**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. 8382 |  | Мирончик П.Д. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2020

**Задание**

Разработайте программу,  решающую задачу точного поиска набора образцов.  
  
**Вход:**  
Первая строка содержит текст *T* ( ).  
Вторая - число *n* (), каждая следующая из *n* строк содержит шаблон из набора .  
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

Вар. 5. Вычислить максимальное количество дуг, исходящих из одной вершины в боре; вы́резать из строки поиска все найденные образцы и вывести остаток строки поиска.

# Структуры данных

*#define ALPHABET\_SIZE 32*

Размер алфавита - разница между первым возможным символом алфавита и последним.

*class Node*

Вершина в дереве алгоритма.

*Node\*Node::son[ALPHABET\_SIZE]*

Массив дочерних вершин. Если вершина *son[c]* не равна *nullptr*, то ребенок есть.

*Node\*Node::go[ALPHABET\_SIZE]*

Переходы их данной вершины по символу (кеш для функции *Axo::getLink*).

*Node\*Node::parent*

Родительская вершина.

*Node\*Node::suffLink*

Ссылка на подходящий для текущего пути (от *root* до вершины) суффикс.

*Node\*Node::up*

Ближайшая сжатая ссылка (указывающая на ближайший суффикс, последний символ которого который является терминалом).

*char Node::charToParent*

Символ между родительской вершиной и текущей.

*bool Node::isLeaf*

Является ли вершина терминальной.

*vector<int> Node::leafPatternNumber*

Список паттернов, кончающихся в данной вершине.

*int Node::index*

Расстояние от вершины до корня.

*int Node::sonsCount*

Количество дуг, которые выходят из вершины.

*class Axo*

Основной класс, занимающийся обработкой данных.

*int Axo::patternsCount*

Количество обработанных паттернов (используется в модифицированном алгоритме).

*Node\*Axo::root*

Корневая вершина дерева.

*Node\*Axo::getSuffLink(Node \*v)*

Функция получения суффикса для вершины *v*. Результат функции кешируется.

*@param v* Вершина, суффикс которой необходимо найти.

*@return* Вершина, являющаяся последним символом максимального суффикса.

*Node\*Axo::getLink(Node \*v, char c)*

Найти максимальный суффикс *[v]+c*. Рассчитанные значения кешируются.

*@param v* Вершина, от коорой строится путь.

*@param c* Символ пути.

*@return* Вершина - последний символ максимального суффикса.

*Node\*Axo::getUp(Node \*v)*

Найти сжатую ссылку для вершины v. Полученные значения кешируются.

*@param v* Вершина, для которой необходимо найти сжатую ссылку.

*@return* Сжатая ссылка для вершины *v*.

*void Axo::addString(const string &s, int number)*

Добавить паттерн в дерево

*@param s* Паттерн

*@param number* Идентификатор паттерна. В случае с добавлением паттернов модифицированного алгоритма в роли идентификатора выступает смещение фрагмента относительно начала паттерна.

*void Axo::setJokerString(const string &s, char joker)*

Заполнить дерево паттерном, использующим символ джокера.

*@param s* Паттерн.

*@param joker* Символ джокера.

*void Axo::processJoker(const string &s, const string &P)*

Обработать строку s при помощи предварительно заполненного дерева модифицированным алгоритмом.

*@param s* Строка, в которой производится поиск.

*@param P* Применяемый паттерн. Данный паттерн не добавляется в дерево, используется лишь его длина.

*vector<pair<int, int>> Axo::process(const string &s)*

Обработать строку s стандартным алгоритмом, используя предзаполненное дерево паттернов.

*@param s* Строка, в которой производится поиск.

*Node\*Axo::findMaxNode()*

Найти вершину, из которой выходит наибольшее количество дуг.

*@return* Вершина, из которой выходит наибольшее количество дуг.

*Node\*Axo::findMaxNode(Node\* v)*

Найти вершину, из которой выходит наибольшее количество дуг.

*@param v* Стартовая вершина - корень дерева, по которому производится поиск.

*@return* Вершина, из которой выходит наибольшее количество дуг.

*int Axo::getSonsCount(Node\* v)*

Получить количество дуг, которые идут из вершины.

*@param v* Вершина, количество дуг из которой необходимо найти.

*@return* Количество дуг, идущих из вершины v

*static bool Axo::cmp(const pair<int, int> &a, const pair<int, int> &b)*

Функция сравнения результатов поиска стандартным способом.

*@param a* Первый элемент

*@param b* Второй элемент

*@return* Меньше ли первый элемент второго

**Алгоритм**

По переданным паттернам строится бор. Каждая вершина бора имеет суффиксную ссылку, сжатую ссылку и сжатую ссылку (это основные существенные моменты для алгоритма).

*|v|* - путь от корня бора до вершины v.

*Суффиксная ссылка* для вершины *v* – это ссылка на такую вершину *u*, что *|u|* является суффиксом *|v|*, при этом нет такой вершины *w*, что *|w|* является суффиксом для *|v|* и *|u| < |w|*.

Сжатая ссылка для вершины *v* – это ближайшая суффиксная ссылка (в цепочке суффиксных ссылок), которая указывает на вершину, являющуюся терминалом.

*getLink(v, c)* – вспомогательная функция, которая ищет такую вершину u, что:

1. *|u| == |w| + c*

2*. |w|* - максимальный суффикс вершины *v*

3. Если такой вершины нет, возвращается *root*

Далее следует обработка строки. На каждом шаге известна текущая вершина бора и индекс текущего символа в строке (а также сам символ).

Инициализация:

*Node\* cur = root;*

На каждом шаге:

1. *cur = getLink(cur, c)*; - ищем максимальный путь, состоящий из суффикса *|cur|* и символа *|c|*

2. Итерируем по цепочке сжатых ссылок, начиная с cur. Вершины, через которые пройдет алгоритм, будут являться терминальными, остается только вычесть из текущего индекса курсора в строке индекс (расстояние от корня) очередной вершины, и получится индекс очередного паттерна в строке.

В случае с использованием символа джокера алгоритм меняется. Во-первых, паттерн может быть всего один. Он делится на подстроки, не содержащие в себе символов джокера – подпаттерны. Из них составляется бор. Теперь идентификатором терминальной вершины (которые хранятся в массиве v.leafPatternNumber) будет являться не индекс, а смещение подпаттерна относительно паттерна с дожкером.

Инициализация:

*int C[s.length()]; // заполнен нулями*

*Node\* cur = root;*

На каждом шаге:

1. *\*аналогично стандартному алгоритму\**

2. Инкрементируем *C[i – offset – suff->index]* – по сути этот индекс (*i – offset – suff->index*) ни что иное, как индекс начала паттерна с джокером, но в данном случае мы сверяли лишь его часто – одиночный подпаттерн. Необходимо проверять, чтобы индекс был неотрицательным.

После выполнения расчетов пробегаемся по массиву C. Если C[i] == patternsCount, то на индексе i нашлось очередное вхождение паттерна с джокером (необходимо проверять, чтобы разница между длиной строки и индексом i была не меньше, чем длина паттерна).

Сложность первого алгоритма:

* Память: O(nq), где n – общая длина слов в словаре, q – Размер алфавита
* Вычислительная: O(nq + H + k), где H – длина текста, k – общая длина всех совпадений.

Сложность второго алгоритма:

* Память: O(nq), где n – общая длина слов в словаре, q – Размер алфавита
* Вычислительная: O(nq + H + k), где H – длина текста, k – общая длина всех совпадений.

# ИДЗ

1. В Node было добавлено поле sonsCount, в котором хранится количество дуг, выходящих из вершины. При вычислении максимального количество дуг из вершины программа рекурсивно проходит все вершины и ищет среди них вершину с наибольшем количеством дуг.

2. Проще будет объяснить непосредственно на коде:

## // Результат поиска, отсортированный по возрастанию item.first и

## // item.second (item – элемент массива)

## vector<pair<int, int>> result = algh.process(T);

## // Отнимаем 1 от индексов, т.к. по условию задачи они нумеруются

## // с 1, нам же нужна нумерация с 0, чтобы было удобнее работать с

## // массивами for (auto& item : result) {

## // item – очередной найденный паттерн item.first--; // Позиция паттерна в строке item.second--;// ID паттерна }

## int start = 0; // Курсор строки, в которой производился поиск int index = 0; // Курсор в массиве cout << "Line without matched substrings: ";

## // Итерируем, пока курсор не вышел за пределы строки while (start < T.length()) {

## // Если мы еще не закончили итерировать по массиву результатов

## // и при этом позиция курсора в строке не превосходит очередного

## // результата, то (не забывайте, массив отсортирован по

## // возрастанию first и second) перед курсором есть паттерн и его

## // нужно обработать: если паттерн накрывает позицию start, нужно

## // сдвинуть start на следующую после паттерна позицию.

## //

## // Если же start меньше очередного индекса паттерна, то можно

## // вывести данный символ, т.к. он не перекрывается никаким

## // паттерном. if (index < result.size() && start >= result[index].first) { start = max<int>(start, result[index].first + P[result[index].second].length()); index++; continue; }

## // Выводим результат, если текущий символ не перекрывается никаким

## // паттерном cout << T[start]; start++; } cout << endl;

# Тестирование

**Стандартный алгоритм**

|  |  |  |
| --- | --- | --- |
| # | input | output |
| 1 | NTAG3TAGTTAGT | 2 22 3 |
| 2 | NTAGT3TAGTTAGT | 2 12 22 35 3 |

**Алгоритм с джокером**

|  |  |  |
| --- | --- | --- |
| # | input | output |
| 1 | ACTANCAA$$A$$ | 1 |
| 2 | ACACAAACXAX | 3 |

**Идз**

|  |  |  |
| --- | --- | --- |
| # | input | output |
| 1 | NTAG3TAGTTAGT | 2 22 3Maximum child count: 1Line without matched substrings: N |
| 2 | NTAGT3TAGTTAGT | 2 12 22 35 3Maximum child count: 1Line without matched substrings: N |

# Вывод

В процессе выполнения работы была решена задача поиска множественных паттернов в строке при помощи алгоритма Ахо-Корасик. Проведено тестирование алгоритма на разных данных. Реализован алгоритм поиска подстроки с символом джокера, основанный на алгоритме Ахо-Корасик.

Также написано решение для индивидуального варианта, содержащее очистку строки от всех вхождений шаблонов и подсчитывающее максимальное количество дуг, исходящих из одной вершины.

# Приложение А. Код класса axo

## #include <vector> #include <string> #include <iostream> #include <algorithm> using namespace std; /\*\* \* Размер алфавита - разница между первым возможным символом алфавита и последним. \*/ #define ALPHABET\_SIZE 32 /\*\* \* Вершина в дереве алгоритма. \*/ class Node { public: /\*\* \* Массив дочерних вершин. Если вершина [c] не равна nullptr, то ребенок есть. \*/ Node \*son[ALPHABET\_SIZE]; /\*\* \* Переходы их данной вершины по символу (кеш для функции Axo::getLink). \*/ Node \*go[ALPHABET\_SIZE]; /\*\* \* Родительская вершина. \*/ Node \*parent = nullptr; /\*\* \* Ссылка на подходящий для текущего пути (от root до вершины) суффикс. \*/ Node \*suffLink = nullptr; /\*\* \* Ближайшая сжатая ссылка (указывающая на ближайший суффикс, последний символ которого который является терминалом). \*/ Node \*up = nullptr; /\*\* \* Символ между родительской вершиной и текущей. \*/ char charToParent = 0; /\*\* \* Является ли вершина терминальной. \*/ bool isLeaf = false; /\*\* \* Список паттернов, кончающихся в данной вершине. \*/ vector<int> leafPatternNumber; /\*\* \* Расстояние от вершины до корня. \*/ int index; /\*\* \* Количество дуг, которые выходят из вершины. \*/ int sonsCount = -1; Node() { for (int i = 0; i < ALPHABET\_SIZE; i++) { son[i] = nullptr; go[i] = nullptr; } } ~Node() { for (auto v : son) delete v; } }; /\*\* \* Основной класс, занимающийся обработкой данных. \*/ class Axo { public: /\*\* \* Количество обработанных паттернов (используется в модифицированном алгоритме). \*/ int patternsCount = 0; /\*\* \* Корневая вершина дерева. \*/ Node \*root = new Node(); ~Axo() { delete root; } /\*\* \* Функция получения суффикса для вершины v. Результат функции кешируется. \* \* @param v Вершина, суффикс которой необходимо найти. \* @return Вершина, являющаяся последним символом максимального суффикса. \*/ Node \*getSuffLink(Node \*v) { if (v->suffLink == nullptr) { if (v == root || v->parent == root) v->suffLink = root; else v->suffLink = getLink(getSuffLink(v->parent), v->charToParent); } return v->suffLink; } /\*\* \* Найти максимальный суффикс [v]+c. Рассчитанные значения кешируются. \* \* @param v Вершина, от коорой строится путь. \* @param c Символ пути. \* @return Вершина - последний символ максимального суффикса. \*/ Node \*getLink(Node \*v, char c) { if (v->go[c] == nullptr) { if (v->son[c] != nullptr) v->go[c] = v->son[c]; else if (v == root) v->go[c] = root; else v->go[c] = getLink(getSuffLink(v), c); } return v->go[c]; } /\*\* \* Найти сжатую ссылку для вершины v. Полученные значения кешируются. \* \* @param v Вершина, для которой необходимо найти сжатую ссылку. \* @return Сжатая ссылка для вершины v. \*/ Node \*getUp(Node \*v) { if (v->up == nullptr) { if (getSuffLink(v)->isLeaf) v->up = getSuffLink(v); else if (getSuffLink(v) == root) v->up = root; else v->up = getUp(getSuffLink(v)); } return v->up; } /\*\* \* Добавить паттерн в дерево \* \* @param s Паттерн \* @param number Идентификатор паттерна. В случае с добавлением паттернов модифицированного алгоритма в роли \* идентификатора выступает смещение фрагмента относительно начала паттерна. \*/ void addString(const string &s, int number) { Node \*cur = root; for (int i = 0; i < s.length(); i++) { char c = s[i] - 'A'; // Добавляем элемент, если его нет. if (cur->son[c] == nullptr) { cur->son[c] = new Node(); cur->son[c]->parent = cur; cur->son[c]->charToParent = c; cur->son[c]->isLeaf = false; cur->son[c]->index = i; } cur = cur->son[c]; } cur->isLeaf = true; cur->leafPatternNumber.push\_back(number); patternsCount++; } /\*\* \* Заполнить дерево паттерном, использующим символ джокера. \* \* @param s Паттерн. \* @param joker Символ джокера. \*/ void setJokerString(const string &s, char joker) { int start = -1; int i = 0; while (true) { if ((i == s.length() || s[i] == joker) && start != -1 && start != i) { addString(s.substr(start, i - start), start); start = -1; } if (i == s.length()) break; if (start == -1 && s[i] != joker) start = i; i++; } } /\*\* \* Обработать строку s при помощи предварительно заполненного дерева модифицированным алгоритмом. \* \* @param s Строка, в которой производится поиск. \* @param P Применяемый паттерн. Данный паттерн не добавляется в дерево, используется лишь его длина. \*/ void processJoker(const string &s, const string &P) { // C[N] - количество паттернов, которые начинаются с индекса N, с учетом смещения относительно паттерна с // джокером. int C[s.length()]; for (int i = 0; i < s.length(); i++) C[i] = 0; Node \*cur = root; // Заполняем массив C for (int i = 0; i < s.length(); i++) { char c = s[i] - 'A'; cur = getLink(cur, c); Node \*suff = cur; while (suff != root) { for (int offset : suff->leafPatternNumber) if (i - offset - suff->index >= 0) C[i - offset - suff->index]++; suff = getUp(suff); } } // Выводим заполненный массив. Если C[i] == patternsCount, то с i-го символа начинаются все паттерны (при учете // смещения относительно основного паттерна с джокером), а значит достигнуто полное совпадение. for (int i = 0; i < s.length(); i++) { if (i + P.length() > s.length()) break; if (C[i] == patternsCount) cout << i + 1 << endl; } } /\*\* \* Обработать строку s стандартным алгоритмом, используя предзаполненное дерево паттернов. \* \* @param s Строка, в которой производится поиск. \*/ vector<pair<int, int>> process(const string &s) { Node \*cur = root; vector<pair<int, int>> result; // Ищем вхождения паттернов в строку for (int i = 0; i < s.length(); i++) { char c = s[i] - 'A'; cur = getLink(cur, c); Node \*suff = cur; while (suff != root) { for (int number : suff->leafPatternNumber) result.emplace\_back(i - suff->index + 1, number); suff = getUp(suff); } } // Сортируем и выводим результаты sort(result.begin(), result.end(), cmp); for (auto &val : result) { cout << val.first << " " << val.second << endl; } return result; } /\*\* \* Найти вершину, из которой выходит наибольшее количество дуг. \* \* @return Вершина, из которой выходит наибольшее количество дуг. \*/ Node\* findMaxNode() { return findMaxNode(root); } /\*\* \* Найти вершину, из которой выходит наибольшее количество дуг. \* \* @param v Стартовая вершина - корень дерева, по которому производится поиск. \* @return Вершина, из которой выходит наибольшее количество дуг. \*/ Node\* findMaxNode(Node\* v) { Node\* maxSon = nullptr; for (Node\* son : v->son) { if (son == nullptr) continue; if (maxSon == nullptr) { maxSon = son; continue; } if (getSonsCount(maxSon) < getSonsCount(son)) maxSon = son; } if (maxSon != nullptr && getSonsCount(maxSon) > getSonsCount(v)) return maxSon; return v; } /\*\* \* Получить количество дуг, которые идут из вершины. \* \* @param v Вершина, количество дуг из которой необходимо найти. \* @return Количество дуг, идущих из вершины v \*/ int getSonsCount(Node\* v) { if (v->sonsCount == -1) { v->sonsCount = 0; for (Node\* son : v->son) if (son) v->sonsCount++; } return v->sonsCount; } /\*\* \* Функция сравнения результатов поиска стандартным способом. \* \* @param a Первый элемент \* @param b Второй элемент \* @return Меньше ли первый элемент второго \*/ static bool cmp(const pair<int, int> &a, const pair<int, int> &b) { if (a.first < b.first) return true; if (a.first > b.first) return false; return a.second < b.second; } };

# Приложение Б. Код основной программы.

## #include <iostream> #include <cmath> #include "axo.cpp" void launchIdz() { string T; cin >> T; int count; cin >> count; string P[count]; Axo algh; for (int i = 0; i < count; i++) { cin >> P[i]; algh.addString(P[i], i + 1); } auto result = algh.process(T); for (auto& item : result) { item.first--; item.second--; } *// Поиск вершины с наибольшим количеством дуг.* int maxSonsCount = algh.getSonsCount(algh.findMaxNode()); cout << "Maximum child count: " << maxSonsCount << endl; int start = 0; int index = 0; *// Вывод строки без вхождений.* cout << "Line without matched substrings: "; while (start < T.length()) { if (index < result.size() && start >= result[index].first) { start = max<int>(start, result[index].first + P[result[index].second].length()); index++; continue; } cout << T[start]; start++; } cout << endl; } */\*\* \* Запустить стандартный алгоритм Ахо-Корасика. \*/* void launchAxo() { string T; cin >> T; int count; cin >> count; Axo algh; for (int i = 0; i < count; i++) { string P; cin >> P; algh.addString(P, i + 1); } algh.process(T); } */\*\* \* Запустить алгоритм Ахо-Корасика с масками. \*/* void launchJoker() { string T; cin >> T; string P; cin >> P; char joker; cin >> joker; Axo algh; algh.setJokerString(P, joker); algh.processJoker(T, P); } int main() { launchIdz(); return 0; }