

专业课程实验报告

课程名称： 算法分析与设计

开课学期： 2020 至 2021 学年 第 一 学期

专业： 软件工程 年级班级： 1班

学生姓名： 宋行健 学号： 222018321062006

实验教师： 曹严元

计算机与信息科学学院 软件学院

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 实验项目名称 | | 动态规划——矩阵连乘 | | | |
| 实验时间 | | 2020年10月27日 | 实验类型 | | □验证性 设计性 □综合性 |
| 一、实验目的   1. 掌握动态规划的基本思想方法； 2. 了解适用于用动态规划方法求解的问题类型，并能设计相应动态规划算法； 3. 掌握动态规划算法复杂性分析方法。   二、实验要求   1. 预习实验指导书及教材的有关内容，掌握动态规划的基本思想； 2. 严格按照实验内容进行实验，培养良好的算法设计和编程的习惯； 3. 认真听讲，服从安排，独立思考并完成实验。   三、实验原理  将待求解问题分解为若干子问题，通过子问题的解得到原问题的解，这是问题求解的有效途径。但是如何实施分解呢？分治策略的基本思想是将规模为的问题分解规模较小的子问题，各子问题相互独立但与原问题求解策略相同。但并不是所有问题都可以这样处理。  问题分解的另一个途径是将求解过程分解为若干阶段（级），一次求解每个阶段即得到原问题的解。通过分解得到的各子阶段不要求相互独立，但希望它们具有相同类型，而且前一阶段的输出可以作为下一阶段的输入。这种策略特别适合求解具有某种最优性质的问题。贪心法属于这类策略：对问题P，其求解过程中各贪心选择步骤构成决策序列， 的最优性仅仅依赖于。贪心法不保证最后求出解的最优性。  动态规划法也是一个分阶段判定决策过程，其问题求解策略的基础是决策过程的最优原理：为达到某问题的最优目标，需要一次作出决策序列。如果是最优的，则对任意，决策子序列也是最优的，即当前决策的最优性取决于其后续决策序列是否最优。由此追溯至目标，再由最终目标决策向上回溯，导出决策序列。因此动态规划方法可以保证问题求解是全局最优的。 | | | | | |
| 三、实验内容与设计（主要内容，操作步骤、算法描述或程序代码）  **矩阵连乘（动态规划解法）**   1. 数据结构：   在本次实验中，选用二维数组的数据结构进行问题的表示。二维数组自然表示矩阵。在函数调用时，将二维数组以指针的形式进行传递，对于矩阵的运算直接通过地址对元素进行取值和运算。因此这种数据结构的表达方式十分直观清晰，但是也有一些不足，例如二维数组所占的内存空间较大，浪费空间资源。   1. 矩阵连乘问题的伪码算法：      1. 矩阵连乘问题C++源代码： 2. #include <iostream> 3. #include <iomanip> 4. #include <malloc.h> 5. #include <string.h> 6. **using** **namespace** std; 8. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 9. \* 函数描述： 构造矩阵（n阶方阵） 10. \* 函数参数： n——矩阵行数 11. m——矩阵列数 12. \* 函数返回： 构造的矩阵指针 13. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 14. **int** \*\*GenerateMatrix(**int** n, **int** m) 15. { 16. **int** \*\*s = (**int** \*\*)malloc(**sizeof**(**int** \*) \* n); 17. **for** (**int** i = 0; i < n; ++i) 18. { 19. s[i] = (**int** \*)malloc(**sizeof**(**int**) \* m); 20. memset(s[i], 0, **sizeof**(**int**) \* m); 21. } 22. **return** s; 23. } 25. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 26. \* 函数描述： 释放内存（n阶方阵） 27. \* 函数参数： p——需要被释放内存的矩阵 28. n——阶数 29. \* 函数返回： void 30. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 31. **void** FreeMatrix(**int** \*\*p, **int** n) 32. { 33. **for** (**int** i = 0; i < n; ++i) 34. { 35. free(p[i]); 36. } 37. free(p); 38. } 40. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 41. \* 函数描述： 打印矩阵（不打印0索引） 42. \* 函数参数： p——被打印矩阵 43. n——行数 44. m——列数 45. \* 函数返回： void 46. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 47. **void** PrintMatrix(**int** \*\*p, **int** n, **int** m) 48. { 49. **for** (**int** i = 1; i < n; ++i) 50. { 51. **for** (**int** j = 1; j < m; ++j) 52. { 53. cout << setw(7) << p[i][j]; 54. } 55. cout << endl; 56. } 57. cout << endl; 58. } 60. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 61. \* 函数描述： 矩阵乘法 a\*b=c 62. \* 函数参数： a——矩阵a 63. b——矩阵b 64. c——结果矩阵c 65. ra——矩阵a的行数 66. ca——矩阵a的列数 67. rb——矩阵b的行数 68. ca——矩阵b的列数 69. \* 函数返回： void 70. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 71. **void** MatrixMultiply(**int** \*\*a, **int** \*\*b, **int** \*\*c, **int** ra, **int** ca, **int** rb, **int** cb) 72. { 73. **if** (ca != rb) // 如果不满足矩阵乘法规则（左列数=右行数），则返回 74. { 75. cout << "Matrix multiplication error." << endl; 76. **return**; 77. } 78. // 矩阵乘法 79. **for** (**int** i = 0; i < ra; ++i) 80. { 81. **for** (**int** j = 0; j < cb; ++j) 82. { 83. **int** tmp = 0; 84. **for** (**int** k = 0; k < ca; ++k) 85. { 86. tmp += a[i][k] \* b[k][j]; 87. } 88. c[i][j] = tmp; 89. } 90. } 91. } 93. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 94. \* 函数描述： 矩阵连乘核心算法 95. \* 函数参数： p——连乘矩阵行列值数组 96. n——连乘矩阵的个数 97. m——储存分步最优值数组 98. s——储存分步最优解，最优值断开的位置 99. \* 函数返回： 最优值 100. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 101. **int** MatrixChain(**int** \*p, **int** n, **int** \*\*m, **int** \*\*s) 102. { 103. **int** i, j; 104. //对角线上的元素都设置成0 105. **for** (i = 1; i <= n; ++i) 106. m[i][i] = 0; 107. **for** (**int** len = 2; len <= n; ++len) 108. { // 遍历连乘长度，从2开始，子问题规模 109. **for** (i = 1; i <= n - len + 1; ++i) 110. { 111. j = i + len - 1; 112. // 将链ij划分为A(i) \* ( A[i+1:j] ) 113. m[i][j] = 0 + m[i + 1][j] + p[i - 1] \* p[i] \* p[j]; 114. s[i][j] = i; 115. **for** (**int** k = i + 1; k < j; ++k) 116. { // 将链ij划分为( A[i:k] ) \* ( A[k+1:j] ) 117. **int** t = m[i][k] + m[k + 1][j] + p[i - 1] \* p[k] \* p[j]; 118. **if** (m[i][j] > t) 119. { 120. m[i][j] = t; 121. s[i][j] = k; 122. } 123. } 124. } 125. PrintMatrix(m, n + 1, n + 1); 126. } 127. **return** m[1][n]; // 返回最优值 128. } 130. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 131. \* 函数描述： 递归求最优解 132. \* 函数参数： s——最优解断开位置矩阵 133. i——连乘矩阵链起始位置 134. j——连乘矩阵链终止位置 135. \* 函数返回： void 136. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 137. **void** Traceback(**int** \*\*s, **int** i, **int** j) 138. { 139. **if** (i == j) 140. **return**; 141. Traceback(s, i, s[i][j]); 142. Traceback(s, s[i][j] + 1, j); 143. cout << "Matrix A" << i << "," << s[i][j] << " and A" << s[i][j] + 1 << "," << j << endl; 144. } 146. **int** main() 147. { 148. // 连乘的矩阵个数 149. **const** **int** n = 6; 150. // n个矩阵连乘，行列顺序。30\*35 35\*15 15\*5 5\*10 10\*20 20\*25 151. **int** p[n + 1] = {30, 35, 15, 5, 10, 20, 25}; 152. **int** \*\*m = GenerateMatrix(n + 1, n + 1); 153. **int** \*\*s = GenerateMatrix(n + 1, n + 1); 154. **int** min = MatrixChain(p, n, m, s); 155. cout << "Min: " << min << endl; 156. PrintMatrix(m, n + 1, n + 1); // 打印储存最优值的矩阵 157. PrintMatrix(s, n + 1, n + 1); // 打印储存最优解的矩阵 158. Traceback(s, 1, n);           // 打印最优解 159. **return** 0; 160. } | | | | | |
| 1. 分析时间复杂度   矩阵连乘算法的时间复杂度主要取决于对连乘长度（len）、子问题起始位置（i）、子问题分割点（k）这三重循环。循环体内的运算的时间复杂度为，而三重循环的时间复杂度为。因此使用动态规划解决矩阵连乘问题的时间复杂度为。  在本算法中，以二维数组为基本的数据结构，因此空间复杂度为。 | | | | | |
| 三、测试数据和执行结果 （在给定数据下，执行操作、算法和程序的结果，可使用数据、图表、截图等给出）  图 1展示的了行列顺序为 (30\*35) (35\*15) (15\*5) (5\*10) (10\*20) (20\*25) 的六个矩阵的连乘结果。由图可知，最小的运算次数为15125次。由最优解的记录矩阵可以看出首先在3号矩阵处断开，分成1~3和4~6两部分。1~3部分在1处断开，4~6部分在5处断开。    图 1 矩阵连乘 | | | | | |
| 四、实验结果分析及总结（对实验的结果是否达到预期进行分析，总结实验的收获和存在的问题等）  通过本次实验，我对动态规划的思想有了更深入、更系统的了解。同时，我选择了用C++语言来实现这些排序算法，对C++中指针的传递、二维数组的传递方式及其指针表达方式、内存分配等代码实现也进行了复习。  矩阵连乘算法的关键在于理解最优值矩阵和最优解矩阵的产生和分析。产生主要是通过连乘长度（len）、子问题起始位置（i）、子问题分割点（k）这三重循环的遍历。每次进行比较后将局部最优的结果储存，同时将子问题的分割方法储存。在分析时，逆向倒退，自顶向下，通过分割点的记录得出最优解。  本次实验用到的动态规划的方法与上次实验的分治法十分类似，二者都要求原问题具有最优子结构性质，都是将原问题分而治之，分解成若干个规模较小很容易解决的子问题，之后将子问题的解合并，形成原问题的解。但是动态规划是将将分解后的子问题理解为相互间有联系，有重叠的部分，需要记忆储存，通常用迭代来做。而分治法是将分解后的子问题看成相互独立的子问题，通常使用递归来解决。 | | | | | |
| 教  师  评  阅 | 实验内容和设计（A-E）： | | |  | |
| 操作过程、算法或代码（A-E）： | | |  | |
| 实验结果（A-E）： | | |  | |
| 实验分析和总结（A-E）： | | |  | |
| 实验成绩（A-E）：  反馈评语： | | | | |