Московский авиационный институт (национальный исследовательский университет)

Факультет компьютерных наук и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №4 по курсу «Дискретный анализ»

Студент: О. А. Мезенин
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б-21

Дата:
Оценка:
Подпись:

Москва, 2023

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.

1 Описание

Алгоритм Бойера-Мура прикладывает паттерн P к тексту T слева направо. Сравнение символов P и T идёт справа налево. Сдвиг паттерна вправо осуществляется согласно одному из двух правил: правило плохого символа (ППС), правило хорошего суффикса (ПХС). Алгоритм выбирает на каждом шагу то правило, которое показало наибольший сдвиг.

Приведём расширенное правило плохого символа: «Когда несовпадение случилось в позиции i образца P и x – несовпадающий символ в T, нужно сдвинуть P вправо, совместив с этим x ближайшее вхождение x в P слева от позиции i» [1]. Если такой позиции не нашлось, сдвигаем на 1. Для реализации заведём хеш-таблицу, в которой ключ – это символ, а значение – упорядоченный массив позиций, на которых стоит этот символ в P. Для поиска ближайшего вхождения x слева от позиции i мы используем бинарный поиск. Правило требует $O(n)$ памяти и $O(n)$ времени на постройку хеш-таблицы, само правило вычисляет сдвиг за $O(\log n)$ времени в худшем случае.

Приведём сильное правило хорошего суффикса: «Пусть строка P приложена к T и подстрока t из T совпадает с суффиксом P, но следующий левый символ уже не совпадает. Найдём, если она существует, 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. Сдвинем P вправо, приложив подстроку t' в P к подстроке t в T. Если t' не существует, то сдвинем левый конец P за левый конец t в T на наименьший сдвиг, при котором префикс сдвинутого образца совпал бы с суффиксом t в T. Если такого сдвига не существует, то сдвинем P на n позиций вправо. Если найдено вхождение P, то сдвинем P на наименьший сдвиг, при котором собственный префикс сдвинутого P совпадает с суффиксом вхождения P и T. Если такой сдвиг невозможен, нужно сдвинуть P на n мест»[1].
Для реализации этого правила понадобятся два массива. Для каждого i пусть $L'(i)$ – наибольшая позиция, меньшая n и такая, что $P[i..n]$ совпадает с суффиксом $P[1..L'(i)]$, а символ, предшествующий этому суффиксу, не равен $P[i-1]$. Если такой позиции нет, то $L'(i) = 0$. Пусть $l'(i)$ обозначает длину наибольшего суффикса $P[i..n]$, который является префиксом P. Если такого суффикса не существует, то $l'(i) = 0$. Тогда алгоритм сдвига по ПХС будет следующим: если при поиске обнаружилось несовпадение в позиции $i-1$ строки P и $L'(i) > 0$, то сдвинем P на $n - L'(i)$ мест вправо. Если $L'(i) = 0$, то смещаем P на $n - l'(i)$ мест. Если первое сравнение даёт несовпадение, то P надо сдвинуть на одно место вправо.[1].
Правило требует $O(n)$ дополнительной памяти и $O(n)$ времени на постройку массивов.

2 Исходный код

Создадим класс `BoyerMooreSearch`. Который будет инициализировать массивы для правил, и у которого будет непосредственно метод поиска.

```
1  using Tchar = std::string;
2
3  class BoyerMooreSearch {
4  // variables
5  private:
6      std::unordered_map<Tchar, std::vector<uint64_t>> badCharPositions;
7      std::vector<uint64_t> goodSuffixShiftsA, goodSuffixShiftsB;
8      std::deque<Tchar> pattern;
9  // methods
10 private:
11     void BadCharPositionsFill();
12     void GoodSuffixShiftAFill();
13     void GoodSuffixShiftBFill();
14     uint64_t BadCharRuleShift(const std::deque<Tchar>& text, uint64_t i, uint64_t j);
15     uint64_t GoodSuffixRuleShift(uint64_t i);
16     std::vector<uint64_t> ZFunction(const std::deque<Tchar>& s);
17     std::deque<Tchar> ReversePattern();
18 public:
19     BoyerMooreSearch(const std::deque<Tchar>& pattern);
20     std::tuple<std::vector<uint64_t>, uint64_t> Search(std::deque<Tchar>& text,
           uint64_t startK);
21 };
```

kv_struct.cpp	
void BadCharPositionsFill()	Функция для заполнения хеш-таблицы `badCharPositions` для ППС
void GoodSuffixShiftAFill()	Функция для заполнения массива `goodSuffixShiftsA` (L') для ПХС
void GoodSuffixShiftBFill()	Функция для заполнения массива `goodSuffixShiftsB` (l') для ПХС
uint64_t BadCharRuleShift(const std::deque<Tchar>& text, uint64_t i, uint64_t j)	Функция для вычисления сдвига по ППС
uint64_t GoodSuffixRuleShift(uint64_t i)	Функция для вычисления сдвига по ПХС
std::vector<uint64_t> ZFunction(const std::deque<Tchar>& s)	Z-функция строки, необходима для заполнения массивов для ПХС
std::deque<Tchar> ReversePattern()	Функция, возвращающая перевёрнутый паттерн, необходима для заполнения массивов для ПХС

std::tuple<std::vector<uint64_t>, uint64_t> Search(std::deque<Tchar>& text, uint64_t startK)	Основная функция поиска в тексте `text` и изначальным сдвигом паттерна на $starkK - n$, возвращает массив позиций, в которых нашлось совпадение, а также число K – позицию, в которой поиск остановился ($K >= m$)
main.cpp	
uint64_t WordsParsing(const std::string& words, std::deque<std::string>& d)	Функция читает строку `words`, делит её на слова, приводя к нижнему регистру, и добавляет в `d`
void EraseLine(uint64_t& lineInfoShift, std::deque<std::pair<uint64_t, uint64_t>>& linesInfo, uint64_t& startK, std::deque<std::string>& text)	Функция удаляет информацию о первой строке в деке `linesInfo`, удаляет слова, принадлежащие этой строке из `text`, и сдвигает `startK` на количество удаляемых слов.

В функции `main` заводится массив с информацией о строках. Все слова записываются в один массив слов, который затем идёт в функцию поиска. Затем исходя из значения K, которое вернула функция поиска, и длин строк, можно удалить строки, к которым паттерн больше не будет прикладываться. Исходя из длин строк и массива *positions*, выводим ответ. Основной цикл функции `main`:

```
1   while (std::getline(std::cin, words)) {
2       uint64_t lineSize = WordsParsing(words, text);
3       ++lineCount;
4       if (lineSize == 0) {
5           continue;
6       }
7       linesInfo.emplace_back(lineCount, lineSize);
8       auto [positions, lastK] = bms.Search(text, startK);
9       startK = lastK;
10
11      uint64_t lineInfoShift = 0;
12      for (uint64_t entryPos: positions) {
13          while (lineInfoShift + linesInfo[0].second <= entryPos) {
14              EraseLine(lineInfoShift, linesInfo, startK, text);
15          }
16          std::cout << linesInfo[0].first << ", " << entryPos - lineInfoShift + 1 <<
                '\n';
17      }
18      while (!linesInfo.empty() && lineInfoShift + linesInfo[0].second <= lastK - n)
            {
19          EraseLine(lineInfoShift, linesInfo, startK, text);
20      }
21  }
```

3 Консоль

```
aprold@SAI:~/Documents/GitHub/MAI-DA/lab4$ cmake -S . -B cmake-build
--The C compiler identification is GNU 11.3.0
--The CXX compiler identification is GNU 11.3.0
--Detecting C compiler ABI info
--Detecting C compiler ABI info -done
--Check for working C compiler: /usr/bin/cc -skipped
--Detecting C compile features
--Detecting C compile features -done
--Detecting CXX compiler ABI info
--Detecting CXX compiler ABI info -done
--Check for working CXX compiler: /usr/bin/c++ -skipped
--Detecting CXX compile features
--Detecting CXX compile features -done
--Configuring done
--Generating done
--Build files have been written to:
/home/aprold/Documents/GitHub/MAI-DA/lab4/cmake-build
aprold@SAI:~/Documents/GitHub/MAI-DA/lab4$ cd cmake-build
aprold@SAI:~/Documents/GitHub/MAI-DA/lab4/cmake-build$ make
[ 33%] Building CXX object CMakeFiles/lab4.dir/main.cpp.o
[ 66%] Building CXX object CMakeFiles/lab4.dir/boyer_moore_search.cpp.o
[100%] Linking CXX executable lab4
[100%] Built target lab4
aprold@SAI:~/Documents/GitHub/MAI-DA/lab4/cmake-build$ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  lab4  Makefile
aprold@SAI:~/Documents/GitHub/MAI-DA/lab4/cmake-build$ ./lab4 <<EOF
>cat dog cat dog bird
CAT dog CaT Dog Cat DOG bird CAT
dog cat dog bird
EOF
1,3
1,8
```

4 Тест производительности

Тесты производительности представляют из себя следующее: алгоритм Бойера-Мура будет сравниваться с поиском с помощью Z-функции (за линейное время). Учитывается также время препроцессинга. Во всех тестах длина текста равна 1000000. В первых трёх тестах длина паттерна равна 1024. Всего тестов будет четыре: случайный паттерн и текст; паттерн и текст с высокой частотой повторений (вероятность 0.99, что следующее слово уже присутствовало в паттерне); паттерн и текст состоят из одного и того же слова; длина паттерна равна 5, паттерн и текст случайные.

```
aprold@SAI:~/Documents/GitHub/MAI-DA/lab4/cmake-build$ ./benchmark <../tests/01.t
Pattern size n = 1024
Text size m = 1000000
Boyer-Moore search time: 2579us
Z-function search time: 285733us
aprold@SAI:~/Documents/GitHub/MAI-DA/lab4/cmake-build$ ./benchmark <../tests/02.t
Pattern size n = 1024
Text size m = 1000000
Boyer-Moore search time: 12687us
Z-function search time: 783276us
aprold@SAI:~/Documents/GitHub/MAI-DA/lab4/cmake-build$ ./benchmark <../tests/03.t
Pattern size n = 1024
Text size m = 1000000
Boyer-Moore search time: 83574888us
Z-function search time: 14778524us
aprold@SAI:~/Documents/GitHub/MAI-DA/lab4/cmake-build$ ./benchmark <../tests/04.t
Pattern size n = 5
Text size m = 1000000
Boyer-Moore search time: 46644us
Z-function search time: 293635us
```

Алгоритм Бойера-Мура проигрывает на 3 тесте, т.к. в этом случае легко заметить, что его сложность равна $O(n * m)$. Сравнивая первый и второй тест, можно сделать вывод, что Бойер-Мур (как и Z-функция) работает хуже в текстах с высокой частотой совпадений, т.к. перед тем, как будет найдено несовпадение, может быть выполнено достаточно много сравнений. Сравнивая первый и четвертый тест, можно сказать, что Бойер-Мур хуже работает с маленькими паттернами, т.к. производятся небольшие сдвиги; при этом время Z-функции почти не увеличилось.

5 Выводы

Выполнив четвёртую лабораторную работу по курсу «Дискретный анализ», узнал об алгоритме Бойера-Мура, в частности о его эвристиках: правила плохого символа и правила хорошего суффикса, в том числе об их сильных версиях. Также приобрёл навыки в написании алгоритма Бойера-Мура и Z-функции, работающей за линейное время.
Алгоритм Бойера-Мура показывает себя с лучшей стороны на случайных данных и большом паттерне. Его сложность в среднем линейна. Также существует множество оптимизаций этого алгоритма [2].

Список литературы

[1] Гасфилд Дэн
Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология / Пер. с англ. И.В.Романовского. — СПб.: Невский Диалект; БХВ-Петербург (ISBN 0-521-58519-8 (англ.); ISBN 5-7940-0103-8 ("Невский Диалект"); ISBN 5-94157-321-9 (БХВ-Петербург))

[2] *Алгоритм поиска строки Бойера – Мура*
URL: `https://ru.wikibrief.org/wiki/Boyer-Moore_string-search_algorithm` (дата обращения: 18.05.2023).