Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №4 по курсу «Дискретный анализ»

Студент: Д. А. Тарпанов
Преподаватель: Н. С. Капралов
Группа: М8О-204Б-19
Дата:
Оценка:
Подпись:

Москва, 2020

Лабораторная работа №4

Задача: Необходимо реализовать один из стандартных алгоритмов поиска образцов для указанного алфавита.
Вариант алгоритма: Поиск одного образца при помощи алгоритма Бойера-Мура.
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.

1 Описание

Алгоритм Бойера-Мура
Алгоритм Бойера-Мура последовательно прикладывает образец P к тексту T и проверяет совпадение символов P с прилежащими символами T. Когда проверка завершается, P сдвигается вправо по T точно так же, как и в наивном алгоритме. Однако алгоритм Бойера-Мура использует три принципа, которых нет в наивном алгоритме: правило сдвига по плохому символу, правило сдвига что по хорошему суффиксу и просмотр справа налево. Совместный эффект этих трёх идей позволяет добиться сложности $O(n+m)$, где n – длина текста, m – длина паттерна.
Просмотр справа налево
При любом прикладывании P к T алгоритм Бойера-Мура проверяет совпадение символов справа налево, а не наоборот, как в наивном алгоритме. Ясно, что если бы образец P сдвигался на одну позицию вправо после каждого несовпадения или обнаружения вхождения, то сложность была бы равна $O(nm)$, ровно как и в наивном алгоритме. Однако два дополнительных правила позволяют делать сдвиги больше чем на одну позицию, и обычными становятся именно большие сдвиги.
Правило плохого символа
Предположим, что паттерн P и текст T несовпали в позиции i, $P[i] = x$, $T[i+j] = y$, $x \neq y$. Тогда сдвинем паттерн таким образом, чтобы крайнее правое вхождение символа y в P совпало с y в T. Если y не встречается в паттерне P, то можно сдвинуть P полностью за точку несовпадения. В этих случаях часть символов вообще не будется проверяться, и метод будет работать за сублинейное время.
Правило хорошего суффикса
Само по себе правило плохого символа хоть и высокоэффективно, но легко деградирует до $O(nm)$ в худшем случае. Поэтому вводится еще одно правило – правило хорошего суффикса. Пусть паттерн P приложен к тексту T и подстрока t из T совпадает с суффиксом P, но следующий левый символ не совпадает. Тогда найдем крайнюю правую копию t' строки t в P, такую что t' не является суффиксом P и символ слева от t' в P отличается от символа слева от t в P, и сдвинем P так, чтобы t' в P и t в T совпали. Если такой копии не существует, то сдвинем левый край P за левый край t из T на наименьший сдвиг, при котором префикс сдвинутого образца совпал бы с суффиксом t в T.

2 Исходный код

Рассмотренные выше правила плохого символа и хорошего суффикса нуждаются в препроцессинге. Для правила плохого символа создадим std::unordered_map, в качестве ключа которой будет выступать слово из паттерна, а значением будет крайнее правое вхождение этого слова в паттерн. Для того чтобы посчитать сдвиги по правилу плохого символа пройдемся по паттерну и для каждого слова присвоим значению индекс текущего слова. С правилом хорошего суффикса все сложнее. Для начала, посчитаем Z-функцию для реверсированного паттерна. Потом найдем N-функцию исходя из Z-функции, i-ый элемент которой будет хранить длину наибольшего суффикса подстроки $P[0..i]$, который является также суффиксом полной строки P. Потом посчитаем l и L, которые уже будем использовать непосредственно для сдвигов. В основной функции считаем паттерн и текст, выполним препроцессинг, и будем искать вхождения паттерна в текст. Для этого создадим две переменных, которые будут отвечать за текущие индексы текста и паттерна (так как по факту, говоря о сдвиге, мы просто увеличиваем эти индексы). В случае несовпадения сдвиг определяется как максимум среди единицы, сдвига, предлагаемого правилом плохого символа и сдвига, предлагаемого правилом хорошего суффикса. Для хранения номеров строк и слов заведем вектор пар, в котором i-ый элемент будет хранить информацию о строке и номере в строке i-го слова.

Листинг main.cpp

```
#include<string>
#include<vector>
#include<unordered_map>
#include<algorithm>
#include<iostream>


void BadCharacterRule(const std::vector<std::string>& pattern, std::unordered_map<std
    ::string, int>& badCharacterShift) {
    int size = pattern.size();
    for(int i = 0; i < size; ++i) {
        badCharacterShift[pattern[i]] = i;
    }
}

void ZFunction(const std::vector<std::string>& pattern, std::vector<int>& zResult) {
    int size = pattern.size();
    std::vector<int> zFunc(size, 0);
    for(int i = 1, left = 0, right = 0; i < size; ++i) {
        if (i <= right) {
            zFunc[i] = std::min(right - i + 1, zFunc[i-left]);
        }
```

```
22          while(i + zFunc[i] < size && pattern[zFunc[i]] == pattern[i + zFunc[i]]) {
23              ++zFunc[i];
24          }
25          if(i + zFunc[i] - 1 > right) {
26              left = i;
27              right = i + zFunc[i] - 1;
28          }
29      }
30      zResult = zFunc;
31  }
32
33  void NFunc(const std::vector<std::string>& pattern, std::vector<int>& nResult) {
34      std::vector<std::string> revPattern(pattern.rbegin(), pattern.rend());
35      std::vector<int> zFunc;
36      ZFunction(revPattern, zFunc);
37      std::vector<int> nFunc(zFunc.size());
38      for(int i = 0; i < revPattern.size(); ++i) {
39          nFunc[i] = zFunc[revPattern.size() - i - 1];
40      }
41      nResult = nFunc;
42  }
43
44  void GoodSuffixRule(const std::vector<std::string>& pattern, std::vector<int>&
        bigLFunc, std::vector<int>& smallLFunc) {
45      std::vector<int> n;
46      NFunc(pattern, n);
47      std::vector<int> l(n.size());
48      std::vector<int> L(pattern.size()+1);
49      int j = 0;
50      for(int i = 0; i < pattern.size()-1; ++i) {
51          if(n[i] != 0) {
52              j = pattern.size()-n[i];
53              l[j] = i;
54          }
55          if (n[i] == i+1) { L[pattern.size()-i-1] = i+1; }
56          else { L[pattern.size()-i-1] = L[pattern.size()-i]; }
57      }
58      bigLFunc = L;
59      smallLFunc = l;
60  }
61
62  int main(){
63      std::cin.tie(nullptr);
64      std::cout.tie(nullptr);
65      std::ios_base::sync_with_stdio(false);
66      std::vector<std::string> text;
67      std::vector<std::string> pattern;
68      std::vector<std::pair<int,int>> wordsID;
69      std::string tempWord;
```

```
    int stringInd = 1, wordInd = 1;
    std::pair<int,int> tempPair;
    std::unordered_map<std::string, int> badCharacterShift;
    char c = getchar();
    while (c > 0) {

        if (c == '\n') {
            if(!tempWord.empty()) {
                pattern.push_back(tempWord);
            }
            break;
        }
        if (c == '\t' || c == ' '){
            if (!tempWord.empty()) {
                pattern.push_back(tempWord);
                tempWord.clear();
            }
        }
        else {
            if ('A' <= c and c <= 'Z') {
                c = c + 'a' - 'A';
            }
            tempWord.push_back(c);
        }
        c = getchar();
    }
    tempWord.clear();
    c = getchar();
    while(c > 0) {
        if (c == '\n') {
            if(!tempWord.empty()) {
                text.push_back(tempWord);
                tempWord.clear();
                tempPair.first = stringInd;
                tempPair.second = wordInd;
                wordsID.push_back(tempPair);
            }
            stringInd++;
            wordInd = 1;
        } else if (c == '\t' || c == ' ') {
            if(!tempWord.empty()) {
                text.push_back(tempWord);
                tempWord.clear();
                tempPair.first = stringInd;
                tempPair.second = wordInd;
                wordsID.push_back(tempPair);
                wordInd++;
            }
        } else {
```

```
119             if ('A' <= c and c <= 'Z') {
120                 c = c + 'a' - 'A';
121             }
122             tempWord.push_back(c);
123         }
124         c = getchar();
125     }
126     if(!tempWord.empty()) {
127         text.push_back(tempWord);
128         tempPair.first = stringInd;
129         tempPair.second = wordInd;
130         wordsID.push_back(tempPair);
131     }
132     if(pattern.empty() || text.empty()) {
133         return 0;
134     }
135     BadCharacterRule(pattern, badCharacterShift);
136     std::vector<int> bigLFunc;
137     std::vector<int> smallLFunc;
138     GoodSuffixRule(pattern, bigLFunc, smallLFunc);
139     std::vector<int> entryVec;
140     int k = pattern.size() - 1;
141     while (k < text.size()) {
142         int i = pattern.size() - 1;
143         int j = k;
144         while ((i >= 0) && (pattern[i] == text[j])) {
145             --i;
146             --j;
147         }
148         if (i == -1) {
149             entryVec.push_back(k-pattern.size()+1);
150             if (pattern.size() > 2) {
151                 k += pattern.size() - bigLFunc[1];
152             } else {
153                 ++k;
154             }
155         } else {
156             int goodSuffixShift = 1;
157             int bCShift = 0;
158             if(badCharacterShift.find(text[j]) != badCharacterShift.end()) {
159                 bCShift = badCharacterShift[text[j]];
160             }
161             if (i != pattern.size() - 1) {
162                 if (smallLFunc[i + 1] > 0) {
163                     goodSuffixShift = pattern.size() - smallLFunc[i + 1] - 1;
164                 } else {
165                     goodSuffixShift = pattern.size() - smallLFunc[i + 1];
166                 }
167             }
```

```
168             k += std::max({goodSuffixShift, i - bCShift, static_cast<int>(1)});
169         }
170     }
171     for(long long i : entryVec) {
172         std::cout << wordsID[i].first << ", " << wordsID[i].second << '\n';
173     }
174 }
```

3 Консоль

```
kng@Legion:/mnt/c/vsc/da/lab4$ make
g++ -std=c++17 -g -O0 -O3 -Wextra -Wall -Werror -Wno-sign-compare -Wno-unused-result
-pedantic -o solution main.cpp
kng@Legion:/mnt/c/vsc/da/lab4$ ./solution
cat dog cat dog bird
CAT dog CaT Dog Cat DOG bird CAT
dog cat dog bird
1,3
1,8
```

4 Тест производительности

Реализованный алгоритм Бойера-Мура сравнивается с наивным алгоритмом поиска. Замеры проводятся на трех тестах: первый тест на 10^3 слов, второй на 10^5, третий на 10^7.

```
1.
BM's time is 0 ms
Naive time is 0 ms
2.
BM's time is 3 ms
Naive time is 2000 ms
3.
BM's time is 24957 ms
Naive time is 314072 ms
```

Видно, что наивный алгоритм поиска сильно проигрывает алгоритму Бойера-Мура. Наивный алгоритм не требует препроцессинга, поэтому на особо синтетических тестах даже может выиграть алгоритм Бойера-Мура, но, как показали замеры, на реальных задачах разрыв огромен.

5 Выводы

Во время выполнения работы я изучил алгоритм Бойера-Мура. Хочется сказать, что в этот раз мне удалось избежать долгих танцев с бубном над обработкой ошибок на чекере и решение прошло с 6-ой попытки. Это достаточно неплохой результат для меня, учитывая то, что прошлые лабораторные я отсылал лишь с 30-ой. Однако стоит отметить, что алгоритм Бойера-Мура сильно сложный для понимания. Алгоритм есть на страницах Гасфилда, и его можно достаточно быстро переписать, но процесс понимания того, как и почему он работает, для того чтобы поправить некоторые места в коде ради исправления ошибок, занял у меня достаточно много времени. Также, стоит сказать, что из 60 выделенных секунд на прохождение тестов, моя реализация потратила лишь 17 в самом сложном случае. Это говорит о том, что алгоритм Бойера-Мура действительно быстро работает.

Список литературы

[1] Гасфилд Дэн. *Строки, деревья и последовательности в алгоритмах: Информатика и вычислетльная биология.* — Издательский дом «БХВ-Петербург». Перевод с английского: И.В. Романовский — 654 с.(дата обращения: 16.12.2020).