Московский авиационный институт (национальный исследовательский университет)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа по курсу "Дискретный анализ" №5

Студент: Клименко В. М.

Преподаватель: Макаров Н. К.

Группа: М8О-203Б-22

Дата: ___________

Оценка: ___________

Подпись: ___________

Содержание

Постановка задачи

Найти самую длинную общую подстроку двух строк с использованием суфф. дерева.

Формат ввода

Две строки.

Формат вывода

На первой строке нужно распечатать длину максимальной общей подстроки, затем перечислить все возможные варианты общих подстрок этой длины в порядке лексикографического возрастания без повторов.

Алгоритм решения

Для построения суффиксного дерева я реализовал алгоритм Укконена за $O(n)$, где n – длина исходной строки.

Поиск реализовал двумя способами:

1. при помощи построении дерева над строкой вида `string1#string2` и поиска общих вершин этих строк при помощи поиска в глубину
2. при помощи поиска всех суффиксов большей строки в дереве, построенном на меньшей строке

Первый алгоритм работает за $O(m + n)$, второй за $O(m \cdot n)$, где m, n – длины строк. На чекере первый алгоритм показал себя хуже и по времени, и по памяти. Скорее всего это произошло из-за реализации первого алгоритма через рекурсию.

Исходный код

```
#include <iostream>

#include <string>
#include <map>
```

```
#include <set>
#include <vector>
#include <stack>

#include <algorithm>

#include <memory>
#include <stdexcept>




// c++ is a fucking joke

struct SuffixTreeNode;
struct SuffixTreeEdge;



struct SuffixTreeNode {
	static size_t currentMaxSuffixIndex;
	// chars to edges
	std::map<char, std::shared_ptr<SuffixTreeEdge>> next;
	std::shared_ptr<SuffixTreeNode> suffixLink = nullptr;
	size_t id = 0;

	SuffixTreeNode();
	SuffixTreeNode(const char value, const size_t start, const
std::shared_ptr<size_t> &end);
	void addEdge(const char value, const size_t start, const
std::shared_ptr<size_t> &end);
	void addEdge(const std::shared_ptr<SuffixTreeNode> &node, const char
value, const size_t start, const std::shared_ptr<size_t> &end);
	std::shared_ptr<SuffixTreeNode> setSuffixLink(const
std::shared_ptr<SuffixTreeNode> &node);

	bool empty();

	friend std::ostream& operator<<(std::ostream &os, const SuffixTreeNode
&node);
};

size_t SuffixTreeNode::currentMaxSuffixIndex = 0;
```

```
struct SuffixTreeEdge {
	size_t start = 0;
	std::shared_ptr<size_t> end = nullptr;
	// edge ends in this node
	std::shared_ptr<SuffixTreeNode> node = nullptr;

	SuffixTreeEdge(const size_t start, const std::shared_ptr<size_t> &end);
	// split the edge at this->start + length (0 < length < *this->end -
this->start), returns shared pointer to a new internal node
	std::shared_ptr<SuffixTreeNode> split(const std::shared_ptr<size_t>
&currentEnd, const size_t length, const char newChar, const char
differentChar);
	size_t getLength();

	friend std::ostream& operator<<(std::ostream &os, const SuffixTreeEdge
&edge);
};
```

```
// implementation of SuffixTreeNode
SuffixTreeNode::SuffixTreeNode() {}

SuffixTreeNode::SuffixTreeNode(const char value, const size_t start, const
std::shared_ptr<size_t> &end) {
	this->next[value] = std::make_shared<SuffixTreeEdge>(start, end);
}

void SuffixTreeNode::addEdge(const char value, const size_t start, const
std::shared_ptr<size_t> &end) {
	this->next[value] = std::make_shared<SuffixTreeEdge>(start, end);
	this->next[value]->node->id = SuffixTreeNode::currentMaxSuffixIndex++;
}

void SuffixTreeNode::addEdge(const std::shared_ptr<SuffixTreeNode> &node, const
char value, const size_t start, const std::shared_ptr<size_t> &end) {
	std::shared_ptr<SuffixTreeEdge> newEdge =
std::make_shared<SuffixTreeEdge>(start, end);
	newEdge->node = node;
	this->next[value] = newEdge;
```

```
}

std::shared_ptr<SuffixTreeNode> SuffixTreeNode::setSuffixLink(const
std::shared_ptr<SuffixTreeNode> &node) {
	this->suffixLink = node;
	return node;
}

bool SuffixTreeNode::empty() {
	return this->next.empty();
}

std::ostream& operator<<(std::ostream &os, const SuffixTreeNode &node) {
	os << "next:\n";
	for (const auto& [key, value] : node.next) {
	os << "\t" << key << " --- " << *value << '\n';
	}
	os << "suffixLink: " << node.suffixLink;
	return os;
}



// implementation of SuffixTreeEdge
SuffixTreeEdge::SuffixTreeEdge(const size_t start, const
std::shared_ptr<size_t> &end) : start(start), end(end),
node(std::make_shared<SuffixTreeNode>()) {}

std::shared_ptr<SuffixTreeNode> SuffixTreeEdge::split(const
std::shared_ptr<size_t> &currentEnd, const size_t length, const char newChar,
const char differentChar) {
	if (length == 0 || length >= this->getLength()) {
	throw std::invalid_argument("length is greater than the substring's
length");
	}

	std::shared_ptr<SuffixTreeNode> internalNode =
std::make_shared<SuffixTreeNode>(newChar, *currentEnd - 1, currentEnd); // node
with the new char
	internalNode->next[newChar]->node->id =
SuffixTreeNode::currentMaxSuffixIndex++;
	internalNode->addEdge(this->node, differentChar, this->start + length,
this->end); // old node

	this->node = internalNode;
```

```
        this->end = std::make_shared<size_t>(this->start + length);

        return internalNode;
}

size_t SuffixTreeEdge::getLength() {
        return *this->end - this->start;
}

std::ostream& operator<<(std::ostream &os, const SuffixTreeEdge &edge) {
        os << ' ' << edge.start << ":" << *edge.end << " (" << edge.end << ")
node: " << edge.node;
        return os;
}




class SuffixTree {
private:
        std::string inputString;
        size_t inputStringLength = 0;

        std::shared_ptr<SuffixTreeNode> root =
std::make_shared<SuffixTreeNode>();

        std::shared_ptr<SuffixTreeNode> activeNode = this->root;
        std::shared_ptr<SuffixTreeEdge> activeEdge = nullptr;
        size_t activeLength = 0;
        size_t depth = 0;

        size_t remainder = 0;

        std::shared_ptr<size_t> end = std::make_shared<size_t>(0);


private:
        //! rule 1
        void ruleOne(const size_t currentCharIndex) {
        if (this->activeNode != this->root || this->depth == 0) {
                throw std::logic_error("activeNode is not root or activeLength is
zero");
        }
```

```
        char nextRemainingChar = this->inputString[currentCharIndex -
this->remainder + 1];

        // apply the rule
        this->depth--;

        const auto entry = this->activeNode->next.find(nextRemainingChar);
        if (entry != this->activeNode->next.end()) {
              this->activeEdge = entry->second;
              this->activeLength = this->depth;
              this->canonicize(currentCharIndex);
        } else {
              this->activeLength = 0;
              this->activeEdge = nullptr;
        }
        }

        //! rule 3
        void ruleThree(const size_t currentCharIndex) {
        if (this->activeNode == this->root) {
              throw std::logic_error("activeNode is root");
        }

        char nextRemainingChar = '\0';

        if (this->remainder - 1 > this->activeLength) {
              nextRemainingChar = this->inputString[currentCharIndex -
this->activeLength];
        } else {
              nextRemainingChar = this->inputString[currentCharIndex -
this->remainder + 1];
        }


        // apply the rule
        if (this->activeNode->suffixLink != nullptr) {
              this->activeNode = this->activeNode->suffixLink;
              this->depth--;
        } else {
              this->activeNode = this->root;
              this->ruleOne(currentCharIndex);
              return;
        }

        const auto entry = this->activeNode->next.find(nextRemainingChar);
```

```
	if (entry != this->activeNode->next.end()) {
		this->activeEdge = entry->second;
		this->canonicize(currentCharIndex);
	} else {
		this->activeEdge = nullptr;
	}
	}

	// rule 1 or 3
	void ruleOneOrThree(const size_t currentCharIndex) {
	if (this->activeNode == this->root && this->activeLength != 0) { //! rule
1
		this->ruleOne(currentCharIndex);
	} else if (this->activeNode != this->root) { //! rule 3
		this->ruleThree(currentCharIndex);
	}
	}

	//! observation 2
	void canonicize(const size_t currentCharIndex) {
	size_t activeEdgeLength = this->activeEdge->getLength();

	while (this->activeLength >= activeEdgeLength) { // substring is too
smol, go further through the edges
		this->activeNode = this->activeEdge->node;
		this->activeLength = this->activeLength - activeEdgeLength;

		if (this->activeLength == 0) {
		this->activeEdge = nullptr;
		return;
		} else {
		this->activeEdge =
this->activeNode->next[inputString[currentCharIndex - this->activeLength]];
		activeEdgeLength = this->activeEdge->getLength();
		}
	}
	}

	// explicitly add all suffixes
	void buildSuffixLinks(std::shared_ptr<SuffixTreeNode>
&previousInternalNode, const size_t currentCharIndex) {
	const char currentChar = this->inputString[currentCharIndex];

	while (this->remainder > 0) {
		if (this->activeLength == 0) { // inserting new edge to the active
```

```
node

            const auto entry = this->activeNode->next.find(currentChar); //* i
HATE performance
            if (entry != this->activeNode->next.end()) {
                  this->activeEdge = entry->second;
                  this->activeLength++;
                  this->depth++;
                  this->canonicize(currentCharIndex);
                  break;
            }

            this->activeNode->addEdge(currentChar, currentCharIndex,
this->end);

            if (this->activeNode != this->root) { //! rule 2
                  previousInternalNode =
previousInternalNode->setSuffixLink(this->activeNode);
            }

            this->remainder--;

            } else { // splitting active edge

            const char checkedChar = this->inputString[this->activeEdge->start
+ this->activeLength];
            if (checkedChar == currentChar) {
                  break;
            }

            std::shared_ptr<SuffixTreeNode> currentInternalNode =
this->activeEdge->split(this->end, this->activeLength, currentChar,
checkedChar);
            previousInternalNode =
previousInternalNode->setSuffixLink(currentInternalNode); //! rule 2
            this->remainder--;

            }

            ruleOneOrThree(currentCharIndex); //! rule 1 or 3 (or none)
      }
      }

      void buildTree() {
      for (size_t phase = 0; phase < this->inputStringLength; ++phase) {
```

```
            // if (phase == 10) break;
            (*this->end)++;
            this->remainder++;

            size_t currentCharIndex = phase;
            const char currentChar = this->inputString[currentCharIndex];


            if (this->activeLength == 0) { // active point is on a node

            const auto entry = this->activeNode->next.find(currentChar);
            if (entry != this->activeNode->next.end()) { // there is an edge
starting with this char
                  this->activeEdge = entry->second; // enter the edge
                  this->activeLength++;
                  this->depth++;
                  this->canonicize(currentCharIndex);
            } else { // there is no edge starting with this char
                  this->activeNode->addEdge(currentChar, phase, this->end); //
create an edge
                  this->remainder--;

                  if (this->activeNode != this->root) {
                        std::shared_ptr<SuffixTreeNode> activeNodePreRule =
this->activeNode;
                        this->ruleThree(currentCharIndex); //! rule 3. rule 1
can't be applied because activeLength is 0
                        this->buildSuffixLinks(activeNodePreRule,
currentCharIndex);
                  }
            }

            } else { // active point is on an edge

            const char checkedChar = this->inputString[this->activeEdge->start
+ this->activeLength];
            if (currentChar == checkedChar) { // char is in the edge's
substring
                  this->activeLength++;
                  this->depth++;
                  this->canonicize(currentCharIndex);
            } else { // char is not in the edge's substring, should split and
create a new internal node
                  std::shared_ptr<SuffixTreeNode> previousInternalNode =
this->activeEdge->split(this->end, this->activeLength, currentChar,
```

```
checkedChar); // create a new node inside the edge
                        this->remainder--;
                        this->ruleOneOrThree(currentCharIndex); //! rule 1 or 3
                        this->buildSuffixLinks(previousInternalNode,
currentCharIndex);
                }

                }
        }
        }

        // return a tuple consisting of two booleans (node starts at string1 and
node starts at string2), a vector of suffix node ids, length, and any leaf node
id
        std::tuple<bool, bool, std::vector<size_t>, size_t, size_t>
findAllLCSInNode(const std::shared_ptr<SuffixTreeNode> &node, const size_t
dividerIndex, const size_t currentDepth) {
        std::vector<size_t> indecies;
        size_t maxLength = 0;

        size_t leafId;

        bool inString1 = false;
        bool inString2 = false;

        for (const auto &[_, edge] : node->next) { // iterate over all edges
                const size_t nodeId = edge->node->id;
                const size_t edgeLength = edge->getLength();
                leafId = nodeId;

                if (edge->node->empty()) { // leaf node at this edge => check its
id and decide whether to keep it or not
                if (nodeId < dividerIndex) { // suffix is in `string1`
                        inString1 = true;
                } else if (nodeId > dividerIndex && nodeId !=
this->inputStringLength - 1) { // suffix is in `string2`
                        inString2 = true;
                }
                } else { // edge ends in an internal node => go deeper, save
edge's string position, indecies and length
                auto [currentInString1, currentInString2, currentIndecies,
currentLength, currentLeafId] = findAllLCSInNode(edge->node, dividerIndex,
currentDepth + edgeLength);
                leafId = currentLeafId;
```

```
            inString1 = inString1 || currentInString1;
            inString2 = inString2 || currentInString2;

            if (currentInString1 && currentInString2 && currentLength >=
maxLength) {
                  if (currentLength > maxLength) { // new length is bigger =>
save only this edge's start
                        maxLength = currentLength;
                        indecies.clear();
                  }
                  indecies.reserve(indecies.size() + currentIndecies.size());
                  for (const size_t &index : currentIndecies) {
                        indecies.push_back(index);
                  }
            }
            }
      }

      if (inString1 && inString2 && maxLength <= currentDepth) {
            maxLength = currentDepth;
            indecies.push_back(leafId);
      }

      return std::make_tuple(inString1, inString2, indecies, maxLength,
leafId);
      }

public:
      SuffixTree(const std::string &inputString) {
      this->inputString = inputString + '\0';
      this->inputStringLength = this->inputString.length();
      this->buildTree();
      }

      // SuffixTree should be initialized with text
      // find all LCS in the suffix tree
      // return indecies of starts of substrings in the pattern and their
length
      std::pair<std::vector<size_t>, size_t> findAllLCS(const std::string
&pattern) {
      const size_t patternLength = pattern.length();

      std::vector<size_t> indecies;
      size_t maxSubstringLength = 0;
```

```
    for (size_t suffixIndex = 0; suffixIndex < patternLength; ++suffixIndex)
{ // search for ith pattern's suffix via dfs from root
        this->activeNode = this->root;
        this->activeEdge = nullptr;
        this->activeLength = 0;

        size_t index;
        for (index = suffixIndex; index < patternLength; ++index) {
        const size_t substringLength = index - suffixIndex;
        const char currentChar = pattern[index];

        if (this->activeLength == 0) {
            const auto entry = this->activeNode->next.find(currentChar);
            if (entry != this->activeNode->next.end()) { // there is an
edge starting with this char
                this->activeEdge = entry->second; // enter the edge
                this->activeLength++;
                if (this->activeLength ==
this->activeEdge->getLength()) {
                this->activeNode = this->activeEdge->node;
                this->activeEdge = nullptr;
                this->activeLength = 0;
                }
            } else {
                if (maxSubstringLength == substringLength) {
                indecies.push_back(suffixIndex);
                } else if (maxSubstringLength < substringLength) {
                maxSubstringLength = substringLength;
                indecies.clear();
                indecies.push_back(suffixIndex);
                }
                break;
            }
        } else {
            const char checkedChar =
this->inputString[this->activeEdge->start + this->activeLength];
            if (currentChar == checkedChar) { // char is in the edge's
substring
                this->activeLength++;
                if (this->activeLength ==
this->activeEdge->getLength()) {
                this->activeNode = this->activeEdge->node;
                this->activeEdge = nullptr;
                this->activeLength = 0;
                }
```

```
                } else { // char is not in the edge's substring, should split and create a new internal node
                    if (maxSubstringLength == substringLength) {
                    indecies.push_back(suffixIndex);
                    } else if (maxSubstringLength < substringLength) {
                    maxSubstringLength = substringLength;
                    indecies.clear();
                    indecies.push_back(suffixIndex);
                    }
                    break;
                }
            }
            }

            if (index == patternLength) {
            size_t substringLength = index - suffixIndex;
            if (maxSubstringLength == substringLength) {
                indecies.push_back(suffixIndex);
            } else if (maxSubstringLength < substringLength) {
                maxSubstringLength = substringLength;
                indecies.clear();
                indecies.push_back(suffixIndex);
            }
            break;
            }

            if (maxSubstringLength >= patternLength - suffixIndex) { // there are no strings, with length bigger than the current maxSubstringLength
            break;
            }
        }

        return std::make_pair(indecies, maxSubstringLength);
        }

        // SuffixTree should be constructed like this: `string1` + '#' + `string2`
        // find all common substrings in `string1` and `string2` with maximal length in the suffix tree
        // return indecies of starts of substrings in the input string and max length
        std::pair<std::vector<size_t>, size_t> findAllLCS() {
        size_t dividerIndex = this->inputString.find('#');
        if (dividerIndex >= this->inputStringLength - 1) {
            throw std::logic_error("tree was not constructed properly for
```

```
finding longest common substrings");
	}

	// dfs until leaf nodes. if a leaf node has id < dividerIndex => the
suffix is in the text, if id > index('#') => the suffix is in the pattern
	// if the suffix is in the pattern and is in text it is a common
substring
	auto [_, __, indecies, maxSubstringLength, ___] =
this->findAllLCSInNode(this->root, dividerIndex, 0);

	return std::make_pair(indecies, maxSubstringLength);
	}
};




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

	if (text.length() > pattern.length()) {
	std::string temp = std::move(pattern);
	pattern = std::move(text);
	text = std::move(temp);
	}

	SuffixTree st(text);

	std::pair<std::vector<size_t>, size_t> foundStrings =
st.findAllLCS(pattern);
	size_t length = foundStrings.second;
	if (length == 0) {
	return 0;
	}

	std::vector<size_t> &starts = foundStrings.first;

	std::set<std::string> substrings;
	for (const size_t &start : starts) {
	substrings.insert(pattern.substr(start, length));
	}

	std::cout << length << '\n';
```

```
        for (const std::string &substring : substrings) {
        std::cout << substring << '\n';
        }
}
```

Тесты

Входные данные:

```
absvcoiaibuabbabbobasobaobababoba
bfasybaioaubcysbauaybababababybvbapbybaipubcxc
```

Выходные данные:

```
5
babab
```

Входные данные:

```
for(size_tj=0;j<n;++j)ult.push_back(currentSubstring);std::vector<std::string>
vectorstringsetsize_tj++push_
```

Выходные данные:

```
7
size_tj
```

Вывод

В ходе лабораторной работы я научился реализовывать алгоритм Укконена для построения суффиксного дерева за линейное время. Также, я научился использовать суффиксное дерево в алгоритмах, в частности – для поиска наибольшей общей подстроки.