МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритм Ахо-Корасик

Студент гр. 8304 _____ Мешков М.А.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2020

Цель работы.

Изучить принцип работы алгоритма Ахо-Корасик на примере решения задачи.

Постановка задачи.

Вариант 4. Реализовать режим поиска, при котором все найденные образцы не пересекаются в строке поиска (т.е. некоторые вхождения не будут найдены; решение задачи неоднозначно).

Задание 1

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

Первая строка содержит текст (T,$1 \le |T| \le 100000$).

Вторая - число n ($1 \le n \le 3000$), каждая следующая из n строк содержит шаблон из набора $P=\{p_1,\ldots,p_n\} 1 \le |p_i| \le 75$

Все строки содержат символы из алфавита {A,C,G,T,N}

Выход:

Все вхождения образцов из P в T.

Каждое вхождение образца в текст представить в виде двух чисел - i p

Где i - позиция в тексте (нумерация начинается с 1), с которой начинается вхождение образца с номером p

(нумерация образцов начинается с 1).

Строки выхода должны быть отсортированы по возрастанию, сначала номера позиции, затем номера шаблона.

Sample Input:

NTAG

3

TAGT

TAG

T

Sample Output:

2 2

2 3

Задание 2

Используя реализацию точного множественного поиска, решите задачу точного поиска для одного образца с джокером.

В шаблоне встречается специальный символ, именуемый джокером (wild card), который "совпадает" с любым символом. По заданному содержащему шаблоны образцу P необходимо найти все вхождения P в текст T.

Например, образец ab??c? с джокером ? встречается дважды в тексте xabvccbababcax.

Символ джокер не входит в алфавит, символы которого используются в T. Каждый джокер соответствует одному символу, а не подстроке неопределённой длины. В шаблон входит хотя бы один символ не джокер, т.е. шаблоны вида ??? недопустимы.

Все строки содержат символы из алфавита {A,C,G,T,N}

Вход:

Текст (T,$1 \leq |T| \leq 100000$)

Шаблон (P,$1 \leq |P| \leq 40$)

Символ джокера

Выход:

Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер).

Номера должны выводиться в порядке возрастания.

Sample Input:

ACTANCA

A$$A$

$

Sample Output:

1

Описание алгоритма.

Алгоритм Ахо-Корасик позволяет найти все вхождения заданных подстрок в другой строке. Поиск вхождений осуществляется с помощью построенного конечного автомата. Переход автомата в какое-либо состояние говорит о наличии вхождения соответствующего данному состоянию искомой подстроки в строке поиска. Переходы осуществляются в соответствии с символами в строке поиска.

Для построения автомата в начале нужно построить бор (префиксное дерево). Построение начинается с корня, каждая искомая строка добавляется в бор, отсутствующие переходы создаются.

Затем построенный бор дополняется суффиксными ссылками, которые ведут к состоянию, соответствующему набольшему префиксу данного бора, который является суффиксом строки текущего состояния. Для нахождения суффиксных ссылок можно использовать суффиксную ссылку родителя. Пусть к данному состоянию ведет символ с. В таком случае необходимо осуществить переход по суффиксной ссылке родителя и если оттуда есть переход, то состояние, в которое ведет этот переход, является искомой суффиксной ссылкой. В противном случае необходимо продолжить осуществление переходов по суффиксной ссылке до тех пор, пока не будет найдена соответствующая ссылка или пока не будет встречено коревое состояние.

Переход по суффиксным ссылкам осуществляется в случае отсутствия во время поиска вхождений соответствующего перехода из текущего состояния по встреченному символу.

При наличии символов-джокеров в искомой строке данную строку необходимо разбить на части разделенные символами-джокерами и обработать каждую часть как отдельную строку для поиска. Во время поиска необходимо запоминать ожидаемую позицию следующей части слова, затем в этой позиции

нужно будет проверить действительно ли там встретилась эта часть, и так пока следующей части не окажется, это будет означать (при условии того что джокеры в начале слова и в конце вмещаются в строку поиска), что слово найдено.

Для реализации индивидуального варианта добавляются к конечному ответу только те вхождения, которые не пересекаются с последним добавленным вхождением (первое вхождение добавляется всегда).

Оценка сложности алгоритма.

Так как осуществление перехода в боре осуществляется за константное время, то в худшем случае сложность построения бора - O(|P|), где |P| - сумма длин всех искомых строк. Сложность установки суффиксных ссылок зависит от количества вершин в боре, поэтому в худшем случае она равна O(|P|). Поиск всех вхождений искомых строк без джокеров требует |T| переходов (|T| - длина строки поиска) — сложность O(|T|). Также нужно учесть сложность обработки совпадений — O(L), где L — общая длина всех совпадений. Общая вычислительная сложность алгоритма в худшем случае O(|P|+|T|+L).

Сложность по памяти равна O(|P|+|T|), так как количество необходимой памяти пропорционально сумме размеров всех искомых строк и строки поиска.

Описание функций и структур данных.

Для решения задачи были реализованы:

Функция findAlphabetIndex для получения индекса символа в алфавите.

Структура Word для хранения информации о искомом слове.

Структура TrieNode для хранения вершины бора.

Функции getFragmentString и getWordString для преобразования части слова и всего слова в строку.

Функция getBfsOrder для получения порядка обхода бора в ширину.

Функция printTrie для печати бора.

Функция prepareTrieForSearching для подготовки бора к поиску.

Функция findWords для непосредственного поиска слов.

Функция main принимает ввод и выводит ответ. Для подробного вывода процесса работы алгоритма при запуске нужно передать опцию -v, для вывода в файл -fo, для ввода из файла -fi. Можно использовать опцию -m для использования индивидуального варианта.

Тестирование.

Таблица 1 — Тестирование задачи 1

Ввод	Вывод
NTAG 3 TAGT TAG T	2 2 2 3
CCCA 1 CC	1 1 2 1

Таблица 2 — Тестирование задачи 2

Ввод	Вывод
ACT A? ? 1	1
ACTANCA A$$A$ $	1
ACGACTACNACG AC*AC *	1 4 7

Таблица 3 — Тестирование индивидуального варианта

Ввод	Вывод
CCCA 1 CC	1 1
AACAA 2 AA CA	1 1 3 2

Выводы.

В ходе работы был реализован алгоритм Ахо-Корасик для нахождения вхождений образов в строке, программа была протестирована на различных входных данных, были получены ожидаемые результаты, была оценена сложность алгоритма - было выяснено, что время работы алгоритма ограничено O(|P|+|T|+L).

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <map>
#include <fstream>
#include <cstring>

const int inputOutputMode = 1; // 1 - первая часть лабораторной, 2 -
вторая

using namespace std;

bool verbose = false;
std::ostream *out;
std::istream *in;
bool myVariant = false;

const int alphabetSize = 5;
const char alphabet[alphabetSize] = {'A', 'C', 'G', 'T', 'N'};
char wildCardChar = '?';

int findAlphabetIndex(char c) {
    for (int i = 0; i < alphabetSize; i++) {
        if (alphabet[i] == c)
            return i;
    }
    return -1;
}

struct Word;

struct TrieNode {
    TrieNode *parent = nullptr;
    int fromParentTransitionAlpha = -1;
    int level = 0;
    TrieNode *nexts[alphabetSize] = {nullptr};
    TrieNode *suffix = nullptr;
    vector<TrieNode *> matches;

    static TrieNode *createNode(TrieNode *parent, int
fromParentTransitionAlpha) {
        auto node = new TrieNode;
        node->parent = parent;
        node->fromParentTransitionAlpha = fromParentTransitionAlpha;
        node->level = (parent != nullptr ? parent->level + 1 : 0);
```

```
        return node;
    }

    void adjustAsFragment() {
        matches = {this};
    }

    vector<Word> words; // Should be used if it is first word fragment.
};

struct FragmentWithWildcardsBefore {
    TrieNode *fragment = nullptr;
    int wildCardsBefore = 0;
};
struct Word {
    int wordLen = 0;
    static int maxWordLen;

    int wordId = -1;
    static int newWordId;
    void adjustAsWord(int wordLen) {
        wordId = newWordId++;
        this->wordLen = wordLen;
        maxWordLen = std::max(wordLen, maxWordLen);
    }

    int wildCardsAfterWord = 0;
    std::vector<FragmentWithWildcardsBefore>
fragmentsWithWildcardsBefore;

    bool hasNextFragment(int fragmentNumber) {
        return fragmentNumber < (fragmentsWithWildcardsBefore.size() -
1);
    }
    TrieNode *getFirstFragment() {
        return fragmentsWithWildcardsBefore.front().fragment;
    }
};
int Word::newWordId = 0;
int Word::maxWordLen = 0;

string getFragmentString(TrieNode *node) {
    if (node == nullptr)
        return "nullptr";
    if (node->parent == nullptr)
        return "root";
    string fragment;
    while (node != nullptr) {
        fragment += alphabet[node->fromParentTransitionAlpha];
```

```
        node = node->parent;
    }
    reverse(fragment.begin(), fragment.end());
    return fragment;
}
string getWordString(Word *word) {
    string wordStr;
    for (auto fragmentWithWildcardsBefore : word-
>fragmentsWithWildcardsBefore) {
        wordStr += string(fragmentWithWildcardsBefore.wildCardsBefore,
wildCardChar);
        wordStr +=
getFragmentString(fragmentWithWildcardsBefore.fragment);
    }
    wordStr += string(word->wildCardsAfterWord, wildCardChar);
    return wordStr;
}

std::vector<TrieNode *> getBfsOrder(TrieNode *root) {
    std::vector<TrieNode *> order = {root};
    for (auto i = 0; i < order.size(); i++) {
        auto currentNode = order[i];
        for (int j = 0; j < alphabetSize; j++) {
            auto next = currentNode->nexts[j];
            if (next != nullptr)
                order.push_back(next);
        }
    }
    return order;
}

void printTrie(TrieNode *root) {
    vector<Word *> words;

    *out << "Trie nodes:" << endl;
    for (auto node : getBfsOrder(root)) {
        *out << getFragmentString(node) << " { level: " << node->level
<< ", "
             << "suffix: " << getFragmentString(node->suffix) << ", "
             << "nextAlphas: {";
        bool firstPrintableChild = true;
        for (int i = 0; i < alphabetSize; i++) {
            auto child = node->nexts[i];
            if (child != nullptr) {
                *out << (firstPrintableChild ? "" : ", ") <<
alphabet[i];
                firstPrintableChild = false;
            }
        }
```

```
        *out << "}, ";
        *out << "matches: {";
        for (int i = 0; i < node->matches.size(); i++) {
            TrieNode *matchNode = node->matches[i];
            *out << (i == 0 ? "" : ", ") <<
getFragmentString(matchNode);
        }
        *out << "}}" << endl;

        for (auto &word : node->words)
            words.push_back(&word);
    }

    *out << "Words:" << endl;
    for (auto word : words) {
        *out << getWordString(word) << " {wordId: " << word->wordId <<
", wordLen: " << word->wordLen << "}" << endl;
    }
    *out << "maxWordLen: " << Word::maxWordLen << endl;
}

TrieNode *insertFragment(TrieNode *root, const string &word, int
&position) {
    auto current = root;
    for ( ; position < word.size(); position++) {
        auto wordAlphaIndex = findAlphabetIndex(word[position]);
        if (wordAlphaIndex < 0)
            break;
        auto &nextCurrent = current->nexts[wordAlphaIndex];
        if (nextCurrent == nullptr)
            nextCurrent = TrieNode::createNode(current, wordAlphaIndex);
        current = nextCurrent;
    }
    if (current != root)
        current->adjustAsFragment();
    return current;
};

int countWildCards(const string &word, int i) {
    int result = 0;
    for ( ; i < word.size() && findAlphabetIndex(word[i]) < 0; i++)
        result++;
    return result;
};

void insertWord(TrieNode *root, const string &wordStr) {
    Word word;
    for (int i = 0; i < wordStr.size(); ) {
        int wildCardsBefore = countWildCards(wordStr, i);
```

```
        i += wildCardsBefore;
        if (i >= wordStr.size()) {
            word.wildCardsAfterWord = wildCardsBefore;
            break;
        }
        auto fragment = insertFragment(root, wordStr, i);
        word.fragmentsWithWildcardsBefore.push_back({fragment,
wildCardsBefore});
    }
    word.adjustAsWord(wordStr.length());
    TrieNode *firstFragment =
word.fragmentsWithWildcardsBefore.front().fragment;
    firstFragment->words.push_back(word);
}

void addSuffixes(TrieNode *root) {
    auto bfsOrder = getBfsOrder(root);
    for (int i = 1; i < bfsOrder.size(); i++) {
        auto currentNode = bfsOrder[i];
        auto suffix = currentNode->parent->suffix;
        while (suffix != nullptr && suffix->nexts[currentNode-
>fromParentTransitionAlpha] == nullptr)
            suffix = suffix->suffix;
        currentNode->suffix = (suffix == nullptr ? root : suffix-
>nexts[currentNode->fromParentTransitionAlpha]);
    }
}

void addMatches(TrieNode *root) {
    auto bfsOrder = getBfsOrder(root);
    for (int i = 1; i < bfsOrder.size(); i++) {
        auto currentNode = bfsOrder[i];
        currentNode->matches.insert(currentNode->matches.end(),
currentNode->suffix->matches.begin(), currentNode->suffix-
>matches.end());
    }
}

void prepareTrieForSearching(TrieNode *root) {
    addSuffixes(root);
    addMatches(root);
}

template <typename T>
class CircularQueue {
public:
    explicit CircularQueue(int capacity) : m_elements(capacity) {}
    T &at(int fromFirstIndex) {
        return m_elements[rangedIndex(fromFirstIndex)];
```

```
    }
    T &front() {
        return m_elements[m_first];
    }
    void pop() {
        m_elements[m_first] = {};
        m_first = rangedIndex(1);
    }

private:
    int rangedIndex(int i) {
        return (m_first + i) % m_elements.size();
    }

    int m_first = 0;
    vector<T> m_elements;
};

struct FragmentExpectation {
    Word *word = nullptr;
    int fragmentNumber = 0;

    int getNextFragmentWithWildCardsBefore() {
        return word->fragmentsWithWildcardsBefore.at(fragmentNumber +
1).wildCardsBefore;
    }
    bool hasNextFragment() {
        return word->hasNextFragment(fragmentNumber);
    }
    TrieNode *getNextFragment() {
        return word->fragmentsWithWildcardsBefore.at(fragmentNumber +
1).fragment;
    };
};
vector<pair<int, Word *>> findWords(TrieNode *trieRoot, const string
&text) {
    vector<pair<int, Word *>> words;
    CircularQueue<multimap<TrieNode *, FragmentExpectation>>
expectations(Word::maxWordLen);

    auto current = trieRoot;
    for (int i = 0; i < text.size(); i++) {
        auto textAlphaIndex = findAlphabetIndex(text[i]);
        while (current != nullptr && current->nexts[textAlphaIndex] ==
nullptr)
            current = current->suffix;
        current = (current == nullptr ? trieRoot : current-
>nexts[textAlphaIndex]);
        expectations.pop();
```

```
        for (auto &fragment : current->matches) {
            for (auto &word : fragment->words)
                expectations.front().insert({word.getFirstFragment(),
{&word, 0}});

            auto expectationRange =
expectations.front().equal_range(fragment);
            for (auto expectationIter = expectationRange.first;
expectationIter != expectationRange.second; expectationIter++) {
                auto expectation = expectationIter->second;
                if (expectation.hasNextFragment()) {
                    int nextFragmentEndPosition =
expectation.getNextFragmentWithWildCardsBefore() +
expectation.getNextFragment()->level;

expectations.at(nextFragmentEndPosition).insert({expectation.getNextFrag
ment(), {expectation.word, expectation.fragmentNumber + 1}});
                }
                else {
                    int wordEnd = i + expectation.word-
>wildCardsAfterWord;
                    int wordStart = wordEnd - expectation.word->wordLen
+ 1;
                    if (wordStart >= 0 && wordEnd < text.length())
                        words.push_back({wordStart, expectation.word});
                }
            }
        }
    }

    return words;
}

vector<pair<int, Word *>> removeOverlays(const vector<pair<int, Word *>>
&sortedFoundedWords, int strLen) {
    int nextGoodPosition = 0;
    vector<pair<int, Word *>> withoutOverlays;
    for (auto wordWithIndex : sortedFoundedWords) {
        auto index = wordWithIndex.first;
        auto word = wordWithIndex.second;
        if (index >= nextGoodPosition) {
            withoutOverlays.push_back(wordWithIndex);
            nextGoodPosition = index + word->wordLen;
        }
    }
    return withoutOverlays;
}
```

```
int main(int argc, char *argv[]) {
    // Настройка ввода и вывода
    ifstream inFile;
    ofstream outFile;
    in = &cin;
    out = &cout;

    // Чтение параметров командной строки
    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-v") == 0)
            verbose = true;
        if (strcmp(argv[i], "-fi") == 0) {
            in = &inFile;
            inFile.open("in.txt");
        }
        if (strcmp(argv[i], "-fo") == 0) {
            out = &outFile;
            outFile.open("out.txt");
        }
        if (strcmp(argv[i], "-m") == 0)
            myVariant = true;
    }

    string text;
    getline(*in, text);

    auto root = TrieNode::createNode(nullptr, -1);

    int wordsNumber;
    if (inputOutputMode == 1)
        *in >> wordsNumber;
    else
        wordsNumber = 1;
    for (int i = 0; i < wordsNumber; i++) {
        string word;
        *in >> word;
        insertWord(root, word);
    }
    if (inputOutputMode == 2)
        *in >> wildCardChar;

    if (verbose) {
        *out << "Trie after inserting matches:" << endl;
        printTrie(root);
        *out << endl;
    }

    prepareTrieForSearching(root);
```

```
    if (verbose) {
        *out << "Trie after searching preparations:" << endl;
        printTrie(root);
        *out << endl;
    }

    auto foundedWords = findWords(root, text);
    auto compareWordIds = [](const pair<int, Word *> &a, const pair<int,
Word *> &b) {
        if (a.first != b.first)
            return a.first < b.first;
        return a.second->wordId < b.second->wordId;
    };
    sort(foundedWords.begin(), foundedWords.end(), compareWordIds);

    if (myVariant)
        foundedWords = removeOverlays(foundedWords, text.length());

    if (verbose) *out << "Founded matches:" << endl;
    for (auto foundedWordWithPosition : foundedWords) {
        int position = foundedWordWithPosition.first;
        Word *word = foundedWordWithPosition.second;
        if (verbose) {
            *out << "Text: " << text << endl;
            *out << "Word: " << string(position, ' ') <<
getWordString(word) << endl;
        }
        else {
            *out << position + 1;
            if (inputOutputMode == 1)
                 *out << " " << word->wordId + 1;
            *out << endl;
        }
    }

    return 0;
}
```