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: А. М. Голев
Преподаватель: С. А. Михайлова
Группа: М8О-201Б

Дата: 09.04.25
Оценка:
Подпись:

Москва, 2025

Лабораторная работа №4

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

Вариант алгоритма: Поиск одного образца при помощи алгоритма Кнута-Морриса-Пратта.

Вариант алфавита: `Числа в диапазоне от 0 до` $2^{32} - 1$.

Запрещается реализовывать алгоритмы на алфавитах меньшей размерности, чем указано в задании.

1 Описание

Требуется написать реализацию алгоритма Кнута-Морриса-Пратта

Основная идея алгоритма КМП заключается в следующем: Предположим, что при некотором выравнивании P около T наивный алгоритм обнаружил совпадение первых i символов из P с их парами из T, а при следующем сравнении было несовпадение. В этом случае наивный алгоритм сдвинет P на одно место и начнет сравнение заново с левого конца P. Но часто можно сдвинуть образец дальше. Например, если P = abcxaводе и при текущем расположении P и T несовпадение нашлось в позиции 8 строки P, то есть возможность сдвинуть P на четыре места без пропуска вхождений P в T. Требуется только место несовпадения в P. Алгоритм Кнута-Морриса-Пратта, основываясь на таком способе рассуждений, и делает сдвиг больше, чем наивный алгоритм. [**?**].

Время работы алгоритма линейно зависит от объема входных данных. [**?**].

Алгоритм КМП состоит из 2-х этапов: препроцессинга образца и непосредственно поиска.

Первому этапу требуется только образец. Для каждой позиции i в образце, вычисляется значение функции $SP'_i(P)$, равное длине наибольшего собственного суффикса подстроки $P[1..i]$, совподающего с префиксом P, при условии, что $P_{SP'_i+1} = P_{i+1}$.

Для вычисления SP'_i используется основная функция препроцессинга $Z_i(P)$. Для каждой позиции i в образце, значение Z-функции равно наибольшей длине собственного префикса P, совпадающего с префиксом $P[i+1..n]$.

Для эффективного вычисления Z-функции в позиции i, используются значения в предыдущих позициях. Z-блоком для позиции i называется строка P[l..r], где $1 \leq l \leq i$, $r = l + SP_l$ и значение r наибольшее. Если i не принадлежит Z-блоку, значение функции в позиции i вычисляется по определению. В противном случае по меньшей мере $min(Z_{i-l}, r - i)$. При этом, если функция больше r - i, строка Z_i может быть больше, поэтому значение функции увеличивается до тех пор, пока это возможно (как при вычислении по определению).

С вычисленным значением Z-функции для позиций $1..n$, значения SP' могут быть вычислены по следующему алгоритму.

```
1 for (i := 1 to n) sp_i := 0
2
3 for (j := n downto 2) {
4     i := j + Z_j(P) - 1
5     sp_i := Z
6 }
```

Наконец, на втором шаге выполняется поиск. Проверка совпадения осуществляется также, как в наивном алгоритме. Однако в случае обнаружения несовпадения на позиции j, образец сдвигается не на 1, а на $j - sp_j$.

2 Исходный код

Код программы начинается с определения структуры Num, хранящей букву (в данном случае число) текста вместе с его позицией: номером строки и отступом в ней.

Далее определяется тип данных TextReader, позволяющий обращаться к произвольной букве текста так, как будто он хранится в массиве. Однако в память сохраняется только порядка n букв текста в кольцевом буфере. (n - длина образца). Такой подход позволяет без существенных изменений реализации алгоритма КМП избавиться от необходимости считывания всего текста в оперативную память.

Далее определяются операции для этого типа, которые будут рассмотрены в таблице ниже. Затем идут функции считывания и препроцессинга образца и функция поиска.

Текст программы завершается функцией main, в которой происходит считывание образца, затем выполняется его препроцессинг, после чего начинается поиск.

```
1  #include <stdio.h>
2  #include <inttypes.h>
3  #include <stdlib.h>
4  #include <stdbool.h>
5
6  #define EXTEND_SIZE 10
7
8  typedef struct Num {
9      uint32_t num;
10     size_t index;
11     int64_t line;
12 } Num;
13
14 typedef struct TextReader {
15     Num *buffer;
16     size_t bufSize;
17     size_t offset;
18     size_t lineNum;
19     size_t lineIndex;
20 } _TextReader, *TextReader;
21
22 TextReader createTextReader(size_t patternSize);
23 void deleteTextReader(TextReader reader);
24 Num nextNum(TextReader reader);
25 Num getNum(TextReader reader, size_t index);
26
27 uint32_t *readSequence(size_t *size);
28 uint16_t *calcZ(uint32_t *s, size_t m);
29 uint16_t *preprocess(uint32_t *p, size_t m);
30
31 void search(uint32_t *P, size_t m, uint16_t *SP);
32
33 int main();
```

main.c	
TextReader createTextReader(size_t patternSize)	Конструктор TextReader
void deleteTextReader(TextReader reader)	Деструктор TextReader
Num nextNum(TextReader reader)	Считывает следующее число из потока ввода, отсчитывает позицию.
Num getNum(TextReader reader, size_t index)	Получает букву в тексте с индексом index.
uint32_t *readSequence(size_t *size)	Считывает образец из потока ввода. Записывает его размер в size.
uint16_t *calcZ(uint32_t *s, size_t m)	Вычисляет массив значений Z-функции для образца.
uint16_t *preprocess(uint32_t *p, size_t m)	Вычисляет массив значений функции SP' для образца.
void search(uint32_t *P, size_t m, uint16_t *SP)	Выполняет поиск в образце по таблице значений SP'. Текст считывается из stdin.
int main()	Точка входа в программу.

3 Консоль

```
smoking_elk@DESKTOP-PJPQAEE:~/discran-labs/lab4$ gcc main.c -o app.out
smoking_elk@DESKTOP-PJPQAEE:~/discran-labs/lab4$ cat test1
34 35 34 35 69
0034 035 00034 35 34
0035 69 78 38 34 35 34 35 69
smoking_elk@DESKTOP-PJPQAEE:~/discran-labs/lab4$ ./app.out <test1
1,3
2,5
```

4 Тест производительности

Тест производительности представляет из себя следующее: поиск образцов с помощью std::find сравнивается с поиском алгоритма КМП. Учитывается время на считывание текста, поскольку исходная реализация выполняет поиск в процессе считывания. Текст состоит из 1 миллиона чисел из ограниченного набора, а образец - из 100 тысяц чисел, среди которых может встречаться одна, которой нет в тексте. Благодаря этому достигается проверка на худшем случае.

```
smoking_elk@DESKTOP-PJPQAEE:~/discran-labs/lab4/benchmark$ make
g++ find.cpp -o find.out
gcc kmp.c -o kmp.out
smoking_elk@DESKTOP-PJPQAEE:~/discran-labs/lab4/benchmark$ make test_find
./find.out <./in.txt | grep "time"
time: 120.197000ms
smoking_elk@DESKTOP-PJPQAEE:~/discran-labs/lab4/benchmark$ make test_kmp
./kmp.out <./in.txt | grep "time"
time: 54.303000ms
```

Как видно, реализованный алгоритм КМП выигрывает у std::find. Вероятно, этому способствует как более эффективная реализация алгоритма поиска, так и более эффективный метод считывания текста. Однако порядки времени работы одинаковые, что позволяет предположить, что метод find также использует один из алгоритмов поиска по образцу за линейное время.

5 Выводы

Выполнив первую лабораторную работу по курсу «Дискретный анализ», я научился реализовывать алгоритм Кнута-Морриса-Пратта для эффективного поиска образца в тексте. Также я научился использовать кольцевые буферы, для значительной оптимизации использования памяти. Кроме того, я научился реализовывать Z-функцию, использующуюся и в других алгоритмах поиска поиска по образцу.

Список литературы

[1] Дэн Гасфилд *Информатика и вычислительная биология* — Издательский дом «Невский диалект», 2003. Перевод с английского: И. В. Романского

[2] *Алгоритм Кнута — Морриса — Пратта — Википедия.*
URL: `http://ru.wikipedia.org/wiki/Алгоритм_Кнута_-_Морриса_-_Пратта` (дата обращения: 08.05.2025).