

Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной
математики

Кафедра вычислительной математики и программирования

Лабораторная работа №4 по курсу «Дискретный анализ»

Студент: М. Ю. Курносов
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б
Дата:
Оценка:
Подпись:

Москва, 2025

Лабораторная работа №4

Задача: Необходимо реализовать один из стандартных алгоритмов поиска образцов для указанного алфавита.

Вариант алгоритма: Поиск одного образца-маски: в образце может встречаться «джокер» (представляется символом ? — знак вопроса), равный любому другому символу. При реализации следует разбить образец на несколько, не содержащих «джокеров», найти все вхождения при помощи алгоритма Ахо-Корасик и проверить их относительное месторасположение.

Вариант алфавита: Слова не более 16 знаков латинского алфавита (регистронезависимые)

Запрещается реализовывать алгоритмы на алфавитах меньшей размерности, чем указано в задании.

Формат ввода: Искомый образец задаётся на первой строке входного файла.

В случае, если в задании требуется найти несколько образцов, они задаются по одному на строку вплоть до пустой строки.

Затем следует текст, состоящий из слов или чисел, в котором нужно найти заданные образцы.

Никаких ограничений на длину строк, равно как и на количество слов или чисел в них, не накладывается.

Формат вывода: В выходной файл нужно вывести информацию о всех вхождениях искомых образцов в обрабатываемый текст: по одному вхождению на строку.

Для заданий, в которых требуется найти только один образец, следует вывести два числа через запятую: номер строки и номер слова в строке, с которого начинается найденный образец. В заданиях с большим количеством образцов, на каждое вхождение нужно вывести три числа через запятую: номер строки; номер слова в строке, с которого начинается найденный образец; порядковый номер образца.

Нумерация начинается с единицы. Номер строки в тексте должен отсчитываться от его реального начала (то есть, без учёта строк, занятых образцами).

Порядок следования вхождений образцов несущественен.

Примеры:

Входные данные:

```
cat ? cat dog
CAT dog CaT
Dog doG dog dOg
Cat doG cat dog cat dog cat Parrot
doG dog DOG DOG dog
```

Результат работы:

1, 1

3, 1

3, 3

1 Описание

Алгоритм Ахо — Корасик — алгоритм поиска подстроки, разработанный Альфредом Ахо и Маргарет Корасик в 1975 году, реализует поиск множества подстрок из словаря в данной строке. Широко применяется в системном программном обеспечении, например, используется в утилите поиска grep.

Алгоритм строит конечный автомат, которому затем передаёт строку поиска. Автомат получает по очереди все символы строки и переходит по соответствующим рёбрам. Если автомат пришёл в конечное состояние, соответствующая строка словаря присутствует в строке поиска.

Несколько строк поиска можно объединить в дерево поиска, так называемый бор (префиксное дерево). Бор является конечным автоматом, распознающим одну строку из m — но при условии, что начало строки известно.

Первая задача в алгоритме — научить автомат «самовосстанавливаться», если подстрока не совпала. При этом перевод автомата в начальное состояние при любой неподходящей букве не подходит, так как это может привести к пропуску подстроки (например, при поиске строки aabab, попадаетея aabaabab, после считывания пятого символа перевод автомата в исходное состояние приведёт к пропуску подстроки — верно было бы перейти в состояние a, а потом снова обработать пятый символ). Чтобы автомат самовосстанавливался, к нему добавляются суффиксные ссылки, нагруженные пустым символом (так что детерминированный автомат превращается в недетерминированный). Например, если разобрана строка aaba, то бору предлагаются суффиксы aba, ba, a. Суффиксная ссылка — это ссылка на узел, соответствующий самому длинному суффиксу, который не заводит бор в тупик (в данном случае a).

Для корневого узла суффиксная ссылка — петля. Для остальных правило таково: если последний распознанный символ — s , то осуществляется обход по суффиксной ссылке родителя, если оттуда есть дуга, нагруженная символом s , суффиксная ссылка направляется в тот узел, куда эта дуга ведёт. Иначе — алгоритм проходит по суффиксной ссылке ещё и ещё раз, пока либо не пройдёт по корневой ссылке-петле, либо не найдёт дугу, нагруженную символом s .

2 Исходный код

main.hpp:

```
1  #include <iostream>
2  #include <string>
3  #include <unordered_map>
4  #include <utility>
5  #include <vector>
6  #include <sstream>
7  #include <algorithm>
8
9  class TTrie;
10
11 class TNode
12 {
13 public:
14     friend TTrie;
15     TNode() : failLink(nullptr), exitLink(nullptr), leaf(false) {}
16
17 private:
18     std::unordered_map<std::string, TNode *> chil; // children
19     TNode *failLink;
20     TNode *exitLink;
21     bool leaf;
22     size_t patSize;
23     std::vector<size_t> pos;
24 };
25
26 class TTrie
27 {
28 public:
29     TTrie() : root(new TNode()) {}
30
31     ~TTrie()
32     void AddPat(std::pair<std::vector<std::string>, size_t> &pat)
33     void failInit(size_t max)
34         std::vector<std::pair<size_t, size_t>> search(std::vector<std::pair<std::string,
35             std::pair<size_t, size_t>>> &text, size_t patCount, size_t patLen)
36     private:
37     void Destroy(TNode *n)
38     void SetLinks(TNode *n, size_t lvl)
39     void Linking(TNode *parent, TNode *nd, std::string nodeSym)
40     TNode *root;
41 };
42
43 std::vector<std::pair<std::vector<std::string>, size_t>> split(std::stringstream &
44     pattern, std::string joker, size_t &wordCount)
45
46 int main()
```

lab4.cpp	
class TTrie	Структура трая
class TNode	Структура узла
TNode()	Конструктор узла
TTrie()	Конструктор трая
$\sim TTrie()$	деструктор трая
void AddPat(std::pair<std::vector<std::string>, size_t> &pat)	добавление узла с паттерном
void failInit(size_t max)	построение ссылок
std::vector<std::pair<size_t, size_t> search(std::vector<std::pair<std::string, std::pair<size_t, size_t>> &text, size_t patCount, size_t patLen)	поиск в трае
void Destroy(TNode *n)	рекурсивное уничтожение от узла n
void SetLinks(TNode *n, size_t lvl)	вспомогательный метод для расстановки ссылок в дереве
void Linking(TNode *parent, TNode *nd, std::string nodeSym)	вспомогательный метод для расстановки ссылок в дереве
std::vector<std::pair<std::vector<std::string>, size_t> split(std::stringstream &pattern, std::string joker, size_t &wordCount)	разбиение паттернов с джокером на подпаттерны
int main()	главная функция программы

3 Консоль

```
me@DESKTOP:/mnt/c/Users/Max Kar/Desktop/МАИ/ДА/2025/lab4$ ./p <3test.txt
1,1
3,1
3,3
me@DESKTOP:/mnt/c/Users/Max Kar/Desktop/МАИ/ДА/2025/lab4$ ./p <2test.txt
1,2
me@DESKTOP:/mnt/c/Users/Max Kar/Desktop/МАИ/ДА/2025/lab4$ ./p <test.txt
1,1
2,1
```

4 Тест производительности

Тест производительности представляет из себя следующее: программа запускается три раза и генерирует текст и паттерны затем ищет паттерны среди текста сначала с помощью нашей реализации поиска затем стандартной реализацией. Время от начала до конца поиска каждой функции выводится на консоль. Размер текста передаётся в программу через аргументы командной строки.

```
me@DESKTOP:/mnt/c/Users/Max Kar/Desktop/МАИ/ДА/2025/lab4$ ./bench 100000
```

```
Text size: 100000 words
```

```
Trie search: 25 ms
```

```
std::search: 2 ms
```

```
me@DESKTOP:/mnt/c/Users/Max Kar/Desktop/МАИ/ДА/2025/lab4$ ./bench 1000000
```

```
Text size: 1000000 words
```

```
Trie search: 252 ms
```

```
std::search: 25 ms
```

```
me@DESKTOP:/mnt/c/Users/Max Kar/Desktop/МАИ/ДА/2025/lab4$ ./bench 100000000
```

```
Text size: 100000000 words
```

```
Trie search: 30938 ms
```

```
std::search: 26337 ms
```

5 Выводы

В результате выполнения лабораторной работы по курсу «Дискретный анализ», были получены навыки использования префиксных деревьев, закреплена работа с утилитой `valgrind` и навыки использования итераторов и стандартных контейнеров языка C++. В ходе выполнения лабораторной работы была реализована структура данных Trie с алгоритмом Ахо-Корасик для поиска множества подстрок в тексте. На малых и средних данных (100К–1М слов): `std::search` работает быстрее. Это ожидаемо, так как Ахо-Корасик требует предварительного построения автомата, что даёт накладные расходы, `std::search` оптимизирован для однократного поиска и использует эффективные алгоритмы. На больших данных (100М слов): Разница сокращается. Причина: алгоритм Ахо-Корасик масштабируется линейно ($O(n + m + k)$), как и `std::search` но на практике быстрее. В итоге если искать несколько паттернов одновременно, Trie-реализация станет выгоднее, так как: Построение автомата делается 1 раз, а поиск работает за линейное время независимо от числа паттернов. В `std::search` пришлось бы делать отдельный проход для каждого паттерна.

Список литературы

- [1] *Префиксное дерево*
URL: https://ru.wikipedia.org/wiki/Префиксное_дерево
(дата обращения: 27.12.2020).
- [2] *Trie, или нагруженное дерево*
URL: <https://habr.com/ru/post/111874/> (дата обращения: 27.12.2020).
- [3] *Алгоритм Ахо-Корасик*
URL: <https://habr.com/ru/post/330644/> (дата обращения: 27.12.2020).
- [4] *Алгоритм Ахо — Корасик*
URL: https://ru.wikipedia.org/wiki/Алгоритм_Ахо_-_Корасик
(дата обращения: 27.12.2020).