**算法分析与设计实验报告**

**第 4 次实验**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 姓名 | 邹林壮 | 学号 | 202208040412 | | 班级 | 计算机科学与技术（拔尖班）2201 |
| 时间 | 2024.11.23 | 地点 | 院楼432 | | | |
| 实验名称 | 实现题 3-11 正则表达式匹配问题 | | | | | |
| 实验目的 | 通过本实验，深入理解和掌握动态规划和回溯法的设计思想，特别是如何结合回溯法来解决正则表达式匹配问题。同时，通过实际编程练习，提高对C++语言的运用能力，以及对算法性能的分析能力。 | | | | | |
| 实验原理 | 正则表达式匹配问题是一个经典的字符串处理问题，目标是构建一个正则表达式，使其可以**匹配尽可能多的操作文件**，同时**避免匹配到任何非操作文件**。。动态规划用于构建匹配表以存储中间结果，而回溯法用于探索所有可能的正则表达式构建路径。 | | | | | |
| 实验步骤 | 1. **确定输入输出**   输入为文件名数n, n行文件名，文件名为英文字母和数字，操作文件（用+标记），非操作文件（用-标记）  输出为最多文件匹配maxmat，匹配最多操作文件且不匹配任何非操作文件的正则表达式s。  约束是使能匹配的待操作数最多，但不能匹配任何不进行操作的文件。   1. **初始化：**   n[0]表示操作文件数，n[1]表示非操作数，将操作文件存储在数组f[1]到f[n[0]]中，将非操作文件存储在数组k[1]到 k[n[1]]中。然后在save函数中遍历所有操作文件，统计每个字符的出现频率，按频率降序排列，从而加快搜索进程。初始化动态规划数组 match[len][i][j]，表示正则表达式长度为 len 时，第i个文件的第j个字符的匹配情况。   1. **判断文件是否匹配**   check函数实现，  遍历所有操作文件，对于当前正则表达式 s：  若字符为 \*，表示可匹配任意长度子串；  若字符为 ?，表示可匹配任意单个字符；  若字符为普通字符，则需要精确匹配。  若正则表达式 s 匹配了更多操作文件，更新匹配数 curmat，将当前正则表达式 s 的匹配状态记录到 match数组中   1. **检测非操作文件**   ok函数中实现，  遍历所有非操作文件，检测当前正则表达式是否匹配：若匹配到任何非操作文件，则当前正则表达式无效。回溯，跳过无效正则表达式。   1. **回溯生成最优正则表达式**   在search函数中实现，  **递归回溯搜索**：在当前正则表达式基础上添加一个字符，若添加 \* 或 ?，重新计算匹配数，按字符频率优先选择高频字符。若新生成的正则表达式更优（匹配更多文件且长度更短），更新为当前最优解。  **剪枝优化**：若当前正则表达式的匹配数小于历史最佳，提前停止搜索；若当前正则表达式长度超过当前最短长度，则跳过。   1. **输出结果**   输出maxmat，表示正则表达式匹配的文件数，输出最优正则表达式 minmat。 | | | | | |
| 关键代码 | 1. **void** save(**char** c, **int** len) { 2. **int** i, j; 3. **for**(i=1; i<=p[len]; i++) 4. **if**(cha[len][i].c == c) { 5. cha[len][i].f++; 6. cha[len][0] = cha[len][i]; 7. j = i; 8. **while**(cha[len][j-1].f < cha[len][0].f) { //将频率小于当前字符频率的字母后移 9. cha[len][j] = cha[len][j-1]; 10. j--; 11. } 12. cha[len][j] = cha[len][0];  //将当前字母放到适当位置 13. **return**; 14. } 15. //如果p[len] = 0，或者字符c第一次出现，则增加一个字符 16. cha[len][++p[len]].c = c; 17. cha[len][p[len]].f = 1; 18. } 20. //计算当前匹配情况 21. **bool** check(**int** len) { 22. **int** i, j, t, k = 0; 23. curmat = 0; 24. **for**(i=1; i<=n[0]; i++) { //遍历操作文件 25. **for**(j=0; j<MAXP; j++) 26. match[len][i][j] = 0;//初始化匹配表 27. **if**(len==1 && s[1]=='\*')//如果是\*且正则长度为1，匹配全部 28. match[len][i][0] = 1; 29. **for**(j=1; j<=f[i].length(); j++) { //操作文件的每个字符 30. **switch**(s[len]) { 31. **case** '\*'://匹配任意长度的子串 32. **for**(t=0; t<=j; t++) 33. **if**(match[len-1][i][t]==1) { 34. match[len][i][j] = 1; //正则表达式的第len个字符与第i个文件的第j个字符的匹配情况 35. **break**; 36. } 37. **break**;  //此处的break不能省略！ 38. **case** '?'://匹配单个字符 39. match[len][i][j] = match[len-1][i][j-1]; 40. **break**; 41. **default**: 42. **if**(s[len]==f[i][j-1])//精准字符匹配 43. match[len][i][j] = match[len-1][i][j-1]; 44. **break**; 45. } 46. } 47. //检查当前文件是否完全匹配 48. **for**(j=f[i].length(); j>=1; j--) { 49. **if**(match[len][i][j] == 1) { 50. k++; 51. **if**(j == f[i].length()) //说明第i个文件与正则表达式匹配 52. curmat++; 53. **break**; 54. } 55. } 56. } 57. //如果当前匹配情况比历史最佳情况差，返回false 58. **if**(k<maxmat || k==maxmat && len>=minlen) 59. **return** **false**; 60. p[len] = 0;//更新字符频率 61. **for**(i=1; i<=n[0]; i++) //对与正则表达式匹配的文件中的字符重新排序，以便正则式下次扩展 62. **for**(j=1; j<f[i].length(); j++) 63. **if**(match[len][i][j]==1) 64. save(f[i][j], len); 65. **return** **true**; 66. } 68. //判断是否匹配非操作文件 69. **bool** ok(**int** len) { 70. **int** i, j, l, t; 71. **for**(l=1; l<=len; l++) { 72. **for**(i=n[0]+1; i<=n[0]+n[1]; i++) { 73. **for**(j=0; j<MAXP; j++) 74. match[l][i][j] = 0;//初始化匹配表 75. **if**(s[1]=='\*' && l==1) 76. match[l][i][0] = 1; 77. **for**(j=1; j<=f[i].length(); j++) { 78. **switch**(s[l]) { 79. **case** '\*': 80. **for**(t=0; t<=j; t++) 81. **if**(match[l-1][i][t]==1) { 82. match[l][i][j] = 1; 83. **break**; 84. } 85. **break**; 86. **case** '?': 87. match[l][i][j] = match[l-1][i][j-1]; 88. **break**; 89. **default**: 90. **if**(s[l]==f[i][j-1]) 91. match[l][i][j] = match[l-1][i][j-1]; 92. **break**; 93. } 94. } 95. } 96. } 97. //如果正则表达式匹配到非操作文件，返回false 98. **for**(i=n[0]+1; i<=n[0]+n[1]; i++)  //如果正则表达式与非操作文件匹配 99. **if**(match[len][i][f[i].length()]==1) 100. **return** **false**; 101. **return** **true**; 102. } 104. //求最优匹配的回溯法 105. //len 为当前正则表达式长度 106. //正则表达式可选字符集的排列顺序先为‘\*’，‘？’，操作文件名序列中出现的所有字符按其频率递减的次序随后 107. **void** search(**int** len) { 108. // 如果当前正则表达式更优(匹配更多文件名/有更短表达式），更新最优解 109. **if**((curmat>maxmat || (curmat==maxmat && len<minlen)) && ok(len)) { 110. maxmat = curmat; 111. minlen = len; 112. **for**(**int** i=0; i<=minlen; i++) 113. minmat[i] = s[i]; 114. } 115. len++; 116. //尝试添加?和\* 117. **if**(len==1 || s[len-1]!='\*') { 118. s[len] = '?'; 119. **if**(check(len)) 120. search(len); 121. s[len] = '\*'; 122. **if**(check(len)) 123. search(len); 124. } 125. // 尝试添加其他字符 126. **for**(**int** i=1; i<=p[len-1]; i++) { 127. s[len] = cha[len-1][i].c; 128. **if**(check(len)) 129. search(len); 130. } 131. } | | | | | |
| 测试结果 | **(1)正确性：**  测试了所给样例，在洛谷上测试，处理最后一个大样例超时，其他都成功运行，保证了正确性。    **(2)复杂度：**  时间复杂度：  **1.读取文件名和初始化**：这部分的时间复杂度是 O(M)，其中 M 是文件名的总数。这是因为代码需要逐个读取每个文件名。  **2.匹配检查 (check 函数)**：对于每个正则表达式长度len，需要检查所有文件名，每个文件名的长度最多为MAXL。因此，这部分的时间复杂度是，其中 N 是文件名的数量，L是文件名的最大长度，P是可能的字符集大小（在这个程序中是 MAXP）。  **3.回溯搜索 (search 函数)：**对于每个正则表达式长度 len，程序尝试添加三种类型的字符：'?'，'\*' 和其他字符。对于每种类型的字符，它都会递归地调用 check 函数。最坏情况下，search 函数的递归深度可以达到MAXL（文件名的最大长度），因此这部分的时间复杂度是，其中来自于对于每个位置，有三种选择（'?' 或 '\*' 或其他字符）。  综合以上，整体时间复杂度大约是。  空间复杂度：  **1.文件名存储：**M是文件名的数量，L是文件名的最大长度，存储文件名的空间复杂度是.  **2.匹配表：**match 数组是一个三维数组，其大小为。  **3.字符频率表：**cha 数组的大小为。  **4.正则表达式存储**：s和minmat数组的大小为 O(P)  整体空间复杂度大约是  (3)构造了不同规模数据集并进行图像化演示：  构造了不同规模和大小的数据集，  数据大小为10    数据大小为100    数据大小为250    分析数据可知，满足时间复杂度 O(M + 3^L \* N \* L \* P)的大致趋势。  这是生成数据的代码   |  | | --- | | import random  import string  def generate\_data\_with\_signs(n, filename):      # 随机生成长度不超过 8 的字符串      def random\_string():          length = random.randint(1, 8)  # 字符串长度在 1 到 8 之间          return ''.join(random.choices(string.ascii\_uppercase, k=length))      # 生成数据      data = [f"{n}"]  # 第一行是数据规模      for \_ in range(n):          string\_part = random\_string()          sign = random.choice(["+", "-"])          data.append(f"{string\_part} {sign}")      # 保存到文件      with open(filename, "w") as file:          file.write("\n".join(data))        print(f"数据已保存到 {filename}")  # 示例：生成 5 行数据并保存到 data.txt  generate\_data\_with\_signs(250, "./lab4/data.txt") | | | | | | |
| 实验心得 | 1. 针对问题较为复杂的问题，要首先明确问题的解决方向和解决框架，不能盲目地做，应该仔细地想结局问题的方法。 2. 回溯法在解决某些问题时的局限性，尤其是在面对具有高复杂度的算法时。在这个问题中，回溯法导致了较高的时间复杂度，这可能在处理大规模数据时成为瓶颈。 3. 认识到了问题规模对算法选择的影响。对于小规模问题，即使复杂度高的算法也能快速解决问题；但对于大规模问题，需要选择更高效的算法。 | | | | | |
| 实验得分 |  | 助教签名 | |  | | |

**附录：完整代码**

|  |
| --- |
| **#include "iostream"**  **#include "string"**  **#include "iomanip"**  **#include "fstream"**  **using namespace std;**  **/\***  **正则表达式可选字符集的排列顺序先为‘\*’，‘？’，操作文件名序列中出现的所有字符按其频率递减的次序随后**  **大体思路：**  **正则表达式为s，当前考察文件为f**  **match(i, j)为s[1,i]与f[1,j]匹配情况**  **如果match(i-1, j-1) = 1, s[i] = '?'**  **match(i-1, j-1) = 1, s[i] = f[j]**  **match(i-1, k) = 1, s[i] = '\*'**  **match(i, j) = 1**  **否则 match(i, j) = 0**  **\*/**  **const int MAXN = 250; //最大文件数**  **const int MAXL = 8; //最大文件名长度**  **const int MAXP = 62; //大写字母 + 小写字母 + 数字**  **int minlen; //最优正则表达式长度**  **int maxmat; //最优正则表达式所能匹配的操作文件数**  **int curmat; //当前正则式所能匹配的操作文件数**  **string f[MAXN+1]; //操作文件 + 非操作文件**  **string k[MAXN+1]; //非操作文件**  **int n[2]; //n[0]为操作文件数，n[1]为非操作文件数**  **int p[MAXP]; //p[len-1] 存储 s[len]可选字符数**  **char s[MAXP]; //存储临时正则表达式**  **char minmat[MAXP]; //最优正则表达式**  **int match[MAXP][MAXN+1][MAXP]; //match[len][i][j]表示正则表达式的第len个字符与第i个文件的第j个字符的匹配情况**  **struct Cha {**  **char c; //字符**  **int f; //字符出现的频率**  **};**  **Cha cha[MAXN][MAXP]; //cha[len][i]表示正则表达式长度为len时的字符排序情况**  **//对操作文件名中出现的字符按出现频率排序存储，以加快搜索进程**  **void save(char c, int len) {**  **int i, j;**  **for(i=1; i<=p[len]; i++)**  **if(cha[len][i].c == c) {**  **cha[len][i].f++;**  **cha[len][0] = cha[len][i];**  **j = i;**  **while(cha[len][j-1].f < cha[len][0].f) { //将频率小于当前字符频率的字母后移**  **cha[len][j] = cha[len][j-1];**  **j--;**  **}**  **cha[len][j] = cha[len][0]; //将当前字母放到适当位置**  **return;**  **}**  **//如果p[len] = 0，或者字符c第一次出现，则增加一个字符**  **cha[len][++p[len]].c = c;**  **cha[len][p[len]].f = 1;**  **}**  **//计算当前匹配情况**  **bool check(int len) {**  **int i, j, t, k = 0;**  **curmat = 0;**  **for(i=1; i<=n[0]; i++) { //遍历操作文件**  **for(j=0; j<MAXP; j++)**  **match[len][i][j] = 0;//初始化匹配表**  **if(len==1 && s[1]=='\*')//如果是\*且正则长度为1，匹配全部**  **match[len][i][0] = 1;**  **for(j=1; j<=f[i].length(); j++) { //操作文件的每个字符**  **switch(s[len]) {**  **case '\*'://匹配任意长度的子串**  **for(t=0; t<=j; t++)**  **if(match[len-1][i][t]==1) {**  **match[len][i][j] = 1; //正则表达式的第len个字符与第i个文件的第j个字符的匹配情况**  **break;**  **}**  **break; //此处的break不能省略！**  **case '?'://匹配单个字符**  **match[len][i][j] = match[len-1][i][j-1];**  **break;**  **default:**  **if(s[len]==f[i][j-1])//精准字符匹配**  **match[len][i][j] = match[len-1][i][j-1];**  **break;**  **}**  **}**  **//检查当前文件是否完全匹配**  **for(j=f[i].length(); j>=1; j--) {**  **if(match[len][i][j] == 1) {**  **k++;**  **if(j == f[i].length()) //说明第i个文件与正则表达式匹配**  **curmat++;**  **break;**  **}**  **}**  **}**  **//如果当前匹配情况比历史最佳情况查，返回false**  **if(k<maxmat || k==maxmat && len>=minlen)**  **return false;**  **p[len] = 0;//更新字符频率**  **for(i=1; i<=n[0]; i++) //对与正则表达式匹配的文件中的字符重新排序，以便正则式下次扩展**  **for(j=1; j<f[i].length(); j++)**  **if(match[len][i][j]==1)**  **save(f[i][j], len);**  **return true;**  **}**  **//判断是否匹配非操作文件**  **bool ok(int len) {**  **int i, j, l, t;**  **for(l=1; l<=len; l++) {**  **for(i=n[0]+1; i<=n[0]+n[1]; i++) {**  **for(j=0; j<MAXP; j++)**  **match[l][i][j] = 0;//初始化匹配表**  **if(s[1]=='\*' && l==1)**  **match[l][i][0] = 1;**  **for(j=1; j<=f[i].length(); j++) {**  **switch(s[l]) {**  **case '\*':**  **for(t=0; t<=j; t++)**  **if(match[l-1][i][t]==1) {**  **match[l][i][j] = 1;**  **break;**  **}**  **break;**  **case '?':**  **match[l][i][j] = match[l-1][i][j-1];**  **break;**  **default:**  **if(s[l]==f[i][j-1])**  **match[l][i][j] = match[l-1][i][j-1];**  **break;**  **}**  **}**  **}**  **}**  **//如果正则表达式匹配到非操作文件，返回false**  **for(i=n[0]+1; i<=n[0]+n[1]; i++) //如果正则表达式与非操作文件匹配**  **if(match[len][i][f[i].length()]==1)**  **return false;**  **return true;**  **}**  **//求最优匹配的回溯法**  **//len 为当前正则表达式长度**  **//正则表达式可选字符集的排列顺序先为‘\*’，‘？’，操作文件名序列中出现的所有字符按其频率递减的次序随后**  **void search(int len) {**  **// 如果当前正则表达式更优(匹配更多文件名/有更短表达式），更新最优解**  **if((curmat>maxmat || (curmat==maxmat && len<minlen)) && ok(len)) {**  **maxmat = curmat;**  **minlen = len;**  **for(int i=0; i<=minlen; i++)**  **minmat[i] = s[i];**  **}**  **len++;**  **//尝试添加?和\***  **if(len==1 || s[len-1]!='\*') {**  **s[len] = '?';**  **if(check(len))**  **search(len);**  **s[len] = '\*';**  **if(check(len))**  **search(len);**  **}**  **// 尝试添加其他字符**  **for(int i=1; i<=p[len-1]; i++) {**  **s[len] = cha[len-1][i].c;**  **if(check(len))**  **search(len);**  **}**  **}**  **int main() {**  **ifstream fin("zz.txt");**  **cout << "输入文件：\n";**  **n[0] = 0;**  **n[1] = 0;**  **p[0] = 0;**  **string str;**  **char ch;**  **while(!fin.eof()) {//读入**  **fin >> str>>ch;**    **if(ch == '+') {**  **f[++n[0]] = str;**  **cout << f[n[0]] << " " << ch << endl;**  **save(f[n[0]][0], 0);**  **} else {**  **k[++n[1]] = str;**  **cout << k[n[1]] << " " << ch << endl;**  **}**  **}**  **for(int i=1; i<=n[1]; i++)**  **f[n[0]+i] = k[i];**  **memset(match, 0, sizeof(match));**  **for(int i=1; i<=n[0]+n[1]; i++)**  **match[0][i][0] = 1;**  **maxmat = 0;**  **minlen = 255;**  **search(0);**  **cout << "匹配最多文件数：" << maxmat << endl;**  **cout << "正则表达式为：";**  **for(int i=0; i<=minlen; i++)**  **cout << minmat[i];**  **cout << endl;**  **fin.close();**  **return 0;**  **}** |