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

Дата:
Оценка:
Подпись:

Москва, 2019

Лабораторная работа №4

Необходимо реализовать один из стандартных алгоритмов поиска образцов для указанного алфавита.
Вариант алгоритма: Поиск одного образца основанный на построении Z–блоков.
Вариант алфавита: Числа в диапазоне от 0 до 232 - 1.

1 Описание

Как сказано в [1]: «У многих алгоритмов сравнения и анализа строк эффективность сильно возрастает из-за пропусков сравнений. Эти пропуски получаются из благодаря изучению внутренней структуры либо образца P, либо текста T. При этом другая строка может даже оставаться неизвестной алгоритму. Эта часть алгоритма называется *препроцессной* фазой. За ней следует фаза *поиска*, на которой информация, полученная в *препроцессной* фазе, используется для сокращения работы по поиску вхождений P в T. Методы, основанные на основном препроцессинге образца P, работают за линейное время. Рассмотрим один из таких методов – поиск образца, основанный на построении Z–блоков».

Определение. Для данной строки S и позиции i > 1 определим $Z_i(S)$, как *длину* наибольшей подстроки S, которая *начинается* в i и совпадает с префиксом S. Другими словами, $Z_i(S)$ – это длина наибольшего *префикса* $S[i..|S|]$, совпадающего с префиксом S.

Определение. Для любой позиции i > 1, в которой $Z_i > 0$, определим *Z–блок* в i как интервал и кончающийся в позиции i + Z_i + 1.

Определение. Для любого i > 1 пусть r_i – крайний правый конец Z–блоков, начинающихся не позднее позиции i. По-другому r_i можно определить, как наибольшее значение j + Z_j - 1 по всем $1 <$ j $\leq$ i, для которых $Z_j > 0$.

Обозначим значение j, в котором достигается значение этого максимума, как l_i. Таким образом, l_i – это позиция *левого конца* Z–блока, кончающегося в r_i.

Вычисление значения Z для строки S (образца P) за линейное время и есть основной препроцессинг.

2 Исходный код

На первой строке входного файла располагается некоторый образец P (*vector<unsigned int> pattern*). В первую очередь находим значение Z–блока для него. Z–блок определим, как вектор *std::vector<int> z*, размер которого равен длине образца и плюс один элемент (значение последнего вычисленного Z_i нашего текста).
Решение не хранить все значения Z_i, где $i = /0, .., T.size() + P.size()/$ было принято, чтобы максимально минимизировать длину Z–блока, т.к. значение каждого последующего Z_i зависит лишь от Z_j, где $0 < j \leq$ P.size().
Так, инициализировав первые P.size() элементов Z некоторыми значениями, переходим ко второму этапу поиска вхождений. Будем считывать текст T в цикле и, когда T.size() станет равен P.size(), найдем значение Z_i, где i- позиция первого считанного символа текста T. Мы считываем текст не до конца, т.к, длина части T, которая потенциально может совпасть с образцом, равна длине этого образца. Далее, первый элемент T, для которого значение Z уже обработано, удаляется и считывается новый символ – тем самым мы двигаемся по тексту T, поочередно находя значения Z.
Так, функция *void ZFunction(vector<unsigned int> pattern, vector<tuple<int, int, unsigned int» text, std::vector<int> z)* получает в качестве параметров ссылку на образец, ссылку на считанный текст и ссылку на найденный Z–блок. Т.к. *ZFunction* вызывается по мере считывания T, переменные l, r, начальные значения которых равны нулю, объявим, как глобальные. Так же, в нашем случае длина Z–блока является постоянной, меняется только значение последнего элемента Z_i, где i = P.size(). Для того, чтобы алгоритм работал корректно и правильно считал l и r, сымитируем прохождение по всей длине текста, объявив индекс k – реальную позицию символа в тексте T. Получим, что для T_k значение Z = Z_i, где i = P.size().
Разобравшись со всеми особенностями этой реализации, рассмотрим подробно работу Z–алгоритма:
Пусть S – строка, полученная конкатенацией P и T. Если текущий индекс, для которого мы хотим посчитать очередное значение Z_i, — это k, мы имеем один из двух вариантов:

1. k > r — т.е. текущая позиция лежит за пределами того, что мы уже успели обработать. Тогда будем искать Z_i тривиальным алгоритмом, т.е. посимвольно сравнивая P и T до первого расхождения.
   Заметим, что в итоге, если Z_i окажется > 0, то мы будем обязаны обновить координаты самого правого отрезка [l, r] — т.к. k + Z_i - 1 гарантированно окажется больше r.

2. k ≤ r — т.е. текущая позиция лежит внутри отрезка совпадения [l, r]. Тогда мы можем использовать уже подсчитанные предыдущие значения Z–блока, чтобы проинициализировать значение Z_i не нулём, а каким-то возможно бОльшим

числом. Для этого заметим, что подстроки S[l ... r] и S[0 ... r - l] совпадают. Это означает, что в качестве начального приближения для Z_i можно взять соответствующее ему значение из отрезка S[0 ... r - l], а именно, значение Z_{k-l}. Однако значение Z_{k-l} может оказаться слишком большим: таким, что при применении его к позиции k оно "вылезет"за пределы границы r. Этого допустить нельзя, т.к. про символы правее r мы ничего не знаем, и они могут отличаться от требуемых.
Таким образом, в качестве начального приближения Z_i безопасно брать значение, равное Max(0, Min(r - k + 1, Z_{k-l})).
Проинициализировав Z_i таким значением, мы снова дальше действуем тривиальным алгоритмом —- потому что после границы r, вообще говоря, могло обнаружиться продолжение отрезка совпадений, предугадать которое одними лишь предыдущими значениями Z–функции мы не могли.

Таким образом, весь алгоритм представляет из себя два случая, которые фактически различаются только начальным значением Z_i: в первом случае оно полагается равным нулю, а во втором —- определяется по предыдущим значениям по указанной формуле. После этого обе ветки алгоритма сводятся к выполнению тривиального алгоритма, стартующего сразу с указанного начального значения.
Вспомогательные функции перечислены в таблице.

lab04.c	
int Min(int first, int second)	Функция поиска минимума
int Max(int first, int second)	Функция поиска максимума

Листинг 1: ZFunction

```
1  void
2  ZFunction(vector<unsigned int> &pattern, vector<tuple<int, int, unsigned int>> &text, std::
       vector<int> &z){
3      vector<unsigned int>::iterator it_pattern = pattern.begin();
4      vector<tuple<int, int, unsigned int>>::iterator it_text = text.begin();
5      z[i] = Max(0, Min(r − k + 1, z[k − l]));
6      while( (it_pattern + z[i]) != pattern.end() && (it_text + z[i])!= text.end() && ∗(it_pattern
           + z[i]) == get<2>(∗(it_text + z[i]))){
7        z[i]++;
8      }
9      if(k + z[i] − 1 > r){
10       l = k;
11       r = k + z[i] − 1;
12     }
```

```
13      if(z[i] == pattern.size()){
14        std::cout << get<0>(*it_text) << ", " << get<1>(*it_text) << std::endl;
15      }
16      text.erase(it_text);
17      k++;
18  }
```

3 Консоль

```
svetlana@svetlana-VirtualBox    /DA/lab4/4 $ cat ../test
7 7 7
7 7 7 7 7 0 0 0 7 7 7
7 7 7 7 7 7 7
7 7 7
1 3 7 7 7 7 7
svetlana@svetlana-VirtualBox    /DA/lab4/4 $ make
g++ 04.cpp -std=c++11 -pedantic -Wall -Werror -Wno-sign-compare -Wno-long-long -Im
-O -o 4
svetlana@svetlana-VirtualBox    /DA/lab4/4 $ ./4 < ../test
1, 1
1, 2
1, 3
1, 9
1, 10
1, 11
2, 1
2, 2
2, 3
2, 4
2, 5
2, 6
2, 7
3, 1
4, 3
4, 4
4, 5
```

4 Тест производительности

Тест производительности программы будет заключаться в сравнении результатов работы Z–алгоритма и STL–функции find. Принцип работы алгоритма с использованием find будет заключаться в том, что к каждой позиции текста T будет применен алгоритм std::find. Так, алгоритм не будет работать за линейное время. Сравним результаты на небольшом тесте в 10 записей и тесте в 10.000 записей. Время выполнения будет записано в миллисекундах.

5 Консоль

```
svetlana@svetlana-VirtualBox   /DA/lab4/4 $ wc -l test*
10001 test1.txt
11 test2.txt
10012 итого
svetlana@svetlana-VirtualBox   /DA/lab4/4 $ ./4 < test1.txt | tail -n 1
ZTime: 0.055133
svetlana@svetlana-VirtualBox   /DA/lab4/4 $ ./find < test1.txt | tail -n 1
STL-findTime: 15.0308
svetlana@svetlana-VirtualBox   /DA/lab4/4 $ ./4 < test2.txt | tail -n 1
ZTime: 0.00272
svetlana@svetlana-VirtualBox   /DA/lab4/4 $ ./find < test2.txt | tail -n 1
STL-findTime: 0.00217
```

Результаты теста производительности:

Data	ZFunction	STL–find
10	2.72	2.17
10.000	55.133	15030.8

Исходя из результатов, можно сделать вывод о том, что на больших входных данных время работы алгоритма с использованием std::find растет очень сильно. Я не нашла в STL-алгоритмах аналогов Z-функции, время выполнения которых было бы линейным.

6 Выводы

Выполнив третью лабораторную работу по курсу «Дискретный анализ», я изучила алгоритм поиска образца в строке с помощью Z–блоков. Он основан на препроцессинге образца – вычислении Z–функции для образца P. Информация, полученная в *препроцессной* фазе, используется для сокращения работы по поиску вхождений P в T. Таким образом, мы минимизируем число сравнений, и время поиска образца становится линейным $O(n)$.

Список литературы

[1] Гасфилд Д. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология. Часть 1.4. Основной препроцессинг за линейное время.