Московский авиационный институт
(национальный исследовательский университет)

Институт информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №4 по курсу «Дискретный анализ»

Студент: И. С. Глушатов
Преподаватель: А. А. Кухтичев
Группа: М8О-207Б-19
Дата:
Оценка:
Подпись:

Москва, 2020

Лабораторная работа №4

Задача: Необходимо реализовать один из стандартных алгоритмов поиска образцов для указанного алфавита. Запрещается реализовывать алгоритмы на алфавитах меньшей размерности, чем указано в задании.

Формат входных данных:
Искомый образец задаётся на первой строке входного файла.

В случае, если в задании требуется найти несколько образцов, они задаются по одному на строку вплоть до пустой строки.

Затем следует текст, состоящий из слов или чисел, в котором нужно найти заданные образцы.

Никаких ограничений на длину строк, равно как и на количество слов или чисел в них, не накладывается.

Формат результата:
В выходной файл нужно вывести информацию о всех вхождениях искомых образцов в обрабатываемый текст: по одному вхождению на строку.

Для заданий, в которых требуется найти только один образец, следует вывести два числа через запятую: номер строки и номер слова в строке, с которого начинается найденный образец. В заданиях с большим количеством образцов, на каждое вхождение нужно вывести три числа через запятую: номер строки; номер слова в строке, с которого начинается найденный образец; порядковый номер образца.

Нумерация начинается с единицы. Номер строки в тексте должен отсчитываться от его реального начала (то есть, без учёта строк, занятых образцами).

Порядок следования вхождений образцов несущественен.

Вариант алгоритма: Поиск одного образца основанный на построении Z-блоков.

Вариант алфавита: Слова не более 16 знаков латинского алфавита (регистронезависимые)

1 Описание

Z-функция от строки S^n длины n – это массив M^n длин, где каждый элемент M_i^n является максимальной длиной общего префикса строки S^n и суффикса этой же строки, начиная с позиции i.

Наивный алгоритм предполагает перебор для каждого M_i^n ответа, пока не обнаружим несовпадение или не дойдем до конца строки.

Моя реализация на Python3 (третья версия нужна была для аннотации типов):

```
1 def z_function(pattern: str) -> list:
2   n = len(pattern)
3   result = [0 for _ in range(n)]
4   for i in range(1, n):
5     while i + result[i] < n and pattern[result[i]] == pattern[i + result[i]]:
6       result[i] += 1
7   return result
```

Сложность такого алгоритма $O(n^2)$. Плохая ассимптотика хорошо наблюдается на строках, типа: "aaaaaaaa". У данного алгоритма происходит много сравнений элементов, которые уже сравнивались, и помимо этого информация об этих сравнениях заложена в массиве до i'ого элемента, но никак не используется. Поэтому для приведения алгоритма к линейной сложности $O(n)$ вводится понятие Z-блоков.

Z-блок – это подстрока строки S^n с максимальной правой границей. Максимальная правая граница определяется предыдущими сравнениями. Если ввести понятия left и right как индексы границ Z-блока, то right будет равняться максимальному индексу элемента, который мы уже сравнивали и при этом сравнение было удачно (т.е. максимальное значение i + zf[i] из всех i, пройденных ранее). Тогда алгоритм будет выглядить следующим образом:

Если i $>$ right, то сравниваем наивным методом.

Если i $<=$ right, то нам достаточно рассмотреть символы за позицией right, так как до этой позиции мы гарантируем все совпадения, потому что вычисляли их до этого.

Таким образом мы пройдемся по каждому символу два раза и сложность алгоритма будет линейной.

2 Исходный код

Так как алфавит состоит по заданию из слов длиной до 16 символов, а при выводе на экран мы должны выводить строку и номер «буквы», с которой найдено вхождение, то я создал структуру с тремя полями, которая хранит само слово, являющимся «буквой», индекс строки на которой оно находится и его порядковый номер в этой строке. Переопределил оператор сравнения «operator==». Константа TLETTER_SIZE определяет максимальный размер слова (по заданию 16 + 1 для знака конца строки).

```
const short TLETTER_SIZE = 17;

struct TLetter {
    char letter[TLETTER_SIZE] {'\0'};
    unsigned int strIndex;
    unsigned int textIndex;

    TLetter() = default;

    TLetter(std::string* s, unsigned int si, unsigned int ti): strIndex(si), textIndex(
        ti) {
        for (unsigned int i = 0; i < s->size(); ++i) {
            letter[i] = (*s)[i];
        }
    };

    TLetter(std::string s, unsigned int si, unsigned int ti): strIndex(si), textIndex(
        ti) {
        for (unsigned int i = 0; i < s.size(); ++i) {
            letter[i] = s[i];
        }
    };

    ~TLetter() = default;

    const bool operator==(const TLetter &rval) const {
        for (unsigned int i = 0; i < TLETTER_SIZE; ++i) {
            if (letter[i] != rval.letter[i]) {
                return false;
            }
        }
        return true;
    }
};
```

Далее сама Z-функции.

```
unsigned int* ZFunction(const std::vector<TLetter> pattern) {
    unsigned int patternSize = pattern.size();
    unsigned int* result = new unsigned int[patternSize] {0};
    unsigned int left = 0;
    unsigned int right = 0;
    for (unsigned int i = 1; i < patternSize; ++i) {
        if (i <= right) {
            result[i] = std::min(right - i, result[i - left]);
        }
        while (i + result[i] < patternSize and pattern[result[i]] == pattern[i + result
            [i]]) {
            ++result[i];
        }
        if (i + result[i] > right) {
```

```
            left = i;
            right = i + result[i];
        }
    }
    return result;
}
```

Маленькая функция, которая переводит символы в нижний регистр, так как по заданию слова регистронезависимые.

```
void ToLower(char* str) {
    if (*str>='A' and *str<='Z') {
        *str -= 'A'-'a';
    }
}
```

В main мы парсим входные данные. Сначала считываем паттерн и записываем в вектор термов, далее добавляем разделяющий символ (пусть будет #) и начинаем считывать текст параллельно записывая в структуру индекст строки и индекс терма в этой строке во время ввода. Перед тем как добавить разделяющий символ мы проверяем, не пустой ли у нас паттерн. В этом случае программа завершается. Перед тем как добавлять очередной терм, проверяем, не пустой ли он. Пустые игнорируем. Далее считаем Z-функцию и выводим ответ.

```
int main() {

    std::vector<TLetter> text;

    unsigned int strNumber = 0;
    unsigned int textNumber = 0;
    unsigned int patternSize = 0;

    std::string letter = "";
    char c;
    while ((c = getchar()) != EOF) {
        if (c == ' ') {
            if (letter != "") {
                text.push_back(TLetter(&letter, strNumber, textNumber));
                ++textNumber;
                ++patternSize;
                letter = "";
            }
        } else if (c == '\n') {
            if (letter != "") {
                text.push_back(TLetter(&letter, strNumber, textNumber));
                ++textNumber;
                ++patternSize;
                letter = "";
            }
            break;
        } else {
            ToLower(&c);
            letter += c;
        }
    }

    if (!text.size()) {
        return 0;
    }

    text.push_back(TLetter(std::string("#"), strNumber, 0));
```

```
38      strNumber = 1;
39      textNumber = 0;
40
41      while ((c = getchar()) != EOF) {
42          if (c == ' ') {
43              if (letter != "") {
44                  text.push_back(TLetter(&letter, strNumber, textNumber));
45                  ++textNumber;
46                  letter = "";
47              }
48          } else if (c == '\n') {
49              if (letter != "") {
50                  text.push_back(TLetter(&letter, strNumber, textNumber));
51                  letter = "";
52              }
53              textNumber = 0;
54              ++strNumber;
55          } else {
56              ToLower(&c);
57              letter += c;
58          }
59      }
60
61      if (letter != "") {
62          text.push_back(TLetter(&letter, strNumber, textNumber));
63      }
64
65      unsigned int* zf = ZFunction(text);
66
67      for (unsigned int i = 0; i < text.size(); ++i) {
68          if (zf[i] == patternSize) { s
69              printf("%d, %d\n", text[i].strIndex, text[i].textIndex+1);
70          }
71      }
72
73      delete []zf;
74      return 0;
75  }
```

3 Консоль

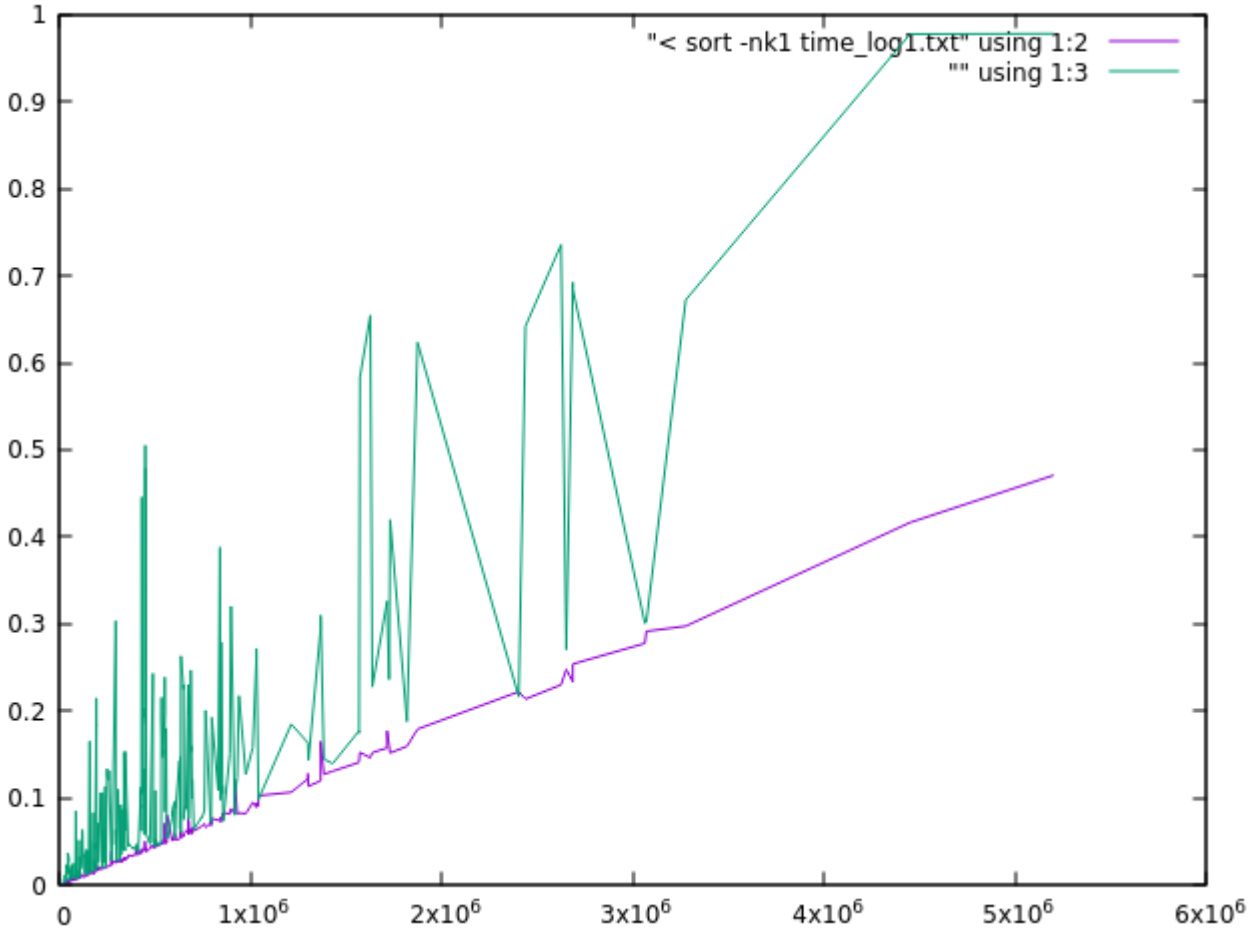
```
igor@igor-Aspire-A315-53G:~/Рабочий стол/c++/DA/lab4$ cat test1.txt && make
run <test1.txt
cat dog cat dog bird
CAT dog CaT Dog Cat DOG bird CAT
dog cat dog bird./main.out
1,3
1,8
igor@igor-Aspire-A315-53G:~/Рабочий стол/c++/DA/lab4$
```

4 Тест производительности

Для того, чтобы сравнить ассимптотику алгоритмов, я написал тестер, генерирующий различные паттерны и тесты. Я сравнивал две реализации Z-функций. С линейной и квадратичной сложностью.

```
1  unsigned int* LowZFunction(const std::vector<TLetter> pattern, unsigned int par_size)
       {
2      unsigned int patternSize = pattern.size();
3      unsigned int* result = new unsigned int[patternSize] {0};
4
5      for (unsigned int i = 1; i < patternSize; ++i) {
6          while (i + result[i] < patternSize and pattern[result[i]] == pattern[i + result
               [i]]) {
7              ++result[i];
8          }
9      }
10     return result;
11 }
12
13 unsigned int* ZFunction(const std::vector<TLetter> pattern, unsigned int par_size) {
14     unsigned int patternSize = pattern.size();
15     unsigned int* result = new unsigned int[patternSize] {0};
16     unsigned int left = 0;
17     unsigned int right = 0;
18     for (unsigned int i = 1; i < patternSize; ++i) {
19         if (i <= right) {
20             result[i] = std::min(right - i, result[i - left]);
21         }
22         while (i + result[i] < patternSize and pattern[result[i]] == pattern[i + result
               [i]]) {
23             ++result[i];
24         }
25         if (i + result[i] > right) {
26             left = i;
27             right = i + result[i];
28         }
29     }
30     return result;
31 }
```

Я сгенерировал около 200 тестов. Скорость выполнения первой и второй функции я замерял с помощью библиотеки std::chrono в секундах, тип double. Сначала оба алгоритма работали с одинаковой скоростью, а наивный метод иногда даже выигрывал. Но я понял ошибку в генерируемых тестах - все слова паттерна были различны и получалось, что тесты не давали возможности продемонстрировать ускорение из-за тривиальности тестов. Изменив генерацию, я провел тесты заново. Результаты времени работы двух функций и количество элементов выводились в файл, с помощью которого я построил график в gnuplot.



Зеленым отмечен график Z-функции, реализованной наивным методом, а синим - с помощью Z-блоков. Как мы видим первый начинает все больше отставать от второго с увеличением количества слов. При этом ненаивная реализация показывает совсем плавный рост, что для меня было удивительно.

5 Выводы

В ходе лабораторной работы я познакомился с алгоритмами Z-функций. Увидел, что функция, использующая Z-блок отличается лишь двумя if'ами и введеним двух целочисленных переменных. Очень долго не понимал, где происходят утечки памяти. Оказалось, что при вызове new[] нужно сразу писать "0". Так же изначально я использовал std::string, но программа "кушала"много памяти, поэтому пришлось использовать массив char'ов. Проваливал тесты из-за считывания пустых термов и неправильного подсчета переносов строк. Учитывание всего этого заставило парсить без std::cin и scanf, что достаточно увеличило код.

Список литературы

[1] *Z-функция.*
URL: https://ru.wikipedia.org/wiki/Z-функция

[2] *Про std::string в C++*
URL: https://en.cppreference.com/w/cpp/string/basic_string

[3] *Еще про Z-функцию*
URL: https://e-maxx.ru/algo/z_function