

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 | | 分支—限界算法（旅行商问题） | | | |
| 实验时间 | | 2021年1月5日 | 实验类型 | | □验证性 设计性 □综合性 |
| 一、实验目的   1. 掌握分支—限界的基本思想方法； 2. 了解适用于用分支—限界方法求解的问题类型，并能设计相应动态规划算法； 3. 掌握分支—限界算法复杂性分析方法，分析问题复杂性。   二、实验要求   1. 预习实验指导书及教材的有关内容，掌握分支—限界的基本思想； 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  分支—限界算法类似于回溯法，也是一种在问题的解空间树上搜索问题解的算法。但两者求解方法有两点不同：第一，回溯法只通过约束条件剪去非可行解，而分支—限界法不仅通过约束条件，而且通过目标函数的限界来减少无效搜索，也就是剪掉了某些不包含最优解的可行解；第二，在解空间树上，回溯法以深度优先搜索，而分支—限界法则以广度优先或最小耗费优先的方式搜索。分支—限界的搜索策略是，在扩展节点处，首先生成其所有的儿子结点（分支），然后再从当前的活结点表中选择下一个扩展结点。为了有效地选择下一扩展结点，以加速搜索进程，在每一活结点处，计算一个函数值（限界），并根据这些已计算出的函数值从当前活结点表中选择一个最有利的结点做为扩展，使搜索朝着解空间树上最优解的分支推进，以便尽快找出一个最优解。  分支—限界法常以广度优先或以最小耗费优先的方式搜索问题的解空间树（问题的解空间树是表示问题皆空间的一颗有序树，常见的有子集树和排序树）。在搜索问题的解空间树时，分支—限界法的每一个活结点只有一次机会成为扩展结点。活结点一旦成为扩展结点，就一次性产生其所有儿子结点。在这些儿子结点中，那些导致不可行解或非最优解的儿子结点将被舍弃，其余儿子结点被加入活结点表中。此后，从活结点表取出下一结点成为当前扩展结点，并重复上述扩展过程，直到找到最优解或活结点表为空时停止。 | | | | | |
| 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）  **旅行商（货郎担）问题分支限界算法**   1. 数据结构：   在本次实验中，选用树的结构对问题进行表示，旅行商问题的解空间树为一棵排列树。树的构造是通过迭代节点（MinHeapNode类），并使用MinHeapNode类中的变量s进行标记解空间树的层数，使用x数组储存当前解。  在算法中使用了STL的优先级队列priority\_queue来储存可行节点的最小堆，堆中每个结点的lcost值是优先队列的优先级，这里重载了“<”操作符，使得以MinHeapNode类的优先级队列可以自动排序。    图 1 最小堆的活节点类   1. 旅行商问题分支限界算法的伪码算法：      1. 解空间树   如下图所示为旅行商问题的解空间树，其中节点的标号（A~J）是按照节点生成顺序进行标注的。最优解使用红线标注。    图 2 解空间树   1. 旅行商问题分支限界算法C++源代码： 2. #include <iostream> 3. #include <vector> 4. #include <cstring> 5. #include <iomanip> 6. #include <queue> 7. #define NoEdge -1 8. #define NN 50 // 可执行的最大的顶点个数 9. **using** **namespace** std; 10. **int** n;                 // 图的顶点个数 11. **int** adjMatrix[NN][NN]; // 图的邻接矩阵 12. **int** v[50];             // 最优解 13. **int** bestC;             // 最优值 14. **int** num = 0;           // 节点序号 16. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 17. \* 函数描述： 数据输入以及内存的初始化 18. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 19. **void** input() 20. { 21. cin >> n; // 输入顶点个数 22. **int** k; 23. memset(adjMatrix, NoEdge, **sizeof**(adjMatrix)); // 邻接矩阵的内存初始化 24. cin >> k;                                     // 输入边的个数; 25. **int** p, q, len; 26. // 初始化邻接矩阵 27. **for** (**int** i = 1; i <= k; ++i) 28. { 29. cin >> p >> q >> len; 30. adjMatrix[p][q] = len; 31. adjMatrix[q][p] = len; 32. } 33. } 35. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 36. \* 函数描述： 格式化打印结果 37. \* 参数描述： res，最优值 38. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 39. **void** printTravel(**int** res) 40. { 41. **if** (res == NoEdge) 42. cout << "======= 无法形成回路 =======" << endl; 43. **else** 44. { 45. cout << "\n======================================\n最短路径为：" << res << endl; 46. **for** (**int** i = 1; i <= n; i++) 47. cout << v[i] << " ---> "; 48. cout << v[1]; 49. } 50. } 52. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 53. \* 类描述：最小堆（队列中元素类型） 54. \* 参数描述： 55. x，用于记录当前解； 56. s，表示节点在排列树中的层次，从排列树的根节点到该节点的路径为x[0:s]， 57. 需要进一步搜索的顶点是x[s+1:n-1]。 58. cc，表示当前费用， 59. lcost，是子树费用的下界， 60. rcost，是x[x:n-1]中顶点最小出边费用和。 61. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 62. **class** MinHeapNode 63. { 64. **public**: 65. **char** name; // 节点的序号 66. **int** rcost, // x[s:n-1]中顶点最小出边费用和 67. lcost, // 子树费用的下界 68. cc;    // 当前费用 69. **int** s,     // 根节点到当前节点的路径为x[0:s] 70. \*x;    // 需要进一步搜索的顶点是x[s+1:n-1] 72. // 构造节点并递增序号 73. MinHeapNode() 74. { 75. num += 1; 76. name = num + 'A'; 77. } 79. // 最小堆中使用下界排序 80. **bool** operator<(**const** MinHeapNode &MH) **const** 81. { 82. **return** lcost > MH.lcost; 83. } 85. // 打印节点信息 86. **void** printNode(priority\_queue<MinHeapNode> pq) 87. { 88. cout << "============== Node: " << name << " ==============" << endl; 89. cout << "最小出边和(rcost)：" << rcost << "\t子树费用的下界(lcost)：" << lcost << "\t当前费用(cc)：" << cc << "  \t节点所在层(s)：" << s << endl; 90. cout << "当前解是(x)："; 91. **for** (**int** i = 0; i < n - 1; ++i) 92. cout << x[i] << "-"; 93. cout << x[n - 1] << endl; 94. // 输出优先级队列 95. **if** (!pq.empty()) 96. { 97. cout << "-- 当前优先队列："; 98. **for** (**int** i = 0; i < pq.size(); ++i) 99. { 100. cout << pq.top().name << "(" << pq.top().lcost << ")-"; 101. pq.pop(); 102. } 103. cout << pq.top().name << "(" << pq.top().lcost << ")" << endl; 104. } 105. **else** 106. { 107. cout << "(优先级队列为空)" << endl; 108. } 109. cout << "-- 当前最优值（bestC）：" << bestC << endl; 110. } 111. }; 113. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 114. \* 算法描述：核心算法 115. 算法开始时创建一个最小堆，表示活节点优先队列。堆中每个节点的lcost 116. 值是优先队列的优先级。接着计算出图中每个顶点的最小费用出边并用Minout记录。 117. 如果所给的有向图中某个顶点没有出边，则该图不可能有回路，算法即告结束。 118. 如果每个顶点都有出边，则根据计算出的Minout作算法初始化。 119. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 120. **int** BBTSP() 121. { 122. priority\_queue<MinHeapNode> pq; // 优先级队列 123. MinHeapNode E;                  // 最小堆节点 125. **int** cc, rcost, MinSum, \*MinOut, b; 126. **int** i, j; 128. MinSum = 0;              // 最小出边费用和 129. MinOut = **new** **int**[n + 1]; // 计算 MinOut[i] = 顶点i的最小出边费用 130. **for** (i = 1; i <= n; i++) 131. { 132. MinOut[i] = NoEdge; // 所有的出边初始化为无连接 133. // 遍历找出 MinOut[i] = 顶点i的最小出边费用 134. **for** (j = 1; j <= n; j++) 135. **if** (adjMatrix[i][j] != NoEdge && (adjMatrix[i][j] < MinOut[i] || MinOut[i] == NoEdge)) 136. MinOut[i] = adjMatrix[i][j]; 137. // 不存在与这个顶点相连接的边 138. **if** (MinOut[i] == NoEdge) 139. **return** NoEdge; 140. MinSum += MinOut[i]; 141. } 143. // 初始化最小堆 144. E.s = 0;          // 根节点到当前节点的路径为x[0:s] 145. E.cc = 0;         // 当前费用为0 146. E.rcost = MinSum; // x[s:n-1]中顶点最小出边费用和 147. E.x = **new** **int**[n]; // 需要进一步搜索的顶点是x[s+1:n-1] 148. // 初始化为顺序搜索 149. **for** (i = 0; i < n; i++) 150. E.x[i] = i + 1; 151. bestC = NoEdge; // 初始化最优值为 NoEdge 153. E.printNode(pq); 154. //搜索排列空间树 155. **while** (E.s < n - 1) //非叶节点 156. { 157. **if** (E.s == n - 2) // 当前扩展节点是叶节点的父节点，判断构成的回路是否最优 158. { 159. **if** (adjMatrix[E.x[n - 2]][E.x[n - 1]] != NoEdge && adjMatrix[E.x[n - 1]][1] != NoEdge && 160. (E.cc + adjMatrix[E.x[n - 2]][E.x[n - 1]] + adjMatrix[E.x[n - 1]][1] < bestC || bestC == NoEdge)) 161. { // 如果更优，则更新费用更小的路 162. cout << "\n||||||||||||||||||||| 到达叶子节点的父节点 ———— 并更新最优解 |||||||||||||||||||||" 163. << endl; 164. E.printNode(pq); 165. bestC = E.cc + adjMatrix[E.x[n - 2]][E.x[n - 1]] + adjMatrix[E.x[n - 1]][1]; 166. E.cc = bestC; 167. E.lcost = bestC; 168. E.s++; 169. pq.push(E); 170. } 171. **else** 172. { 173. cout << "\n||||||||||||||||||||||| 到达叶子节点的父节点 ———— 不更新 |||||||||||||||||||||||" 174. << endl; 175. E.printNode(pq); 176. **delete**[] E.x; // 舍弃需要进一步搜索的节点 177. } 178. } 179. **else** // 产生当前扩展节点儿子节点 180. { 181. cout << "\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 开始一个新节点扩展 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n" 182. << endl; 183. **for** (i = E.s + 1; i < n; i++) 184. { // 广度优先搜索，进行子节点的扩展 185. MinHeapNode N; 186. **if** (adjMatrix[E.x[E.s]][E.x[i]] != NoEdge)   // E.x[E.s] 是当前要扩展的父节点，E.x[i] 是被遍历的子节点 187. {                                            // 可行儿子节点 188. cc = E.cc + adjMatrix[E.x[E.s]][E.x[i]]; // 当前费用 = 之前费用 + 新增费用 189. rcost = E.rcost - MinOut[E.x[E.s]];      // 更新最小出边费用和 190. b = cc + rcost;                          // 下界（限界函数） 191. **if** (b < bestC || bestC == NoEdge)        // 子树可能含最优解 节点插入最小堆 192. { 193. N.s = E.s + 1; // 进入下一层 194. N.cc = cc; 195. N.lcost = b; 196. N.rcost = rcost; 197. N.x = **new** **int**[n]; 198. **for** (j = 0; j < n; j++) 199. N.x[j] = E.x[j]; 200. // 获得新的路径【换位】 201. N.x[E.s + 1] = E.x[i]; 202. N.x[i] = E.x[E.s + 1]; 203. pq.push(N); // 加入优先队列 204. N.printNode(pq); 205. } 206. } 207. } 208. **delete**[] E.x; //完成节点扩展 209. } 210. **if** (pq.empty()) // 堆已空 211. **break**; 212. E = pq.top(); // 取下一扩展节点 213. pq.pop(); 214. } 216. **if** (bestC == NoEdge) // 无回路 217. **return** NoEdge; 218. **for** (i = 0; i < n; i++) // 将最优解复制到v[1:n] 219. v[i + 1] = E.x[i]; 220. **while** (pq.size()) // 释放最小堆中所有节点 221. { 222. E = pq.top(); 223. pq.pop(); 224. **delete**[] E.x; 225. } 226. **return** bestC; 227. } 229. **int** main() 230. { 231. input(); 232. **int** res = BBTSP(); 233. printTravel(res); 234. } 235. /\* 236. 4 237. 6 238. 1 2 30 239. 1 3 6 240. 1 4 4 241. 2 3 5 242. 2 4 10 243. 3 4 20 244. \*/ | | | | | |
| 1. 分析时间复杂度   旅行售货员问题如果不进行剪枝的情况下，具有种排列方式，是一棵n阶排列数，因此时间复杂度为。使用约束函数和限界函数对解空间树进行剪枝，可以省去大量不必要的计算，经过剪枝后的算法复杂度为。  在解决本问题是选用的是二维数组来储存图的邻接矩阵，因此空间复杂度为。   1. 与动态规划进行比较   分支限界法是从出发城市开始累加，是一种自顶向下的计算方式；而动态规划法是从目标城市往回计算走过的路程，是一种自底向上的方法。  动态规划其实也是一种穷举的过程，它需要将整个解空间树遍历后才能等到答案，但是分支限界法可以通过限界函数的剪枝大大减少计算次数，效率更高。 | | | | | |
| 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）  输入的第一行是节点数，第二行是边数k，之后的k行是边的信息，前两个数是边的两端的顶点，第三个数表示这两个节点之间的距离。图 3中给出了一个四个节点的测试用例示意图，从输出可以看出最短的哈密顿回路为25，最优解为1→3→2→4→1。    图 3 用例示意图  对于分支限界法解决旅行商问题，我认为重要的不是结果，而是剪枝和跳层的过程，因此我在输出时，把每一个节点的信息都输出在命令行中，如下图所示，其中节点的命名序号与图 2 解空间树的节点序号一致，根据构造节点是顺序进行命名排列。从命令行的输出中可以清楚地看出活节点的进堆和出堆的情况，还可以看出解空间树的构建层次。          图 4 旅行商问题分支限界过程 | | | | | |
| 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，我对分支限界算法有了更深入的了解。同时，我选择了用C++语言来实现旅行商（货郎担）算法，对C++的STL中的优先级队列priority\_queue进行了复习。  分支限界算法，类似于回溯法，但是回溯法是深度优先搜索，且回溯法是一种无目的性的搜索，只是通过约束函数和限界函数进行剪枝，从而简化操作。但是分支限界法是一种广度优先搜索的算法，在本次实验中又使用了优先队列的方式，通过优先级的排序，使得本算法是一种有目的性的，保证每一步都是尽可能接近最优解的算法。  对于优先级队列的分支限界法，我认为最难理解的是算法在解空间树上跳层的过程和活节点进堆和出堆的过程。因此，我在活节点的类中增加了展示当前信息的函数printNode()：    通过这个函数可以实时了解到当前节点的最小出边和(rcost）、子树费用的下界(lcost)、当前费用(cc)、节点所在层(s)、当前解数组(x)、当前优先队列的情况和当前全局最优解的情况。输出的结果展示在图 4。 | | | | | |
| 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： | | | | |