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: М. С. Гаврилов
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

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-маски: в образце может встречаться «джокер» (представляется символом ? — знак вопроса), равный любому другому символу. При реализации следует разбить образец на несколько, не содержащих «джокеров», найти все вхождения при помощи алгоритма Ахо-Корасик и проверить их относительное месторасположение.

Вариант алфавита: Числа в диапазоне от 0 до 232 - 1

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 Ахо-Корасик осуществляет поиск нескольких подстрок в данной строке. Перед началом поиска производится построение структуры trie из искомых подстрок. Затем на основе полученного trie строится конечный автомат, состояниями которого является текущая позиция (буква) в структуре. Переходы в автомате осуществляются либо от узла-родителя к потомку (если узел с буквой, к которой мы намерены перейти находится в списке потомков текущего узла), либо по суффиксной ссылке текущего узла (во всех остальных случаях). Суффиксная ссылка – это указатель на такой узел, путь от корневого узла до коего содержит последовательность букв, соответствующую суффиксу слова, образованного путем от корня до узла, суффисной ссылкой которого рассматриваемый указатель является. Под суффиксом здесь подразумевается часть слова, начало которой не соответствует первой букве слова, а конец – соответствует последней. В поле суффиксной ссылки записывается та ссылка, которая ведет по наиболее длинному суффиксу из возможных. Если ни одного суффикса, являющегося началом какого-либо слова выделить не удалось, то суффиксная ссылка ведет в корень.

Теперь в ходе поиска необходимо совершать переход в ту букву, которая попалась в тексте, и если это невозможно, переходить по суффиксным ссылкам. Таким образом, даже если слово началось уже внутри другого рассматриваемого слова оно будет обнаружено.

Также следует учитывать, что если слово началось и закончилось внутри другого рассматриваемого слова, то оно все еще будет упущено. Чтобы этого избежать, необходимо на каждом шаге пробегать по суффиксным ссылкам от текущей буквы до корня и, если на это пути попадется терминальная вершина, отмечать вхождение. Чтобы не было необходимости совершать слишком много переходов при такой проверке эти ссылки можно упростить еще при нахождении, объединив все ссылки, ведущие не в терминальные вершины или нули и записав их в отдельное поле.

При поиске вхождения шаблона, состоящего из нескольких образцов и джокеров я вычислял для каждого образца его позицию относительно начала шаблона и при обнаружении вхождения инкрементировал элемент специального массива, расположенный под номером позиции, на которой может начинаться шаблон, в который входил бы найденный образец. Если после завершения обработки текста в какой-нибудь позиции массива находилось число, равное числу образцов в шаблоне, то в этом месте начинается вхождение шаблона в текст.

2 Исходный код

Основа алгоритма – структура Trie. Создаем тип узла: TTrieNode.

```
1
2  class TTrieNode {
3  private:
4      std::vector<TTrieNode*> children;
5      std::vector<unsigned long> term;
6      unsigned long litera = 0;
7      TTrieNode* suffix;
8      TTrieNode* parent;
9      TTrieNode* Uplink;
10 public:
11     TTrieNode(unsigned long inp) {
12         litera = inp;
13         suffix = nullptr;
14         Uplink = nullptr;
15         parent = nullptr;
16     }
17 }
```

В поле term хранятся все маркеры терминальности, которые в свою очередь содержат позицию данного образца в шаблоне.
В поле Uplink содержится сжатая суффиксная ссылка.

Методы для работы с узлом	
Функция	описание
unsigned long liter()	возвращает литеру текущего узла
std::vector<TTrieNode*> child()	возвращает массив, содержащий указатели на потомков
bool hasChild(unsigned long a)	проверяет наличие потомка, содержащего литеру a
TTrieNode* child(unsigned long a)	возвращает указатель на потомка, содержащего литеру a
std::vector<unsigned long> Term()	возвращает маркеры терминальности текущего узла
void addChild(TTrieNode* adb)	добавляет потомка теущему узлу
void setterm(unsigned long count)	устанавливает маркер терминальности узла в значение count
TTrieNode* back()	возвращает суффиксную ссылку
TTrieNode* Up()	возвращает сжатую суффиксую ссылку
TTrieNode* Parent()	возвращает указатель на родителя

Теперь создадим класс TTrie:

```
1
2  class TTrie {
3  private:
4      TTrieNode* root;
5      unsigned long maxSize;
6      long long count = 0;
7  public:
8      TTrie() {
9          root = new TTrieNode(0);
10         maxSize = 0;
11         count = 0;
12         root->setSuff(root);
13     }
14 }
```

Методы для работы с нагруженным деревом	
Функция	описание
unsigned long Count()	возвращает число узлов дерева
long add(std::vector<unsigned long> word, unsigned long id)	добавляет узел
void print()	выводит дерево на экран
void suffixate()	расставляет суффиксные ссылки
TTrieNode* Root()	возвращает корень дерева

Вспомогательные функции, работающие на уровне узлов	
Функция	описание
TTrieNode* sufGetByG(TTrieNode* root, TTrieNode* forWhom)	вычисляет суффиксную ссылку через операцию перехода
TTrieNode* upGet(TTrieNode* root, TTrieNode* forWhom)	вычисляет сжатую ссылку
TTrieNode* nextGet(TTrieNode* root, TTrieNode* forWhom, unsigned long to)	вычисляет переход по литере

Все эти функции должны использоваться в ходе поиска, без предварительного вычисления суффиксных ссылок. В программе также есть возможность предрассчета ссылок, однако так как в итоговой версии она не используется приводить вспомогательные функции для нее я не буду.

Собственно, функция поиска:

```
std::vector<int> search(TTrie* trie,std::vector<unsigned long> text) {
    std::vector<int> ret(text.size(),0);
```

//сюда будут записываться позиции, в которых может начинаться вхождение шаблона

```
    TTrieNode* cur = trie->Root();

    for (unsigned long i = 0; i < text.size(); ++i) {
        cur = nextGet(trie->Root(),cur,text[i]);
```

//осуществляется переход по литерам текста

```
        if (cur->isTerm()) {

            std::vector<unsigned long> curT = cur->Term();
            for(unsigned long k =0;k < curT.size();++k) {
```

//если данный образец встречается в шаблоне неоднократно, то он генерирует насколько потенциальных стартов вхождения. Нужно записать их все

```
                if(i + 1 >= curT[k]) {
                    ret[i-curT[k]+1]++;
                    if (DEBUG)printf("uppos %lu\n",i + 1 - curT[k]);
                }
            }
        }

        TTrieNode* cheker = upGet(trie->Root(),cur);
```

//С помощью перехода по сжатым суффиксным ссылкам проверяем, не пропустили ли мы вхождение

```
        while(cheker != trie->Root()) {

            if (cheker->isTerm()) {
                if (DEBUG)printf("\t\tpattern found (%lu) from %lu to %lu\n", cheker->
                    Term()[0], i - reconst(trie->Root(), cheker, 0) + 2, i + 1);

                std::vector<unsigned long> curT = cheker->Term();
                for(unsigned long k = 0;k < curT.size();++k) {
                    if(i + 1 >= curT[k]) {
                        ret[i + 1 - curT[k]]++;
                    }
                }
            }

            cheker = upGet(trie->Root(),cheker);
        }
    }
```

```
17
18
19      return ret;
20  }
```

В ходе работы программы производится считывание образцов, в ходе него заполняется Trie и формируется шаблон:

```
1   while (scanf("%c", &sq) != EOF) {
2       if ((sq >= '0') && (sq <= '9')) {
3           if(writeReady) {
4               inpword.push_back(inp);
5               ++len;
6               inp = 0;
7               if (DEBUG)printf("pushed\n");
8               writeReady = false;
9           }
10          inp *= 10;
11          inp += sq - '0';
12          lastq = 0;
13      }
14      else
15      if (sq == ' ' && lastq == 0) {
16          writeReady = true;
17      }
18      else
19      if (sq == '?') {
20          if (lastq == 0) {
21              if(writeReady) {
22                  inpword.push_back(inp);
23                  ++len;
24                  inp = 0;
25                  if (DEBUG)printf("pushed\n");
26                  writeReady = false;
27              }
28              count += len;
29              patCOUNT++;
30              mainTrie.add(inpword,count);
31              jPattern.push_back(count);
32              len = 0;
33              inpword.clear();
34              if (DEBUG)printf("added\n");
35
36              jPattern.push_back(-1);
37          }
38          else {
39              jPattern.push_back(-1);
40          }
41          lastq = 1;
42          ++count;
```

```
43      }
44      else
45      if (sq == '\n') {
46          if (lastq == 0) {
47              inpword.push_back(inp);
48              ++len;
49              count += len;
50              patCOUNT++;
51              mainTrie.add(inpword,count);
52              jPattern.push_back(count);
53              len = 0;
54              inpword.clear();
55              if (DEBUG)printf("added\n");
56          }
57          break;
58      }
59  }
```

Затем считывается текст, и производится поиск. Суффиксные ссылки строятся в ходе работы. После завершения поиска итоговый массив проверяется на наличие в нем признаков вхождений шаблона:

```
1   if(preRes.size() >= count && count != 0)
2   for(unsigned long i=0;i<=preRes.size()-count;++i) {
3       while(i+1 >= textmodula[modulapos]) {
4           modulapos++;
5       }
6       if(preRes[i] == patCOUNT) {
7           printf("%lu, %lu\n", modulapos, i+2 - textmodula[modulapos-1]);
8
9       }
10  }
```

3 Тест производительности

Проводим тест производительности. Я создал простой генератор тестов, создающий файл, состоящий из первой строки заданного размера (в ней могут встречаться знаки вопроса) и набора из заданного числа чисел в следующих строках. В самой программе поиска я замерял только время работы функции search.

```
max@max-Swift:~/Рабочий стол/ДА/lab4/DA_lab_4-main$ ./lab4bg.exe 5 100000000
done
max@max-Swift:~/Рабочий стол/ДА/lab4/DA_lab_4-main$ ./lab4b.exe <test_a
search complete| 6324ms
max@max-Swift:~/Рабочий стол/ДА/lab4/DA_lab_4-main$ ./lab4bg.exe 5 50000000
done
max@max-Swift:~/Рабочий стол/ДА/lab4/DA_lab_4-main$ ./lab4b.exe <test_a
search complete| 3148ms
```

Вначале был сгенерирован файл с текстом в 100000000 символов, затем – вдвое меньше, 50000000. видим, что зависимость линейная.

4 Выводы

В ходе выполнения этой лабораторной я понял, насколько важно изучить максимум доступной теоретической информации, прежде чем приступать к работе: вместо того чтобы несколько недель пытаться отладить заведомо неэффективный метод хранения отдельных записей о вхождении образцов и последующего их анализа я бы мог повнимательнее прочитать одну из статей, что я использовал для изучения алгоритма и заметить, что в ней также приведено общее описание куда более эффективного метода работы со строкой, содержащей джокеры, который в итоге и был реализован. Также я получил опыт работы с алгоритмами поиска подстроки в строке, что, несомненно, пригодится мне когда-нибудь.

Список литературы

[1] *std::map — cppreference.com.*
URL: `https://en.cppreference.com/w/cpp/algorithm/stable_sort` (дата обращения: 22.11.2020)

[2] *Алгоритм Ахо-Корасик — Викиконспекты.*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Ахо-Корасик` (дата обращения: 09.11.2020)