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, 2021

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 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 $O(m)$.

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.

```
  #include <iostream>
#include "kmp.hpp"

std::vector<size_t> CountPrefixFunction (const std::vector<std::string>& str) {
    size_t n = str.size();
    std::vector<size_t> sp(n);
    sp[0] = 0;
    size_t lastPrefix = 0;
    for (size_t i = 1; i < n; ++i) {
        while ((lastPrefix > 0) && (str[i] != str[lastPrefix])) {
            lastPrefix = sp[lastPrefix - 1];
        }
        if (str[i] == str[lastPrefix]) {
            ++lastPrefix;
        }
        sp[i] = lastPrefix;
    }
    return sp;
}
```

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

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

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

```
    size_t occurrances = 0;

    size_t lastPrefix = 0;
    for (size_t lineIdx = 0; lineIdx < text.size(); ++lineIdx) {
        for (size_t wordIdx = 0; wordIdx < text[lineIdx].size(); ++wordIdx)
        {
            while ((lastPrefix > 0) && (!EqualToNextPatternWord(pattern, text[lineIdx][
                wordIdx], lastPrefix))) {
                lastPrefix = patternPrefix[lastPrefix - 1];
            }

            if (pattern[lastPrefix] == text[lineIdx][wordIdx]) {
                lastPrefix++;
            }

            if (lastPrefix == pattern.size()) {
                ++occurrances;

                #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 сслов.

```
#include <iostream>
#include "kmp.hpp"

bool isSpace(char c) {
    return ((c == ' ') || (c == '\t') || (c == '\n'));
}

enum TState {
    inStr,
    betweenStrs
};

int main(int argc, const char * argv[]) {
```

```
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(nullptr);

    std::vector<std::vector<std::string>> text;
    std::vector<std::string> pattern;

    bool firstLine = true;
    TState state = betweenStrs;
    std::vector<std::string> currentLine;
    std::string currentStr;

    char letter = getchar();
    while (letter != EOF) {
        switch (state) {
            case betweenStrs:
                if (!isSpace(letter)) {
                    state = inStr;
                    break;
                }

                if (letter == '\n') {
                    if (firstLine) {
                        pattern = std::move(currentLine); //
                        firstLine = false;
                    } else {
                        text.push_back(std::move(currentLine)); //
                    }
                    currentLine.clear();
                }
                letter = getchar();
                break;

            case inStr:
                if (isSpace(letter)) {
                    currentLine.push_back(std::move(currentStr)); //
                    currentStr.clear();
                    state = betweenStrs;
                    break;
                }
                currentStr.push_back(std::tolower(letter));
                letter = getchar();
                break;
        }
    }

    KMPSearch(text, pattern);

    return 0;
}
```

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

```cpp
#include <iostream>
#include <chrono>
#include "kmp.hpp"

using timeDuration = std::chrono::nanoseconds;

std::string RandomString(size_t maxLength) {
    size_t length = rand() % maxLength;
    auto RandChar = []() -> char {
        const char charSet[] =
        "0123456789"
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        const size_t maxIdx = (sizeof(charSet) - 1);
        return charSet[rand() % maxIdx];
    };
    std::string str(length, 0);
    std::generate_n(str.begin(), length, RandChar);
    return str;
}

void RandomLine(size_t maxCount, std::vector<std::string>& strs) {
    std::vector<std::string> result;
    const size_t maxStrLen = 17;
    size_t count = rand() % maxCount;
    for (size_t i = 0; i < count; ++i) {
        strs.push_back(RandomString(maxStrLen));
    }
}

int main(int argc, const char * argv[]) {
    std::srand(static_cast<unsigned int>(std::time(0)));

    std::cout << "Enter amount of lines: ";
    std::cin >> linesCnt;

    std::string patternStr = "one two three four4 5fiv5e 9120asD one two three";
    std::vector<std::string> pattern = {
        std::string("one"), std::string("two"), std::string("three"), std::string("
            four4"),
        std::string("5fiv5e"), std::string("9120asD"), std::string("one"),
        std::string("two"), std::string("three")
    };

    std::vector<std::vector<std::string>> text;
    std::string textStr;

```

```
46      const int maxLineLen = 20;
47      const int patternFrequency = 5;
48      int occurrances = 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              ++occurrances;
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 OCCURRANCES = " << occurrances << 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 OCCURRANCES = 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 OCCURRANCES = 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 четвёртую 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», 200ц. 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).