Лабораторная работа № 4 по курсу дискретного анализа: поиск образца в строке

Выполнил студент группы 08-208 МАИ *Сизонов Артём.*

Условие

Кратко описывается задача:

1. Общая постановка задачи: Необходимо реализовать один из стандартных алгоритмов поиска образцов для указанного алфавита.

2. Вариант задания: Поиск большого количества образцов при помощи алгоритма Ахо-Корасик, слова не более 16 знаков латинского алфавита (регистронезависимые) (5-1).

Метод решения

Решение данное задачи разделено на этапы: вставка слов в trie, расстановка ссылок в trie, поиск паттернов в тексте. При этом, считывание символов происходит параллельно с вставкой в trie и поиском по тексту (с помощью функции GetWord). Это сделано для экономии памяти.

Вставка слов в trie. Происходит считывание слов до тех пор, пока не обнаружится пустая строка. Для навигации по trie используется дополнительная переменная present, имеющая тип указателя на вершину trie (TTrieNode*), которая определяет, в какое место в trie будет добавлено считанное слово.

Если считано слово и оно не пустое, то оно добавляется в trie сразу за вершиной present. Переменная present обновляется на только что созданную вершину. Для хранения последующих вершин используется контейнер std::map <std::string, TTrieNode> next, который есть у каждой вершины trie. Также, при добавлении каждой вершины для неё устанавливается параметр distance, который равен расстоянию от корня trie до добавляемой вершины.

Если считано пустое слово, то добавление не происходит.

Далее, если считанное слово является последним в строке, то в текущую вершину present, в её параметр number записывается порядок паттерна, а переменная present обновляется на корень trie (root).

Расстановка ссылок в trie. Расстановка ссылок в созданном trie происходит с использованием поиска в ширину. Сначала в очередь добавляется корень. Далее берётся элемент (обозначим через tn) из очереди, если он является корнем или сыном корня,

то его ссылка указывает на корень. В противном случае происходит поиск вершины, на которую будет указывать ссылка:

1. Создаётся временная переменная TTrieNode* temp_link, которой приравнивается ссылка отца вершины tn.

2. Переходим по ссылкам до тех пор, пока не придём в корень или не найдём такую вершину, один из следующих слов которой совпадает со словом текущей вершины tn.

3. Приравниваем ссылке вершины tn, указатель на вершину, который следует за temp_link и соответствует слову вершины tn.

4. Добавляем в очередь указатели на все вершины, следующие за tn.

5. Снова берём элемент из очередь.

Поиск паттернов в тексте. Для определения номера строки, в которой находится найденный паттерн и номер слова в строке, с которого начинается паттерн, объявляем целочисленные переменные number_of_word и number_of_string, в которых будут храниться номера текущего слова и строки соответсвенно. Также объявляется std::vector < int > length_of_strings, где для каждого i: length_of_string[i] - length_of_string[i - 1] — количество слов в строке номер i. Объявляем TTrieNode* node в котором будет храниться вершина, на которой мы находимся. Алгоритм:

1. Считываем слово. Переходим к пунктам 2-5.

2. Если оно является пустым и последним в строке, от увеличиваем number_of_strings на 1, а в вектор length_of_string добавляем number_of_word - 1. Переходим к пункту 1.

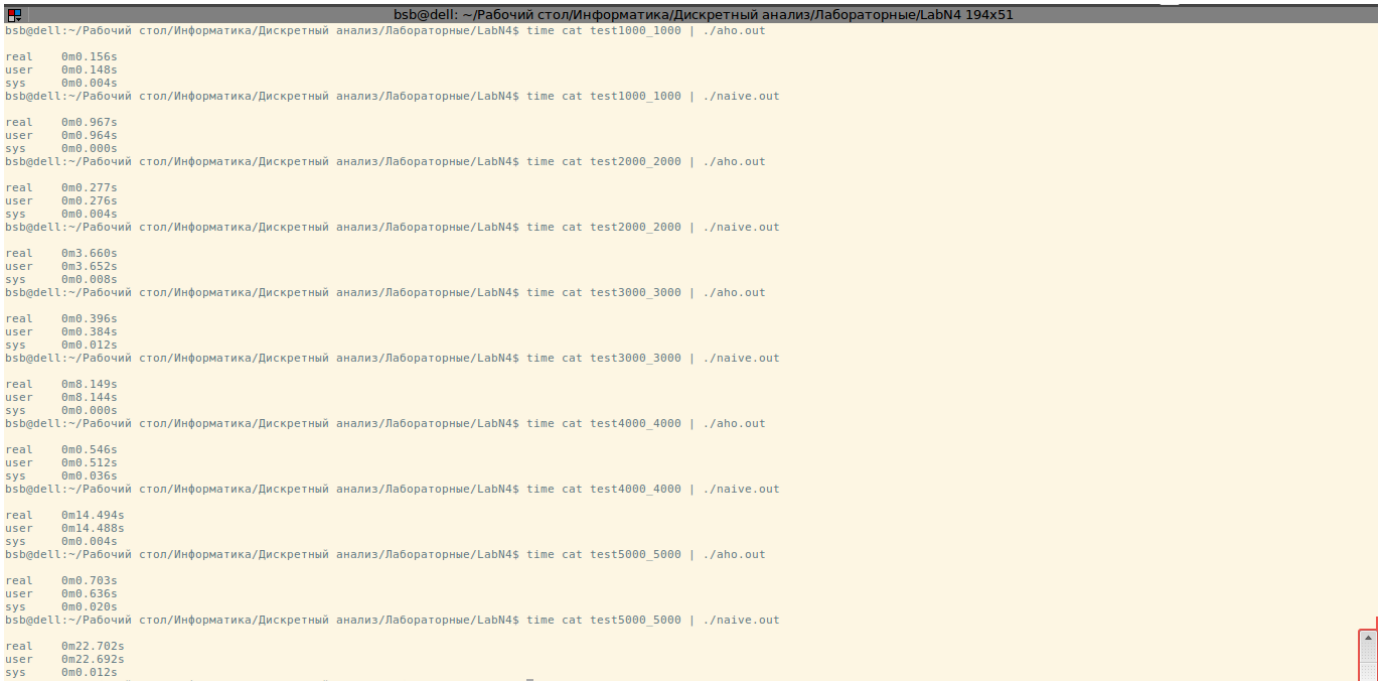
3. Если оно не является пустым и существует следующая за node вершина, помеченная входным словом, то создаём очередь всех найденных паттернов (Ps). Если следующая за node вершина (node_next), отмеченная входным словом является концом какого-либо паттерна, то добавляем её в очередь. Далее, нужно пройти по всем ссылкам, начинаная с node_next, до корня. В процессе, если встречаем вершину, являющуюся концом паттерна, то добавляем её в очередь. Последовательно извлекаем из очереди все вершины и выводим для каждой соотвествующие параметры, вычисляя номер строки и номер слова в строке. Далее, если слово является последним в строке, то повторяем действия из пункта 2 и увеличиваем переменную number_of_word на единицу. Переходим к пункту 1.

4. Если слово не является пустым и не существует следующей за node вершины, помеченной входным словом и в данный момент мы находимся в корне, то, если это

слово является последним в строке, то повторяем действия из пункта 2 и увеличиваем переменную number_of_word на единицу. Переходим к пункту 1.

5. Если слово не является пустым и не существует следующей за node вершины, помеченной входным словом и в данный момент мы не находимся в корне, то проходим по ссылке, переходя к пунктам 2-5.

Бенчмарк



Описание программы

Программа написана одним файлом.
Структуры данных:
class TTrie — класс, описывающий trie.
struct TTrieNode — структура, являющаяся вершиной trie.
Функции:
void CreateTTrieNode(TTrieNode, std::string, TTrieNode*)* — функция, инициализирующая объект TTrieNode.
TTrie::TTrie() — конструктор TTrie.
TTrie:: TTrie() — деструктор TTrie.
void TTrie::Search() — поиск паттернов в тексте.
void TTrie::SetLink() — установка ссылок для вершин.

void TTrie::Insert() — вставка в trie.
std::string GetWord(int status)* — функция, возвращающая слово и устанавливающая в status определённое значение.
int BinarySearch(std::vector < int >&, int, int, int) — бинарный поиск. Используется для нахождения номера строки при поиске паттернов в тексте.

Дневник отладки

Посылка 1: *Status:* Wrong answer at test 01.t. Проблемы: программа не учитывала, что входные слова могут быть разделены произвольным количеством пробелов.
Посылка 2: *Status:* Wrong answer at test 01.t. Проблемы: программа также не учитывает, что входные слова могу быть разделены. Исправления предыдущей версии программы не исправили ошибку.
Посылка 3: *Status:* Wrong answer at test 02.t. Проблемы: отправлена программа только с выводом, чтобы проверить, что вывод в первом тесте корректный.
Посылки 4-7: *Status:* Wrong answer at test 01.t. Проблемы: программа также не учитывает, что входные слова могу быть разделены. Исправления предыдущей версии программы не исправили ошибку.
Посылка 8: *Status:* Wrong answer at test 06.t. Проблемы: найдена ошибка в алгоритме поиска паттернов в тексте.
Посылка 9: *Status:* Wrong answer at test 04.t. Проблемы: найдена ошибка в алгоритме поиска паттернов в тексте. Исправления предыдущей версии программы не исправили ошибку.
Посылка 10: *Status:* Wrong answer at test 10.t. Проблемы: найдена вторая ошибка в алгоритме поиска паттернов в тексте.
Посылка 11: *Status:* OK.
Во всех следующий посылках убирались коментарии, исправлялись мелкие недочёты.

Недочёты

Выявленных недочётов нет.

Выводы

С помощью алгоритма Ахо-Корасик можно эффективным образом найти все вхождения нескольких строк (паттернов) а заданную строку (текст). Данный алгоритм широко применяется в системном программном обеспечении и используется в антивирусах и другом программном обеспечени, где требуется быстро найти вхождения заранее известных строк. Его преимущество в том, что он позволяет за линейное время произвести поиск. Вычислительная сложность алгоритма (если хранить переходы в виде массива) — $O(m + n + k)$, расход памяти — $O(n * \delta)$, где m — длина текста, n — общая длина всех паттернов, δ — размер алфавита, k — полное число вхождений. В реализованном

алгоритме для хранения переходов используется std::map, поэтому его вычислительная сложность — $O((m+n) * log(\delta) + k)$, а расход по памяти — $O(n)$.