Московский авиационный институт (национальный исследовательский университет)

Факультет компьютерных наук и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №4 по курсу «Дискретный анализ»

Студент: С. Ю. Свиридов
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б-22

Дата:
Оценка:
Подпись:

Москва 2024

Лабораторная работа №4

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

Вариант алгоритма: Поиск одного образца при помощи алгоритма Кнута-Морриса-Пратта.

Вариант алфавита: Числа в диапазоне от 0 до $2^{32} - 1$.

Запрещается реализовывать алгоритмы на алфавитах меньшей размерности, чем указано в задании.

Формат ввода

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 вывода

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.

1 Описание

Алгоритм Кнута — Морриса — Пратта (КМП-алгоритм) — эффективный алгоритм, осуществляющий поиск подстроки в строке, используя то, что при возникновении несоответствия само слово содержит достаточно информации, чтобы определить, где может начаться следующее совпадение, минуя лишние проверки. Время работы алгоритма линейно зависит от объёма входных данных, то есть разработать асимптотически более эффективный алгоритм невозможно. Алгоритм был разработан Д. Кнутом и В. Праттом и, независимо от них, Д. Моррисом.

Алгоритм работает на основе префикс функции, выполняющей поиск всех возможных суффиксов, которые являются также его префиксами, и использование этой информации для сдвига указателя поиска при несовпадении символов.

В своей лабораторной работе я использовал реализацию алгоритма, работающего на основе так называемого массива "Пи". О массиве будет рассказано далее.

2 Исходный код

Опишем алгоритм КМП, который будет принимать на вход два вектора, являющихся шаблоном поиска и текстом, внутри которого будет выполнен поиск вхождения шаблона. Перед поиском, нужно создать массив ПИ для того, чтобы знать как сдвигать указатели в случае несовпадения слов. Для этого создадим пустой вектор длиной как шаблон и первый его элемент сделаем нулем. Далее создадим две переменных, которые будут являться индексами в массиве ПИ.

После чего начинается сравнение элементов в векторе, хранящем в себе шаблон для поиска. Массив ПИ заполняется в зависимости от результатов сравнения и значений индексов. В конце алгоритма, как только был пройден весь массив `pattern`, сравнение прекращается и массив ПИ считается заполненным. Заполненный массив ПИ хранит в себе длину наибольшего суффикса подстроки `pattern[i]` (подстрока до индекса i в шаблоне), который является также префиксом этой подстроки.

То есть например в случае, если шаблоном является вектор вида: `[11, 45, 11, 45, 90]`, то массив ПИ будет иметь вид: `[0, 0, 1, 2, 0]`. Все потому, что в строке `[11, 45, 11]` максимальная длина суффикса, совпадающего с префиксом равняется 1. А в строке `[11, 45, 11 ,45 ]` такая длина равняется 2.

После формирования массива ПИ начинается поиск вхождений. Начинаем сравнивать все слова в `pattern` со словами из `text`. В случае несовпадения происходит обращение к массиву ПИ с целью сместить индексы и не сравнивать ненужные слова. Таким образом, поиск выполняется быстрее. Как только вхождение найдено, в вектор `result` добавляется номер слова, с которого вхождение началось. Далее поиск продолжается с целью поиска следующего вхождения; для этого просто меняем значение индекса j по правилу `j = PI[j - 1]`. Это позволит нам избегать возврата к началу шаблона и вернуться к нужному слову. После этого алгоритм вновь начинает поиск вхождений, но уже не с начала шаблона, а после УЖЕ СОВПАВШЕЙ его части.

```
1  std::vector<int> KMP(const std::vector<unsigned long long>& pattern, std::vector<
       unsigned long long>& text) {
2      std::vector<int> PI(pattern.size());
3
4      PI[0] = 0;
5      int j = 0, i = 1;
6      while (i < static_cast<int>(pattern.size())) {
7          if (pattern[i] != pattern[j]) {
8              if (j == 0) {
9                  PI[i] = 0;
10                 i++;
11             }
12             else {
13                 j = PI[j - 1];
14             }
```

```
        }
        else {
            PI[i] = j + 1;
            i++;
            j++;
        }
    }

    i = 0;
    j = 0;
    std::vector<int> result;
    while (i < static_cast<int>(text.size())) {
        if (text[i] == pattern[j]) {
            i++;
            j++;
            if (j == static_cast<int>(pattern.size())) {
                result.push_back(i - j + 1);
                j = PI[j - 1];
            }
        }
        else {
            if (j > 0) {
                j = PI[j - 1];
            }
            else {
                i++;
            }
        }
    }
    return result;
}
```

Далее рассмотрим основную функцию программы, которая формирует корректный ввод и вывод информации. Первой строкой пользователь вводит образец для поиска в тексте. С помощью обычного цикла while пользователь вводит числа, которые ему нужны и далее, когда будет введен символ переноса строки, ввод образца будет закончен и начнется ввод текста.

```
int main() {
    std::vector<unsigned long long> pattern;
    unsigned long long num;

    while (std::cin >> num) {
        pattern.push_back(num);
        char nextchar = std::cin.get();
        while (nextchar == ' ') {
            nextchar = std::cin.get();
        }
        if (nextchar == '\n') {
```

```
12              break;
13          }
14          else {
15              std::cin.unget();
16          }
17      }
18      ...
```

Далее начинается ввод текста. Поскольку он может быть сколь угодно большим, логично было бы считать его по частям, а не целиком, ведь в таком случае, может не хватить памяти компьютера. Итак, вновь начинается обычный цикл. В нем пользователь вводит числа. Допустим ввод незначащих нулей и длинных пробелов между словами. В цикле считается количество строк, в том числе и пустых, а слова добавляются в вектор `text`. Как только размер вектора `text` в два раза превысит размер вектора `pattern`, то запускается алгоритм КМП, который ищет вхождения образца в эту часть текста. Далее, результат работы алгоритма КМП отправляется в вектор `result`. Затем каждый элемент этого вектора увеличивается на значение переменной `summ`, которая хранит в себе число, на единицу больше разности между размерами считанного текста, и образца. Оно суммируется и получается, что в нем хранится число считанных слов для каждой итерации цикла. Прибавление нужно для того, чтобы в результирующем векторе хранились индексы вхождений образца, начиная с начала всего текста, а не с начала текущей его части. Далее сам текст копируется таким образом, чтобы не пропустить вхождение. В новую часть текста записываются сначала последние несколько слов предыдущего текста, а именно `text.size() - pattern.size() + 1`, а затем уже следующие введнные символы. Так сделано для того, чтобы не пропустить ни одного вхождения. Итак, после окончания этого цикла, когда весь текст прочитан, мы получаем итоговый вектор, хранящий в себе все номера слов, с которых начинаются вхождения образца во всем тексте, а так же еще некоторые вспомогательные векторы. Первый вектор `NotEmptyLinesNumbers` хранит в себе номера непустых строк. Второй вектор `SummOfNumbersForEachLine` хранит в себе сумму слов относительно каждой непустой строки, то есть n-й элемент вектора будет содержать в себе сумму слов на 0, 1, ..., n-1, n непустых строках. По размерам он совпадает с вектором `NotEmptyLinesNumbers`. Третий вектор `LineNumbersQuantity` хранит в себе количетство элементов на каждой непустой строке. По размерам он также совпадает с вектором `NotEmptyLinesNumbers`. Эти три вектора формируются в цикле while. Затем с помощью них формируются следующих два вектора, которые помогут сделать правильный вывод результатов. Оба они имеют длину, равную количеству строк, в том числе и пустых. Вектор `NumsQuantityOfEveryLines` хранит в себе количество элементов для каждой, В ТОМ ЧИСЛЕ И ПУСТОЙ, строки. А вектор `NumbersSummForAllLines` хранит в себе сумму слов на 0, 1, ..., n-1, n строках, В ТОМ ЧИСЛЕ И ПУСТЫХ. Перед финальным выводом результатов, нужно еще раз запустить алгоритм КМП для оставшейся части текста, если таковая имеется. А

таковая будет иметься в случае, если в тексте содержится нечетное количество слов. Для вывода нужно рассмотреть каждый элемент вектора FinalResult, а именно, в цикле сравнить каждый его элемент с элементами вектора `NumbersSummForAllLines`. Как только будет выполнено условие (FinalResult[k] <= NumbersSumForAllLines[j]), означающее, что номер вхождения (номер слова) меньше или равно сумме слов на 0, 1, ..., j-1, j строках, В ТОМ ЧИСЛЕ И ПУСТЫХ, мы выводим `j - 1`, то есть номер строки, в которой имеется вхождение. А чтобы корректно вывести номер слова (для каждой строки нумерация слов должна начинаться с нуля), нужно просто из элемента `FinalResult[k]` вычесть количество слов на всех предыдущих строках до j. Это делается с помощью функции `CalculateSumm`. Таким образом получается правильный вывод.

```
 ...

while(std::getline(std::cin, line)) {
      if (line.empty()) {
          QuantityOfLines++;
          continue;
      }

      std::stringstream ss(line);
      while (ss >> num) {
          not_empty = true;
          SummNumbersOfLine++;
          numbersOfLine++;
          text.push_back(num);
          if (text.size() >= pattern.size() * 2) {
              result = KMP(pattern, text);
              for (int res = 0; res < result.size(); res++) {
                  result[res] += summ;
              }
              FinalResult.insert(FinalResult.end(), result.begin(), result.end());

              summ += text.size() - pattern.size() + 1;
              text = copy(text, text.size() - pattern.size() + 1);

          }
      }

      QuantityOfLines++;
      NotEmptyLinesNumbers.push_back(QuantityOfLines);
      SummOfNumbersForEachLine.push_back(SummNumbersOfLine);
      LineNumbersQuantity.push_back(numbersOfLine);
      numbersOfLine = 0;
  }

  if (pattern.empty() || !(not_empty)) {
      return 0;
```

```
    }

    result = KMP(pattern, text);
    for (int i = 0; i < result.size(); i++) {
        result[i] += summ;
    }

    int max = NotEmptyLinesNumbers.back();

    std::vector<int> NumsQuantityOfEveryLines(max, 0);
    std::vector<int> NumbersSumForAllLines(max, 0);
    for (int i = 0; i < NotEmptyLinesNumbers.size(); i++) {
        NumsQuantityOfEveryLines[NotEmptyLinesNumbers[i] - 1] = LineNumbersQuantity[i];
        NumbersSumForAllLines[NotEmptyLinesNumbers[i] - 1] = SummOfNumbersForEachLine[i
            ];
    }

    FinalResult.insert(FinalResult.end(), result.begin(), result.end());
    for (int k = 0; k < FinalResult.size(); k++) {
        for (int j = 0; j < NumbersSumForAllLines.size(); j++) {
            if (FinalResult[k] <= NumbersSumForAllLines[j]) {
                std::cout << j + 1 << ", " << FinalResult[k] - CalculateSumm(
                    NumsQuantityOfEveryLines, j) << std::endl;
                break;
            }
        }
    }

    return 0;
}
```

Описание вспомогательных функций для корректного считывания и вывода результата:

```
int CalculateSumm(std::vector<int>& array, int idx) {
    int sum = 0;
    for (int i = 0; i < idx; i++) {
        sum += array[i];
    }
    return sum;
}

std::vector<unsigned long long> copy(std::vector<unsigned long long>& text, int idx) {
    std::vector<unsigned long long> newText;
    for (int i = idx; i < text.size(); i++) {
        newText.push_back(text[i]);
    }
    return newText;
}
```

3 Консоль

```
stepan@stepan-ASUS:~/Рабочий стол/учеба/prog4sem/discran/laba4/src$ ./a.out
11 11



0000000124                                          11

000000011



11                11        11 124 12           0182        0011       11


11
4,2
6,1
10,1
10,2
10,7
10,8
```

4 Тест производительности

Тесты производительности представляют из себя следующее: будет дан шаблон и тексты разной длины. Сравиваться будут время работы наивного алгоритма поиска подстроки в строке и алгоритма Кнута-Морриса-Пратта.

```
stepan@stepan-ASUS:~/Рабочий стол/учеба/prog4sem/discran/laba4/src$ ./a.out
Text length: 9504
Naive Algorithm Working time: 136ms
KMP Algorithm Working time: 104ms

Text length: 47520
Naive Algorithm Working time: 627ms
KMP Algorithm Working time: 476ms

Text length: 142560
Naive Algorithm Working time: 1919ms
KMP Algorithm Working time: 1391ms
```

Как видно из результатов теста, алгоритм Кнута-Морриса-Пратта оказался быстрее наивного на всех тестах. Этот результат был ожидаемым, поскольку сложность алгоритма КМП составляет $\Theta(n + m)$, где n - длина текста, а m - длина образца. Так происходит потому, что массив ПИ формируется за $\Theta(m)$ и затем, на его основе выполняется поиск вхождений. А сложность наивного алгоритма - $\Theta(n * m)$, потому что для каждой позиции текста длины n мы сравниваем все символы шаблона длины m. Отсюда такая разница в скорости.

5 Выводы

Выполнив четвертую лабораторную работу по курсу «Дискретный анализ», я еще раз поработал с алгоритмом КМП. Протестировал его, написал код и оптимищировал его для считивания и обработки больших данных с минимальными затратами по памяти. В целом, сам алгоритм довольно простой, но вот реализация правильного чтения данных вызвало у меня затруднения, однако, несмотря на это, я считаю, что справился с поставленной задачей.

Список литературы

[1] *Алгоритм Кнута — Морриса — Пратта*
URL: https://ru.wikipedia.org/wiki/Алгоритм_

[2] *КМП на python*
URL: https://www.youtube.com/watch?v=S2I0covkyMct=440s/