Лабораторная работа №4 по курсу Дискретного Анализа: Поиск за линейное время

Выполнил студент группы 08-208 МАИ Жерлыгин Максим Андреевич

Условие

Необходимо реализовать один из стандартных алгоритмов поиска образцов для указанного алфавита.
Вариант алгоритма: Поиск одного образца при помощи алгоритма Кнута-Морриса-Пратта.
Вариант алфавита: Слова не более 16 знаков латинского алфавита (регистронезависимые)
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.

Метод решения

По заданию требуется реализовать алгоритм поиска подстроки в строке за линейное время. Для начала найдем префикс функцию, которая будет для каждого элемента паттерна ставить в соответствие число - максимальную длину суффикса строки от первого до элемента, который равен префиксу всей строк. С помощью данной функции при КМП обходе строка патерна будет сдвигаться на число элементов заданное перфикс функцией.

Кроме этого, так как в здании сказано, что файл может быть сколь угодной длины, значит нужно научить программу обрабатывть поток частями, при этом не теряя и не перебирая несколько раз возможные вхождения.

«Самый известный алгоритм с линейным временем для задачи точного совпадения предложен Кнутом, Моррисом и Праттом. Хотя и метод редко используется и часто на практике уступает методу Бойера-Мура (и другим), он может быть просто объяснён, и его линейная оценка времени легко обосновываеся. Кроме того, он создаёт основу для известного алгоритма Ахо-Корасика.».

Фаза препроцессинга требует $O(m)$ времени и памяти, где m - длина шаблона. В итоге, время работы алгоритма оценивается как $O(m + n)$, где n - длина текста, имея оценку по памяти равную $O(m)$.

Исходный код

Выделим следующие стадии написания кода:

1. Реализация ввода
2. Реализация префикс-функции
3. Реализация КМП

В файле kmp.cpp реализуем функцию вычисления префикс функции для образца.

```
1  std::vector<size_t> CountPrefixFunction
2                      (const std::vector<std::string>& str) {
3      size_t n = str.size();
4      std::vector<size_t> sp(n);
5      sp[0] = 0;
6      size_t lastPrefix = 0;
7      for (size_t i = 1; i < n; ++i) {
8          while ((lastPrefix > 0) && (str[i] != str[lastPrefix])) {
9              lastPrefix = sp[lastPrefix - 1];
10         }
```

```
11          if (str[i] == str[lastPrefix]) {
12              ++lastPrefix;
13          }
14          sp[i] = lastPrefix;
15      }
16      sp.push_back(0);
17      return sp;
18  }
```

Затем займёмся непосредственной реализацией поиска.
Стоит отметить, что функция EqualToNextPatternWord используется как обёртка для получения следующей строки из образца

```
1  bool EqualToNextPatternWord(const std::vector<std::string>& pattern,
       const std::string& str, size_t idx) {
2      if (idx >= pattern.size()) {
3          return false;
4      }
5      return pattern[idx] == str;
6  }
```

И сам алгоритм КМП

```
1  size_t KMPSearch(const std::vector<std::vector<std::string>>& text,
       const std::vector<std::string>& pattern) {
2      std::vector<size_t> patternPrefix = CountPrefixFunction(pattern);
3      size_t occurrances = 0;
4
5      size_t lastPrefix = 0;
6      for (size_t lineIdx = 0; lineIdx < text.size(); ++lineIdx) {
7          for (size_t wordIdx = 0; wordIdx < text[lineIdx].size(); ++
               wordIdx)
8          {
9              while ((lastPrefix > 0) && (!EqualToNextPatternWord(pattern
                   , text[lineIdx][wordIdx], lastPrefix))) {
10                 lastPrefix = patternPrefix[lastPrefix - 1];
11             }
12
13             if (pattern[lastPrefix] == text[lineIdx][wordIdx]) {
14                 lastPrefix++;
15             }
16
17             if (lastPrefix == pattern.size()) {
18                 ++occurrances;
19
20                 #ifndef BENCH
```

```
                    size_t entryLine = lineIdx;
                    long long entryWord = wordIdx - (pattern.size() - 1);
                    while(entryWord < 0) {
                        --entryLine;
                        entryWord += text[entryLine].size();
                    }
                    std::cout << entryLine + 1 << ", " << entryWord + 1 <<
                       std::endl;
                   #endif
                }
            }
        }
        return occurrances;
}
```

В main.cpp будем сохранять входной текст посимвольно, используя конечный автомат. При этом, образец сохраняется в виде вектора слов, а текст в виде вектора векторов слов.

Пример работы

```
mmaxim2710@DESKTOP-RDPBU3D:/mnt/c/Users/mmaxi/Desktop/coursera/DA_ex/
    lab4$ make
g++ -std=c++17 -pedantic -Wall -O2 -c main.cpp -o main.o
g++ -std=c++17 -pedantic -Wall -O2 -c kmp.cpp -o kmp.o
g++ -std=c++17 -pedantic -Wall -O2 main.o kmp.o -o solution
mmaxim2710@DESKTOP-RDPBU3D:/mnt/c/Users/mmaxi/Desktop/coursera/DA_ex/
    lab4$ ./solution
hey pls find me
Big cat and dog
and a rabbit
and so now Hey pls
Find me
3, 4
```

Вывод

Выполняя данную работу, я встретил проблему с лишним выделением памяти. Дело в том, что если для нахождения паттерн P в строке T склеим паттерн со строкой вот так P@T и вызовем для нее префикс-функцию для заполнения массива, то мы сразу же повысим требования по памяти в несколько раз. Дополнительной памяти "правильная"реализация алгоритма требует $O(N)$, где N — длина паттерна. Изначальный же способ требует $O(N + M)$ дополнительной памяти, где N — длина образца, M — длина текста.

Роль @ играет $'\backslash 0'$ на конце образца. Нам достаточно просто идти по тексту и запоминать только предыдущее значение префикс функции.

Альтернатива КМП - алгортим Бойера-Мура, который часто работает быстрее в несколько раз (до N раз быстрее, где N — длина образца — хотя иногда может работать и медленнее, чем КМП — худшая оценка у него выше). Если первый символ в КМП не совпадает, мы всегда смещаемся на единицу, если последний символ в БМ не совпадает, и этого символа нет в образце — мы смещаемся сразу на длину образца. И даже если есть одинаковые символы, мы почти всегда смещаемся больше, чем на единицу.