字符串匹配算法 实验报告

吴佳龙 2018013418

摘要

本次实验对 Brute-Force、KMP、BM 三种字符串匹配算法进行了算法描述和编程实现，验证了算法实现的结果正确性，并比较了不同算法运行的效率。另外，还实现了方便用户测试的命令行界面。

1 问题

实现并测试 Brute-Force、KMP、BM 三种字符串匹配算法。要求字母表至少包含有 26 个英文字母，0-9 数字以及若干英文标点符号。

2 实验环境

操作系统：Windows 10

IDE：Visual Studio 2019

处理器：3.1 GHz 双核 Intel Core i5

3 算法分析

本次实验共实现了 Brute-Force、KMP、BM 三种字符串匹配算法。

3.1 Brute-Force 算法

算法描述 枚举所有模式串可能出现的位置，并从该位置起逐个检查模式串与文本串是否匹配。

时间复杂度分析 设模式串 P 的长度为 m，文本串 T 的长度为 n，则模式串可能匹配的位置共有 $n-m+1$ 个，最坏情况下需要 m 次比较才能确定模式串匹配。因此复杂度为 $O(nm)$。

空间复杂度分析 除了 P 和 T，额外的辅助空间为 $O(1)$。

3.2 KMP 算法

算法描述 通过定义模式串的前缀函数（即 P_q 最长的 border 的长度）

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

当文本串的当前字符与模式串的当前字符 $P[k+1]$ 不匹配时，跳转到 $\pi[k]$ 尝试继续匹配。

算法分为计算 π 和匹配两部分。

算法的正确性与伪代码详见课件。

时间复杂度分析 课件第 41 页的伪代码计算 π 的时间复杂度为 $\Theta(m)$，这可以通过 aggregate method 分析的得出：第 9 行 k 最多增加 $O(m)$ 次，因此第 7 行中 k 也最多减少 $O(m)$ 次。或者也可以通过将伪代码中的 k 作为每个 q 对应的状态的势函数分析得出。

课件第 40 页中的匹配算法的复杂度为 $\Theta(n)$，类似上面分析第 40 页伪代码中的 q 的增加和减少即可得。

空间复杂度分析 需要 $O(m)$ 的辅助空间存储 π。

3.3 BM 算法

算法描述 BM 算法对 Brute-Force 算法进行了改进，它尝试从模式串的右端向左检验是否与文本串匹配，并分析这种检验匹配的方式的性质，设计了两种优化，分别是 Bad-character Shift 和 Good-Suffix Shift，可以跳过一些不可能的匹配位置。

BM 算法的伪代码和描述详见课件，但是课件中缺少了 COMPUTE-OSUFF 函数的算法描述和伪代码，在下面给出。

Osuff 数组计算 Osuff[i] 定义为：P_i 与 P 的最长公共后缀的长度：

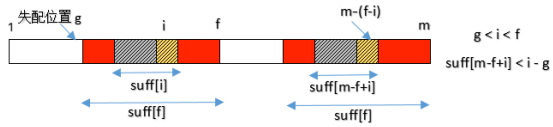
$$\max\{k : P[i-k+1..i] = P[m-k+1..m]\}$$

一种暴力的做法是，直接从 $P[i]$ 和 $P[m]$ 开始，然后从左逐个字符进行匹配，这样的复杂度是 $O(m^2)$ 的。我们需要利用之前已经得到的信息。

令 f 表示上一次进行暴力匹配时的起始位置 i，g 表示上一次暴力匹配时的失配位置，则对于当前的 i 有如下的性质：

Property 1. 若 $i > g$ 且 $Osuff[m-(f-i)] < i-g$, 则 $Osuff[i] = Osuff[m-(f-i)]$

Proof. 如下图 [1]，$P[i]$ 与 $P[m-f+i]$ 匹配，若 Osuff$[m-(f-i)] < i-g$ 则 Osuff[i] 的左端不会超过 g。 □



当以上条件不满足时，我们只需从 g 开始继续匹配即可，伪代码如下：

```
1  COMPUTE−OSUFF(P):
2      m = P.length
3      Osuff[m] = m
4      g=m, f=0
5      for i = m−1 down to 1:
6          if i>g and Osuff[m−(f−i)] < i−g:
7              Osuff[i] = Osuff[m−(f−i)]
8          else
9              if (i<g) g=i
10             f = i
11             while (g && P[g] == P[m−(i
                   −g)])
12                 g=g−1
13             Osuff[i] = i−g
```

注意到，g 的值只会下降，因此 12 行至多执行 $O(m)$ 次，该函数的复杂度是 $\Theta(m)$

时间复杂度分析 计算 Osuff, bmBc, bmGs 的复杂度分别为 $\Theta(m), \Theta(|\Sigma|), \Theta(m)$，因此总的预处理的复杂度为 $\Theta(m+|\Sigma|)$。

匹配时，最好的情况下，失配时可以跳过 m 个位置，为 $\Omega(n/m)$，最坏的情况下与暴力相同，复杂度为 $O(nm)$。

空间复杂度分析 需要 $O(m+|\Sigma|)$ 的额外空间存储 Osuff, bmBc, bmGs。

4 结果分析

4.1 结果正确性

首先我们验证了算法实现的正确性，验证方法为：随机生成指定规模的模式串与文本串，并调用不同算法，比较他们计算的模式串出现的所有位置是否一致。

实验表明，对于随机生成的数据，三种算法给出的结果总是相同的，由此可以认为我们对于算法的实现是无误的。

4.2 计算时间

4.2.1 一般情况下的测试数据

数据生成 给定 n 和 m，一般情况下的测试数据生成过程为：先随机生成 P 和一定数量的长度在 m 左右的串，合并成单词集；然后不断从单词集中随机选取，拼接在 T 的末尾，直到 T 的长度超过 n 并截断超过的部分。

这种方式较为合理地模拟了现实中字符串匹配算法的应用场景。

结果分析 选取 $m = 10$，$n \in [10^7, 10^8]$，算法的运行时间如图 1。

可以看到，运行时间 KMP > Brute-Force > BM。这是由于在这样的数据下，模式串匹配的位置不多，Brute-Force 算法在内循环能够达到 $O(m)$ 的情况很少，且 Brute-Force 算法的常数因子很小，因而快过了 KMP 算法。

而 BM 算法在这样的数据下，则能够接近 $\Omega(n/m)$ 的下界，比 Brute-Force 和 KMP 快 3-5 倍。

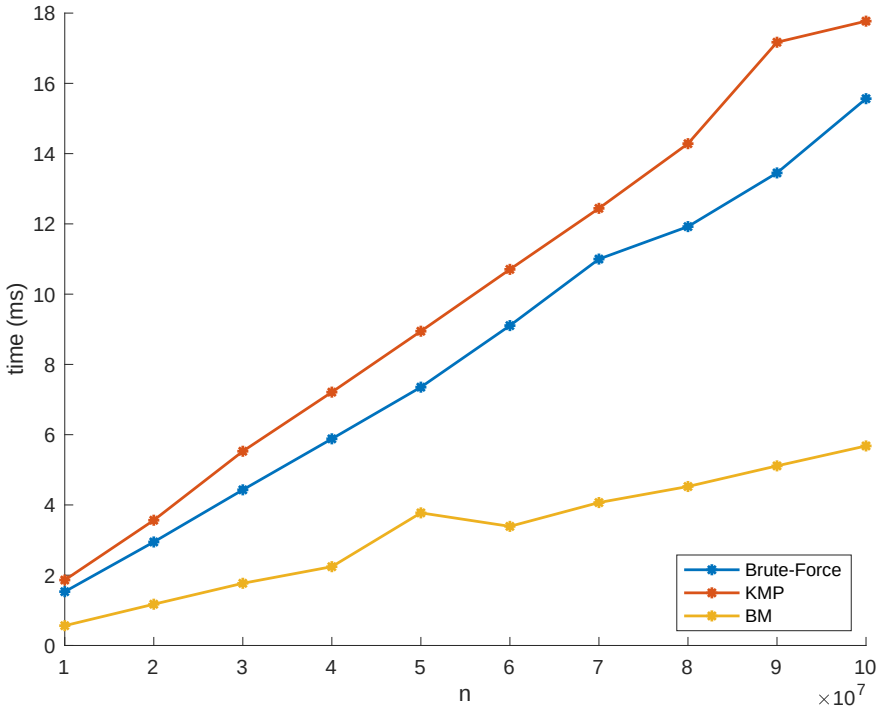


Figure 1: 一般情况下的测试数据，不同算法的运行时间

4.2.2 极端情况下的测试数据

数据生成 给定 n 和 m，极端情况下的测试数据生成为：选取字符集中的一个字符，将 P 和 T 全部填充为该字符。

结果分析 选取 $m = 10$，$n \in [10^7, 10^8]$，算法的运行时间如图 2。

可以看到，运行时间 BM > Brute-Force > KMP。在这种极端的数据下，BM 和 Brute-Force 都达到了上界 $O(nm)$，且 BM 的常数因子较大。而 KMP 算法的时间复杂度是 $\Theta(n+m)$ 的，比两者都快。

5 总结：不同算法的比较

Brute-Force 实现简单，且在一般情况下不会达到最坏的复杂度，常数因子小，一般情况下甚至能快过 KMP 算法。

KMP $\Theta(n+m)$ 的复杂度，具有理论意义。但是在实际应用场景中，可能反而不及 Brute-Force 效率高。

BM 在理想情况下，能达到 $\Omega(n/m)$ 的复杂度，且实际场景中大多接近这一种理想的情况，因而极具实用意义。但是最坏情况下仍然能达到 $O(nm)$，其变体 Turbo-BM [2] 能将复杂度降低到 $O(n)$，从而解决了这一缺点。

References

[1] Boyer-Moore 算法 _ 百度百科, https://baike.baidu.com/item/Boyer-%20Moore%E7%AE%97%E6%B3%95/16548374?fr=aladdin

[2] Turbo-BM algorithm, http://igm.univ-mlv.fr/ lecroq/string/node15.html

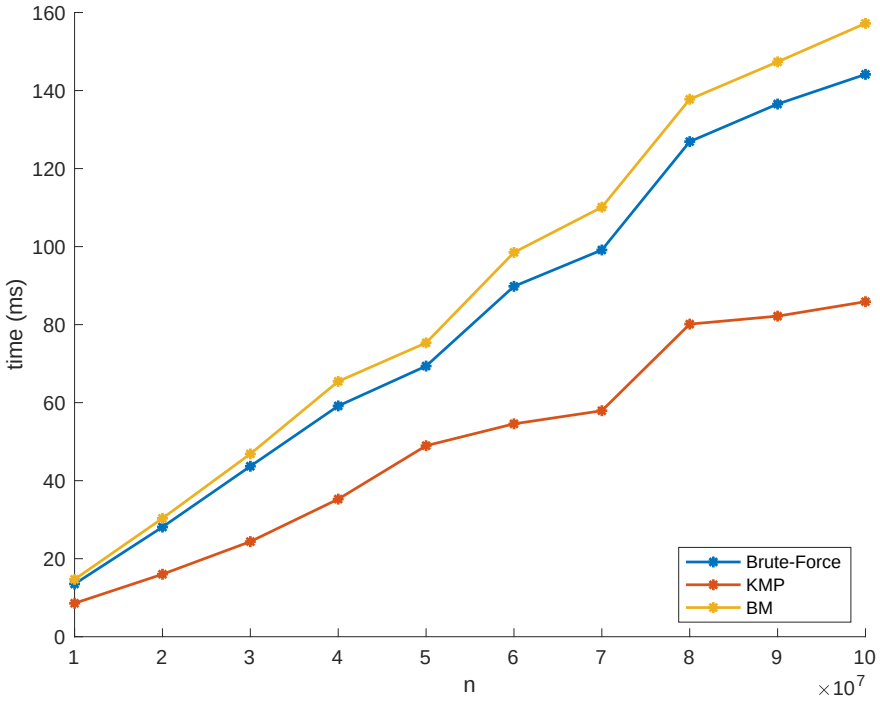


Figure 2: 极端情况下的测试数据，不同算法的运行时间