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
Вариант: 4

Студент гр. 3388 _____ Потапов Р.Ю.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2025

Цель работы:

Изучить теоретические основы алгоритма Ахо-Корасика. Разработать алгоритм с учетом варианта.

Задание:

Разработайте программу, решающую задачу точного поиска набора образцов.

Входные данные:

Первая строка содержит текст (T,1≤|T|≤100000).

Вторая - число n*n* (1≤n≤30001≤*n*≤3000), каждая следующая из n*n* строк содержит шаблон из набора P={p1,…,pn}1≤|pi|≤75

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

Выходные данные:

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 1:

NTAG

3

TAGT

TAG

T

Sample Output 1:

2 2

2 3

Выполнение работы

Описание алгоритма:

Алгоритм решения задачи коммивояжёра (TSP) методом ветвей и границ с двумя нижними оценками строится по принципу рекурсивного перебора и раннего отсечения (branch and bound). Он стремится найти оптимальный гамильтонов цикл минимальной стоимости, последовательно добавляя новые города к текущему маршруту и используя оценки, чтобы сократить пространство поиска. Ниже приводятся основные этапы работы алгоритма и краткое описание функций, задействованных в коде.

Алгоритм начинается с города 0, формируя частичное решение, и на каждом шаге добавляет одну ещё не посещённую вершину. При этом рассчитывается «нижняя граница» (bound), чтобы понять, возможно ли улучшить уже известное лучшее решение. Если bound оказывается не меньше текущего лучшего результата, ветвь поиска прекращается (отсечение), иначе алгоритм рекурсивно углубляется, пытаясь продолжить маршрут.

Основные этапы работы:

1 Построение префиксного дерева (Trie): создаётся корневой узел с пустым префиксом, после чего каждый шаблон последовательно вставляется в бор. При вставке для каждой буквы текущего шаблона из текущей вершины либо переходим по существующему ребру, либо создаём новую вершину, а в конце шаблона добавляем его индекс в список output этой вершины.

2 Построение ссылок неудачи (failure links) и терминальных ссылок (dict_link): обходом в ширину (BFS) сначала все дочерние узлы корня получают fail→корень и dict_link=None, затем для каждой вершины и каждого её

ребра по символу ищется наибольший собственный суффикс через следование fail-ссылок; полученный узел становится fail для данного ребёнка, а dict_link указывает либо на узел fail, если в нём есть output, либо наследует dict_link родителя.

3 Поиск непересекающихся вхождений: текущее состояние автомата устанавливается в корень, заводятся пустые множества blocked и result. Для каждого символа текста выполняются переходы по children с «откатом» по fail при отсутствии ребра, затем в текущей вершине сначала обрабатываются собственные выходы из output, потом по цепочке dict_link собираются дополнительные совпадения. Полученный список шаблонов сортируется по убыванию длины, и первые не пересекающиеся с уже заблокированными позициями вхождения блокируются (их позиции добавляются в blocked и пара (начало, номер шаблона) записывается в result), после чего алгоритм переходит к следующему символу.

4 Итоговый вывод: после просмотра всего текста список result сортируется по возрастанию позиции и вывода шаблона и печатается в требуемом формате (1-based индексы).

Описание функций:

1 **Node.init(self, prefix='')**

Конструктор узла бора. Инициализирует словарь переходов children = {}, поля навигации fail = None и dict_link = None, список собственных совпадений output = [], сохраняет строку-префикс от корня до этой вершины prefix, присваивает уникальный id для отладки.

2 **build_trie(patterns)**

Построение префиксного дерева (Trie) из списка шаблонов. Создаёт корень с пустым префиксом, затем для каждого шаблона и каждой его буквы либо

переходит по существующему ребру, либо создаёт новую вершину, выводя сообщения о создании. В конце каждого шаблона добавляет его индекс в output текущей вершины и печатает итоговую структуру всех узлов. Возвращает корень бора.

3 **build_failure_links(root)**

Установка суффиксных (fail) и терминальных (dict_link) ссылок. Обходит вершины бора в ширину, инициализирует всех детей корня так, чтобы fail→root и dict_link=None, затем для каждого ребра по символу ищет наибольший собственный суффикс через следование fail-ссылок, присваивает найденную вершину в u.fail, после чего если у u.fail есть свои совпадения, направляет u.dict_link на u.fail, иначе наследует u.fail.dict_link. Печатает подробные логи для каждой установки.

4 **aho_corasick_non_overlapping(text, patterns)**

Основная функция поиска. Вызывает build_trie и build_failure_links, затем последовательно перебирает символы входного текста, переходя по ребрам и при отсутствии по fail, собирает в текущей вершине сначала собственные совпадения из output, затем дополнительные через цепочку dict_link, сортирует найденные шаблоны по убыванию длины, проверяет на непересечение с уже заблокированными позициями (blocked) и блокирует новые, записывая пару (start, pattern_index) в result. Параллельно выводит подробный лог каждого шага и возвращает отсортированный список вхождений.

5 **main()**

Считывает вход: первую строку — текст, вторую — число шаблонов, затем сами шаблоны. Вызывает aho_corasick_non_overlapping для выполнения поиска с отладочным выводом и печати финальных результатов.

Оценка сложности алгоритма:

Временная сложность

Оценка временной сложности нашего алгоритма сводится к двум основным фазам: построению структуры и собственно поиску. Построение бора вместе с вычислением fail- и dict_link-ссылок обрабатывает каждый символ каждого шаблона лишь константное число раз и выполняется за время $O(\sum|patterns|)$, занимая память $O(\sum|patterns|)$. Поиск по тексту в среднем работает за $O(|text|)$ амортизированно, поскольку для каждого символа мы совершаем несколько переходов по children и fail (в сумме не более некоторого константного множителя от числа символов), а затем обрабатываем найденные совпадения: сортировка списка совпавших шаблонов и проверка блокировки позиций требуют времени $O(k \log k + L)$, где k — число совпавших в данной позиции шаблонов, а L — их максимальная длина. В худшем же случае, когда на каждой позиции текста совпадают все шаблоны, общая сложность поиска вырастает до $O(|text| \cdot (P \log P + L_{max}))$, где P — число шаблонов, однако на практике при умеренном количестве совпадений и ограниченной длине шаблонов алгоритм ведёт себя близко к $O(|text| + \sum|patterns|)$. Пространственная сложность остаётся полиномиальной: $O(\sum|patterns|)$ на структуру бора плюс $O(|text|)$ на блокировку позиций.

Тестирование

Таблица 1. Тестирование.

Входные данные	*Выходные данные*
ushers he she 2	2 2
niggers 2 ger gg	3 2
ushersushersushers 4 he she rsu er	2 2 5 3 8 2 11 3 14 2
trololobombon 3 lol lob bom	4 1 8 3

Исследование

1 В ходе данной лабораторной работы был реализован и подробно исследован алгоритм многопаттернного поиска Ахо–Корасик в варианте 4 — с требованием поиска непересекающихся вхождений. Цель алгоритма — за один проход по тексту находить все вхождения набора шаблонов и при этом обеспечивать, что найденные фрагменты не пересекают друг друга, выбирая на каждом шаге жадно самое длинное подходящее совпадение.

2 Построение структуры автомата

Алгоритм начинает с построения префиксного дерева (Trie) по всем шаблонам: каждый шаблон последовательно вставляется в бор, создавая новые вершины по новым буквам и помечая в списке output индексы шаблонов в узлах, соответствующих их концам. Затем в порядке обхода в ширину (BFS) устанавливаются суффиксные ссылки fail (для перехода к наибольшему собственному суффиксу при несовпадении) и терминальные ссылки dict_link (для быстрого перебора всех шаблонов, заканчивающихся в данной вершине или её суффиксных предках).

3 Поиск непересекающихся вхождений

При сканировании текста алгоритм хранит текущее состояние автомата (с вершины в боре) и множество blocked заблокированных позиций. Для каждого символа текста выполняется переход по ребру children или, в случае отсутствия перехода, по цепочке fail до корня. Оказавшись в новой вершине, алгоритм собирает все завершённые шаблоны через output и dict_link, сортирует их по убыванию длины и проверяет по каждому, не пересекает ли оно уже заблокированные участки текста. Первое «чистое» вхождение блокирует соответствующие позиции и вносит пару (позиция начала, номер шаблона) в результат.

4 Отчётность и отладочные выводы

Для наглядности работы алгоритма в коде предусмотрена печать: при

построении бора — создание каждой вершины и её префикса, при установке ссылок — назначения fail и dict_link, при поиске — детали каждого перехода, найденные шаблоны и причины их принятия или отбрасывания, а также текущее состояние множества заблокированных позиций. Это позволяет полностью отследить ход выполнения и подтвердить корректность непересекающегося варианта.

5. Функции в методе ветвей и границ

– **Node.init(self, prefix='')**. Конструктор узла бора: создаёт пустой словарь переходов children, устанавливает fail = None для ссылки неудачи и dict_link = None для терминальной ссылки, инициализирует пустой список output для индексов шаблонов, сохраняет строку-префикс от корня до этой вершины prefix и присваивает уникальный id (увеличивая счётчик), что позволяет в отладке однозначно идентифицировать каждую вершину.
– **build_trie(patterns)**. Построение префиксного дерева: создаёт корневую вершину с пустым префиксом, затем для каждого шаблона и каждой буквы либо переходит по уже существующему ребру, либо создаёт новую вершину, выводя сообщения об их создании, и в конце каждого шаблона добавляет его индекс в output текущей вершины. В завершение печатает полную структуру бора (id, prefix, дети, output) и возвращает корень.
– **build_failure_links(root)**. Установка суффиксных и терминальных ссылок: выполняет обход в ширину по вершинам бора, сначала для всех детей корня присваивает fail→root и dict_link=None, затем для каждого ребра по символу находит через цепочку fail наибольший собственный суффикс и устанавливает u.fail на соответствующую вершину (или на корень), после чего, если в u.fail.output есть шаблоны, направляет u.dict_link = u.fail, иначе наследует u.dict_link = u.fail.dict_link. При каждом присвоении выводит подробный лог (id, prefix, куда указывает fail и dict_link).
– **aho_corasick_non_overlapping(text, patterns)**. Основной поиск: сначала вызывает build_trie и build_failure_links, затем инициализирует пустые

множества blocked (для уже занятых позиций) и result (для найденных вхождений), устанавливает current = root и последовательно проходит по каждому символу текста, делая переходы по children с «откатом» по fail при несовпадении. В каждой достигнутой вершине собирает сначала свои совпадения из output, затем дополнительные через цепочку dict_link, сортирует их по убыванию длины, проверяет на непересечение с blocked и при нахождении первого допустимого блокирует соответствующие позиции, добавляя пару (start, pattern_index) в result. Параллельно выводит отладочные сообщения о каждом шаге, найденных шаблонах, причинах принятия или пропуска. В конце возвращает отсортированный по позиции и номеру шаблона список вхождений.

– **main()**. Интерфейс ввода-вывода: считывает первую строку как text, вторую — число шаблонов n, затем n строк шаблонов, и запускает aho_corasick_non_overlapping(text, patterns), который печатает полный лог работы и финальные результаты.

6. Сложность

Сложность: построение структуры (Trie + fail- и dict_link-ссылок) требует времени $O(\sum|patterns|)$ — мы один раз вставляем все шаблоны и один раз обходим их вершины в ширину, настраивая переходы и ссылки. Поиск по тексту амортизированно занимает $O(|text| + \tau)$, где τ — общее число найденных совпадений: каждый символ обрабатывается переходами по children или fail, а затем для каждого совпадения выполняется константный набор операций (проверка блокировки и запись результата). В худшем же случае при каждой позиции текста совпадают все P шаблонов, и их сортировка и проверка непересечения займут $O(P \log P + L_{max})$ времени, так что общий худший-сценарий — $O(|text| \cdot (P \log P + L_{max}))$. На практике число совпадений и длина шаблонов ограничены, поэтому алгоритм ведёт себя близко к $O(|text| + \sum|patterns|)$. Память: структура бора требует $O(\sum|patterns|)$, а множество

заблокированных позиций — O(|text|), так что общая пространственная сложность остаётся O(∑|patterns| + |text|).

Вывод

В ходе работы реализован «вариант 4» Aho–Corasick для непересекающихся вхождений: строится trie, настраиваются fail и dict_link, и детально логируется ход поиска. Во время обхода текста применяется жадный выбор самых длинных шаблонов и блокировка занятых позиций, что исключает пересечения. Амортизированная сложность близка к O(|text| + ∑|patterns| + τ), память — O(∑|patterns| + |text|). Такой подход сочетает наглядность работы алгоритма и высокую практическую эффективность.