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: Л. Я. Вельтман
Преподаватель: А. А. Кухтичев
Группа: М8О-207Б

Дата:
Оценка:
Подпись:

Москва, 2019

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», 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 от 0 до $2^{32} - 1$.

1 Описание

Для реализовации поиска одного образца с джокером нужно применить алгоритм Ахо-Корасик с некоторой модификацией. В дереве ключей будет храниться не один образец, а все подстроки образца, не содержащие джокеры. Далее нужно построить для дерева связи-неудач для каждой вершины в порядке обхода в ширину. Связь-неудач связывает в дереве две вершины, первая вершина получена после прохода по тексту и совпадения символов текста с каким-то образцом до следующей вершины, в которой найдено не совпадение, а вторая вершина соответствует символу другого образца, префикс которого является максимальным суффиксом уже пройденного пути. Также при построении дерева нужно найти длины и позиции начала подстрок образца, не содержащих джокеров. Позиции понадобятся при поиске образца в тексте и для проверки относительного месторасположения подстрок. Препроцессинг для построения связей неудач работает за O(n), где n - сумма длин всех подстрок образца без джокеров.

Пусть ? - маска, обозначающая любой символ. Например, шаблон ab?c? встречается на позициях 2 и 6 строки xabvcabbcax. Если количество ? ограничено сверху константой, то шаблоны с масками могут быть выявлены за линейное время. Для этого надо обнаружить безмасочные куски шаблона Q:

1. Пусть Q1, ..., Qk - набор подстрок Q, разделенных масками, и пусть l1, ..., lk - их стартовые позиции в Q.
2. for i := 1 to m do C [i] := 0;
3. Используя алгоритм Ахо-Корасик, находим подстроки Q: когда находим Qi в тексте T на позиции j, увеличиваем на единицу C [j - li + 1].
4. Каждое i, для которого C [i] = k, является стартовой позицией появления шаблона Q.

2 Исходный код

1. main.cpp (содержит основной метод main, в нем считывается образец и текст, вызываются методы Create, CreateFailLinks, Search).
2. TBor.h (описание класса TAhoCorasick и класса TBorNode).
3. TBor.cpp (реализация функций создания бора, создания связей-неудач и поиска образца в тексте).

TBor.cpp	
void Create(const std::vector<std::string>&);	Создание бора.
void CreateFailLinks();	Построение связей-неудач.
void Search(const std::vector<unsigned long>&, const int, std::vector< std::pair<int, int»);	Поиск образца в тексте.
int main (int argc, const char * argv[])	Точка входа в программу.

```
1  class TAhoCorasick {
2  private:
3      class TBorNode {
4      public:
5          TBorNode();
6          virtual ~TBorNode() {};
7
8          std::shared_ptr<TBorNode> fail;
9          std::map<unsigned long, std::shared_ptr<TBorNode>> links;
10         std::vector<int> out;
11     };
12
13     std::shared_ptr<TBorNode> root;
14     std::vector<int> lengthSubstr;
15     int withoutJoker;
16
17 public:
18     TAhoCorasick();
19     virtual ~TAhoCorasick() {};
20
21     void Create(const std::vector<std::string>&);
22     void CreateFailLinks();
23     void Search(const std::vector<unsigned long>&, const int, std::vector< std::pair<
           int, int>>);
24 };
```

3 Консоль

```
MacBook-Pro-Lina:da4 linuxoid$ make
g++ -std=c++11 -Wall -Werror -Wno-sign-compare -Wno-unused-result -O3 -o da4
main.cpp TBor.cpp
MacBook-Pro-Lina:da4 linuxoid$ cat test1
468 145 890 408 119
385 507 448 282 670 357 276 485 445 824 356 966 545 462 512 540
205 752 933 132 36 261 267 76 713 870 749 446 592 982 253 175
200 978 683 444 135 125 295 44 17 384 281 230 725 860 101 409 42
775 206 79 376 515 86 315 597 726 536 3 626 330 299 905 838 847
68 113 811 841 908 484 74 561 103 383 550 403 189 130 695 332
803 863 148 751 582 307 746 738 342 815 474 861 1000 140 945 806
250 928 352 696 142 335 595 862 662 565 893 581 256 949 77 126
733 163 393 45 882 82 614 671 340 845 895 703 13 950 812 274 794
698 168 883 577 823 293 819 425 185 562 972 580 390 260 846 88
968 809 226 560 687 313 538 164 150 325 805 397 447 452 919 399
272 176 591 306 431 722 767 648 621 666 137 849 303 120 607 112
700 309 128 639 727 842 697 158 41 552 258 481 194 1 162 645 83
264 654 23 432 627 706 606 628 704 935 927 548 900 504 782 463
754 116 178 367 837 50 778 288 218 816 160 435 220 777 57 749
859 493 777 533 853 502 66 579 492 434 205 130 110 442 746 517
768 192 68 567 563 307 673 907 858 769 252 773 368 137 306 401
177 117 89 793 19 565 603 116 377 986 729 350 162 969 747 805
558 955 721 199 436 628 515 958 154 251 381 836 985 272 807 4
864 610 544 806 14 730 779 767 281 444 258 253 844 323 78 23 966
MacBook-Pro-Lina:da4 linuxoid$ ./da4 <test1
MacBook-Pro-Lina:da4 linuxoid$ cat test2
2 3 4 5 6 ? ? ? ?
2 3 4 5 6 7 9 12
2 3 4 5 6 7 10 13
2 3 4 5 6 8 11 14
MacBook-Pro-Lina:da4 linuxoid$ ./da4 <test2
1,1
2,1
MacBook-Pro-Lina:da4 linuxoid$ cat test3
1188 1950 ? 1751
1557 1029 1365 1331 1499 1220 1109 1669 1923 1077 1081
1188 1950 1434 1937 1086 1925 1751 1169 1476 1073 1938 1984 1834
1169 1476 1073 1938 1984 1834 1188 1950 1434 1937 1086 1925 1751
1188 1950 1073 1751
```

```
23787 2388 1212 98239
MacBook-Pro-Lina:da4 linuxoid$ ./da4 <test3
4,1
```

4 Тест производительности

Для теста производительности я использую метод find в std::vector из STL для сравнения со своим модифицированным поиском Ахо-Корасик. В алгоритм find передается две точки: старт и конец поиска. Между точками находится диапазон, в котором находятся значения. Если встретился элемент, который равен третьему аргументу, то выполняется какое-то условие. Выходит, что этот метод работает за линейное время, так как просматривает каждое число в тексте.

Тест производительности представляет из себя следующее: 1 тест - образец состоит только из джокеров (длина 10 символов). Текст содержит в себе 100 строк по 100 чисел в каждой строке. 2 тест - образец состоит только из джокеров (длина 10 символов). Текст содержит в себе 10000 строк по 100 чисел в каждой строке. 3 тест - числа и джокеры. Длина образца равна 5. Текст состоит из 10 строк по 100 чисел в каждой строке. 4 тест - числа и джокеры. Длина образца равна 5. Текст состоит из 100 строк по 100 чисел в каждой строке. 5 тест - числа и джокеры. Длина образца равна 5. Текст состоит из 10000 строк по 100 чисел в каждой строке. 6 тест - образец не содержит джокеров. Длина образца 10 чисел. Текст 100 строк по 100 чисел. 7 тест - образец не содержит джокеров. Длина образца 10 чисел. Текст 10000 строк по 100 чисел в каждой строке.

```
MacBook-Pro-Lina:da4 linuxoid$ ./da4 <test1
Aho-Corasick search time is: 0.022964 seconds

MacBook-Pro-Lina:da4 linuxoid$ ./da4 <test2
Aho-Corasick search time is: 2.26397 seconds

MacBook-Pro-Lina:da4 linuxoid$ ./da4 <test3
Aho-Corasick search time is: 0.00203 seconds

MacBook-Pro-Lina:da4 linuxoid$ ./da4 <test4
Aho-Corasick search time is: 0.000478 seconds

MacBook-Pro-Lina:da4 linuxoid$ ./da4 <test5
Aho-Corasick search time is: 0.020691 seconds

MacBook-Pro-Lina:da4 linuxoid$ ./da4 <test6
Aho-Corasick search time is: 0.000289 seconds
std::find search time is: 1.1e-05 seconds

MacBook-Pro-Lina:da4 linuxoid$ ./da4 <test7
Aho-Corasick search time is: 0.020152 seconds
```

```
std::find search time is: 2e-06 seconds
```

Анализируя время выполнения данных тестов можно сделать вывод, что алгоритм Ахо-Корасик с джокерами показывает хороший результат по времени как на маленьких, так и на больших данных. Но в сравнении с алгоритмом find в 7 тесте АК проигрывает по времени. Засчет препроцессинга получается дерево ключей, состоящее из пути длины 100, поэтому на работу тратится такое количество времени. Также немаловажную роль могло сыграть то, что в тексте не было ни одного вхождения образца.

5 Выводы

Таким образом, выполнив данную лабораторную работу, я научилась решать задачу поиска образца в заранее известном тексте.

В итоге, временная оценка построения суффиксного дерева O(m*logs), где m - длина текста, s - размер алфавита. Все вхождения образца в текст находятся за O(n*logk), где n - длина образца, k - количество ребёр суффиксного дерева.

Несмотря на то, что данный алгоритм является одним из самых простых в понимании алгоритмов для построения суффиксных деревьев и использует online подход, у него есть серьёзные недостатки, из-за которых его нечасто используют на практике. Размер суффиксного дерева сильно превосходит входные данные, поэтому при очень больших входных данных алгоритм Укконена сталкивается с проблемой memory bottleneck problem(другое её название thrashing). Также алгоритм предполагает, что дерево полностью должно быть загружено в оперативную память. Если же требуется работать с большими размерами данных, то становится не так просто модифицировать алгоритм, чтобы он не хранил всё дерево в ней.

Суффиксные деревья имеют широкую область применения. С помощью него можно найти количество различных подстрок данной строки, наибольшую общую подстроку двух строк, суффиксный массив, статистику совпадений. Все вышеприведенные задачи решаются за линейное время.

Список литературы

[1] *Сопоставление множеств и алгоритм Ахо-Корасик.*
URL: `http://aho-corasick.narod.ru`.
(дата обращения: 20.11.2018).

[2] *Алгоритм Ахо — Корасик.*
URL: `https://ru.wikipedia.org/wiki/Алгоритм_Ахо_-_Корасик`.
(дата обращения: 20.11.2018).

[3] *Алгоритм Ахо-Корасик.*
URL: `https://habr.com/post/198682/`.
(дата обращения: 20.11.2018).