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

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритм Ахо-Корасик

Студент гр. 9383 _____ Арутюнян С.Н.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Изучить алгоритм Ахо-Корасик без джокера и с джокером и реализовать его на практике.

Основные теоретические положения.

Бор — древовидная структура данных для хранения некоторого множества строк. Из каждого узла, не являющегося листом, есть переход в другой узел по какому-то символу из алфавита. Это дерево строится так, что для каждой строки из исходного множества существует путь от корня до листа, в котором каждый переход определяется текущим символом заданной строки.

Алгоритм Ахо-Корасик решает задачу поиска произвольного количества подстрок в исходном тексте. Он строит конечнный автомат, который последовательно получает все символы текста и переходит по необходимым ребрам. Если автомат пришел в терминальное (конечное) состоние, мы можем говорить, что подстрока найдена в тексте.

Задание.

1. Первая строка содержит текст T, вторая — число n и каждая следующая из n строк содержит шаблон из набора P = {p1, …, pn}. Разработать программу , решающую задачу точного поиска набора образцов.

2. На вход подается текст T, шаблон P и символ джокера. Символ джокера не входит в алфавит, символы которого используются в T ({A, C, G, T, N}). Символ джокера «совпадает» с любым символом. Используя реализацию точного множественного поиска, решите задачу точного поиска для одного образца с джокером.

Вариант 2. Подсчитать количество вершин в автомате; вывести список найденных образцов, имеющих пересечения с другими найденными образцами в строке поиска.

Ход работы:

1. Был разработан класс BohrNode, представляющий собой состояние автомата, строящегося в алгоритме Ахо-Корасик:

1) Проходимся по всем строкам из набора строк.

2) Инициализируем текущий узел в качестве корневого.

3) Проходимся по всем символам текущей строки.

4) Если из текущего узла есть переход по текущему символу, инициализируем текущий узел в качестве узла, в который мы можем перейти по этому символу.

5) Если из текущего узла нет перехода по текущему символу, создаем новый узел и добавляем переход по текущему символу из текущего узла к новому узлу. Инициализируем текущий узел в качетсве нового узла.

2. Был разработан непосредственно алгоритм Ахо-Корасик, на вход которому подается множество строк, стартовое состояние автомата и исходный текст:

1) Инициализируем текущий узел в качестве корневого.

2) Проходимся по всем символам исходного текста.

3) Если из текущего узла есть переход по текущему символу текста, совершаем этот переход.

4) Если такого перехода нет, переходим по суффиксной ссылке.

5) Если новый узел конечный, это означает, что мы нашли подстроку в исходном тексте. Обновляем ответ. Проходимся по конечным ссылкам до тех пор, пока не дойдем до корня (это нужно для того, чтобы не пропустить подстроки, являющиеся подстроками других подстрок из исходного множества строк).

6) Если новый узел не является конечным, переходим к шагу 3.

3. Были разработаны тесты с использованием библиотеки Catch2.

Сложность алгоритма Ахо-Корасик — O(|T| + |M| * log|M| + |k|). Здесь |T| - длина исходного текста, |M| - суммарная длина образцов, |k| - общая длина всех совпадений. Можно понять, что такая сложность получается из построения бора и последующего прохода по исходному тексту.

Описание функций и структур данных:

1. Был разработан класс BohrNode, представляющий собой состояние (узел) автомата. У него есть следующие поля:

static std::shared_ptr<BohrNode> root — статическое поле-корень построенного автомата.

char symbol_to_parent — символ, по которому можно попасть в текущее состояние из его родителя

int pattern_index — индекс образца в векторе образцов (нужно только для терминальных состояний)

std::weak_ptr<BohrNode> parent — указатель на родителя

std::weak_ptr<BohrNode> suffix_reference — суффиксная ссылка

std::unordered_map<char, std::shared_ptr<BohrNode>> children — дети текущего состояния (узла)

std::unordered_map<char, std::weak_ptr<BohrNode>> go — мапа «хороших» переходов из текущего состояния (нужно для того, чтобы при совершении переходов не рассматривать разные случаи переходов напрямую или по суффиксным ссылкам)

И следующие методы:

BohrNode() = default — конструктор по умолчанию.

BohrNode(std::weak_ptr<BohrNode> parent, char symbol_to_parent) — конструктор, инициализирующий информацию о родителе.

std::size_t CountStates() const — подсчет количества состояний.

void AddChild(char arc, std::shared_ptr<BohrNode> child) — добавление нового ребенка.

std::shared_ptr<BohrNode> GetChild(char arc) — возврат указателя на ребенка текущего узла по символу arc.

const std::unordered_map<char, std::shared_ptr<BohrNode>>& GetChildren() const — возврат мапы с детьми.

const std::shared_ptr<BohrNode>& GetChild(char arc) const — константный аналог метода GetChild

std::weak_ptr<BohrNode> MakeTransition(char arc) — совершение перехода по символу arc с использованием мапы go

std::weak_ptr<BohrNode> GetSuffixReference() - возврат суффиксной ссылки.

bool HasChild(char arc) const — проверка существования прямого перехода по символу arc.

bool IsRoot() const — проверка, является ли текущий узел корнем.

bool IsFinal() const — проверка, является ли текущий узел терминальным.

void SetPatternIndex(int pattern_index_) - установка индекса образца из вектора образцов.

int PatternIndex() const — возврат индекса образца из вектора образцов.

Тесты

1. Тест **Building bohr. Test 1.**

Этот тест проверяет правильность построения бора по набору строк «TAGT», «TAG, «T».

2. Тест **Building bohr. Test 2.**

Этот тест проверяет правильность построения бора по набору строк «N», «NA», «NG», «NP», «K», «KN».

3. Тест **Counting states. Test 1.**

Этот тест проверяет правильность подсчета количества состояний в боре, построенного по набору строк «TAGT», «TAG», «T».

4. Тест **Counting states. Test 2.**

Этот тест проверяет правильность подсчета количества состояний в боре, построенного по набору строк «N», «NA», «NG», «NP», «K», «KN».

5. Тест **Searching.**

Этот тест проверяет правильность поиска подстрок «TAGT», «TAG», «T» в тексте «NTAG».

Рис. 1. Правильность прохождения всех тестов.

Примеры работы программы

```
NTAG
3
TAGT
TAG
T
2 2
2 3
Количество вершин в автомате: 4
Найденные пересечения:
Пересечение TAG и T с индекса 1
```

Рис. 2. Пример работы программы 1.

```
AHOCORASICK
3
CORAS
ICK
SICKS
4 1
9 2
Количество вершин в автомате: 13
Найденные пересечения:
```

Рис. 3. Пример работы программы 2.

```
IWANTFIVEINMYJOURNAL
3
IWANTFIVE
WANT
JOURNAL
1 1
2 2
14 3
Количество вершин в автомате: 20
Найденные пересечения:
Пересечение IWANTFIVE и WANT с индекса 1
```

Рис. 4. Пример работы программы 3.

Выводы.

В выполненной лабораторной работе был изучен и применен на практике алгоритм Ахо-Корасик. В процессе выполнения лабораторной работы я осознал, что необязательно строить автоматы наивным образом, т. е. только с прямыми связями между состояниями, ведь можно воспользоваться такими умными оптимизацями, как суффиксные ссылки.

Приложение А

main.cpp

```
#include <iostream>
#include <string>
#include <vector>

#include "bohr.h"
#include "aho_corasick.hpp"


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

    int n;
    std::vector<std::string> substrings;
    std::cin >> n;
    getchar();

    substrings.reserve(n);
    for (int i = 0; i < n; ++i) {
        std::string substring;
        std::getline(std::cin, substring);
        substrings.push_back(std::move(substring));
    }

    auto forest = BohrNode::BuildForest(substrings);
    auto answer = AhoCorasick(forest, substrings, text);

    for (const auto& aho_pair : answer)
        std::cout << aho_pair.start_index << " " <<
aho_pair.pattern_index << std::endl;

    std::cout << "Количество вершин в автомате: " << forest-
>CountStates() << std::endl;

    std::vector<AhoPair> found =
std::vector<AhoPair>{answer.begin(), answer.end()};
    std::cout << "Найденные пересечения:" << std::endl;
```

```
    for (int i = 0; i < found.size() - 1; ++i) {
        if (found[i].start_index - 1 +
substrings[found[i].pattern_index - 1].size() >
found[i+1].start_index - 1) {
            std::cout << "Пересечение " <<
substrings[found[i].pattern_index - 1] << " и "
                      << substrings[found[i+1].pattern_index - 1]
<< " с индекса "
                      << found[i+1].start_index - 1
                      << std::endl;
        }
    }

    return 0;
}
```

bohr.h

```
#pragma once

#include <memory>
#include <vector>
#include <string>
#include <unordered_map>


class BohrNode : std::enable_shared_from_this<BohrNode> {
public:
    BohrNode() = default;
    BohrNode(std::weak_ptr<BohrNode> parent, char
symbol_to_parent);

    std::size_t CountStates() const;

    void AddChild(char arc, std::shared_ptr<BohrNode> child);
    std::shared_ptr<BohrNode> GetChild(char arc);

    const std::unordered_map<char, std::shared_ptr<BohrNode>>&
GetChildren() const;

    const std::shared_ptr<BohrNode>& GetChild(char arc) const;
```

```
    std::weak_ptr<BohrNode> MakeTransition(char arc);

    std::weak_ptr<BohrNode> GetSuffixReference();

    bool HasChild(char arc) const;
    bool IsRoot() const;
    bool IsFinal() const;

    void SetPatternIndex(int pattern_index_);
    int PatternIndex() const;

    static std::shared_ptr<BohrNode> BuildForest(const
std::vector<std::string>& patterns);

private:
    static std::shared_ptr<BohrNode> root;

    char symbol_to_parent = -1;
    int pattern_index = -1;

    std::weak_ptr<BohrNode> parent = std::weak_ptr<BohrNode>();
    std::weak_ptr<BohrNode> suffix_reference =
std::weak_ptr<BohrNode>();

    std::unordered_map<char, std::shared_ptr<BohrNode>> children;
    std::unordered_map<char, std::weak_ptr<BohrNode>> go;
};
```

bohr.cpp

```
#include "bohr.h"

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

BohrNode::BohrNode(std::weak_ptr<BohrNode> parent, char
symbol_to_parent)
    : parent(std::move(parent)),
symbol_to_parent(symbol_to_parent)
{
```

```
}

std::size_t BohrNode::CountStates() const {
    std::size_t result = children.size();
    for (const auto& record : children)
        result += record.second->CountStates();
    return result;
}

void BohrNode::AddChild(char arc, std::shared_ptr<BohrNode> child)
{
    children[arc] = std::move(child);
}

std::shared_ptr<BohrNode> BohrNode::GetChild(char arc) {
    return HasChild(arc) ? children.at(arc) : nullptr;
}

const std::shared_ptr<BohrNode>& BohrNode::GetChild(char arc)
const {
    return children.at(arc);
}

const std::unordered_map<char, std::shared_ptr<BohrNode>>&
BohrNode::GetChildren() const {
    return children;
}

std::weak_ptr<BohrNode> BohrNode::MakeTransition(char arc) {
    if (go.count(arc) == 0) {
        if (children.count(arc) != 0)
            go[arc] = children.at(arc);
        else if (IsRoot())
            go[arc] = root;
        else
            go[arc] = GetSuffixReference().lock()-
>MakeTransition(arc);
    }

    return go.at(arc);
```

```
}

std::weak_ptr<BohrNode> BohrNode::GetSuffixReference() {
    if (suffix_reference.lock() == nullptr) {
        if (IsRoot())
            suffix_reference = root;
        else if (parent.lock()->IsRoot())
            suffix_reference = parent;
        else
            suffix_reference = parent.lock()-
>GetSuffixReference().lock()->MakeTransition(symbol_to_parent);
    }

    return suffix_reference;
}


bool BohrNode::HasChild(char arc) const {
    return children.count(arc) != 0;
}

bool BohrNode::IsRoot() const {
    return parent.lock() == nullptr;
}

bool BohrNode::IsFinal() const {
    return pattern_index >= 0;
}

void BohrNode::SetPatternIndex(int pattern_index_) {
    pattern_index = pattern_index_;
}

int BohrNode::PatternIndex() const {
    return pattern_index;
}


std::shared_ptr<BohrNode> BohrNode::BuildForest(
    const std::vector<std::string>& patterns
```

```
) {
    root = std::make_shared<BohrNode>();

    for (int pattern_index = 0; pattern_index < patterns.size(); +
+pattern_index) {
        auto current = root;
        const std::string& pattern = patterns[pattern_index];

        for (int i = 0; i < pattern.size(); ++i) {
            if (!current->HasChild(pattern[i]))
                current->AddChild(pattern[i],
std::make_shared<BohrNode>(current, pattern[i]));
            current = current->GetChild(pattern[i]);
        }

        current->SetPatternIndex(pattern_index);
    }

    return root;
}
```

aho_corasick.hpp

```
#include <string>
#include <vector>
#include <memory>
#include <set>

#include "bohr.h"


struct AhoPair {
    std::size_t start_index;
    std::size_t pattern_index;

    bool operator<(const AhoPair& other) const {
        return std::tie(start_index, pattern_index) <
std::tie(other.start_index, other.pattern_index);
    }
};
```

```
static std::set<AhoPair> AhoCorasick(
    const std::shared_ptr<BohrNode>& root, const
std::vector<std::string>& patterns,
    const std::string& text
) {
    std::set<AhoPair> result;
    std::weak_ptr<BohrNode> current = root;

    for (int i = 0; i < text.size(); ++i) {
        std::weak_ptr<BohrNode> next = current.lock()-
>GetChild(text[i]);

        while (!next.lock()) {
            if (current.lock()->IsRoot()) {
                next = current;
                break;
            }

            current = current.lock()->GetSuffixReference();
            next = current.lock()->GetChild(text[i]);
        }

        current = next;
        while (!next.lock()->IsRoot()) {
            if (next.lock()->IsFfinal()) {
                result.insert(
                        {i - patterns[next.lock()-
>PatternIndex()].size() + 2,
                         static_cast<std::size_t>(next.lock()-
>PatternIndex()) + 1}
                );
            }
            next = next.lock()->GetSuffixReference();
        }
    }

    return result;
}
```