Лабораторная работа № 4 по курсу : Дискретный анализ

Выполнил студент группы М8О-208Б-17 МАИ *Милько Павел.*

Условие

- Постановка задачи:

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

- Вариант задания

  Алгоритм: Ахо-Корасик

  Алфавит: Слова менее 17 знаков латинского алфавита(без учёта регистра).

Метод решения

Алгоритм Ахо-Корасик оказался не очень сложным в реализации, так как легко делится на несколько частей:

- Создание структуры дерева ключей
- Добавление суффиксных ссылок в дерево ключей
- Поиск

Дерево ключей:

Для такого дерева даже есть отдельное название – бор. Он представляет из себя префиксное дерево, с дополнениями в виде суффиксной ссылки, ссылки выхода и собственно порядковый номер ключа. Для обозначения правил перехода я использовал std::map – словарь, который реализован в виде красно-чёрного дерева

Сам алгоритм вставки ключа в дерево тривиален и реализуется в несколько строк:

```
curr = root;
while (line >> temp) {
    if (!curr->links.count(temp))
        curr->links[temp] = std::shared_ptr<TAho::TBor>(new
                                                TAho::TBor(root.get()));
    curr = curr->links[temp];
}
curr->key_num = ++key_count;
```

Создание суффиксных ссылок и ссылок выхода:

Тут уже алгоритм более интересный. Он основан на поиске в ширину и использует на последующих шагах уже вычисленные ранее суффиксные ссылки.

Изначально все вершины бора (кроме корня), имеют суффиксные ссылки, указывающие на корень.

Алгоритм основан на обходе дерева в ширину, для корректности обхода использована структура данных очередь. Изначально в очереди лежит корень, он же обрабатывается первым, после обработки корня в очереди находятся все его дети, в том порядке, в котором они были обработаны.

На каждом шаге алгоритм рассматривает по очереди всех возможных детей текущего узла и если по суффиксной ссылке текущего узла найдётся такая же строка перехода, что и в ребёнка, то суффиксная ссылка для ребёнка найдена, иначе нужно пройти по следующей суффиксной ссылке, выходящей из суффиксной ссылки родителя. По суффиксным ссылкам передвигаться пока не будет найдена совпадающая строка перехода, либо пока суффиксная ссылка не станет *nullptr* (ссылка из корня). В последнем случае не нужно менять суффиксную ссылку ребёнка, так как она уже показывает на корень.

Построение ссылок выхода можно вставить сразу после создания суффиксной ссылки для узла. Если ссылка является ключом, то ссылка выхода будет указывать на него, иначе на его ссылку выхода.

Далее, для корректности обхода в глубину нужно добавить в очередь только что обработанного ребёнка.

Поиск:

Поиск так же тривиален, нужно лишь написать правило перехода от узла (состояния) к узлу (состоянию):

```
void TAho::Next(std::string& str)
{
    while (state) {
        if (state->links.count(str)) {
            state = state->links[str].get();
            return;
        }
        state = state->suff_link;
    }
    state = root.get();
}

void TAho::Search(std::vector<std::string>& lines,
                  std::vector<std::pair<int, int>>& result)
{
    Suffix();

    size_t size = lines.size();
    for (ull i = 0; i < size; ++i) {
        Next(lines[i]);
        Show(i, result);
    }
}
```

Дневник отладки

При отправке на чекер было сразу выявлено нескольно недостатков:

- нужно было обрабатывать все строки для поиска как цельный текст, а не по отдельности (упало на первом же тесте)
- В тексте могут встречаться пустые строки (у меня программа завершала работу при нахождении пустой строки)
- Time limit exceeded at test 14.t. Не хватило времени, я расстроился и пошёл делать оптимизации.

  Так как в этом тесте на вход подавались очень длинные ключи, то решил начать с оптимизации ввода. До этого считывание строки происходило в функции $main$, и слова записывались в вектор строк. Неэффективно, подумал я, и сделал вставку "сходу", без записи в вектор.

  Это не помогло, лимит по времени снова оказался не пройден. Когда я уже почти убедил себя переписывать программу на Си, увидел, что в функцию поиска аргументы передаются по значению, а не по ссылке, что весьма плохо отражается на быстродействии. Это исправление чекеру понравилось, и дальше были только косметические правки.

Выводы

Поиск подстроки в строке довольно распространённая задача в наши дни. Поиск нескольких образцов встречается уже реже, но иногда бывает полезен, например для поиска различных форм слова в тексте (статистика использования).

Для перехода от вершины к вершине использовался словарь map, он использовался из за простоты использования и ради избежания расходов памяти (если сопоставить каждому возможному слову индекс массива, то затраты памяти будут огромны при незначительном приросте к производительности). Вставка в map и поиск в нём занимают $O(log_2 k)$ по времени и $O(k)$ по памяти, где k – это количество вершин текущего узла.

Несмотря на вкусности этого алгоритма иногда (почти всегда) бывает проще использовать регулярные выражения, так как они намного проще и привычнее, чем собственноручное создание алгоритма.