实验4：串匹配算法

PB16111485 张劲暾

1. 实验要求

实验内容: 字符串匹配问题, 文本串 T 的长度为 n ,对应的模式串 P 的长度为 m , 字符串均是随机生成的字符 (A-F ,共 6 种不同字符) 。

(n, m) 共取五组数据 : $(2^5, 2)$, $(2^8, 3)$, $(2^{11}, 4)$, $(2^{14}, 5)$, $(2^{17}, 6)$ 。

算法: Rabin-Karp 算法; KMP 算法; Boyer-Moore-Horspool 算法

性能分析:

a) 用适当的方法,或工具记录算法在执行时所消耗的时间;

b) 根据不同输入规模时记录的数据,画出算法在不同输入规模下的运行时间曲线图,并分析不同的串匹配算法的时间曲线。

2. 实验环境

编译环境: gcc (Ubuntu 7.2.0-8ubuntu3.2) 7.2.0

机器内存: 7.7 GiB

时钟主频: Intel Core™ i7-6500U CPU @ 2.50GHz × 4

3. 实验过程

(1) 编写简单的 Python 程序生成随机文本串和模式串:

```python
import numpy as np
import numpy.random as random

def generate_random_str(randomlength=16):
    """
    生成一个指定长度的随机字符串
    """
    random_str = ''
    base_str = 'ABCDEF'
    length = len(base_str) - 1
    for i in range(randomlength):
        random_str += base_str[random.randint(0, length)]
    return random_str

output = open("./PB16111485-project4/input/input_string.txt",'w')
text = generate_random_str(pow(2,5))
pattern = generate_random_str(2)
output.write(text + ',' + pattern + ';')
text = generate_random_str(pow(2,8))
pattern = generate_random_str(3)
output.write(text + ',' + pattern + ';')
text = generate_random_str(pow(2,11))
pattern = generate_random_str(4)
output.write(text + ',' + pattern + ';')
text = generate_random_str(pow(2,14))
pattern = generate_random_str(5)
output.write(text + ',' + pattern + ';')
text = generate_random_str(pow(2,17))
pattern = generate_random_str(6)
output.write(text + ',' + pattern + ';')

output.close()
```

(2) 编写 C 语言函数实现各种字符串匹配算法:

依次编写实现三种字符串匹配算法的核心代码和辅助函数，详见第四部分。

(3) 在 main 函数中对不同长度的随机文本串和模式串在不同的字符串匹配算法上依次测试并记录相关数据:

依次按文本串和模式串规模读取数据进行测试，测试代码如下:

```
int main(){
    FILE* InputFile = fopen("../../input/input_string.txt","r");
    FILE* OutputFile = fopen("../../output/BMH/output.txt","w");
    int text_size[GROUPS] = {(int)pow(2.0,5.0),(int)pow(2.0,8.0),(int)pow(2.0,11.0),(int)pow(2.0,14.0),(int)pow(2.0,17.0)};
    int pattern_size[GROUPS] = {2,3,4,5,6};
    int i;
    struct timeval start,end;
    for(i = 0;i < GROUPS;i++){
        char* text = (char*)malloc(text_size[i] * sizeof(char));
        char* pattern = (char*)malloc(pattern_size[i] * sizeof(char));
        fread(text,sizeof(char),text_size[i],InputFile);
        fgetc(InputFile);
        fread(pattern,sizeof(char),pattern_size[i],InputFile);
        fgetc(InputFile);
        fprintf(OutputFile,"%d %d ",text_size[i],pattern_size[i]);
        FIRSTFINDPOSITION = -1;
        printf("The %dth Search:\n",i + 1);
        gettimeofday(&start,NULL);
        BMHMatcher(text,pattern,text_size[i],pattern_size[i]);
        gettimeofday(&end,NULL);
        fprintf(OutputFile,"%d %ldus\n",FIRSTFINDPOSITION,(end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec));
        free(text);
        free(pattern);
    }
    fclose(InputFile);
    fclose(OutputFile);
}
```

文本串和模式串规模

读取文本串和模式串

字符串匹配测试

记录用时

(4) 分析数据,对比分析:

对实验得到的字符串匹配结果,检查发现基本正确,用时分析详见第五部分。

4. 实验关键代码截图（结合文字说明）

(1) Rabin-Karp 算法:

```
void RabinKarpMatcher(char* T,char* P,int n,int m,int d,int q){
    //A-F.6种不同的字符，对应6进制的0-5
    int h = ((int)pow((double)d,(double)(m-1))) % q;
    int p = 0;
    int t = 0;
    int i;
    for(i = 0;i < m;i++){
        p = (d * p + (P[i] - 'A')) % q;
        t = (d * t + (T[i] - 'A')) % q;
    }
    for(i = 0;i <= n-m;i++){
        if(p == t){
            if(memcmp(P,&T[i],m) == 0){
                printf("Found at %d\n",i);
                if(FIRSTFINDPOSITION == -1) FIRSTFINDPOSITION = i;
            }
        }
        if(i < n-m){
            t = (d*(t - (T[i] - 'A')*h) + (T[i + m] - 'A')) % q;
            if(t < 0) t += q;
        }
    }
}
```

计算模式串 Hash 值和文本串初始 Hash 值

更新文本串 Hash 值

非常重要的一步，前面算出的 t 可能是负值

(2) KMP 算法:

```
int* ComputePrefixFunction(char* P,int m){
    int* Pai = (int*)malloc((m + 1) * sizeof(int));
    Pai[1] = 0;
    int k = 0;  //匹配的模式串指针
    int q;      //被匹配的模式串指针
    for(q = 2;q <= m;q++){
        while(k > 0 && P[k+1] != P[q]){
            k = Pai[k];
        }
        if(P[k+1] == P[q]){
            k++;
        }
        Pai[q] = k;
    }
    return Pai;
}
void KMPMatcher(char* T,char* P,int n,int m){
    int* Pai = ComputePrefixFunction(P,m);
    int q = 0;  //匹配的模式串指针
    int i;      //被匹配的模式串指针
    for(i = 1;i <= n;i++){
        while(q > 0 && P[q + 1] != T[i]){
            q = Pai[q];
        }
        if(P[q + 1] == T[i]){
            q++;
        }
        if(q == m){
            printf("Found at %d\n",i - m);
            if(FIRSTFINDPOSITION == -1) FIRSTFINDPOSITION = i - m;
            q = Pai[q];
        }
    }
    free(Pai);
}
```

用自身匹配自身，从而计算得到前缀函数

一次线性扫描匹配文本串中的模式串

(3) BMH 算法:

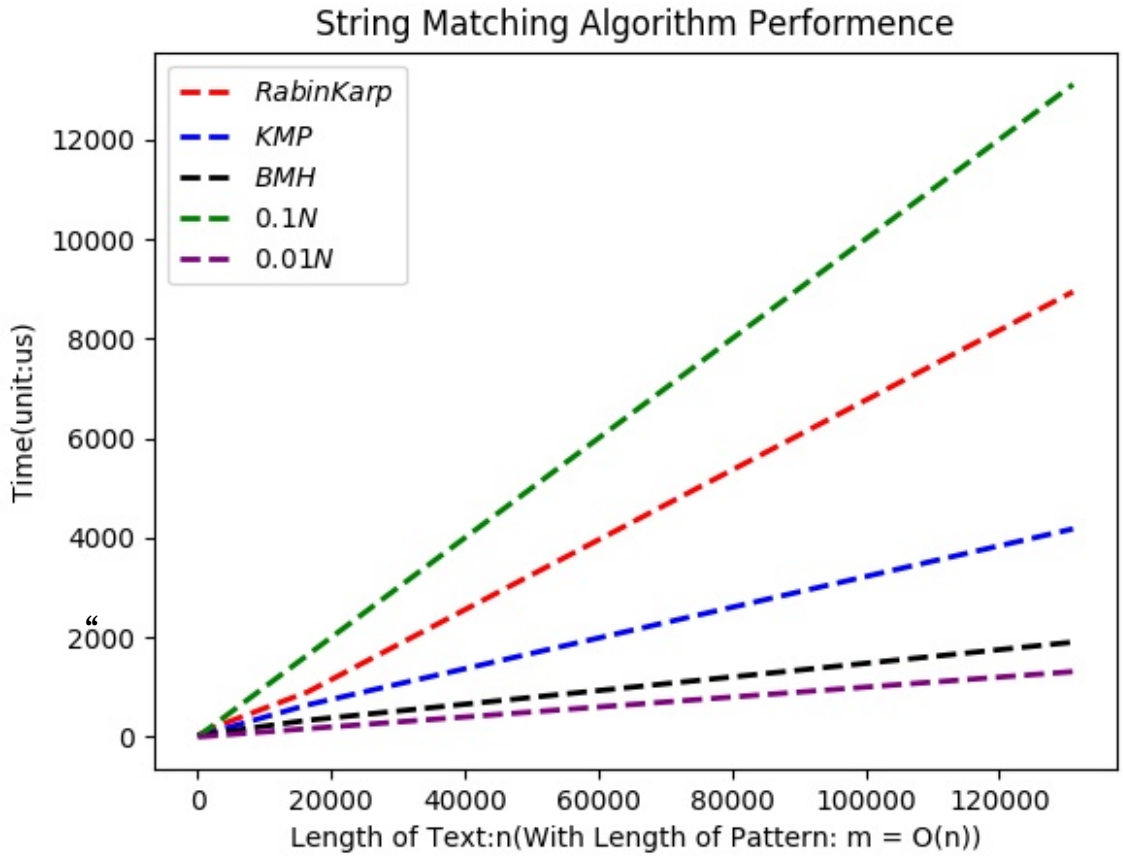
```
int* BadCharacterCompute(char* P,int m){
    int* BC = (int*)malloc(ASIZE * sizeof(int));
    int i;
    for(i = 0;i < ASIZE;i++){
        BC[i] = m;
    }
    for(i = 0;i < m-1;i++){
        BC[P[i] - 'A'] = m - 1 - i;
    }
    return BC;
}
void BMHMatcher(char* T,char* P,int n,int m){
    int* BC = BadCharacterCompute(P,m);
    int j = 0;
    while(j <= n - m){
        if(P[m - 1] == T[j + m - 1] && memcmp(P,&T[j],m) == 0){
            printf("Found at %d\n",j);
            if(FIRSTFINDPOSITION == -1) FIRSTFINDPOSITION = j;
        }
        j += BC[T[j + m - 1] - 'A'];
    }
    free(BC);
}
```

“坏字符”偏移值的计算

用“坏字符”偏移值加速文本串扫描速度

5. 实验结果、分析（结合相关数据图表分析）

匹配结果基本正确，用时分析如图：



可以看到，当 m = O(n) 或者说 m << n 时，三种字符串匹配算法的用时表现都是 O(n)数量级，而且隐含的常数系数较小，性能表现优越，其中 KMP 算法由于不需要 Hash 值计算，优于 RabinKarp 算法，而 BMH 算法相比于 KMP 算法，则用“坏字符”偏移量加速了文本串扫描速度，所以性能表现更加优越。

6. 实验心得

(1) 熟悉了三种字符串匹配算法的编程实现,提高了编程能力,加深了对于各个字符串匹配算法的特性的认识。

(2) 发现了 RabinKarp 算法中简单的文本串 Hash 值计算可能导致负值的问题，提高了对算法工程应用的了解。