

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 | | 贪心算法——二元归并树 | | | |
| 实验时间 | | 2020年11月10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. 了解适用于用贪心法求解的问题类型，并能设计相应贪心法算法； 3. 掌握贪心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  有一类问题是要从所有的允许解中求出最优解，其策略之一是“贪心法”，即逐次实施“贪心选择”：在每个选择步骤上做出的选择都是当前状态下最优的。贪心选择依赖于在此之前所做出的选择，但不依赖于后续步骤所需要的选择，即不依赖于后续待求解子问题。显然，这种选择方法是局部最优的，但不是从问题求解的整体考虑进行选择，因此不能保证最后所得一定是最优解。贪心法是求解问题的一种有效方法，所得到的结果如果不是最优的，通常也是近似最优的。 | | | | | |
| 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）  **二元归并树贪心算法**   1. 数据结构：   在本次实验中，选用vector和二叉树的结构对问题进行表示。  C++语言中的STL里面的vector是一种连续的空间存储，可以使用“[]”操作符快速的访问随机的元素，快速的在末尾插入元素，但是在序列中间的插入，删除元素要慢，其vector的性质类似于一个数组。这里选用vector而不是链表的原因是，在算法中需要多次查找最小的两个数，需要频繁进行随机访问元素。  二叉树的数据结构是自定义的，为了简单方便，此处并未用到链表的指针形式来构造二叉树，而是直接用一个struct结构存储了每个节点的内容和其父节点、子节点分别在上述vector中的索引值。这里类似于一个哈希散列映射。    图 1 储存内容、父节点、子节点的struct结构   1. 二元归并树贪心算法的伪码算法：      1. 二元归并树贪心算法C++源代码： 2. #include <iostream> 3. #include <iomanip> 4. #include <vector> 5. #include <algorithm> 6. #define INT\_MAX 0x7fffffff 7. **using** **namespace** std;  10. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 11. \* 函数描述： 打印 vector 12. \* 函数参数： arr——向量 13. \* 函数返回： void 14. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 15. **void** printVector(vector<**int**>& arr) 16. { 17. **for** (vector<**int**>::iterator it = arr.begin(); it != arr.end(); it++) 18. { 19. cout << setw(5) << \*it << '\t'; 20. } 21. cout << "\n" << endl; 22. }   26. **class** MergeTree 27. { 28. **private**: 29. **struct** MergeNode 30. { 31. **int** weight; // 节点权重，记录数 32. **int** parent; 33. **int** lchild, rchild; 34. }; 36. **public**: 37. // 构造函数 38. MergeTree(vector<**int**> arr) 39. { 40. **int** n = arr.size(); // 求得当前 arr 的节点个数 41. count = 2 \* n - 1;  // 二叉树的总节点数; 42. MoveTotal = 0;      // 移动总量初始化为0 43. Node = **new** MergeNode[count]; 44. // 使用传入的向量构造叶子节点 45. **for** (**int** i = 0; i < size(arr); ++i) 46. { 47. LeafNodes.push\_back(arr[i]); 48. } 49. } 51. MergeNode\* Node; 52. **int** count;             // 总节点数 53. **int** MoveTotal;         // 移动总量 54. vector<**int**> LeafNodes; // 节点数据 55. // 核心算法：构造最优二元归并树 56. **void** Tree(); 57. // 从 LeafNodes 选取两个最小数 58. **void** selectMin(**int**& s1, **int**& s2); 59. // 打印树 60. **void** printTree(); 61. // 标记已经归并的节点，把 LeafNodes 的 i 位置置位为正无穷 62. **void** labelNode(vector<**int**>& arr, **int** i) { arr[i] = INT\_MAX; } 63. };  66. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 67. \* 函数描述： 选择 vector 中两个最小的数，并将索引号储存在 s1 和 s2 中 68. \* 函数参数： s1, s2——储存当前的两个最小的数的内存地址 69. \* 函数返回： void 70. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 71. **void** MergeTree::selectMin(**int**& s1, **int**& s2) 72. { 73. **int** Min1 = INT\_MAX, Min2 = INT\_MAX; 74. **for** (**int** i = 0; i < LeafNodes.size(); ++i) 75. { 76. **if** (LeafNodes[i] < Min1) 77. { 78. **if** (Min1 < Min2) 79. { 80. Min2 = Min1; 81. s2 = s1; 82. } 83. Min1 = LeafNodes[i]; 84. s1 = i; 85. } 86. **else** **if** (LeafNodes[i] < Min2) 87. { 88. Min2 = LeafNodes[i]; 89. s2 = i; 90. } 91. **else** 92. **continue**; 93. } 94. } 96. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 97. \* 函数描述： 构造二元归并的最优解法树 98. \* 函数参数： null 99. \* 函数返回： void 100. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 101. **void** MergeTree::Tree() 102. { 103. sort(LeafNodes.begin(), LeafNodes.end()); // 将 arr 升序排列 104. **int** n = LeafNodes.size();                 // 求得当前 arr 的节点个数 105. //初始化，所有节点没有父节点和左右子节点 106. **for** (**int** i = 0; i < count; ++i) 107. { 108. Node[i].parent = -1; 109. Node[i].lchild = -1; 110. Node[i].rchild = -1; 111. } 112. **for** (**int** i = n; i < count; ++i) 113. { 114. **int** i1, i2; 115. selectMin(i1, i2); 116. Node[i1].weight = LeafNodes[i1]; 117. Node[i2].weight = LeafNodes[i2]; 118. Node[i1].parent = i; 119. Node[i2].parent = i; 120. **int** mid = Node[i1].weight + Node[i2].weight; 121. Node[i].lchild = i1; 122. Node[i].rchild = i2; 123. Node[i].weight = mid; 124. // 把 i1 和 i2 位置置位为正无穷 125. labelNode(LeafNodes, i1); 126. labelNode(LeafNodes, i2); 127. // 插入新的节点 128. LeafNodes.insert(LeafNodes.begin() + i, mid); 129. // 累加移动总量 130. MoveTotal += mid; 131. } 132. } 134. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 135. \* 函数描述： 打印最优解法树 136. \* 函数参数： null 137. \* 函数返回： void 138. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 139. **void** MergeTree::printTree() 140. { 141. cout << "移动总量为：" << MoveTotal << endl; 142. **for** (**int** i = 0; i < count; ++i) 143. { 144. cout << setw(7) << "节点：" << Node[i].weight << "\t"; 145. cout << setw(7) << "父节点：" << Node[Node[i].parent].weight << "\t"; 146. cout << setw(7) << "左：" << Node[Node[i].lchild].weight << "\t"; 147. cout << setw(7) << "右：" << Node[Node[i].rchild].weight << endl; 148. } 149. }  152. **int** main() 153. { 154. vector<**int**> arr; 155. **int** test[] = { 15, 30, 10, 8, 40, 13 }; 156. **for** (**int** i = 0; i < size(test); ++i) 157. { 158. arr.push\_back(test[i]); 159. } 161. cout << "需要进行二元归并的数据：" << endl; 162. printVector(arr); 164. // 使用 vector 数据实例化一个归并树 165. MergeTree mt(arr); 166. // 进行二元最优归并 167. mt.Tree(); 168. // 打印结果 169. mt.printTree(); 171. **return** 0; 172. } | | | | | |
| 1. 分析时间复杂度   二元归并树的贪心算法的时间复杂度主要取决于vector中最小值的寻找和生成归并树时对每一个节点的遍历。个归并文件所生成的二叉归并树有个节点，需要步归并操作来进行循环，同时也会有步查找最小值的操作。循环体内的运算的时间复杂度为，查找最小值的操作的复杂度为，因此上述两重循环的时间复杂度为。综上，使用贪心算法解决二元归并树问题的时间复杂度为。  在本算法中，以vector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 2展示了需要归并的文件长度为{ 15, 30, 10, 8, 40, 13 }的6个文件所产生的二元归并树结果。由图可知，最小的移动总量次数为278次。    图 2 二元归并树结果  图 3展示了将上述运行结果还原成二叉树示意图。其中使用方块表示已知节点，所有的已知节点都在叶子节点的位置。使用圆圈表示生产的新节点，即为内部节点。所有的内部节点之和为移动总量278。    图 3 归并树示意图 | | | | | |
| 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，我对贪心算法有了更深入的了解。同时，我选择了用C++语言来实现二元归并树算法，对C++中STL的运用和数据集结构中二叉树的知识也进行了复习。  二元归并树算法，其贪心重点体现在每次选取的归并文件都是最小的文件。直观地看，先选择权值小的，所以权值小的结点被放置在树的较深层（叶子节点），而权值较大的离根较近，这样一来，在计算树的带权路径长度时，自然会具有最小的带权路径长度。  本次实验时，我也查阅了相关资料，了解到这种最优二叉归并树有一个更加学术的名字，叫做哈夫曼树（Huffman），它的定义是由个带权叶子结点构成的所有二叉树中带权路径长度最小的二叉树。其定义便是二元归并树的最优解。这种数据结构在现实生活中十分常用，例如哈夫曼编码、数据的解压缩等等。  在本实验中，我遇到的最大问题就是数据结构的选择问题，经过比较，我最终选择了使用STL中现成的vector结构来储存数据，并使用一个类似哈希的散列映射来记录二叉树的信息。Vector的优点是其容量是动态的，方便插入新的节点。在构建二叉树时，之前《数据结构》课上讲的是使用链表来构建，但是其代码量很大，而且该问题只是记录二叉树的信息，并不需要一个很完整的二叉树链表去实现其增删改查的功能，所以只是简单的用一个struct记录了其父节点和子节点在vector中的索引信息。  关于元归并树我也查了一些资料，其思路和二元的类似，都是将数据排序后寻找最小的个元素进行归并。但是，有一个细节要注意的就是，在最后一次选取的时候可能元素个数不满足个。为了解决这个问题，在算法的开始，我们就先取掉几个数，使得剩下的数正好能每次取个取完。 | | | | | |
| 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： | | | | |