Лабораторная работа №4 по курсу дискретного анализа: Строковые алгоритмы

Выполнил студент группы 08-203 **МАИ** *Арусланов Кирилл Антонович*

Условие

Общая постановка задачи: Реализовать строковый алгоритм для поиска образца (pattern) в тексте, состоящем из слов, с использованием Z-алгоритма. Алфавит: латинские буквы (регистронезависимые), слова не более 16 знаков.

Вариант задания: Поиск одного образца на основе построения Z-блоков. Алфавит: слова не более 16 знаков латинского алфавита (регистронезависимые).

Метод решения

Для решения задачи использован Z-алгоритм, который позволяет находить все вхождения образца в тексте за линейное время (O(n)), где (n) — длина строки (S = pattern + '\x01' + text). Алгоритм работает следующим образом:

1. **Предобработка текста и образца**:
    - Текст и образец токенизируются на слова с использованием `std::istringstream`.
    - Все символы приводятся к нижнему регистру для регистронезависимости.
    - Слова длиннее 16 символов отбрасываются.
    - Между словами в итоговой строке вставляется ровно один пробел.
2. **Построение строки** (S): Образец и текст объединяются через специальный разделитель `\x01`, который не встречается в алфавите.
3. **Вычисление Z-массива**: Для строки (S) вычисляется массив (Z), где (Z[i]) — длина наибольшего префикса строки (S), совпадающего с подстрокой, начинающейся в позиции (i).
4. **Поиск совпадений**: Если (Z[i] = |pattern|), то в позиции (i - |pattern| - 1) текста начинается вхождение образца.
5. **Преобразование позиций**: Используется массив `charMap` для сопоставления позиций символов в строке с номерами строк и слов в исходном тексте.

Особенностью Z-алгоритма по сравнению с алгоритмами Кнута-Морриса-Пратта (КМП) и Бойера-Мура является то, что он вычисляет массив совпадений для всей строки (S) за один проход, не требуя предварительного построения префикс-функции (как в КМП) или таблицы смещений (как в Бойере-Муре). Это делает Z-алгоритм проще в реализации для случаев, когда нужно найти все вхождения образца, но может быть менее эффективным на больших текстах с редкими совпадениями, где Бойер-Мур использует обратный поиск для ускорения.

Описание программы

Программа реализована в одном файле на C++ и состоит из следующих компонентов:

- **Типы данных**:
    - `struct TAnswer { int strPos, wordPos; }` — структура для хранения позиции вхождения (номер строки и номер слова).
    - `struct Token { std::string word; int line, idx; }` — структура для хранения токенов текста (слово, номер строки, индекс слова в строке).
    - `std::vector<std::pair<int,int>> charMap` — массив для сопоставления позиций символов с номерами строк и слов.
- **Основные функции**:
    - `std::vector<int> computeZ(const std::string& s)` — вычисляет Z-массив для строки (s). Использует технику "окна совпадений" с переменными (l) и (r) для оптимизации.
    - Встроенные функции `to_lower` и `split_into_words` реализованы через `std::transform` и `std::istringstream`.
- **Архитектура**:
    1. Чтение образца из первой строки ввода.
    2. Токенизация образца с фильтрацией слов по длине и приведением к нижнему регистру.
    3. Чтение текста построчно, токенизация с сохранением номеров строк и слов.
    4. Сборка строки (S) и массива `charMap`.
    5. Вычисление Z-массива и поиск вхождений.
    6. Вывод результатов в формате "номер строки, номер слова".

Дневник отладки

Процесс разработки сопровождался следующими этапами и исправлениями:

1. **Первый вариант**: Построчное склеивание текста включало некорректные вхождения. Тесты не проходили.
2. **Добавили нормализацию**: Использовали `operator>>` для токенизации, но потребовалась посимвольная логика для точного соответствия позиций.
3. **Проблема `catcatcat`**: Одиночный образец-слово не учитывал перекрывающиеся вхождения. Рассматривали Z-алгоритм по токенам, но это усложнило реализацию.
4. **Ограничение длины**: Изначально не отбрасывали слова длиннее 16 символов, что привело к WA на тестах с "длинными" лексемами. Добавили фильтрацию.
5. **Финальная версия**: Полная токенизация с фильтром по длине, сборка текста с единичными пробелами, использование `charMap` и классического Z-алгоритма. Все тесты пройдены.

Тест производительности

Для оценки производительности были проведены замеры времени работы программы на текстах различной длины:

- (10^3) слов — 0.0908354 секунды.
- (10^5) слов — 0.247076 секунды.
- (10^6) слов — 1.20815 секунды.
- (10^7) слов — 14.1217 секунды.

Анализ линейности:

- От (10^5) до (10^6) слов (увеличение объёма в 10 раз) время выросло с 0.247076 до 1.20815 секунд, то есть в (1.20815 / 0.247076 \approx 4.89) раза. Это меньше ожидаемого роста в 10 раз, что может быть связано с накладными расходами на ввод-вывод.
- От (10^6) до (10^7) слов (увеличение объёма в 10 раз) время выросло с 1.20815 до 14.1217 секунд, то есть в (14.1217 / 1.20815 \approx 11.69) раза, что ближе к ожидаемому линейному росту.

На больших объёмах данных рост времени становится более линейным, что соответствует теоретической сложности Z-алгоритма (O(n)), где (n) — длина строки (S). Замеры проводились на стандартном оборудовании с учётом операций ввода-вывода.

Выводы

Z-алгоритм показал себя как эффективный инструмент для поиска образца в тексте за линейное время. Он применим в задачах обработки текстов, таких как поиск в редакторах или анализ данных. Основные трудности возникли при обработке граничных случаев (длинные слова), что потребовало тщательной предобработки. Программирование алгоритма оказалось умеренно сложным, потребовало внимания к деталям отладки.