Лабораторная работа № 5 по курсу дискретного анализа: суффиксные деревья

Выполнил студент группы 08-303 МАИ: *Арусланов Кирилл*

Условие

1. **Общая постановка задачи**
   Даны две строки `s1` и `s2`. Требуется найти все общие подстроки максимальной длины — такие строки, которые одновременно встречаются в `s1` и `s2` и имеют наибольшую длину.

2. **Вариант задания**
   Вариант 5: Использовать суффиксное дерево для поиска наибольшей общей подстроки.
   Формат ввода: Две строки.
   Формат вывода: На первой строке нужно распечатать длину максимальной общей подстроки, затем перечислить все возможные варианты общих подстрок этой длины в порядке лексикографического возрастания без повторов.

Метод решения

Решение состоит из двух этапов:

1. **Построение обобщённого суффиксного дерева (Generalized Suffix Tree)**
   Для строки TEXT = s1 + '$' + s2 + '#' используется алгоритм Укконена, который работает за время `O(n)`, где `n = |s1| + |s2| + 2`.

Основные идеи алгоритма:

- **Активная точка** — тройка `(active_node, active_edge, active_length)`, задающая текущее место вставки.

- **Суффиксные ссылки** — позволяют быстро переходить к следующему расширению.
- **Прыжки по счетчику (Skip/Count walkDown)** — перескакивание через целые рёбра, если активная длина больше длины ребра.
- **Start:end** — храним не строки как в трае, а индексы начала подстроки и ее конца.
- **Общий конец листьев** — все листья используют общий указатель `leaf_end`, который обновляется при добавлении символа.

2. **Поиск LCS**

- Выполняется обход DFS и пометка узлов: если в поддереве узла встречаются суффиксы обеих строк, значит путь от корня до этого узла — общая подстрока.
- Среди всех таких узлов выбирается максимальная глубина.
- Подстроки длины `max_len` собираются в `std::set` для упорядочивания и уникальности.

Описание программы

Основные структуры

- **Node**:
- `start`, `*end` — границы подстроки.
- `link` — суффиксная ссылка.
- `next` — переходы (`unordered_map<char,int>`).
- `example_s1`, `example_s2` — индексы суффиксов строк.
- **Вектор** `nodes` хранит все узлы. Корень имеет индекс 0.

Основные функции

- `extend(pos)` — расширение дерева на символ `text[pos]`.
- `build(text)` — построение дерева.
- `setSuffixIndicesAndExamples(...)` — DFS, отмечающий принадлежность суффиксов.

- `findMaxLen(...)` и `collectStrings(...)` — поиск максимальной длины и всех LCS.

Дневник отладки

1. Первичная реализация не находила строку `"bay"` для теста `xabay / xabcbay`. Ошибка — отсутствовал лист для одного суффикса.
2. Исправлено двойное удаление `leaf_end`.
3. Убрана проверка через `std::string::find` (медленная и избыточная).
4. Оптимизировано выделение памяти и исправлены ошибки в заполнении `example_s1`, `example_s2`.

Тест производительности

Данные: случайные строки длиной 1MB, 2MB, 4MB, 6MB.

Результаты:

Размер строк	Время построения (с)	Время поиска (с)	Память (MB)
1 MB	4.4	1.7	411
2 MB	8.2	4.5	719
4 MB	22.2	11.3	1342
6 MB	36.8	16.5	2045

Вывод: время и память растут почти линейно с увеличением входных данных, что подтверждает теоретическую сложность алгоритма Укконена.

Недочёты

- Используется `unordered_map` для переходов; можно, наверное, ускорить с помощью массива фиксированного размера.
- DFS рекурсивный, есть риск переполнения стека.

- На лекциях рассматривался более быстрый способ построения обобщенного суффиксного дерева, который у меня не получилось реализовать.

Выводы

Реализованный алгоритм построения суффиксного дерева по Укконену применим во множестве задач, связанных с обработкой строк.
К типовым задачам относятся:

- поиск наибольшей общей подстроки двух строк (как в данной работе);
- поиск всех вхождений подстроки в тексте;
- нахождение различных повторов и палиндромов;
- сжатие данных и построение индексов для быстрых текстовых запросов.

Алгоритм отличается линейной асимптотикой по времени и памяти, но обладает большой константой и требует аккуратной реализации.
Сложность программирования достаточно высокая: необходимо поддерживать активную точку, корректно управлять суффиксными ссылками и концами рёбер.

В ходе выполнения работы возникли трудности с корректным обновлением листьев и управлением памятью. После устранения ошибок алгоритм стал корректно и эффективно находить LCS.
Таким образом, задача была решена, а пройденные тесты подтверждают теоретическую линейную сложность алгоритма.