних вершин из­меняющегося дерева, кроме внутренней вершины, введенной последней. Она получит свою суффиксную связь в конце следующего продолжения.

Заметим, что конец полной строки S[1..i] должен быть в листе дерева Ti, так как S[1..i] — самая длинная строка в нем. Следовательно, конец этого суффикса най­ти легко. Представим строку S[1..i] в виде ха, где х — один символ, а а — оставшаяся подстрока (возможно, пустая), и пусть (v, 1) — дуга дерева, входящая в лист 1. Опи­шем подробно второе продолжение. Пусть y обозначает дуговую метку дуги (v, 1). Для того, чтобы найти конец а, требуется пройти вверх от листа 1 до вершины v, далее по суффиксной связи из v в s(v), а затем от s(v) — вниз по пути (который может состоять больше чем из одной дуги) с меткой y. Конец этого пути и является концом а. В конце пути а дерево изменяется по правилам продолжения суффикса.

При продолжении любой строки S[j..i] до S[j..i + 1] при j > 2 следуем той же общей идее. Начиная в конце строки S[j — 1..i] в текущем дереве, поднимаемся вверх не больше, чем на одну вершину, попадая либо в корень, либо в вершину v, из которой выходит суффиксная связь. Далее, пусть y — дуговая метка этой дуги; в случае, если v те корень, проходим суффиксную связь из v в s(v) и спускаемся по дереву из s(v) то пути, помеченному y, к концу S [j..i]. И наконец, продолжаем суффикс до S[j..i + 1] по правилам продолжения.

На шаге 2 продолжения j + 1 алгоритм идет вниз из вершины s(v) по пути с меткой y. При буквальной реализации прохождение вдоль y занимает время, про­порциональное |y|, числу символов в этом пути. Однако простой прием, именуемый скачком по счетчику, уменьшит время перехода до приблизительно пропорциональ­ного числу вершин в пути. Отсюда будет следовать, что время на все проходы вниз за фазу не превзойдет О(m).

Алгоритм отдельного продолжения:

1. Найти в конце строки S[j — 1..i] или выше его первую вершину v, которая либо имеет исходящую суффиксную связь, либо является корнем. Для этого необходимо пройти вверх от конца S[j — 1..i] в текущем дереве не более чем на одну дугу. Пусть y обозначает строку между v и концом S[j — 1..i] (возможно, пустую).
2. Если v те корень, пройти суффиксную связь из v в вершину s(v) и спуститься из s(v) те пути строки y. Если v — корень, пройти по пути для S[j..i] из корня (как в наивном алгоритме).
3. Обеспечить вхождение строки S [j..i] S(i +1) в дерево, используя правила про­должения.
4. Если в продолжении j — 1 была создана новая внутренняя вершина w (по правилу продолжения 2), то строка а должна кончаться в вершине s(w), конце суффиксной связи из w. Создать эту суффиксную связь (w, s(w)).

Пусть g обозначает длину y. Никакие две метки дуг, выходящих из s(u), те могут начинаться с одного и того же символа, так что первый символ y должен быть начальным только у одной дуги, выходящей из s(v). Пусть g' — чис­ло символов в этой дуге. Если g’<g, то алгоритму не требуется просматривать остальные символы дуги – он просто перепрыгивает к ее концу. Затем g полается равным g — g', h = g' + 1, и просматриваются выходящие дуги для того, чтобы най­ти среди них правильное продолжение (с начальным символом, равным символу h строки y)- По достижении дуги, в которой g те превосходит g', алгоритм переходит к символу g на дуге и завершается, поскольку путь y из s(v) кончается на этой дуге, строго в g символах вниз по метке.

Эффект от использования скачков по счетчику будет в обеспечении перехода от вершины пути y к другой вершине за константное время. Полное время прохода по пути становится при этом пропорциональным числу вершин, а не символов в нем.

При использовании скачков по счетчику любая фаза алгоритма Укконена занимает время О(m). Всего фаз m, так что суффиксные связи в алгоритме Укконена обеспечивают время работы О(m2).

1. Нахождение максимальной повторяющейся подстроки:

В суффиксном дереве у одного узла не может быть больше чем одной дуги, начинающейся одинакового символа, и поэтому если в тексте будут повторяющиеся подстроки, то они будут располагаться на одном и том же пути. Этот путь в суффиксном дереве пройдет через один или несколько внутренних узлов вниз по дереву (ниже точки, где подстрока заканчивается на том пути).

Мы можем заметить, что самая длинная повторная подстрока закончится во внутреннем узле, который является самым дальним от корня (т.е. самый глубокий узел в дереве), потому что длина подстроки - длина этикетки пути от корня до того внутреннего узла.

Так нахождение самой длинной повторной подстроки сводится к нахождению самого глубокого узла в дереве, и затем получение метки пути от корня до того самого глубокого внутреннего узла.

1. Вывод результатов работы.
2. Текст программы:

// Нахождение максимальной повторяющейся подстроки

#include <iostream>

#include <fstream>

#include <string>

#define MAX\_CHAR 256

using namespace std;

struct SuffixTreeNode {

struct SuffixTreeNode \*children[MAX\_CHAR];

//указатель на другой узел по суффиксной ссылке

struct SuffixTreeNode \*suffixLink;

/\*(start, end) интервал определяет ребро, которым кадый

узел соединен с родительским узлом. Хранится в ребенке \*/

int start;

int \*end;

/\*для листьев хранится индекс суффикса для пути от корня до листа\*/

int suffixIndex;

};

typedef struct SuffixTreeNode Node;

string text;

//char text[MAX\_CHAR];

Node \*root = NULL; //указатель на вершину

Node \*lastNewNode = NULL; //lastNewNode указывает на созданный внутренний узел

Node \*activeNode = NULL;

int activeEdge = -1; //индекс символа

int activeLength = 0;

int remainingSuffixCount = 0; //как много суффиксов еще добавить в дерево

int leafEnd = -1;

int \*rootEnd = NULL;

int \*splitEnd = NULL;

int strSize = -1; //длина введенной строки

Node \*newNode(int start, int \*end)

{

Node \*node = (Node\*)malloc(sizeof(Node));

int i;

for (i = 0; i < MAX\_CHAR; i++)

node->children[i] = NULL;

/\*Для root suffixLink будет установлен в NULL

Для внутренних узлов suffixLink будет установлен в root

по умолчанию в текущем расширения и может измениться \*/

node->suffixLink = root;

node->start = start;

node->end = end;

/\*suffixIndex по умолчанию -1\*/

node->suffixIndex = -1;

return node;

}

int edgeLength(Node \*n) {

if (n == root)

return 0;

return \*(n->end) - (n->start) + 1;

}

int walkDown(Node \*currNode)

{

/\*Если activeLength больше, чем текущая длина ребра,

то следующий внутренний узел - activeNode, activeEdge

и activeLength соответственно представляют ту же activePoint

(APCFWD)

\*/

if (activeLength >= edgeLength(currNode))

{

activeEdge += edgeLength(currNode);

activeLength -= edgeLength(currNode);

activeNode = currNode;

return 1;

}

return 0;

}

void extendSuffixTree(int pos)

{

leafEnd = pos;

remainingSuffixCount++;

lastNewNode = NULL;

//добавление всех суффиксов, что еще не были добавлены, по одному в дерево

while (remainingSuffixCount > 0) {

if (activeLength == 0)

activeEdge = pos; //APCFALZ

// нет выходящего ребра

// activeEdge из activeNode

if (activeNode->children[text[activeEdge]] == NULL)

{

//создание нового ребра листа

activeNode->children[text[activeEdge]] =

newNode(pos, &leafEnd);

if (lastNewNode != NULL)

{

lastNewNode->suffixLink = activeNode;

lastNewNode = NULL;

}

}

// выходящее ребро, начинающееся с activeEdge

// из activeNode

else

{

// получить следующий узел в конце ребра, начинающегося

// с activeEdge

Node \*next = activeNode->children[text[activeEdge]];

if (walkDown(next))//Do walkdown

{

continue;

}

if (text[next->start + activeLength] == text[pos])

{

if (lastNewNode != NULL && activeNode != root)

{

lastNewNode->suffixLink = activeNode;

lastNewNode = NULL;

}

activeLength++;

break;

}

splitEnd = (int\*)malloc(sizeof(int));

\*splitEnd = next->start + activeLength - 1;

//новый внутренний узел

Node \*split = newNode(next->start, splitEnd);

activeNode->children[text[activeEdge]] = split;

split->children[text[pos]] = newNode(pos, &leafEnd);

next->start += activeLength;

split->children[text[next->start]] = next;

if (lastNewNode != NULL)

{

lastNewNode->suffixLink = split;

}

lastNewNode = split;

}

/\* Один суффикс был добавлен в дерево, декремент количество

суффиксов на добавление\*/

remainingSuffixCount--;

if (activeNode == root && activeLength > 0) //APCFER2C1

{

activeLength--;

activeEdge = pos - remainingSuffixCount + 1;

}

else if (activeNode != root) //APCFER2C2

{

activeNode = activeNode->suffixLink;

}

}

}

//печать от и до

void print(int i, int j)

{

int k;

for (k = i; k <= j; k++)

printf("%c", text[k]);

}

//Вывод дерева и расстановка суффиксных индексов

//Вывод в DFS стиле

//Печатается каждое ребро с его суффиксным индексом

void setSuffixIndexByDFS(Node \*n, int labelHeight)

{

if (n == NULL) return;

if (n->start != -1) //не корень

{

//Печать значения на ребре от родителя до текущего узла

//printf("\nNode: ");

print(n->start, \*(n->end));

}

int leaf = 1;

int i;

for (i = 0; i < MAX\_CHAR; i++)

{

if (n->children[i] != NULL)

{

//if (leaf == 1 && n->start != -1)

// printf(" [%d]\n", n->suffixIndex);

//текущий узел не лист, т.к. имеет выходящее ребро

leaf = 0;

setSuffixIndexByDFS(n->children[i], labelHeight +

edgeLength(n->children[i]));

}

}

if (leaf == 1)

{

n->suffixIndex = strSize - labelHeight;

printf(" [%d]\n", n->suffixIndex);

}

}

//освобождение памяти

void freeSuffixTreeByPostOrder(Node \*n)

{

if (n == NULL)

return;

int i;

for (i = 0; i < MAX\_CHAR; i++)

{

if (n->children[i] != NULL)

{

freeSuffixTreeByPostOrder(n->children[i]);

}

}

if (n->suffixIndex == -1)

free(n->end);

free(n);

}

/\*Построение дерева и вывод значений ребер с суффиксными индексами.

Индекс для листовых ребер >=0, для не листовых = -1\*/

void buildSuffixTree()

{

strSize = text.length();//strlen(text);//

int i;

rootEnd = (int\*)malloc(sizeof(int));

\*rootEnd = -1;

/\*индексы root -1, т.к. у него нет родителя\*/

root = newNode(-1, rootEnd);

activeNode = root; //перый activeNode будет корнем

for (i = 0; i<strSize; i++)

extendSuffixTree(i);

int labelHeight = 0;

setSuffixIndexByDFS(root, labelHeight);

}

//обход для нахождения длиннейшей повторяющейся подстроки

void doTraversal(Node \*n, int labelHeight, int\* maxHeight, int\* substringStartIndex)

{

if (n == NULL)

{

return;

}

int i = 0;

if (n->suffixIndex == -1) //если узел внутренний

{

for (i = 0; i < MAX\_CHAR; i++)

{

if (n->children[i] != NULL)

{

doTraversal(n->children[i], labelHeight +

edgeLength(n->children[i]), maxHeight,

substringStartIndex);

}

}

}

else if (n->suffixIndex > -1 &&

(\*maxHeight < labelHeight - edgeLength(n)))

{

\*maxHeight = labelHeight - edgeLength(n);

\*substringStartIndex = n->suffixIndex;

}

}

//алго поиска длиннейшей повторяющейся подстроки

void getLongestRepeatedSubstring()

{

int maxHeight = 0;

int substringStartIndex = 0;

doTraversal(root, 0, &maxHeight, &substringStartIndex);

printf("Longest Repeated Substring is: ");

int k;

for (k = 0; k<maxHeight; k++)

printf("%c", text[k + substringStartIndex]);

if (k == 0)

printf("No repeated substring");

printf("\n");

}

int main()

{

ifstream inpFile("input.txt");

//string inpStr;

if (inpFile.is\_open()) {

getline(inpFile, text);

text += "$";

inpFile.close();

}

buildSuffixTree();

getLongestRepeatedSubstring();

//освобождение памяти

freeSuffixTreeByPostOrder(root);

cout << text;

system("pause");

return 0;

}

1. Тесты

