实验七 允许字符误差的 KMP 算法改进

熊恪峥 22920202204622

2021 年 12 月 17 日

目录

1 允许一个字符误差的 KMP 算法

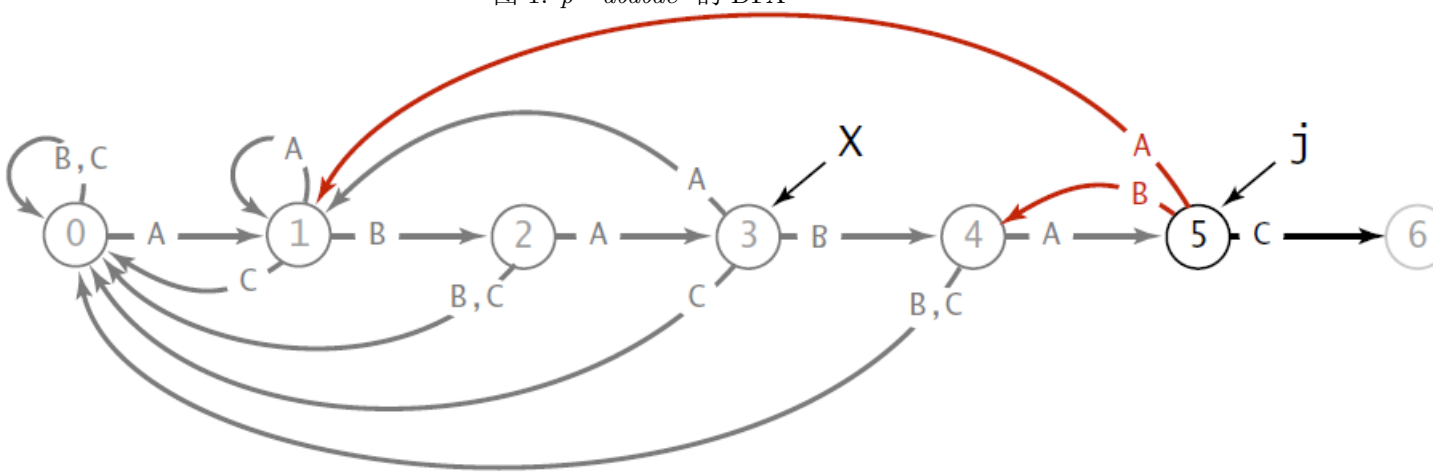
KMP 一般认为有两种可能的实现，包括使用失配函数的实现和使用确定有限状态自动机（DFA, Deterministic finite automaton）的实现，它们的时间复杂度和空间复杂度基本相同，但 DFA 的实现有较大的常数。使用 DFA 的实现的匹配过程就是模拟 DFA 的运行过程，因而相当简单，易于进行修改。因此以下的改进基于 *Algorithms, 4th Edition* 中的 DFA 实现的 KMP 算法。代码 1是该实现构建 DFA 的预处理过程。它用二维数组的方式存储 DFA。

代码 1: 构建 DFA 的预处理代码

```
1       static inline constexpr size_t PATTERN_LEN = 128;
2
3       int dfa[CHAR_MAX][PATTERN_LEN]{};
4
5       void construct(const std::string& pattern)
6       {
7               dfa[pattern.at(0)][0] = 1;
8               for (int i = 1, x = 0; i < pattern.length(); i++)
9               {
10                      for (char c = 0; c < CHAR_MAX; c++)
11                      {
12                              dfa[c][i] = dfa[c][x];
13                      }
14
15                      dfa[pattern.at(i)][i] = i + 1;
16                      x = dfa[pattern.at(i)][x];
17              }
18      }
```

例如对于模式串 *p='ababac'*，构建出的 DFA 如图 1。

图 1: *p='ababac'* 的 DFA



从 DFA 的构造中容易知道，当失配时目标的状态值小于当前状态值。在这种情况首次发生时，如果跳过一个字符能够达到终止状态（情况 1），或者能达到下一个匹配状态（情况二），就跳过一个字符，这样就能实现允许一个字符的误差。

代码 2: 允许一个错误的 KMP 匹配算法

```
1       int match(const std::string& pattern, const std::string& text, int pos)
2       {
3               int i = pos, j = 0;
4
5               bool e = false;
6               for (; i < text.length() && j < pattern.length(); i++)
7               {
```

```
8                       //情况一
9                       if (dfa[text.at(i)][j] <= j && j == pattern.length() - 1)
10                      {
11                              e = true;
12                              j++;
13                              i++;
14                              break;
15                      }
16                      //情况二
17                      else if (dfa[text.at(i)][j] <= j && i + 1 < text.length() && j + 1 < pattern.
                            length() &&
18                      text.at(i + 1) == pattern.at(j + 1) && !e)
19                      {
20                              e = true;
21                              j++;
22                      }
23                      else
24                      {
25                              j = dfa[text.at(i)][j];
26                      }
27              }
28
29              if (j == pattern.length())
30              {
31                      return i - pattern.length();
32              }
33
34              return text.length();
35      }
```

1.1 该改进算法的正确性

1. 当主串 T 中没有和模式串 p 中差一个字符匹配的子串时，该算法和使用 DFA 的 KMP 实现相同，因此是正确的。

2. 当主串 T 中存在和模式串 p 中差一个字符匹配的子串时，遇到该失配字符前该算法的行为与和使用 DFA 的 KMP 实现相同，跳过一个适配字符之后该算法的行为也和使用 DFA 的 KMP 实现相同，因此是正确的。

或者可以从 DFA 图出发论证该改进的正确性，该改进相当于在每两个状态之间添加一个转移，转移的条件是已经跳过的字符数量小于 1 个。那么原始的 DFA 是新的 DFA 的一个子图，因此它不影响正常的字符匹配。由新加的转移的条件可以知道，它允许有一个误差的输入到达终止状态。因此它的正确性是显然的。

实际测试表明该代码确实能正确完成允许一个失配字符的模式匹配功能。

1.2 改进算法的时间复杂度

该算法的预处理过程与使用 DFA 的 KMP 的预处理过程实现相同，是 $O(MR)$，其中 R 是字母表的长度，是一个常数。而匹配过程最多比普通的 KMP 多出一次比较，因此时间复杂度亦没有改变。

2 允许多个字符误差的 KMP 算法

使用相同的思路可以将该算法扩展成允许多个字符误差的匹配算法，代码 3中匹配函数的参数 e_allow 控制允许多少个字符的误差。它在发生失配时会试图检查跳过不超过 e_allow 个字符能否到达终止状态或者一个匹配状态。

代码 3: 允许指定错误数量的 KMP 匹配算法

```
1       int match(const std::string& pattern, const std::string& text, int pos, int e_allow)
2       {
3               int i = pos, j = 0;
```

```
4
5               int e = 0;
6               for (; i < text.length() && j < pattern.length(); i++)
7               {
8                       if (dfa[text.at(i)][j] <= j && e < e_allow)
9                       {
10
11                              int e_cnt = 0, ni = i, nj = j;
12                              while (ni < text.length() && nj < pattern.length() && text.at(ni) !=
                                    pattern.at(nj) && e_cnt < e_allow)
13                              {
14                                      ni++;
15                                      nj++;
16                                      e_cnt++;
17                              }
18
19                              if (ni < text.length() && nj < pattern.length() && text.at(ni) == pattern
                                    .at(nj))
20                              {
21                                      i = ni - 1;
22                                      j = nj;
23                                      e = e_cnt;
24                              }
25                              else if (nj == pattern.length())
26                              {
27                                      i = ni;
28                                      j = nj;
29                                      e = e_cnt;
30                                      break;
31                              }
32                              else
33                              {
34                                      j = dfa[text.at(i)][j];
35                              }
36                      }
37                      else
38                      {
39                              j = dfa[text.at(i)][j];
40                      }
41              }
42
43              if (j == pattern.length())
44              {
45                      return i - pattern.length();
46              }
47
48              return text.length();
49      }
```

该算法的正确性说明同上，但时间复杂度并不与普通的 KMP 实现相同，它与要允许的错误数量 E 和主串 T 中存在的符合条件的字串数量有关。