Лабораторная работа № 5 по курсу дискрeтного анализа: суффиксные деревья

Выполнил студент группы М8О-308Б-20 МАИ *Борисов Ян*.

# Условие

Кратко описывается задача:

1. Реализовать алгоритм Укконена для построения суффиксного дерева за линейное время.
2. Вариант задания: поиск наибольшей общей подстроки у двух строк.

# Метод решения

# 1)Реализую класс вершины trie, состоящий из:

* + - 1. Map, содержащей символы с которых начинаются ребра и ссылки на вершины, которые соединены этими ребрами с текущей вершиной.
      2. Суффиксной ссылки.
      3. Позиции начала и конца символов, содержащихся на ребре в исходном тексте.
      4. Индекс суффикса в тексте, начиная от корня до листа.

1. При создании новой вершины для нее выделяется память, суффиксная ссылка по умолчанию ведет в корень, интервал ребра устанавливается между двумя переданными в функцию создания вершины позициями, индекс суффикса по умолчанию принимает значение -1.
2. Функция добавления в суффиксное дерево реализована согласно алгоритму Укконена: мы добавляем все суффиксы префикса строки, начинающейся с обрабатываемой в тексте позиции, при этом если при увеличении глобального индекса, какой то суффикс не находится в дереве, мы добавляем его в ручную через алгоритм добавления в trie, и устанавливаем значение суффиксной ссылки добавленной раннее вершины на текущую вершину.
3. Затем, используя алгоритм прохода дерева в глубину мы расставляем индексы суффиксов
4. Используя алгоритм прохода дерева в глубину, мы помечаем каждую вершину значением той строки, к которой она принадлежит, те вершины, которые помечены и 1, и 2 строкой, являются для них общими. Мы берем самые глубокие такие вершины, путь от корня до них будет представлять собой наибольшие общие подстроки для двух данных строк.

# Описание программы

В моей программе один файл main.cpp:

#include <iostream>

#include <vector>

#include <map>

#define MAX\_CHAR 28

struct SuffTreeNode {

std::map<char, SuffTreeNode\*> children;

// pointer to other node

SuffTreeNode \*suffLink;

// interval of symbols on edge connecting nodes

int begin;

int \*end;

// index of suffix in text from root to leaf

int suffIndex;

};

std::string text;

SuffTreeNode \*root = nullptr;

// newly created internal node

SuffTreeNode \*lastAddedNode = nullptr;

// node where we are now

SuffTreeNode \*currentNode = nullptr;

// index of current character in input text

int activeEdge = -1;

int activeLength = 0;

// number of remaining suffixes for adding to tree

unsigned int remainingSuffixToAdd = 0;

// index of end in leaves to increment in each iteration of suffix adding

int leafEnd = -1;

int \*splitEnd = nullptr;

int size = -1;

unsigned int size1;

SuffTreeNode\* createNewNode(int start, int\* end) {

auto \*node = new SuffTreeNode;

*/\*for(auto & i : node->children) {*

*i = nullptr;*

*}\*/*

// by default suffixLink will lead to root

node->suffLink = root;

// set edge interval

node->begin = start;

node->end = end;

//if node is not a leaf it's suffixIndex = -1

node->suffIndex = -1;

return node;

}

int edgeLength(SuffTreeNode\* n) {

if(n == root) {

return 0;

}

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

}

int getNextNode(SuffTreeNode\* currNode) {

if (activeLength >= edgeLength(currNode)) {

activeEdge += edgeLength(currNode);

activeLength -= edgeLength(currNode);

currentNode = currNode;

return 1;

}

return 0;

}

void addToSuffTree(int pos) {

// extending of leaves in tree

leafEnd = pos;

remainingSuffixToAdd++;

lastAddedNode = nullptr;

// add all suffixes in tree

while(remainingSuffixToAdd > 0) {

if(activeLength == 0) {

activeEdge = pos;

}

// There is no outgoing edge starting with activeEdge from currentNode

auto find = currentNode->children.find(text[activeEdge]);

if(find == currentNode->children.end()) {

currentNode->children[text[activeEdge]] = createNewNode(pos,

&leafEnd);

if(lastAddedNode != nullptr) {

lastAddedNode->suffLink = currentNode;

lastAddedNode = nullptr;

}

}

// There is an outgoing edge starting with activeEdge from currentNode

else {

// Get the next node at the end of edge starting with activeEdge

SuffTreeNode\* next = currentNode->children[text[activeEdge]];

if(getNextNode(next)) {

//Start from next node (the new currentNode)

continue;

}

// current character being processed is already on the edge

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

if(lastAddedNode != nullptr && currentNode != root) {

lastAddedNode->suffLink = currentNode;

lastAddedNode = nullptr;

}

activeLength++;

break;

}

// when we are on the edge and there is no next character on it, so

// we need to split our edge

splitEnd = new int(next->begin + activeLength - 1);

// New internal node

SuffTreeNode\* split = createNewNode(next->begin, splitEnd);

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

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

next->begin += activeLength;

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

if(lastAddedNode != nullptr) {

lastAddedNode->suffLink = split;

}

lastAddedNode = split;

}

remainingSuffixToAdd--;

if(currentNode == root && activeLength > 0) {

activeLength--;

activeEdge = pos - remainingSuffixToAdd + 1;

}

else if (currentNode != root) {

currentNode = currentNode->suffLink;

}

}

}

void printTree(int i, int j) {

int k;

*/\*for (k = i; k <= j; ++k) {*

*std::cout << text[k];*

*}\*/*

}

void dfsSuffIndexSet(SuffTreeNode\* n, int labelHeight) {

if (n == nullptr) {

return;

}

*/\*if (n->begin != -1) { //A non-root node*

*Print the label on edge from parent to current node*

*printTree(n->begin, \*(n->end));*

*}\*/*

int leaf = 1;

for (auto const& key : n->children) {

if (n->children[key.first] != nullptr) {

*/\*if (leaf == 1 && n->begin != -1) {*

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

*}\*/*

//Current node is not a leaf as it has outgoing

//edges from it.

leaf = 0;

dfsSuffIndexSet(n->children[key.first], labelHeight +

edgeLength(n->children[key.first]));

}

}

// node is a leaf and we set index of suffix

if (leaf == 1) {

n->suffIndex = size - labelHeight;

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

}

}

void freeSuffTree(SuffTreeNode\* n) {

if (n == nullptr) {

return;

}

for (auto const& key : n->children) {

if (n->children[key.first] != nullptr) {

freeSuffTree(n->children[key.first]);

}

}

if (n->suffIndex == -1) {

delete(n->end);

}

delete(n);

}

void buildSuffixTree() {

size = text.length();

// Root is a node with start and end indices -1

root = createNewNode(-1, new int(-1));

currentNode = root; // First currentNode will be root

for (int i = 0; i < size; ++i) {

addToSuffTree(i);

}

int labelHeight = 0;

dfsSuffIndexSet(root, labelHeight);

}

int dfsCommonSubstringsFind(SuffTreeNode \*n, int labelHeight, unsigned short\* maxHeight,

std::vector<int>\* substringStartIndex) {

if(n == nullptr) {

return -5;

}

int ret = -1;

if(n->suffIndex < 0) { // if it is internal node

for (auto const& key : n->children) {

if(n->children[key.first] != nullptr) {

ret = dfsCommonSubstringsFind(n->children[key.first],labelHeight + edgeLength( n->children[key.first]),maxHeight, substringStartIndex);

if(n->suffIndex == -1) {

n->suffIndex = ret;

}

else if((n->suffIndex == -2 && ret == -3) ||

(n->suffIndex == -3 && ret == -2) ||

n->suffIndex == -4) {

n->suffIndex = -4; // mark node as I II

// find the deepest node

if(\*maxHeight < labelHeight) {

substringStartIndex->clear();

\*maxHeight = labelHeight;

substringStartIndex->push\_back( \*(n->end) -

labelHeight + 1);

}

else if (\*maxHeight == labelHeight) {

substringStartIndex->push\_back( \*(n->end) -

labelHeight + 1);

}

}

}

}

}

else if(n->suffIndex > -1 && n->suffIndex < size1) { //suffix of I

return -2; //mark node as I

}

else if(n->suffIndex >= size1) { //suffix of II

return -3; //mark node as II

}

return n->suffIndex;

}

void getLongestCommonSubstring() {

auto \*substringStartPositions = new std::vector<int>;

auto \*answers = new std::vector<std::string>;

unsigned short maxHeight = 0;

dfsCommonSubstringsFind(root, 0, &maxHeight, substringStartPositions);

unsigned short k;

std::cout << maxHeight << "**\n**";

for(int index : \*substringStartPositions) {

std::string answer;

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

answer += text[k + index];

}

answers->push\_back(answer);

}

delete substringStartPositions;

answers->erase( unique(answers->begin(), answers->end() ),

answers->end() );

for(const std::string& answer : \*answers) {

std::cout << answer << std::endl;

}

delete answers;

}

int main() {

std::string string2;

std::cin >> text >> string2;

size1 = text.length() + 1;

text = text + '{' + string2 + '|';

buildSuffixTree();

getLongestCommonSubstring();

freeSuffTree(root);

return 0;

}

# Дневник отладки

Алгоритм не проходил 13 тест, происходило превышения лимита по использованию памяти, проблема была в том, что в каждой вершине я хранил массив из 28 ячеек(26 символов латинского алфавита и 2 sentinel символа), при этом не все ячейки на самом деле были нужны(так как не из каждого узла выходили ребра начинающиеся со всех символов алфавита), они заполнялись null и отнимали много лишней памяти. Я произвел замену массива на map, что оказалось гораздо эффективнее по памяти, и алгоритм сразу же прошел все тесты.

# Тест производительности

Померить время работы кода лабораторной и теста производительности на разных объёмах входных данных. Сравнить результаты. Проверить, что рост времени работы при увеличении объема входных данных согласуется с заявленной сложностью.

Время работы алгоритма (N = M):

А) N = M = 1 => 0.000003 c.

Б) N = M = 10 => 0.00003 c, отношение к предыдущему ~10

В) N = M =100 => 0.00035 c, отношение к предыдущему ~10

Г) N = M = 1000 => 0.004 c, отношение к предыдущему ~10

Д) N = M = 10000 => 0.047 c, отношение к предыдущему ~10

E) N = M = 100000 => 0.564 c, отношение к предыдущему ~10

Ж) N = M = 100000 => 7.625 c, отношение к предыдущему ~13

Из данных тестов мы можем заметить, что при увеличении размера входных данных время работы увеличивается пропорционально (линейно). Значит моя реализация алгоритма действительно имеет линейное время.

# Недочёты

Основной недочёт – весь код лабораторной находится в одном файле, в связи с отсутствием сложностей загрузки его на чекер.

# Выводы

Выполнив данную лабораторную работу, я познакомился с практическим применением суффиксного дерева, построением его за линейное время с помощью алгоритма Укконена, научился искать наибольшую подстроку для двух строк, используя суффиксное дерево.