

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: И. О. Ильин
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б-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 Кнута-Морриса-Пратта.

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

Формат входных данных: Искомый образец задаётся на первой строке входного файла. 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. Следует вывести два числа через запятую: номер строки и номер слова в строке, с которого начинается найденный образец. 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

Требуется написать реализацию алгоритма Кнута-Морриса-Пратта поиска подстроки в строке. В качестве алфавита выступают слова не более 16 знаков латинского алфавита (регистронезависимые).

Как сказано в [1]: «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 обосновывается. 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 $O(m)$.

2 Исходный код

Выделим следующие стадии написания кода:

1. Реализация ввода
2. Реализация функции, выполняющей препроцессинг образца
3. Реализация алгоритма КМП
4. Реализация бенчмарка

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

```
1 | #include <iostream>
2 | #include "kmp.hpp"
3 |
4 | std::vector<size_t> CountPrefixFunction (const std::vector<std::string>& str) {
5 |     size_t n = str.size();
6 |     std::vector<size_t> sp(n);
7 |     sp[0] = 0;
8 |     size_t lastPrefix = 0;
9 |     for (size_t i = 1; i < n; ++i) {
10 |         while ((lastPrefix > 0) && (str[i] != str[lastPrefix])) {
11 |             lastPrefix = sp[lastPrefix - 1];
12 |         }
13 |         if (str[i] == str[lastPrefix]) {
14 |             ++lastPrefix;
15 |         }
16 |         sp[i] = lastPrefix;
17 |     }
18 |     return sp;
19 | }
```

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 | }
7 |
8 | size_t KMPSearch(const std::vector<std::vector<std::string>>& text, const std::vector<
   | std::string>& pattern) {
9 |     std::vector<size_t> patternPrefix = CountPrefixFunction(pattern);
```

```

10     size_t occurrences = 0;
11
12     size_t lastPrefix = 0;
13     for (size_t lineIdx = 0; lineIdx < text.size(); ++lineIdx) {
14         for (size_t wordIdx = 0; wordIdx < text[lineIdx].size(); ++wordIdx)
15             {
16                 while ((lastPrefix > 0) && (!EqualToNextPatternWord(pattern, text[lineIdx][
17                     wordIdx], lastPrefix))) {
18                     lastPrefix = patternPrefix[lastPrefix - 1];
19                 }
20
21                 if (pattern[lastPrefix] == text[lineIdx][wordIdx]) {
22                     lastPrefix++;
23                 }
24
25                 if (lastPrefix == pattern.size()) {
26                     ++occurrences;
27
28                     #ifndef BENCH
29                     size_t entryLine = lineIdx;
30                     long long entryWord = wordIdx - (pattern.size() - 1);
31                     while(entryWord < 0) {
32                         --entryLine;
33                         entryWord += text[entryLine].size();
34                     }
35                     std::cout << entryLine + 1 << ", " << entryWord + 1 << std::endl;
36                     #endif
37                 }
38             }
39     return occurrences;
40 }

```

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.

```

1  #include <iostream>
2  #include "kmp.hpp"
3
4  bool isSpace(char c) {
5      return ((c == ' ') || (c == '\t') || (c == '\n'));
6  }
7
8  enum TState {
9      inStr,
10     betweenStrs
11 };
12
13 int main(int argc, const char * argv[]) {

```

```

14     std::ios_base::sync_with_stdio(false);
15     std::cin.tie(nullptr);
16
17     std::vector<std::vector<std::string>> text;
18     std::vector<std::string> pattern;
19
20     bool firstLine = true;
21     TState state = betweenStrs;
22     std::vector<std::string> currentLine;
23     std::string currentStr;
24
25     char letter = getchar();
26     while (letter != EOF) {
27         switch (state) {
28             case betweenStrs:
29                 if (!isspace(letter)) {
30                     state = inStr;
31                     break;
32                 }
33
34                 if (letter == '\n') {
35                     if (firstLine) {
36                         pattern = std::move(currentLine); //
37                         firstLine = false;
38                     } else {
39                         text.push_back(std::move(currentLine)); //
40                     }
41                     currentLine.clear();
42                 }
43                 letter = getchar();
44                 break;
45
46             case inStr:
47                 if (isspace(letter)) {
48                     currentLine.push_back(std::move(currentStr)); //
49                     currentStr.clear();
50                     state = betweenStrs;
51                     break;
52                 }
53                 currentStr.push_back(std::tolower(letter));
54                 letter = getchar();
55                 break;
56         }
57     }
58
59     KMPSearch(text, pattern);
60
61     return 0;
62 }

```

Для бенчмарка напомним функции генерации случайного слова (*RandomString*) и случайной строки (*RandomLine*), с помощью которых сгенерируем текст для тестирования. Сравниваем время и количество найденных вхождений с *std::string::find*.

```

1  #include <iostream>
2  #include <chrono>
3  #include "kmp.hpp"
4
5  using timeDuration = std::chrono::nanoseconds;
6
7  std::string RandomString(size_t maxLength) {
8      size_t length = rand() % maxLength;
9      auto RandChar = []() -> char {
10         const char charSet[] =
11             "0123456789"
12             "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
13         const size_t maxIdx = (sizeof(charSet) - 1);
14         return charSet[rand() % maxIdx];
15     };
16     std::string str(length, 0);
17     std::generate_n(str.begin(), length, RandChar);
18     return str;
19 }
20
21 void RandomLine(size_t maxCount, std::vector<std::string>& str) {
22     std::vector<std::string> result;
23     const size_t maxStrLen = 17;
24     size_t count = rand() % maxCount;
25     for (size_t i = 0; i < count; ++i) {
26         str.push_back(RandomString(maxStrLen));
27     }
28 }
29
30 int main(int argc, const char * argv[]) {
31     std::srand(static_cast<unsigned int>(std::time(0)));
32
33     std::cout << "Enter amount of lines: ";
34     std::cin >> linesCnt;
35
36     std::string patternStr = "one two three four4 5fiv5e 9120asD one two three";
37     std::vector<std::string> pattern = {
38         std::string("one"), std::string("two"), std::string("three"), std::string("
39         four4"),
40         std::string("5fiv5e"), std::string("9120asD"), std::string("one"),
41         std::string("two"), std::string("three")
42     };
43
44     std::vector<std::vector<std::string>> text;
45     std::string textStr;

```

```

46  const int maxLineLen = 20;
47  const int patternFrequency = 5;
48  int occurrences = 0;
49  for (size_t i = 0; i < linesCnt; ++i) {
50      std::vector<std::string> line;
51      RandomLine(maxLineLen, line);
52
53      if ((rand() % patternFrequency) == 0) {
54          ++occurrences;
55          size_t idx = 0;
56          if (line.size() != 0) {
57              idx = rand() % line.size();
58          }
59          auto it = std::begin(line);
60          while(idx > 0) {
61              ++it;
62              --idx;
63          }
64          line.insert(it, std::begin(pattern), std::end(pattern));
65      }
66
67      text.push_back(line);
68      for (std::string str : line) {
69          textStr += (str + " ");
70      }
71  }
72
73  std::cout << "TOTAL OCCURRENCES = " << occurrences << std::endl;
74
75  std::chrono::time_point<std::chrono::system_clock> start, end;
76  int64_t findTime;
77  int64_t kmpTime;
78
79  start = std::chrono::system_clock::now();
80  size_t pos = textStr.find(patternStr, 0);
81  size_t findRes = 0;
82  while(pos < textStr.size()) {
83      ++findRes;
84      pos = textStr.find(patternStr, pos+1);
85  }
86  end = std::chrono::system_clock::now();
87  findTime = std::chrono::duration_cast<timeDuration>(end - start).count();
88
89  start = std::chrono::system_clock::now();
90  size_t kmpRes = KMPSearch(text, pattern);
91  end = std::chrono::system_clock::now();
92  kmpTime = std::chrono::duration_cast<timeDuration>(end - start).count();
93
94  std::cout << "===== " << std::endl;

```



```

95 | std::cout << " KMP SEARCH RESULT = " << kmpRes << std::endl;
96 | std::cout << " KMP SEARCH TIME = " << kmpTime << " ms" << std::endl;
97 | std::cout << "STRING::FIND RESULT = " << findRes << std::endl;
98 | std::cout << " STRING::FIND TIME = " << findTime << " ms" << std::endl;
99 | std::cout << "===== " << std::endl;
100 |
101 | return 0;
102 | }

```

3 Консоль

```

MacBook-Pro:da_lab_04 mr-ilin$ make clean
rm -f *.o solution benchmark
MacBook-Pro:da_lab_04 mr-ilin$ 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
MacBook-Pro:da_lab_04 mr-ilin$ ./solution
Hey pls find me

Big cat and a dog
and a rabbit
and now hey pls
find
me
4,3
MacBook-Pro:da_lab_04 mr-ilin$ ./solution
a aa b aa a
a aa b aa a aa b
aa a
aa b
1,1
1,5

```

4 Тест производительности

Тест производительности представляет из себя генерацию случайного текста с повторениями заданного шаблона и дальнейшее выполнение поиска с помощью алгоритма КМП и `std::string::find`. Сравнивается время и количество найденных образцов.

```
MacBook-Pro:da_lab_04 mr-ilin$ make clean
rm -f *.o solution benchmark
MacBook-Pro:da_lab_04 mr-ilin$ make test
g++ -std=c++17 -pedantic -Wall -O2 -DBENCH -c benchmark.cpp -o benchmark.o
g++ -std=c++17 -pedantic -Wall -O2 -DBENCH -c kmp.cpp -o kmp.o
g++ -std=c++17 -pedantic -Wall -O2 -DBENCH benchmark.o kmp.o -o benchmark
MacBook-Pro:da_lab_04 mr-ilin$ ./benchmark
Enter amount of lines: 2000
TOTAL OCCURRENCES = 422
=====
KMP SEARCH RESULT = 422
KMP SEARCH TIME = 164000 ms
STRING::FIND RESULT = 422
STRING::FIND TIME = 69000 ms
=====
MacBook-Pro:da_lab_04 mr-ilin$ ./benchmark
Enter amount of lines: 4000
TOTAL OCCURRENCES = 791
=====
KMP SEARCH RESULT = 791
KMP SEARCH TIME = 304000 ms
STRING::FIND RESULT = 791
STRING::FIND TIME = 113000 ms
=====
```

Можно сделать вывод, что алгоритм компилятора `g++` работает быстрее. Однако, алгоритм КМП, по результатам тестов, сохраняет свою временную сложность.

5 Выводы

Выполнив четвертую лабораторную работу по курсу «Дискретный анализ», я ещё раз познакомился с алгоритмом КМП (прошлый раз был на 1 курсе). Окончательно разобрался во всех тонкостях его работы. Примечательным был тот факт, что в качестве элемента для сравнения был использован не символ, а целое слово. Это позволяет легче вычислять номер строки и слова, с которого начинается вхождение, при этом не имея сильных потерь в производительности.

Список литературы

- [1] Дэн Гасфилд. *Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология*. — Издательский дом «Невский Диалект», 2000ц. Перевод с английского: И. В. Романовского. — 654с. (ISBN 5-7940-0103-8 (рус.))
- [2] *Алгоритм Кнута-Морриса-Пратта* — Вики университета ИТМО.
URL:
https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Кнута-Морриса-Пратта
(дата обращения: 08.12.2020).