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: К.С. Саженов
Преподаватель: Н. С. Капралов
Группа: М8О-208Б

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 Кнута-Морриса-Пратта.

Вариант ключа: `Числа от 0 до` $2^{32} - 1$.

1 Описание

Требуется написать реализацию поиска образца в строке, используя алгоритм Кнута-Морриса-Пратта с алфавитом в виде чисел от 0 до $2^{32} - 1$.

Для начала исходная строка конкатенируется с образцом в таком формате: $p+\#+s$. Основная идея в поиске подстрок данного алгоритма заключается в том, что предварительно считается, т.н. «префиксная функция». Идея данной функции заключается в том, что на i-й позиции она содержит максимально возможное число k, такое, что $s[0 \dots k-1] = s[i-k+1 \dots i]$, где s - исодная строка. Имея исходную строку $p+\#+s$, мы заключаем, что наша задача сводится к задаче поиска значнеия элемена в префикс-функции, которое будет равно длине образца.

В наивном алгоритме префикс-функции будет учавствовать, по меньшей мере, 2 цикла и одно сравнение строк, что не очень эффективно ($O(n^3)$). В ходе некоторых рассуждений, можно заключить, что $\pi[i+1]$ никак не может превысить число $\pi[i]+1$, что позволяет нам уже добиться сложности $O(n^2)$.

Затем, можно заметить, что если $s[i+1] = s[\pi[i]]$, то мы можем с уверенностью сказать, что $\pi[i+1] = \pi[i]+1$. Однако же если $s[i+1] \neq s[\pi[i]]$, тогда следует попытаться подобрать подстроку меньшей длины, которая будет такой наибольшей, что $j < \pi[i]$, и будет по-прежнему выполняться префикс-свойство в позиции i ($s[0 \dots j-1] = s[i-j+1 \dots i]$).

Формула для нахождения такого j: $j = \pi[j-1]$. Причем применять её следует, пока $s[i] \neq s[j]$, либо пока $j > 0$(в ином случае префикс-функция в данной позиции равна 0).

2 Исходный код

Файл `main.cpp`:

```cpp
#include <iostream>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

using TValue = uint64_t;
const TValue shebang = (1ull << 32ull);

// todo: can replace with getline with three args
vector<TValue> GetClause() {
  TValue c;
  vector<TValue> cl;
  std::string line;
  if(!std::getline(cin, line)){
    throw exception();
  }
  std::istringstream iss(line);
  while ( iss >> c) {
    cl.push_back(c);
  }
  return cl;
}

void BuildZFunction(vector<TValue> &pattern, vector<TValue> &pi){
  for (size_t i = 1; i < pattern.size(); ++i) {
    TValue j = pi.at(i-1);
    while(j > 0 && pattern.at(i) != pattern.at(j)){
      j = pi.at(j-1);
    }
    if(pattern.at(i) == pattern.at(j)) j++;
    pi.at(i) = j;
  }
}


int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(nullptr); cout.tie(nullptr);

  vector<TValue> pattern = GetClause();
  pattern.push_back(shebang);
  vector<TValue> pi(pattern.size(), 0);
  BuildZFunction(pattern, pi);

```

```
47    vector<size_t> rowsLength;
48
49    size_t wordInd(0), rowInd(0);
50    TValue curPi = 0;
51
52    while(true){
53      vector<TValue> row;
54      try {
55        row = GetClause();
56      } catch (exception& err) {
57        break;
58      }
59  // if(row.empty()) continue;
60      for (auto& el: row) {
61        while(curPi > 0 && el != pattern.at(curPi)){
62          curPi = pi.at(curPi - 1);
63        }
64        if(el == pattern.at(curPi)) ++curPi;
65        if(curPi == pattern.size()-1){
66          size_t ro = rowInd + 1, wo = wordInd + 1;
67          int64_t ind = -1;
68          size_t pattern_size = pattern.size()-1;
69          while(pattern_size > wo){
70            pattern_size -= wo;
71            ro--;
72            wo = *next(rowsLength.end(), ind--);
73          }
74          wo -= pattern_size - 1;
75          cout << ro << ", " << wo << '\n';
76        }
77        wordInd++;
78      }
79      wordInd = 0;
80      rowInd++;
81      rowsLength.push_back(row.size());
82    }
83  }
```

3 Консоль

```
sakost@sakost-pc ~/university/2 course/diskran/lab2 <master*>$ cmake-build-debug/lab4
1 1 1 2 1
1 1 1 2 1 1 1 2 1
1 1 2
0 1 2 1 10
1

1

1 2

1
0
1,1
1,5
4,1

sakost@sakost-pc ~/university/2 course/diskran/lab2 <master*>$ cmake-build-debug/lab4
1
1 2 1 2 1 1

1

0
1
1,1
1,3
1,5
1,6
3,1
7,1
```

4 Тест производительности

Тест производительности будет проводится из консоли, где `bench` файл – программа с наивной реализацией поиска по строке, а `a.out` – программа с алгоритмом КМП. Тестирование производится с помощью утилиты `time`. В тесте содержится 5000 строк, на каждой 100 случайных чисел. Чтобы не засорять вывод ответами, я перенаправляю стандартный вывод в `/dev/null`.

```
sakost@sakost-pc ~/university/2 course/diskran/lab4$ time ./a.out <input.txt
>/dev/null
./a.out <input.txt >/dev/null  5,07s user 0,02s system 99% cpu 5,089 total
sakost@sakost-pc ~/university/2 course/diskran/lab4$ time ./bench <input.txt
>/dev/null
./bench <input.txt >/dev/null  8,15s user 0,30s system 99% cpu 8,461 total
```

Общее время работы в секундах указано последними числами: это `5.089` у КМП и `8.461` у наивной реализации соответственно.

Алгоритм КМП оказался чуть более, чем в полтора раза быстрее наивной реализации поиска даже на таких маленьких данных. Т.к. генерация данных тоже занимает время, я не стал генерировать большее количество строк в тесте, поскольку данное количество вполне показательно.

Данный результат и следовало ожидать, поскольку сложность этих алгоритмов значительно отличается($O(n+m)$ у КМП и $O(nm)$ у наивного алгоритма, где n - количество символов в тексте, а m - количество символов в образце).

5 Выводы

Выполнив четвертую лабораторную работу по курсу «Дискретный анализ», я узнал, как работают алгоритмы поиска образца в строке, а именно `КМП`.

Данный алгоритм лично для меня является одним из самых интуитивно-понятных алгоритмов поиска образца в строке за линейное время. Также, используемая в данном контексте π-функция имеет и множество применений в других алгоритмах, в том числе, и других алгоритмах поиска образцов в строке. Более того, я выделил для себя некоторые хорошие стороны алгоритма:

1. Нет ограничений на алфавит
2. Алгоритм является, что называется, «онлайновым»(об этом ниже)
3. Работает за линейное время
4. Реализуется в несколько десятков строк кода

Хочу выделить пункт 2 – данный пункт означает, что алгоритму не требуются сразу все данные на ввод и он может подсчитывать вхождения непосредственно при вводе текста. Такая оптимизация достигается путем подсчета префикс-функции только для «образца» и разделяющего элемента, а затем в цикле обрабатывается каждый элемент последовательности из текста(и данная обработка происходит один и только один раз), что мне показалось очень практичным.

Список литературы

[1] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 2-е издание.* — Издательский дом «Вильямс», 2007. Перевод с английского: И. В. Красиков, Н. А. Орехова, В. Н. Романов. — 1296 с. (ISBN 5-8459-0857-4 (рус.))

[2] *Алгоритм Кнута — Морриса — Пратта — Википедия*
URL: `https://ru.wikipedia.org/wiki/Алгоритм_Кнута_-_Морриса_-_Пратта` (дата обращения 05.12.2020).

[3] *MAXimal :: algo :: Префикс-функция. Алгоритм Кнута-Морриса-Пратта*
URL: `https://e-maxx.ru/algo/prefix_function` (дата обращения 05.12.2020).