

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по лабораторной работе №4
по дисциплине «Построение и анализ алгоритмов»
Тема: Алгоритм Кнута-Морриса-Пратта

Студент гр. 8303

Пушпышев А.И.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020



Цель работы.

Реализовать алгоритм Кнута-Морриса-Пратта, найти индексы вхождения подстроки в строку, а также разработать алгоритм проверки двух строк на циклический сдвиг.

Вариант 1.

Подготовка к распараллеливанию: работа по поиску разделяется на k равных частей, пригодных для обработки k потоками (при этом длина образца гораздо меньше длины строки поиска).

Задание.

Реализуйте алгоритм КМП и с его помощью для заданных шаблона  и текста T  найдите все вхождения P в T .

Вход:

Первая строка – P

Вторая строка – T

Выход:

Индексы начал вхождений P в T , разделенных запятой, если P не входит в T , то вывести -1.

Пример входных данных

aba

ababa

Пример выходных данных

0, 2

Описание алгоритма КМП.

На вход алгоритма передается строка-образец, вхождения которой нужно найти, и строка-текст, в которой нужно найти вхождения.

Оптимизация – строка-текст считывается посимвольно, в памяти хранится текущий символ.

Алгоритм сначала вычисляет префикс-функцию строки-образца.

Далее посимвольно считывается строка-текст. Переменная-счетчик изначально $k = 0$. При каждом совпадении k -го символа образца и i -го символа текста счетчик увеличивается на 1. Если $k =$ размер образца, значит вхождение найдено. Если очередной символ текста не совпал с k -ым символом образца, то сдвигаем образец, причем точно знаем, что первые k

символов образца совпали с символами строки и надо сравнить $k+1$ -й символ образца (его индекс k) с i -м символом строки.

Сложность алгоритма по операциям: $O(m + n)$, m – длина образца, n – длина текста.

Сложность алгоритма по памяти: $O(m)$, m – длина образца.

Описание функций и структур данных.

```
char* read_pattern(char *filename)
```

Функция считывания из файла.

```
void* seek_substring_KMP (void *ptr)
```

Функция, реализующая префикс-функцию строки и алгоритм КМП. Принимает на вход указатель массива входных данных для потоков.

```
struct Args{};
```

Структура, содержащая информацию об исходной строке поиска, искомой строки и границах данного куска строки, в которой работает данный поток.

Разбиение исходной строки на куски происходит путем формирования подстрок (для оптимизации по памяти будем запоминать начало и конец таких строк, а не сам отрезок). Для начала определимся с количеством кусков: это результат деления всей строки, в которой проводится поиск на количество потоков, для корректировки решения в случае, если с точки зрения математики число получается не целочисленным, мы используем округление вверх. Начала кусков, будут соответствовать номеру потока N с учетом длины кусков NUM , таким образом начальные позиции кусков в исходной строке формируются как $N * NUM$ для соответствующего куска. В случае с формированием координат следует выделить два случая. Первый случай — это последний поток, поэтому координату последнего куска приравниваем к последнему символу строки, по которой производится поиск. Второй случай — предыдущие потоки. Формируется как сумма начальной позиции, длины куска на 1 поток и размер искомой строки. В таком случае мы избегаем того, что искомая строка может быть наложена на стык, она будет находиться либо в одном, либо в другом потоке.

Тестирование.

Так как гарантируется, что искомая строка много меньше строки, в которой производится поиск, поэтому тестирование проводилось на количестве потоков равному трем.

| Входные данные | Вывод |
|--------------------------------------|---|
| ababbababbbbaaabababab bab | Size of search.txt = 21 Size of str.txt = 3 Threads count = 3 Part of this thread is Part of this thread is aabbbbaaabababa ab Pos = 1 Pos = 4 Pos = 6 Part of this thread is bababa Pos = 14 Pos = 16 Work time = 0.000693 |
| qwerqwerqwerqwer werqwerqwerqwerq | Size of search.txt = 16 Size of str.txt = 16 Threads count = 3 Part of this thread is qwerqwerqwerqwerqwerq Pos = 1 Part of this thread is erqwerqwerqwerqwer Part of this thread is qwerqwerqwe Work time = 0.000675 |
| qwerqwer qwerqwer | Size of search.txt = 8 Size of str.txt = 8 Threads count = 3 Part of this thread is qwerqwerqwer Pos = 0 |

| | |
|----------------------------|--|
| | Part of this thread is werqwerqwer Part of this thread is erqwe Work time = 0.000717 |
| qwertyuiopuiopuiop uiop | Size of search.txt = 18 Size of str.txt = 4 Threads count = 3 Part of this thread is qwertyuio Part of this thread is uiopuiopu Pos = 6 Pos = 10 Part of this thread is opuio Work time = 0.000843 |

Выводы.

В ходе выполнения лабораторной работы был реализован алгоритм КМП и подготовка к распараллеливанию. Для убедительности решения код был написан с использованием POSIX threads. Программа учитывает, что искомая подстрока может находиться на стыке двух потоков, и работает корректно.

ПРИЛОЖЕНИЕ А.

Исходный код

```
#include <stdio.h>
#include <pthread.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>

struct Args
{
    bool flag_cycle;
    bool* flag_used;
    const char *pattern;//исходный поиск
    char *line;//искомое
    long int x0;//начало
    long int x1;//конец
};

char* read_pattern(char *filename)
{
    FILE *fp;
    long int lSize;
    char *buffer;
    fp = fopen ( filename , "rb" );
    if( !fp )
        perror(filename),exit(1);
    fseek( fp , 0L , SEEK_END);
    lSize = ftell( fp );
    rewind( fp );
    printf("Size of %s = %ld\n", filename, lSize);
    /* allocate memory for entire content */
    buffer = calloc( 1, lSize+1 );
    if( !buffer ) fclose(fp),fputs("memory alloc fails",stderr),exit(1);

    /* copy the file into the buffer */
    if( 1 != fread(buffer , lSize, 1 , fp) )
        fclose(fp),free(buffer),fputs("entire read fails",stderr),exit(1);
    fclose(fp);
    return buffer;
}

void* seek_substring_KMP (void *ptr)
{
    struct args * a = (struct args*)ptr;
    long int ofst = 0;
    long int M = strlen(a->line);
    long int * pi =(long int *)malloc(M * sizeof(long int)); //динамический массив длины M

    //Вычисление префикс-функции
    pi[0] = 0;
    printf("Part of this thread is ");
    for (int p = a->x0; p < a->x1; p++)
        printf("%c", a->pattern[p]);
    printf("\n");
    for(long int i = 1; i < M; i++)
    {
        while(ofst > 0 && a->line[ofst] != a->line[i])
            ofst = pi[ofst - 1];
```

```

        if(a->line[ofst] == a->line[i])
            ofst++;
        pi[i] = ofst;
    }
    //поиск
    for(long int i = a->x0, j = 0; i < a->x1; i++)
    {
        while(j > 0 && a->line[j] != a->pattern[i])
            j = pi[j - 1];

        if(a->line[j] == a->pattern[i])
            j++;
        if (j == M)
        {
            if(a->flag_cycle) {
                if(i-j+1 < M || *a->flag_used){
                    printf("Pos = %ld\n", i - j + 1);
                    *a->flag_used = true;
                }
            } else printf("Pos = %ld\n", i - j + 1);
        }
    }
    free (pi); /* освобождение памяти массива pi */
    pthread_exit(NULL);
}

int main(int argc, char** argv)
{
    /*
    if(argc!=4){
        return 0;
    }*/
    argv[1] = "search.txt";
    argv[2] = "str.txt";
    argv[3] = "4";

    char * pattern = read_pattern(argv[1]); //чтение строки
    char * line = read_pattern(argv[2]); //чтение искомого образца
    bool flag = false;

    if (strlen(pattern) == strlen(line)){
        char * buf = (char*)malloc(sizeof(char)*(strlen(pattern)*2+1));
        buf = strncat(buf, pattern, strlen(pattern) + 1);
        buf = strncat(buf, pattern, strlen(pattern) + 1);
        buf[strlen(pattern)*2 + 1] = 0;
        free(pattern);
        pattern = buf;
        flag = true;
    }

    int threads_count = atoi(argv[3]); //заданное количество потоков

    printf("Threads count = %d\n", threads_count);
    int NUM = 1 + (strlen(pattern)/(threads_count)); //количество кусков
    struct args * a = (struct args*)malloc(threads_count*sizeof(struct args)); //объект
    аргументов
    pthread_t *threads = (pthread_t*)malloc(threads_count*sizeof(pthread_t)); //массив
    потоков
    int error_code;

    bool check = false;

```

```

for(int i = 0; i < threads_count; i++)
{
    //соответствие каждого куска потоку
    a[i].flag_used = &check;
    a[i].flag_cycle = flag;
    a[i].pattern = pattern;
    a[i].line = line;
    a[i].x0 = i * NUM;
    if(i == threads_count - 1)
    {
        a[i].x1 = strlen(pattern) - 1;
    }
    else
    {
        a[i].x1 = i * NUM + strlen(line) + NUM - 1;
    }
}
clock_t t = clock();
//создание потоков
for(int i = 0 ; i < threads_count; i++)
{
    error_code = pthread_create( &threads[i], NULL, seek_substring_KMP, (void*) &a[i]);
    if(error_code)
    {
        fprintf(stderr,"Error - pthread_create() return code: %d\n",error_code);
        exit(0);
    }
    //else printf("Thread %d is created\n",i);

}
//ожидание завершения потоков
for(int i = 0; i < threads_count; i++)
{
    pthread_join(threads[i],NULL);
}
t = clock() - t;
free(a);
free(threads);
printf("Work time = %f\n",((float)t)/CLOCKS_PER_SEC);
return 0;
}

```