

算法基础project4实验报告

PB19071535徐昊天

一.实验内容

1. KMP算法
2. Rabin-Karp算法

二.实验设备和环境

- windows操作系统, VMware虚拟机
- vscode、dev-c++
- Excel

三.实验方法和步骤

1.KMP算法

- 处理文件

本次实验须通过相对路径读取 `input` 文件夹内的文件数据, 经过算法处理后得到输出数据并通过相对路径写入 `output` 文件夹。涉及文件处理的代码如下:

```
1  FILE *fp1,*fp2,*fp3;
2  // 打开输入输出文件
3  if((fp1 = fopen("../input/4_1_input.txt","r"))==NULL)
4  {
5      printf("cannot open input file\n");
6      exit(0);
7  }
8  if((fp2 = fopen("../output/result.txt","w"))==NULL)
9  {
10     printf("cannot open result.txt\n");
11     exit(0);
12 }
13 if((fp3 = fopen("../output/time.txt","w"))==NULL)
14 {
15     printf("cannot open time.txt\n");
16     exit(0);
17 }
```

```

18 .....
19 .....
20 .....
21 //关闭输入输出文件
22 fclose(fp1);
23 fclose(fp2);
24 fclose(fp3);

```

- 前缀函数

模式的前缀函数 π 包含有模式与其自身的位移进行匹配的信息。这些信息可用于避免在朴素的字符串匹配算法中，对无用位移进行测试。模式P的前缀函数是函数满足如下公式：

$$\pi[q] = \max(k : k < q \text{ 且 } P_k \supset P_q)$$

即 $\pi[q]$ 是P[q]的真后缀P的最长前缀的长度。实现算法如下：

```

1 void compute_prefix_function()
2 {
3     int m = strlen(P) - 1;
4     int i, q;
5     for (i = 0; i < m; i++)
6         next[i] = 0;
7     int k = 0;
8     for (q = 2; q < m + 1; q++)
9     {
10         while (k > 0 && P[k + 1] != P[q])
11         {
12             k = next[k];
13         }
14         if (P[k + 1] == P[q])
15         {
16             k++;
17         }
18         next[q] = k;
19     }
20 }

```

- KMP匹配算法

调用辅助过程compute_prefix_function来计算 π ，并匹配字符串。实现算法如下：

```

1 void KMP_matcher(FILE *fp)
2 {
3     int n = strlen(T) - 1;
4     int m = strlen(P) - 1;
5     compute_prefix_function();
6     int q = 0;
7     int i, num = 0;
8     for (i = 1; i <= n; i++)
9     {
10         while (q > 0 && P[q + 1] != T[i])
11             q = next[q];
12         if (P[q + 1] == T[i])

```

```

13     q++;
14     if (q == m)
15     {
16         pos[num++] = i - m + 1;
17         q = next[q];
18     }
19 }
20 }

```

- 输出信息

根据以上KMP算法函数，利用 `num` 记录匹配次数，`pos` 数组记录所有匹配的T中开始位置，`next` 数组记录 π 的函数值，并按照要求输出到 `result.txt` 文件中。实现代码如下：

```

1     fprintf(fp, "%d\n", num);
2     for (i = 1; i < m + 1; i++)
3         fprintf(fp, "%d ", next[i]);
4     fprintf(fp, "\n");
5     for (i = 0; i < num; i++)
6         fprintf(fp, "%d ", pos[i]);
7     fprintf(fp, "\n\n");

```

- 记录算法运行时间

利用头文件 `#include<time.h>` 中 `clock()` 函数记录时间，结果以毫秒为单位，实现代码如下：

```

1     clock_t start, end;
2     double time;
3     start = clock();
4     KMP_matcher(fp2);
5     end = clock();
6     time = (double)(end - start) / (CLOCKS_PER_SEC);
7     fprintf(fp3, "%f\n", time);

```

得到时间单位为毫秒。

2.Rabin-Karp算法

- 处理文件

本次实验须通过相对路径读取 `input` 文件夹内的文件数据，经过算法处理后得到输出数据并通过相对路径写入 `output` 文件夹。涉及文件处理的代码如下：

```

1     FILE *fp1,*fp2,*fp3;
2     // 打开输入输出文件
3     if((fp1 = fopen("../input/4_2_input.txt","r"))==NULL)
4     {
5         printf("cannot open input file\n");
6         exit(0);
7     }

```

```

8   if((fp2 = fopen("../output/result.txt","w"))==NULL)
9   {
10      printf("cannot open result.txt\n");
11      exit(0);
12   }
13   if((fp3 = fopen("../output/time.txt","w"))==NULL)
14   {
15      printf("cannot open time.txt\n");
16      exit(0);
17   }
18   .....
19   .....
20   .....
21   //关闭输入输出文件
22   fclose(fp1);
23   fclose(fp2);
24   fclose(fp3);
25   return 0;

```

- 实现Rabin-Karp算法

在该算法中，所有的字符都是d进制的数字。首先需将h初始化为m数位窗口中高位数字的值，其次计算出p的值为 $P[1..m] \bmod q$ ，t的值为 $T[1..m] \bmod q$ ，最后用for循环迭代所有可能的位移i，保持着如下的循环不变量：

$$t_i = T[i + 1..i + m] \bmod q$$

实现代码如下：

```

1   void rabin_karp_matcher(int d, int q)
2   {
3       int n = strlen(T) - 1;
4       int m = strlen(P) - 1;
5       int i, j;
6       int h = 1;
7       num_r = num_f = 0;
8       for (i = 0; i < m - 1; i++)
9       {
10          h = (h * d) % q;
11      }
12      int p = 0;
13      for (i = 0; i < 4100; i++)
14          t[i] = 0;
15      for (i = 1; i <= m; i++)
16      {
17          p = ((d * p + P[i]) % q + q) % q;
18          t[0] = ((d * t[0] + T[i]) % q + q) % q;
19      }
20      for (i = 0; i <= n - m; i++)
21      {
22          if ((p - t[i]) % q == 0)
23          {
24              for (j = 1; j <= m; j++)
25              {
26                  if (P[j] != T[i + j])
27                      break;
28              }

```

```

29         if (j == m + 1)
30         {
31             pos[num_r++] = i + 1;
32         }
33         else
34         {
35             num_f++;
36         }
37     }
38     if (i < n - m)
39     {
40         t[i + 1] = (((d * (t[i] - T[i + 1] * h) + T[i + m + 1]) % q) + q) % q;
41     }
42 }
43 }
44

```

- 输出信息

根据以上Rabin-Karp算法函数，利用 `num_r` 记录匹配次数，`pos` 数组记录所有匹配的T中开始位置，`num_f` 记录伪命中次数，并按照规定输出到 `result.txt` 文件中。实现代码如下：

```

1     for (j = 0; j < 4; j++)
2     {
3         start = clock();
4         rabin_karp_matcher(d[j], q[j]);
5         end = clock();
6         timer[j][i] = (double)(end - start) / (CLOCKS_PER_SEC);
7         if (j == 0)
8             fprintf(fp2, "%d\n", num_r);
9             fprintf(fp2, "%d ", num_f);
10    }
11    fprintf(fp2, "\n");
12    for (j = 0; j < num_r; j++)
13        fprintf(fp2, "%d ", pos[j]);
14    fprintf(fp2, "\n\n");

```

- 记录算法运行时间

利用头文件 `#include <time.h>` 中 `clock()` 函数记录时间，结果以毫秒为单位，并利用二维数组 `timer` 记录不同(n,m)和(d,q)情况下的运行时间,实现代码如下：

```

1     for (j = 0; j < 4; j++)
2     {
3         start = clock();
4         rabin_karp_matcher(d[j], q[j]);
5         end = clock();
6         timer[j][i] = (double)(end - start) / (CLOCKS_PER_SEC);
7     }

```

得到时间单位为毫秒。

四.实验结果和分析

1.KMP算法

- result文件结果

执行程序后 `result.txt` 文件打印的结果如下图所示：

```
126-徐昊天-PB19071535-project4 > ex1 > output > result.txt
1 2
2 0 0 0 1 2 1 1 2
3 72 189
4
5 2
6 0 1 0 0 1 0 0 0 1 2 3 1 0 1 2 3
7 234 295
8
9 2
10 0 1 0 0 1 0 0 1 2 3 4 5 2 3 1 0 1 2 2 3 4 0 1 0 1 2 3 1 2 3 4 5
11 441 698
12
13 2
14 0 0 0 1 1 2 3 4 5 1 1 2 1 2 1 2 3 0 0 1 1 2 3 0 1 2 3 4 2 3 4 5 6 7 8 9 10 2 3 0 1 2 1 1 1 1 1 2 3 0 0 1 2 1 1 1 2 1 1 2 1 1 2 3 0
15 928 1908
16
17 2
18 0 1 2 3 0 0 1 2 0 1 2 0 1 0 1 2 3 4 4 4 5 1 2 0 1 0 0 1 2 3 0 0 0 1 2 3 4 4 5 6 7 8 9 0 1 0 1 2 3 4 4 4 5 6 7 0 0 0 1 0 0 1 0 1 0 1 0 1 2 3
19 1124 2727
20
21
```

- 时间复杂度分析

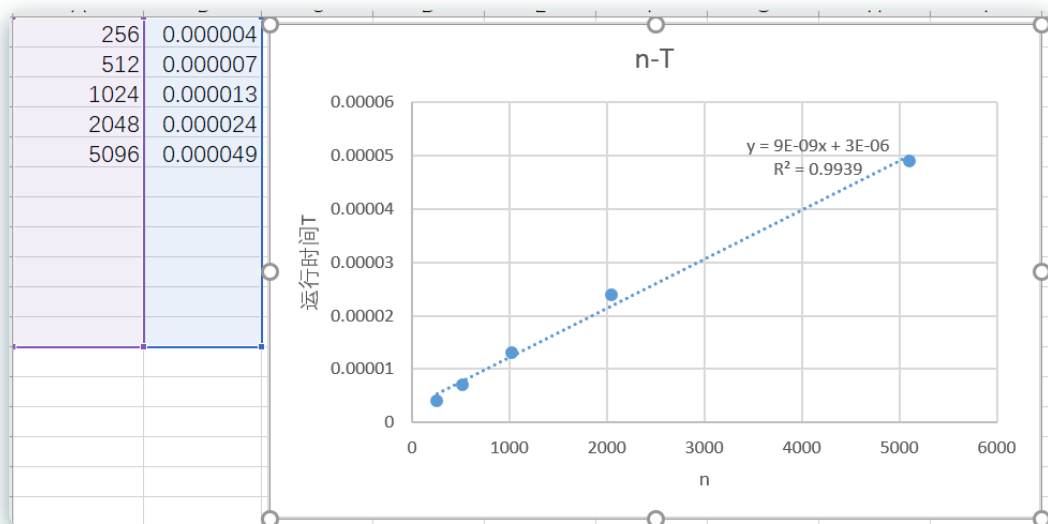
将代码放入虚拟机中通过以下指令编译运行：

```
1 | gcc main.c -o main
2 | ./main
```

得到 `time.txt` 中时间如下图所示：

```
time.txt
~/ex1/output
保存(S)
time.txt x result.txt x result.txt x time.txt x
1 0.000004
2 0.000007
3 0.000013
4 0.000024
5 0.000049
纯文本 制表符宽度：8 第 1 行，第 8 列 插入
```

KMP算法计算前缀函数的复杂度是 $O(m)$ ，实际匹配的复杂度是 $O(n)$ 。考虑到算法总复杂度是 $O(m+n)$ ， n 比 m 大得多，故以 n 为横坐标，直接画总运行时间的曲线。数据线性拟合如下图所示：



由上图可知， n 和运行时间 T 呈线性关系，符合理论时间复杂度。

2.Rabin-Karp算法

- result文件结果

执行程序后 **result.txt** 文件打印的结果如下图所示：

```
126-徐昊天-PB19071535-project4 > ex2 > output > result.txt
1 2
2 15 0 15 0
3 79 197
4
5 2
6 40 2 34 0
7 162 261
8
9 2
10 74 1 77 0
11 400 687
12
13 2
14 145 1 134 0
15 624 1788
16
17 2
18 324 3 296 1
19 1476 2609
20
21
```

- 时间复杂度分析

将代码放入虚拟机中通过以下指令编译运行：

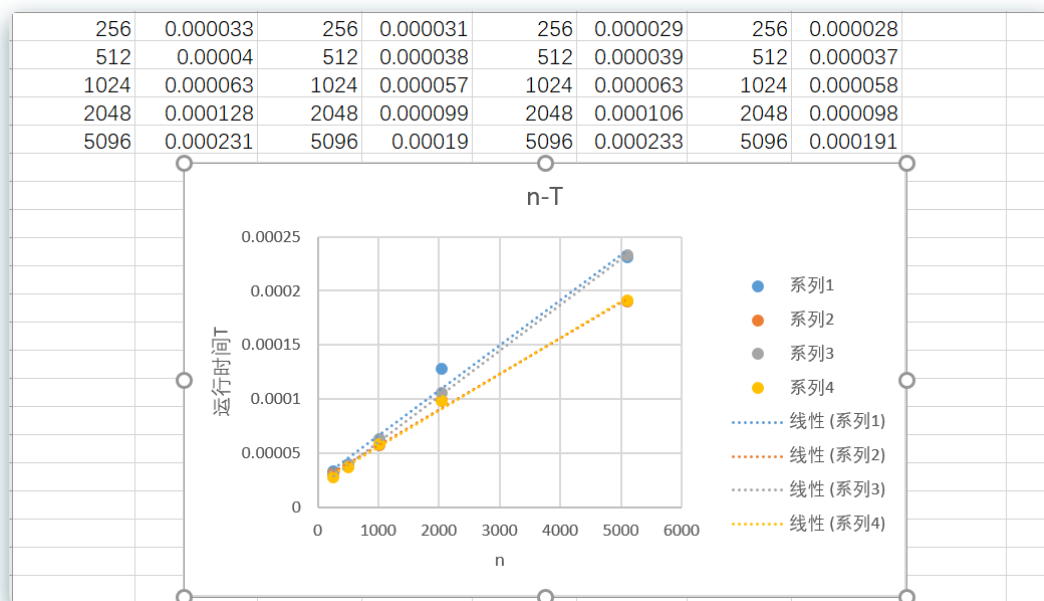
```
1 gcc main.c -o main
2 ./main
```

得到 `time.txt` 中时间如下图所示：

```
time.txt  x  result.txt  x  result.txt  x  time.txt  x
1 (2,13)
2 0.000033 0.000040 0.000063 0.000128 0.000231
3 (2,1009)
4 0.000031 0.000038 0.000057 0.000099 0.000190
5 (10,13)
6 0.000029 0.000039 0.000063 0.000106 0.000233
7 (10,1009)
8 0.000028 0.000037 0.000058 0.000098 0.000191
```

纯文本 制表符宽度: 8 第 4 行, 第 8 列 插入

RK算法最坏复杂度是 $O(mn)$ ，考虑到 n 比 m 大得多，这里可以直接以 n 作横坐标，对应不同的 (d,q) 分别拟合不同曲线。数据线性拟合如下图所示：



由上图可知，在四种 (d,q) 情况下， n 和运行时间 T 均呈线性关系，符合理论时间复杂度。

五.实验收获与感想

1. 通过实现经典算法实现串匹配算法，对以上算法有了更深刻的理解和更灵活的运用。
2. 通过处理较为复杂的字符串及其相关算法，提升了对C语言的运用和调试能力。