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: «Ахо-Корасик»

Студент гр. 3343 _____ Бондаренко Ф. А.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2025

Цель работы.

Изучить алгоритм Ахо-Корасик для нахождения вхождения всех образцов в строке. Решить задачу поиска набора подстрок в тексте. Реализовать программу для нахождения шаблонов с масками.

Задание.

Задание 1.

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

Вторая - число n ($1 \leq n \leq 3000$), каждая следующая из n строк содержит шаблон из набора $P=\{p1, \ldots, pn\} 1 \leq |pi| \leq 75$,

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

Индивидуальный вариант.

Вариант 7: вывод графического представления автомата.

Выполнение работы.

Описание реализованных функций для алгоритма Ахо-Корасик:

- *struct Vertex* — структура вершины бора (префиксного дерева). В реализации алгоритма является состоянием автомата. В поля структуры входят:
  - *parentIdx* — индекс вершины родителя в боре;
  - *symbol* — символ, по которому пришли в текущую вершину;
  - *suffixLink* — суффиксная ссылка;
  - *outputLink* — конечная (терминальная) ссылка;
  - *isTerminal* — флаг, отвечающий за то, что вершина в боре — конечная: т. е. на ней заканчивается слово, что было добавлено в префиксное дерево;
  - *patternIndices* — вектор индексов шаблонов, что заканчиваются в данной вершине: нужен для разрешения конфликта состояния, при котором в префиксное дерево было добавлено два (или более) одинаковых по значению шаблона;
  - *next* — *std::map*, что хранит прямые переходы по символу: т. е. индексы вершин, в которые можно попасть напрямую, без использования суффиксальных ссылок;
  - *autoMove* — std::map, что хранит сохраненные переходы по символу: данные переходы могут использовать суффиксальные ссылки.

- *class Trie* — префиксное дерево (бор), которое используется в качестве конечного детерминированного автомата в алгоритме Ахо-Корасик:
  - *vertices* — вектор указателей вершин бора.
  - *patternsCount* — число шаблонов, что используется на данный момент в дереве.
  - *void push(const std::string &pattern, bool record)* — функция добавления строки (шаблона) в бор.

- *bool find(const std::string &pattern, bool record)* — функция поиска шаблона в дереве.
- *getVerticesCount*, *getPatternsCount*, *getPatternsCount* — геттеры для получения информации о префиксном дереве.

- *namespace ahocorasic* — пространство имен, отвечающее за функции для алгоритма Ахо-Корасик:
  - *std::vector<std::pair<int, int>> search(const std::string& text, const std::vector<std::string>& patterns, bool record)* — функция стандартного поиска набора шаблонов в тексте по алгоритму Ахо-Корасика. Использует префиксное дерево в качестве детерминированного конечного автомата (конечное множество состояний, из каждого состояния возможен только один переход).
  - *std::vector<int> search(const std::string& text, const std::string& pattern, char wildcard, bool record)* — функция для поиска шаблонов с маской. Использует стандартную функцию для поиска шаблонов в тексте.
  - *buildAutomaton(Trie& trie, bool record)* — функция для построение автомата из заданного бора: "пробрасывает" все ссылки в боре. Необходима для индивидуального задания.
- В безымянном пространстве имен (необходимое для инкапсуляции) прописаны следующие функции:
  - *int getSuffixLink(Trie& trie, int idx_V, bool record)* — функция для получения суффиксальной ссылки.
  - *int getOutputLink(Trie& trie, int idx_V, bool record)* — функция для получения конечной (терминальной) ссылки.
  - *getAutoMove(Trie& trie, int idx_V, char symbol, bool record)* — функция для задания перехода из одного состояния автомата в другое по заданному символу.

- *Trie initTrie(const std::vector<std::string>& patterns, bool record)* — функция для инициализации префиксного дерева: добавляет набор шаблонов в бор.
- *std::pair<std::vector<std::string>, std::vector<int>> makePartition(const std::string& pattern, char wildcard, bool record)* — функция для разбиения шаблона с масками на безмасочные куски, с сохранением индексов вхождений этих кусков.

- *namespace visualize* — пространство имен, служащее для графического представления автомата:
  - *void automatonToDot(Trie& trie, const std::string& filename)* — функция, что "переводит" автомат в картинку. Для отрисовки использовалась библиотека *Graphviz*.
- В данном пространстве (*visualize*) так же присутствует еще и безымянное:
  - *char* createVertexLabel(const Vertex& vertex, int vertexId)* — функция для задачи названия для вершины графа.
  - *Agnode_t* createNode(Agraph_t* g, int id, const Vertex& v)* — функция для создания узла (вершины) графа.
  - *void addEdge(Agraph_t* g, Agnode_t* from, Agnode_t* to, const char* color, const char* style, const char* label = nullptr)* — функция добавления ребра графа.

Описание алгоритма Ахо-Корасик:

Основная задача алгоритма — построения конечного детерминированного автомата: т. е. такой математической модели, в которой определено:

- Конечное множество состояний.
- Функция переходов: которая для каждой пары "текущего состояния и критерия перехода (символа)" однозначно определяет следующее состояние.

- Начальное состояние.
- Множество конечных (принимающих) состояний.

В алгоритме, переход из состояний осуществляется по 2 параметрам — текущей вершине v и символу ch. по которому нам надо сдвинуться из этой вершины.

Зачем нужен такой автомат?

Для нахождения всех строк бора, что являются суффиксами текущей подстроки обрабатываемого текста. После этого, осуществляется переход из состояния автомата v в состояние u по следующему символу из текста.

Для создания автомата, необходимо ввести суффиксальные ссылки:

- *failure links* — обычная суффиксальная ссылка — для узла, соответствующего строке S, суффиксальная ссылка указывает на узел, соответствующий наибольшему собственному суффиксу строки S, который также является префиксом какого-либо шаблона в боре. Однако не обязательно, что эта вершина будет соответствовать концу какого-либо шаблона.
- *terminal links* — "хорошая" (терминальная) суффиксальная ссылка — дополнительная оптимизация, которая напрямую связывает узлы с конечными состояниями шаблонов, достижимые через суффиксальные ссылки.

Оба вида ссылок позволяют эффективно "перепрыгивать" в правильное место в дереве при обработке текста, когда очередное совпадение невозможно.

Проблема обычных суффиксальных ссылок: при проверки всех суффиксов, необходимо пройти всю цепочку и проверить каждую вершину на

признак конца шаблона. Это может быть неэффективно, особенно если в боре много "промежуточных" вершин. Для строки из состояния v можно найти v.length() суффиксов, а переход из состояний может просто увеличивать на 1 длину этой строки. Следовательно, получается квадратичная асимптотика относительно N — длины текста.

Для решения этой проблемы как раз и вводят конечные суффиксальные ссылки. Число "скачков" при использование таких ссылок уменьшится и станет пропорционально количеству искомых вхождений, оканчивающихся в этой позиции.

Алгоритм нахождения failure link:

Для вычисления суффиксальной ссылки текущей вершины v (подстроки s[i..j]) необходимо:

- Перейти в родителя v — parent. Запомнить символ symbol, по которому из parent можно попасть в v.
- Из parent перейти по суффиксальной ссылке в вершину relative (предположим, что она найдена).
- После перехода текущее состояние — такой префикс, что является суффиксом для подстроки s[i..j-1].
- Если из этого префикса есть переход по символу symbol в вершину u, то значение суффиксальной ссылки для вершины v будет вершина u (по-сути применение 1ого свойства префикс функции из КМП).
- Иначе: из relative перейти по суффиксальной ссылке, повторяя предыдущие пункты (используем 2ое свойство префикс функции).

Свойства префикс функции:

Пусть P(T, i) = k, тогда:

- Если T[i + 1] == T[k + 1], то P(T, i + 1) = k + 1
- T[1..P(T, k)] — префикс-суффикс строки T[1..i]

Алгоритм получения terminal link:

- Аналогичен алгоритму получения для failure link, но добавляется условие на проверку того, что вершина, в которую мы пришли — конечная (терминальная).

По функции получения суффиксальной ссылки можно построить функцию перехода из одного состояния автомата в другое: при переходе необходимо найти вершину u, которая обозначает наидлиннейшую строку, состоящую из суффикса строки v (возможно нулевого) + символа ch. Если такого в боре нет, то идем в корень.

Алгоритм обычного поиска:

- Задать состояние автомата u (т.е. вершину в боре).
- Итерироваться по символам текста, изменяя состояние автомата: т.е. переходить из текущей вершины u в вершину v по текущему символу текста.
- На этой же итерации проходить по всем конечным ссылкам.

Для того чтобы найти все вхождения в текст заданного шаблона с масками Q, необходимо обнаружить вхождения в текст всех его безмасочных кусков. Введем обозначения:

- $\{Q_1,\dots,Q_k\}$ — набор подстрок Q, разделенных масками.
- $\{l_1,\dots,l_k\}$ — набор стартовых позиций подстрок из набора $\{Q_1,\dots,Q_k\}$ в Q.
- C — массив; C[i] — количество встретившихся в тексте безмасочных подстрок шаблона, который начинается в тексте на позиции i.

Тогда появление подстроки Q_i в тексте на позиции j будет означать возможное появление шаблона на позиции $j-l_i+1$.

Алгоритм поиска шаблона с масками:

- Используя алгоритм Ахо-Корасик, находим безмасочные подстроки шаблона Q: когда находим Qi в тексте T на позиции j, увеличиваем на единицу $C[j-l_i+1]$
- Каждое i, для которого C[i] == k, является стартовой позицией появления шаблона Q в тексте.

Оценка сложности алгоритма.

Обычного алгоритма Ахо-Корасик:

- По времени: $O((M+N)\cdot\log_2 k+t)$, где M — число вершин в боре (сумма длин всех уникальных шаблонов), N — длина текста, k — размер алфавита, t — количество вхождений шаблонов в текст:
  - из-за std::map:
    - Добавление переходов при построении бора: $O(M\cdot log_2 k)$.
    - Поиск по тексту: для каждого символа выполняется *getAutoMove()*, что дает: $O(N\cdot log_2 k)$.
    - Обработка всех найденных вхождений шаблонов в текст (прохождение по цепочке конечных ссылок при поиске): $O(t)$
- По памяти: $O(M+N)$, где M — число вершин в боре (сумма длин всех уникальных шаблонов), N — длина текста: количество вычислений переходов автомата пропорционально длине строки.

Алгоритма Ахо-Корасика для нахождения шаблонов с маской:

- По времени: $O((M+N)\cdot\log_2 k+t+N)$, где M — суммарная длина всех частей шаблона после разбиения (без учета wildcard), N — длина текста, k — размер алфавита, t — количество всех найденных вхождений подстрок-частей в тексте:
  - Разбиение шаблона на части: $O(P)$ (где P = M + число джокеров в шаблоне).
  - Построение автомата для частей шаблона: $O(M\cdot log_2 k)$.

- Поиск всех вхождений частей шаблона с помощью автомата: $O(N \cdot log_2 k + t)$.
- Агрегация результатов и финальная проверка полных совпадений: $O(t+N)$.

- По памяти: $O(M+N+t)$, где M — суммарная длина всех частей шаблона после разбиения (без учета wildcard), N — длина текста, t — количество всех найденных вхождений подстрок-частей в тексте:
  - Память для хранения автомата: $O(M)$.
  - Память для массива счетчиков (count): $O(N)$.
  - Память для хранения позиций всех промежуточных совпадений: $O(t)$.

Индивидуальное задание:

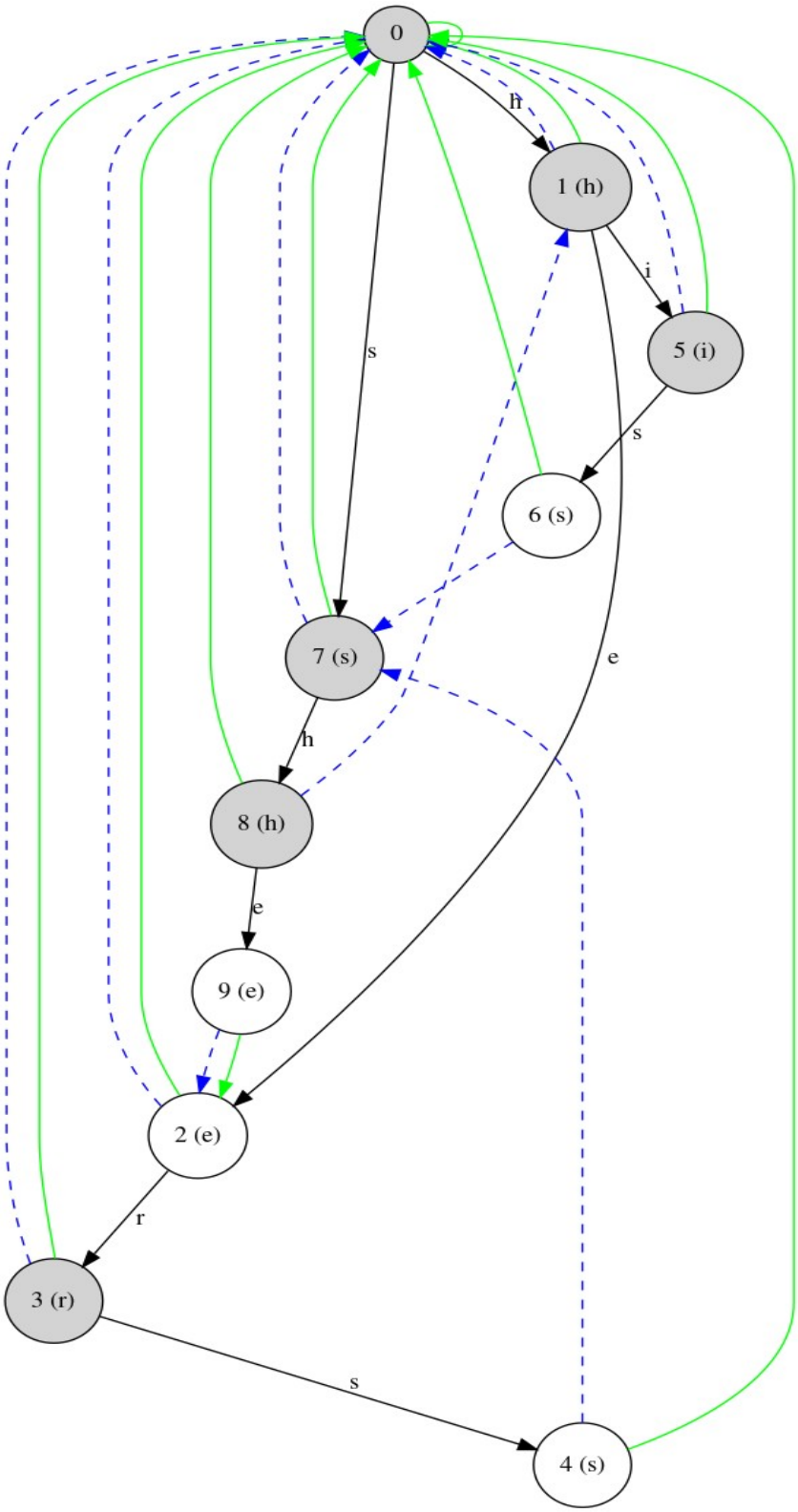

Рисунок 1 — графическое представление автомата.

- Черные сплошные стрелки — прямые переходы по символам.
- Синие пунктирные стрелки — суффиксальные ссылки.
- Зеленые сплошные стрелки — терминальные ссылки.

Тестирование.

Входные данные	Выходные данные	Комментарий
NTAG 3 TAGT TAG T	2 2 2 3	Успех для стандартного поиска
ushers 4 hers he his she	2 4 3 1 3 2	Успех для стандартного поиска
aaaaaaa 3 aaa aa a	1 1 1 2 1 3 2 1 2 2 2 3 3 1 3 2 3 3 4 1 4 2 4 3 5 1 5 2 5 3 6 2 6 3 7 3	Успех для стандартного поиска
ACTANCA A$$A$	1	Успех для поиска с масками

$		
ABVGDEV ??V ?	1 5	Успех для поиска с масками

Вывод.

Изучен принцип работы алгоритма Ахо-Корасик. Реализованы программы для нахождения набора шаблонов в тексте и нахождения шаблона с масками.

ПРИЛОЖЕНИЕ А

Файл *main.cpp*:

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <map>
#include <string>
#include <utility>
#include <vector>

#include "AhoCorasick.hpp"
#include "Trie.hpp"
#include "Visualize.hpp"

#define VISUAL_DIR "./pic"

void defaultSearch() {
  std::string text;

  int n;
  std::cin >> text >> n;

  std::vector<std::string> patterns(n);
  for (int i = 0; i < n; i++) {
    std::cin >> patterns[i];
  }

  std::vector<std::pair<int, int>> result =
      ahocorasick::search(text, patterns, false);

  std::sort(result.begin(), result.end());
  for (const auto& match : result) {
      std::cout << match.first + 1 << ' ' << match.second + 1 <<
std::endl;
  }
}

void wildcardSearch() {
  std::string text;
  std::string pattern;
  char wildcard;

  std::cin >> text >> pattern >> wildcard;

     std::vector<int> result = ahocorasick::search(text, pattern,
wildcard, false);

  std::sort(result.begin(), result.end());
  for (const auto& match : result) {
    std::cout << match + 1 << std::endl;
  }
}

void defaultStep() {
  std::string text;

  int n;
```

```
  std::cin >> text >> n;

  std::vector<std::string> patterns(n);
  for (int i = 0; i < n; i++) {
    std::cin >> patterns[i];
  }

  std::vector<std::pair<int, int>> result =
      ahocorasick::search(text, patterns, true);
  std::sort(result.begin(), result.end());
  for (const auto& match : result) {
      std::cout << match.first + 1 << ' ' << match.second + 1 <<
std::endl;
  }
}

void wildcardStep() {
  std::string text;
  std::string pattern;
  char wildcard;

  std::cin >> text >> pattern >> wildcard;

     std::vector<int>  result  =  ahocorasick::search(text,  pattern,
wildcard, true);

  std::sort(result.begin(), result.end());
  for (const auto& match : result) {
    std::cout << match + 1 << std::endl;
  }
}

void visualizeAutomaton() {
  int n;
  std::cin >> n;

  std::vector<std::string> patterns(n);
  for (int i = 0; i < n; i++) {
    std::cin >> patterns[i];
  }

  Trie trie;
  for (const auto& pattern : patterns) {
    trie.push(pattern, false);
  }

  ahocorasick::buildAutomaton(trie, false);
       visualize::automatonToDot(trie,   std::string(VISUAL_DIR)   +
"/automaton");
}

int main(int argc, char* argv[]) {
  if (argc < 2) {
    return 1;
  }

  std::map<std::string, std::function<void()>> modes = {
      {"run-default", defaultSearch},
      {"run-wildcard", wildcardSearch},
```

```
      {"step-default", defaultStep},
      {"step-wildcard", wildcardStep},
      {"visualize", visualizeAutomaton}};

  std::string mode = argv[1];

  auto it = modes.find(mode);
  if (it != modes.end()) {
    it->second();
  }

  return 0;
}
```

Файл *Vertex.hpp*:

```
#ifndef VERTEX_HPP_
#define VERTEX_HPP_

#include <map>
#include <vector>

struct Vertex {
  int parentIdx;
  char symbol;

  int suffixLink;
  int outputLink;

  bool isTerminal;
  std::vector<int> patternIndices;

  std::map<char, int> next;
  std::map<char, int> autoMove;

  Vertex(bool isTerminal, char symbol)
      : parentIdx(-1),
        symbol(symbol),
        suffixLink(-1),
        outputLink(-1),
        isTerminal(isTerminal),
        patternIndices() {}
};

#endif  // VERTEX_HPP_
```

Файл *Trie.hpp*:

```
#ifndef TRIE_HPP_
#define TRIE_HPP_

#include <string>
#include <vector>

struct Vertex;

class Trie {
 private:
  std::vector<Vertex *> vertices;
  int patternsCount;
```

```
 public:
  Trie();

  void push(const std::string &pattern, bool record);
  bool find(const std::string &pattern, bool record);

  int getVerticesCount() { return vertices.size(); }
  int getPatternsCount() { return patternsCount; }
  Vertex &getVertex(int num) { return *vertices[num]; };

  ~Trie();
};

#endif  // TRIE_HPP_
```

Файл *AhoCorasick.hpp*:

```
#ifndef AHOCORASICK_HPP_
#define AHOCORASICK_HPP_

#include <string>
#include <utility>
#include <vector>

class Trie;

namespace ahocorasick {

std::vector<std::pair<int, int>> search(
    const std::string& text, const std::vector<std::string>& patterns,
    bool record);

std::vector<int> search(const std::string& text, const std::string&
pattern,
                        char wildcard, bool record);

void buildAutomaton(Trie& trie, bool record);

}  // namespace ahocorasick

#endif  // AHOCORASICK_HPP_
```

Файл *Visualize.hpp*:

```
#ifndef VISUALIZE_HPP_
#define VISUALIZE_HPP_

#include "Trie.hpp"

namespace visualize {
void automatonToDot(Trie& trie, const std::string& filename);
}  // namespace visualize

#endif  // VISUALIZE_HPP_
```

Файл *Trie.cpp*:

```
#include "Trie.hpp"

#include <iostream>
#include <map>
```

```
#include "Vertex.hpp"

Trie::Trie() : vertices(1, new Vertex(false, '\0')), patternsCount(0)
{}

void Trie::push(const std::string& pattern, bool record) {
  if (record) {
    std::cout << std::endl;
    std::cout << "================================" << std::endl;
    std::cout << "[Trie::push] вход" << std::endl;
    std::cout << "pattern:  [" << pattern << "]" << std::endl;
  }

  int idx_V = 0;
  for (char symbol : pattern) {
    if (record) {
      std::cout << "--------------------------------" << std::endl;
      std::cout << "symbol: [" << symbol << "]" << std::endl;
      std::cout << "idx_V:  [" << idx_V << "]" << std::endl;
       std::cout << "vertices.size(): [" << vertices.size() << "]" <<
std::endl;
    }

    if (!vertices[idx_V]->next.count(symbol)) {
      if (record) {
           std::cout << "Нет прямого перехода из текущей вершины" <<
std::endl;
        std::cout << "Создаем новую вершину" << std::endl;
      }

      Vertex* u = new Vertex(false, symbol);
      u->parentIdx = idx_V;
      vertices.push_back(u);
      vertices[idx_V]->next[symbol] = vertices.size() - 1;
    }

    idx_V = vertices[idx_V]->next[symbol];
    if (record) {
      std::cout << "idx_V:         [" << idx_V << "]" << std::endl;
        std::cout << "vertices size: [" << vertices.size() << "]" <<
std::endl;
      std::cout << "--------------------------------" << std::endl;
    }
  }

  patternsCount++;
  vertices[idx_V]->isTerminal = true;
  vertices[idx_V]->patternIndices.push_back(patternsCount - 1);

  if (record) {
    std::cout << "[Trie::push] выход" << std::endl;
     std::cout << "pattern: [" << pattern << "] добавлен в бор" <<
std::endl;
    std::cout << "================================" << std::endl;
    std::cout << std::endl;
  }
}

bool Trie::find(const std::string& pattern, bool record) {
```

```
  if (record) {
    std::cout << std::endl;
    std::cout << "================================" << std::endl;
    std::cout << "Поиск шаблона в боре" << std::endl;
    std::cout << "pattern: [" << pattern << "]" << std::endl;
  }

  int idx_V = 0;
  for (char symbol : pattern) {
    if (record) {
      std::cout << "--------------------------------" << std::endl;
      std::cout << "symbol: [" << symbol << "]" << std::endl;
      std::cout << "idx_V:  [" << idx_V << "]" << std::endl;
       std::cout << "vertices.size(): [" << vertices.size() << "]" <<
std::endl;
    }

    if (!vertices[idx_V]->next.count(symbol)) {
      if (record) {
         std::cout << "Нет перехода по символу - шаблон не найден" <<
std::endl;
        std::cout << "================================" << std::endl;
        std::cout << std::endl;
      }
      return false;
    }

    idx_V = vertices[idx_V]->next[symbol];
  }

  bool found = vertices[idx_V]->isTerminal;
  if (record) {
      std::cout << (found ? "Шаблон найден: достигнута терминальная
вершина"
                          : "Шаблон не найден: достигнута нетерминальная
вершина")
                << std::endl;
    std::cout << "================================" << std::endl;
    std::cout << std::endl;
  }
  return found;
}

Trie::~Trie() {
  for (auto idx_V : vertices) delete idx_V;
}
```

Файл *AhoCorasick.cpp*:

```
#include "AhoCorasick.hpp"

#include <iostream>

#include "Trie.hpp"
#include "Vertex.hpp"

namespace ahocorasick {

namespace {
```

```
int getAutoMove(Trie& trie, int idx_V, char symbol, bool record);

int getSuffixLink(Trie& trie, int idx_V, bool record) {
  Vertex& vertex_V = trie.getVertex(idx_V);
  if (record) {
    std::cout << std::endl;
    std::cout << "================================" << std::endl;
    std::cout << "[getSuffixLink] вход" << std::endl;
    std::cout << "idx_V:      [" << idx_V << "]" << std::endl;
      std::cout << "symbol:        [" << vertex_V.symbol << "]" <<
std::endl;
      std::cout << "parentIdx:   [" << vertex_V.parentIdx << "]" <<
std::endl;
      std::cout << "suffixLink: [" << vertex_V.suffixLink << "]" <<
std::endl;
    std::cout << "--------------------------------" << std::endl;
  }

  if (vertex_V.suffixLink == -1) {
    if (record) {
          std::cout << "Суффиксной ссылки для этой вершины нет" <<
std::endl;
    }

    if (idx_V == 0 || vertex_V.parentIdx == 0) {
      if (record) {
         std::cout << "Попали либо в корень, либо в потомка корня" <<
std::endl;
      }

      vertex_V.suffixLink = 0;
    } else {
      if (record) {
           std::cout << "Переходим по суффиксной ссылке родителя" <<
std::endl;
      }

      vertex_V.suffixLink =
             getAutoMove(trie, getSuffixLink(trie, vertex_V.parentIdx,
record),
                         vertex_V.symbol, record);
    }
  }

  if (record) {
    std::cout << "[getSuffixLink] выход" << std::endl;
      std::cout << "suffixLink: [" << vertex_V.suffixLink << "]" <<
std::endl;
    std::cout << "================================" << std::endl;
    std::cout << std::endl;
  }
  return vertex_V.suffixLink;
}

int getOutputLink(Trie& trie, int idx_V, bool record) {
  Vertex& vertex_V = trie.getVertex(idx_V);
  if (record) {
    std::cout << std::endl;
    std::cout << "================================" << std::endl;
```

```
    std::cout << "[getOutputLink] вход" << std::endl;
    std::cout << "idx_V:      [" << idx_V << "]" << std::endl;
      std::cout << "outputLink: [" << vertex_V.outputLink << "]" <<
std::endl;
    std::cout << "--------------------------------" << std::endl;
  }

  if (vertex_V.outputLink == -1) {
    if (record) {
       std::cout << "Конечная ссылка для этой вершины неопределена" <<
std::endl;
      std::cout << "--------------------------------" << std::endl;
    }

    int idx_U = getSuffixLink(trie, idx_V, record);
    if (record) {
       std::cout << "suffixLink для idx_V=[" << idx_V << "]" << " ->
idx_U=["
                 << idx_U << "]" << std::endl;
      std::cout << "--------------------------------" << std::endl;
    }

    if (idx_U == 0) {
      if (record) {
        std::cout << "Попали в корень" << std::endl;
      }

      vertex_V.outputLink = 0;
    } else {
      if (record) {
        std::cout << "Проверяем isTerminal у idx_U" << std::endl;
      }

      Vertex& vertex_U = trie.getVertex(idx_U);
      if (vertex_U.isTerminal) {
        if (record) {
           std::cout << "Найдена конеченая вершина -> outputLink=[" <<
idx_U
                     << "]" << std::endl;
        }

        vertex_V.outputLink = idx_U;
      } else {
        if (record) {
             std::cout << "Конечная вершина не найдена -> продолжаем
поиск через "
                       "рекурсию"
                     << std::endl;
        }
        vertex_V.outputLink = getOutputLink(trie, idx_U, record);
      }
    }
  }

  if (record) {
    std::cout << "[getOutputLink] выход" << std::endl;
      std::cout << "outputLink: [" << vertex_V.outputLink << "]" <<
std::endl;
    std::cout << "================================" << std::endl;
```

```
    std::cout << std::endl;
  }
  return vertex_V.outputLink;
}

int getAutoMove(Trie& trie, int idx_V, char symbol, bool record) {
  Vertex& vertex_V = trie.getVertex(idx_V);
  if (record) {
    std::cout << std::endl;
    std::cout << "================================" << std::endl;
    std::cout << "[getAutoMove] вход" << std::endl;
    std::cout << "idx_V:  [" << idx_V << "]" << std::endl;
    std::cout << "symbol: [" << symbol << "]" << std::endl;
    std::cout << "--------------------------------" << std::endl;
  }

  if (!vertex_V.autoMove.count(symbol)) {
    if (vertex_V.next.count(symbol)) {
      if (record) {
        std::cout << "Есть прямой переход -> next_state=["
                  << vertex_V.next[symbol] << "]" << std::endl;
      }

      vertex_V.autoMove[symbol] = vertex_V.next[symbol];
    } else {
      if (record) {
          std::cout << "Нет прямого перехода, идём по suffixLink" <<
std::endl;
      }

      vertex_V.autoMove[symbol] =
          (idx_V == 0) ? 0
                       : getAutoMove(trie, getSuffixLink(trie, idx_V,
record),
                                     symbol, record);
    }
  }

  if (record) {
    std::cout << "[getAutoMove] выход" << std::endl;
    std::cout << "next state: [" << vertex_V.autoMove[symbol] << "]"
              << std::endl;
    std::cout << "================================" << std::endl;
    std::cout << std::endl;
  }
  return vertex_V.autoMove[symbol];
}

Trie initTrie(const std::vector<std::string>& patterns, bool record) {
  if (record) {
    std::cout << std::endl;
    std::cout << "================================" << std::endl;
    std::cout << "[initTrie] вход" << std::endl;
      std::cout << "patterns size: [" << patterns.size() << "]" <<
std::endl;
    std::cout << "--------------------------------" << std::endl;
  }

  Trie trie;
```

```
  for (size_t i = 0; i < patterns.size(); ++i) {
    if (record) {
      std::cout << "patterns[" << i << "]: [" << patterns[i] << "]"
                << std::endl;
      std::cout << "--------------------------------" << std::endl;
    }
    trie.push(patterns[i], record);
  }

  if (record) {
    std::cout << "[initTrie] выход" << std::endl;
    std::cout << "vertices size: [" << trie.getVerticesCount() << "]"
              << std::endl;
    std::cout << "patterns count: [" << trie.getPatternsCount() << "]"
              << std::endl;
    std::cout << "================================" << std::endl;
    std::cout << std::endl;
  }
  return trie;
}

std::pair<std::vector<std::string>, std::vector<int>> makePartition(
    const std::string& pattern, char wildcard, bool record) {
  if (record) {
    std::cout << std::endl;
    std::cout << "================================" << std::endl;
    std::cout << "[makePartition] вход" << std::endl;
    std::cout << "pattern:  [" << pattern << "]" << std::endl;
    std::cout << "wildcard: [" << wildcard << "]" << std::endl;
    std::cout << "--------------------------------" << std::endl;
  }

  std::vector<std::string> pieces;
  std::vector<int> positions;

  std::string current;
  for (size_t i = 0; i < pattern.length(); ++i) {
    if (pattern[i] == wildcard) {
      if (!current.empty()) {
        pieces.push_back(current);
        int pos = int(i - current.length());
        positions.push_back(pos);

        if (record) {
          std::cout << "Кусок: [" << current << "]" << std::endl;
          std::cout << "pos:   [" << pos << "]" << std::endl;
                  std::cout  <<  "--------------------------------"  <<
std::endl;
        }

        current.clear();
      }
    } else {
      current.push_back(pattern[i]);
    }
  }
  if (!current.empty()) {
    pieces.push_back(current);
    int pos = int(pattern.size() - current.size());
```

```
    positions.push_back(pos);

    if (record) {
      std::cout << "Кусок: [" << current << "]" << std::endl;
      std::cout << "pos:   [" << pos << "]" << std::endl;
      std::cout << "--------------------------------" << std::endl;
    }
  }

  if (record) {
    std::cout << "[makePartition] выход" << std::endl;
     std::cout << "Общее число кусков без джокера: [" << pieces.size()
<< "]"
              << std::endl;
    std::cout << "================================" << std::endl;
    std::cout << std::endl;
  }

  return {pieces, positions};
}

}  // namespace

std::vector<std::pair<int, int>> search(
    const std::string& text, const std::vector<std::string>& patterns,
    bool record) {
  if (record) {
    std::cout << std::endl;
    std::cout << "================================" << std::endl;
    std::cout << "[search-default] вход" << std::endl;
    std::cout << "text:          [" << text << "]" << std::endl;
      std::cout << "patterns size: [" << patterns.size() << "]" <<
std::endl;
    std::cout << "--------------------------------" << std::endl;
  }

  Trie trie = initTrie(patterns, record);
  std::vector<std::pair<int, int>> result;
  int state = 0;

  for (size_t i = 0; i < text.length(); i++) {
    state = getAutoMove(trie, state, text[i], record);

    if (record) {
      std::cout << "Шаг i:  [" << i << "] " << std::endl;
      std::cout << "symbol: [" << text[i] << "]" << std::endl;
      std::cout << "state:  [" << state << "]" << std::endl;
      std::cout << "--------------------------------" << std::endl;
    }

    for (int u = state; u != 0; u = getOutputLink(trie, u, record)) {
      Vertex& vertex_U = trie.getVertex(u);
      if (vertex_U.isTerminal) {
        for (int patternIdx : vertex_U.patternIndices) {
          int symbolIdx = i - patterns[patternIdx].length() + 1;
          result.push_back({symbolIdx, patternIdx});

          if (record) {
            std::cout << "Шаблон совпал" << std::endl;
```

```
                std::cout << "symbolIdx:   [" << symbolIdx << "]" <<
std::endl;
                std::cout << "patternIdx: [" << patternIdx << "]" <<
std::endl;
          }
        }
      }
    }
  }

  if (record) {
    std::cout << "[search-default] выход" << std::endl;
    std::cout << "Общее число совпадений: [" << result.size() << "]"
              << std::endl;
    std::cout << "================================" << std::endl;
    std::cout << std::endl;
  }

  return result;
}

std::vector<int> search(const std::string& text, const std::string&
pattern,
                        char wildcard, bool record) {
  if (record) {
    std::cout << std::endl;
    std::cout << "================================" << std::endl;
    std::cout << "[search(wildcard)] вход" << std::endl;
    std::cout << "text:     [" << text << "]" << std::endl;
    std::cout << "pattern:  [" << pattern << "]" << std::endl;
    std::cout << "wildcard: [" << wildcard << "]" << std::endl;
    std::cout << "--------------------------------" << std::endl;
  }

  auto [pieces, positions] = makePartition(pattern, wildcard, record);

  size_t occurrencesCount = text.size() - pattern.size() + 1;
  std::vector<int> count(occurrencesCount, 0);

    std::vector<std::pair<int, int>> matches = search(text, pieces,
record);

  if (record) {
    std::cout << "--------------------------------" << std::endl;
       std::cout << "pieces.size(): [" << pieces.size() << "]" <<
std::endl;
     std::cout << "Число возможных стартов: [" << occurrencesCount <<
"]"
               << std::endl;
         std::cout << "Всего найдено вхождений кусочков: [" <<
matches.size() << "]"
               << std::endl;
    std::cout << "--------------------------------" << std::endl;
  }

  for (auto [pos, pieceIdx] : matches) {
    int start = pos - positions[pieceIdx];
    if (start >= 0 && (size_t)start < occurrencesCount) {
      count[start]++;
```

```
      if (record) {
            std::cout << "pieceIdx:       [" << pieceIdx << "]" <<
std::endl;
        std::cout << "pos:          [" << pos << "]" << std::endl;
        std::cout << "start:        [" << start << "]" << std::endl;
        std::cout << "count[start]: [" << count[start] << std::endl;
        std::cout << "--------------------------------" << std::endl;
      }
    } else if (record) {
      std::cout << "Игнорируем кусок" << std::endl;
      std::cout << "pieceIdx: [" << pieceIdx << "]" << std::endl;
      std::cout << "pos:      [" << pos << "] (выходит за границы)"
                << std::endl;
      std::cout << "--------------------------------" << std::endl;
    }
  }

  std::vector<int> result;
  for (size_t i = 0; i < occurrencesCount; ++i) {
    if (count[i] == (int)pieces.size()) {
      if (record) {
         std::cout << "Полное совпадение на старте: [" << i << "]" <<
std::endl;
        std::cout << "--------------------------------" << std::endl;
      }
      result.push_back(i);
    }
  }

  if (record) {
    std::cout << "[search(wildcard)] выход" << std::endl;
    std::cout << "Число полных совпадений: [" << result.size() << "]"
              << std::endl;
    std::cout << "================================" << std::endl;
    std::cout << std::endl;
  }

  return result;
}

void buildAutomaton(Trie& trie, bool record) {
  if (record) {
    std::cout << std::endl;
    std::cout << "================================" << std::endl;
    std::cout << "[buildAutomaton] вход" << std::endl;
    std::cout << "Число вершин: [" << trie.getVerticesCount() << "]"
              << std::endl;
    std::cout << "--------------------------------" << std::endl;
  }

  for (int i = 0; i < trie.getVerticesCount(); ++i) {
    if (record) {
       std::cout << "Обработка вершины с индексом [" << i << "]" <<
std::endl;
      std::cout << "--------------------------------" << std::endl;
    }

    getSuffixLink(trie, i, record);
```

```
    getOutputLink(trie, i, record);

    if (record) {
      std::cout << "Вершина [" << i << "] обработана" << std::endl;
      std::cout << "--------------------------------" << std::endl;
    }
  }

  if (record) {
    std::cout << "[buildAutomaton] выход" << std::endl;
        std::cout  <<  "Автомат  Ахо-Корасик  полностью  построен"  <<
std::endl;
    std::cout << "================================" << std::endl;
    std::cout << std::endl;
  }
}

}  // namespace ahocorasick
```

Файл *Visualize.cpp*:

```
#include "Visualize.hpp"

#include <graphviz/gvc.h>

#include <cstdio>
#include <cstring>
#include <vector>

#include "Vertex.hpp"

namespace visualize {

namespace {
const char* color_white = "white";
const char* color_lightgray = "lightgray";
const char* color_black = "black";
const char* color_blue = "blue";
const char* color_green = "green";

char* createVertexLabel(const Vertex& vertex, int vertexId) {
  char* buffer = new char[32];

  if (vertex.symbol) {
    sprintf(buffer, "%d (%c)", vertexId, vertex.symbol);
  } else {
    sprintf(buffer, "%d", vertexId);
  }

  return buffer;
}

Agnode_t* createNode(Agraph_t* g, int id, const Vertex& v) {
  char node_name[16];
  sprintf(node_name, "%d", id);

  Agnode_t* node = agnode(g, (char*)node_name, 1);
  char* label = createVertexLabel(v, id);
```

```
  agsafeset(node, (char*)"label", label, (char*)"");
  agsafeset(node, (char*)"shape", (char*)"circle", (char*)"");
  agsafeset(node, (char*)"style", (char*)"filled", (char*)"");
  agsafeset(node, (char*)"fillcolor",
                                v.isTerminal   ?   (char*)color_white   :
(char*)color_lightgray,
            (char*)"");

  delete[] label;

  return node;
}

void addEdge(Agraph_t* g, Agnode_t* from, Agnode_t* to, const char*
color,
             const char* style, const char* label = nullptr) {
  Agedge_t* edge = agedge(g, from, to, nullptr, 1);

  agsafeset(edge, (char*)"color", (char*)color, (char*)"");
  agsafeset(edge, (char*)"style", (char*)style, (char*)"");

  if (label) agsafeset(edge, (char*)"label", (char*)label, (char*)"");
}

}  // namespace

void automatonToDot(Trie& trie, const std::string& filename) {
  GVC_t* gvc = gvContext();
  Agraph_t* g = agopen((char*)"AhoCorasick", Agdirected, nullptr);

  agsafeset(g, (char*)"rankdir", (char*)"TB", (char*)"");

  std::vector<Agnode_t*> nodes;
  for (int i = 0; i < trie.getVerticesCount(); ++i) {
    nodes.push_back(createNode(g, i, trie.getVertex(i)));
  }

  for (int i = 0; i < trie.getVerticesCount(); ++i) {
    Vertex& v = trie.getVertex(i);
    for (auto& [sym, next] : v.next) {
      char lbl[2] = {sym, '\0'};
      addEdge(g, nodes[i], nodes[next], color_black, "solid", lbl);
    }
  }

  for (int i = 1; i < trie.getVerticesCount(); ++i) {
    if (trie.getVertex(i).suffixLink != -1) {
            addEdge(g,  nodes[i],  nodes[trie.getVertex(i).suffixLink],
color_blue,
              "dashed");
    }
  }

  for (int i = 0; i < trie.getVerticesCount(); ++i) {
    if (trie.getVertex(i).outputLink != -1) {
            addEdge(g,  nodes[i],  nodes[trie.getVertex(i).outputLink],
color_green,
              "solid");
    }
```

```
  }

  char output[256];
  sprintf(output, "%s.png", filename.c_str());
  gvLayout(gvc, g, (char*)"dot");
  gvRenderFilename(gvc, g, (char*)"png", output);

  gvFreeLayout(gvc, g);
  agclose(g);
  gvFreeContext(gvc);
}

}  // namespace visualize
```

Файл *Makefile*:

```
CXX=g++
CXXFLAGS=-std=c++20 -Wall -Wextra -pedantic -g -O0
LDFLAGS=-lcgraph -lgvc
EXT=cpp

SRC_DIR=src
OBJ_DIR=obj
VISUAL_DIR=pic

SRC_FILES=$(wildcard $(SRC_DIR)/*.$(EXT))
OBJ_FILES=$(patsubst $(SRC_DIR)/%.$(EXT),$(OBJ_DIR)/%.o,$(SRC_FILES))

EXEC=$(OBJ_DIR)/exec
INCLUDES=-Iinclude

$(shell mkdir -p $(OBJ_DIR))
$(shell mkdir -p $(VISUAL_DIR))

all: $(EXEC)

$(EXEC) : $(OBJ_FILES)
	$(CXX) $(CXXFLAGS) $(INCLUDES) -o $@ $^ $(LDFLAGS)

$(OBJ_DIR)/%.o : $(SRC_DIR)/%.$(EXT)
	$(CXX) $(CXXFLAGS) $(INCLUDES) -c $< -o $@

run-default: $(EXEC)
	./$(EXEC) run-default

run-wildcard: $(EXEC)
	./$(EXEC) run-wildcard

step-default: $(EXEC)
	./$(EXEC) step-default

step-wildcard: $(EXEC)
	./$(EXEC) step-wildcard

visualize: $(EXEC)
	./$(EXEC) visualize

valgrind: $(EXEC)
	@valgrind   --leak-check=full   --show-leak-kinds=all   --track-
origins=yes --error-exitcode=1 ./$(EXEC) run-default
```

```
clean:
	rm -rf $(OBJ_DIR)/*.o $(EXEC)

clean-visual:
	rm -rf $(VISUAL_DIR)/*

.PHONY: all run-default run-wildcard step-default step-wildcard
valgrind clean clean-visual
```