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: П. А. Харьков
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

Москва, 2020

Лабораторная работа №4

Задача: Необходимо реализовать один из стандартных алгоритмов поиска образцов для указанного алфавита.

Вариант алгоритма: Поиск большого количества образцов при помощи алгоритма Ахо-Корасик.

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

1 Описание

Требуется написать алгоритм Ахо-Корасик для поиска подстроки в строке.
Существует много алгоритмов для поиска подстроки в строке, таких как: алгоритм Кнута-Морриса-Пратта, Бойера-Мура, Апостолико-Джанкарло. Но все они полезны для поиска одной подстроки в тексте, так как их сложность может возрасти в n раз, если необходимо будет искать n подстрок. Это связано с тем, что для поиска каждой подстроки необходимо будет заново пройтись по тексту.
На помощь приходит алгоритм Ахо-Корасика, согласно [1] его сложность равна $O(m+t+a)$, где m - сумма длин всех подстрок, t - длина текста, a - количество ответов. Алгоритм Ахо-Корасика несложен: сначала из подстрок строится обычный бор, затем его преобразовываем, добавляя суффиксные ссылки и терминальные (их ещё называют сжатыми суффиксными) ссылки, и после этого просто просто идем по тексту, переходя по узлам бора.

2 Исходный код

Вставка строки в бор происходит следующим способом, сначала мы находимся в корне и начинаем идти по строке:

1. Если в детях текущего узла нет слова, на котором мы остановились, то вставляем его.
2. Если есть, то переходим в узел со значением этого слова.
3. Переходим к следующему слову.

Когда строка закончилась, мы сохраняем в узел, в котором мы остановились, информацию о том, что это «терминал», то есть конец слова.
Затем в бор нам необходимо добавить суффиксные ссылки. Идти по бору мы будем поиском в ширину:

1. Сначала находим суффиксную ссылку для узла:
   (a) Переходим к родителю.
   (b) Идём по суффиксным ссылкам, пока в узле по суффиксной ссылке не будет ребёнка со значеним рассматриваемого узла. Если ребёнок есть, то суффиксной ссылкой указываем на это узел.
   (c) Если мы пришли в корень и у корня нет нужного ребёнка, то суффиксная ссылка - корень.
2. Затем необходимо определить терминальную ссылку:
   - Если узел по суффиксной ссылке - «терминальный», то терминальной ссылкой указываем на него.
   - В другом случае терминальная ссылка равна терминальной ссылке узла по суффиксной ссылке.
3. Добавляем детей текущего узла в очередь и рассматриваем относительно них.

Поиск подстрок происходит следующим способом. Сначала рассматриваемый узел - корень, мы идём по слова в строке:

1. Если текущего слова нет в детях текущего узла, то переходим по суффиксным ссылкам, пока не найдем необходимого ребёнка или не придем в корень.
2. Если текущий узел - «терминальный», то значит подстрока совпала и сохраняем узел в результат.

3. Если у текущего узла есть терминальная ссылка, то сохраняем узлы по ним в результат.

4. Переходим к следующему слову в строке.

Таблица функций:

main.cpp	
void SplitLine(string& line, vector<string>& vec)	Функция, разбивающая строку на вектор слов.
aho_corasick.cpp	
VNode* VNode::GetChild(string& str)	Функция, возвращающая указатель на ребёнка со значением *str*
void VTrie::AddPattern(vector<string>& pattern, TUll line)	Функция, добавляющая слова в бор.
VNode* VTrie::GetSuffix(VNode* node, string& val)	Функция, возвращающая суффиксную ссылку для узла.
void VTrie::MakeSuffixes()	Функция, обрабатывающая бор: определяющая суффиксные ссылки.
void VTrie::SearchPatterns(vector<VInput>& text, vector<VResults>& res)	Функция, выполняющая поиск подстрок в тексте.
void VTrie::DeleteTrie(VNode* node)	Функция, удаляющая бор.

Структуры и классы без реализаций их методов:

```
1  struct VInput
2  {
3      std::string Str;
4      unsigned long long Line;
5      unsigned long long Number;
6  };
7
8  struct VResults{
9      unsigned long long Line;
10     unsigned long long Word;
11     unsigned long long Sample;
12     VResults(unsigned long long line, unsigned long long word, unsigned long long
           sample):
13             Line(line), Word(word), Sample(sample){}
14 };
15
16 struct VNode{
17     std::string Value;
18     unsigned long long Line;
19     unsigned long long Length;
```

```
20      VNode* Par;
21      VNode* SuffLink;
22      VNode* TerminalLink;
23      std::map<std::string, VNode*> Childs;
24      std::map<std::string, VNode*> Next;
25
26      VNode():Value(""), Line(0), Length(0), Par(nullptr), SuffLink(nullptr),
            TerminalLink(nullptr){}
27
28      VNode(std::string& val, VNode* p):
29                      Value(val), Line(0), Length(0), Par(p), SuffLink(nullptr),
                            TerminalLink(nullptr){}
30
31      VNode* GetChild(std::string& str);
32      bool IsTerminal();
33  };
34
35  class VTrie{
36      VNode* GetSuffix(VNode* node, std::string& val);
37      void DeleteTrie(VNode* root);
38
39      VNode* Root;
40  public:
41      void AddPattern(std::vector<std::string>& pattern, unsigned long long line);
42      void MakeSuffixes();
43      void SearchPatterns(std::vector<VInput>& text, std::vector<VResults>& res);
44
45      VTrie(): Root(new VNode());
46      ~VTrie();
47  };
```

3 Консоль

```
p.kharkov$ make
g++ -std=c++14 -pedantic -Wall -c aho_corasick.cpp -o aho_corasick.o
g++ -std=c++14 -pedantic -Wall main.cpp aho_corasick.o -o laba4
p.kharkov$ cat tests/main
cat dog cat dog
CAT dog CaT
Dog doG dog dOg

Cat doG cat dog  cat dog cat Parrot
doG dog DOG DOG   dog
p.kharkov$ ./laba4 <tests/main
1,1,2
1,1,1
1,3,2
1,3,1
1,5,2
2,1,3
2,2,3
```

4 Тест производительности

Тест производительности представляет из себя следующее: поиск образцов с помощью алгоритма Ахо-Корасика сравнивается с поиском с помощью std::find. Время построения бора и суффиксных ссылок в нём не учитывается. С помощью std::find ищутся все вхождения образца в тексте, а не одно. Тест состоит из 100 образцов, длина которых может быть от 2 до 10 слов, и текста, состоящего из 10^5 слов. Каждое слово состоит из 1 или 2 букв.

```
p.kharkov$ make benchmark
g++ -std=c++14 -pedantic -Wall -c aho_corasick.cpp -o aho_corasick.o
g++ -std=c++14 -pedantic -Wall benchmark.cpp aho_corasick.o -o benchmark
p.kharkov$ ./benchmark <tests/01.t
aho time: 42ms
find time: 91ms
```

Как видно, алгоритм Ахо-Корасика выиграл у std::find почти в 2 раза. Скорее всего, это связано с тем, что алгоритм Ахо-Корасика за один проход по тексту находит все подстроки, в то время, как std::find необходимо для каждого образца заново идти по всему тексту.

5 Выводы

Выполнив четвертую лабораторную работу по курсу «Дискретный анализ», я лучше разобрался с алгоритмами поиска подстрок в подстроке, особенно в алгоритме Ахо-Корасика, так как реализовывал его. Это полезно, так как в будущем это может помочь мне с выбором правильного решения для поиска подстроки. К примеру, если мне необходимо будет искать сразу несколько подстрок в строке, то, конечно же, я буду использовать алгоритм Ахо-Корасика. А если будет необходимо искать только одну подстроку, то для простоты можно реализовать и алгоритм Кнута-Морриса-Пратта.

Так как алфавитом являлись слова длины до 16 символов, то я решил использовать map для их хранения, но сложность моего алгоритма возросла, как минимум, до $O((n + t) * lg(n) + a)$. Но после я вспомнил, что существует такая структура, как $unordered\_map$, основанная на хеш-таблице. И так как сложность вставки и поиска по ней, в среднем $O(1)$, то в моем случае это гораздо лучше.

Список литературы

[1] *Алгоритм Ахо-Корасик - Викиконспект*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Ахо-Корасик`
(дата обращения: 16.12.2020)