

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 | | 回溯算法——n皇后问题和旅行售货员问题 | | | |
| 实验时间 | | 2020年12月8日 | 实验类型 | | □验证性 设计性 □综合性 |
| 一、实验目的   1. 掌握回溯法的基本思想方法； 2. 了解适用于用回溯法求解的问题类型，并能设计相应回溯法算法； 3. 掌握回溯法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，掌握回溯法的基本思想； 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  回溯法是最常用的解题方法，有“通用的解题法”之称。当要解决的问题有若干可行解时，则可以在包含问题所有解的空间树中，按深度优先的策略，从根节点出发搜索解空间树。算法搜索至解空间树的任一结点时，总是先判断该结点是否肯定不包含问题的解。如果肯定不包含，则跳过对以该结点为根的子树的搜索，继续查找该结点的兄弟结点，若它的兄弟结点都不包含问题的解，则返回其父结点——这个步骤称为回溯。否则进入一个可能包含解的子树，继续按深度优先的策略进行搜索。这种以深度优先的方式搜索问题的解的算法称为回溯法。它本质上是一种穷举法，但由于在搜索过程中不断略过某些显然不合适的子树，所以搜索的空间大大少于一般的穷举，故它适用于解一些组合数较大的问题。  回溯法也可以形式化地描述如下：假设能够用*n*元组表示一个给定问题*P*的解，其中。如果*n*元组的子组 满足一定的约束条件，则称为部分解。如果它已经是满足约束条件的部分解，则添加形成新的子组，并检查它是否满足约束条件，若仍满足则继续添加，并以此类推。如果所有的都不满足约束条件，那么去掉，回溯到的位置，并去掉当前的，另选一个，组成新的子组，并判断其是否满足约束条件。如此反复下去，直到得到解或者证明无解为止。 | | | | | |
| 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）  **n皇后问题回溯算法**   1. 数据结构   在本次实验中，选用二维数组对问题进行表示，来模拟放置皇后的棋盘。在运算求解的时候，是使用深度优先搜索的解空间树对问题进行递归回溯求解。   1. 回溯临界条件   回溯的临界条件是当前的位置不符合放皇后的规则（该行、该列、斜角线已经存在皇后），则进行回溯。回溯的终止条件是，到达了解空间树的叶子节点，即将最后一个皇后放置完毕。   1. 解空间树      1. n皇后问题回溯算法的伪码算法      1. n皇后问题回溯算法C++源代码 2. #include <iostream> 3. #include <iomanip> 4. #define N 8     // N 为皇后个数 5. **using** **namespace** std; 6. **int** num = 0;  9. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 10. \* 函数描述： 输出棋盘 11. \* 参数描述： Chess[N][N]——棋盘数组 12. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 13. **void** printChessboard(**int** Chess[N][N]) 14. { 15. **for** (**int** i = 0; i < N; i++) 16. { 17. **for** (**int** j = 0; j < N; j++) 18. { 19. **if** (Chess[i][j] == 1) 20. cout << setw(2) << 'q'; 21. **else** 22. cout << setw(2) << '\_'; 23. } 24. cout << endl; 25. } 26. } 28. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 29. \* 函数描述： 判定是否满足放皇后的规则 30. \* 参数描述： Chess[N][N]——棋盘数组 31. \*           r——行数 32. \*           c——列数 33. \* 返回值：布尔类型，是否满足放皇后的规则 34. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 35. **bool** isValid(**int** Chess[N][N], **int** r, **int** c) 36. { 37. // 判断一列上有无皇后 38. **for** (**int** i = 0; i < N; i++) 39. { 40. **if** (i == r) 41. **continue**; 42. **else** **if** (Chess[i][c] == 1) 43. **return** **false**; 44. } 46. //判断左对角 47. **int** j = c - 1; 48. **for** (**int** i = r - 1; i >= 0 && j >= 0; i--) 49. { 50. **if** (Chess[i][j] == 1) 51. **return** **false**; 52. j--; 53. } 54. j = c + 1; 55. **for** (**int** i = r + 1; i < N && j < N; i++) 56. { 57. **if** (Chess[i][j] == 1) 58. **return** **false**; 59. j++; 60. } 62. //判断右对角 63. j = c + 1; 64. **for** (**int** i = r - 1; i >= 0 && j < N; i--) 65. { 66. **if** (Chess[i][j] == 1) 67. **return** **false**; 68. j++; 69. } 70. j = c - 1; 71. **for** (**int** i = r + 1; i < N && j >= 0; i++) 72. { 73. **if** (Chess[i][j] == 1) 74. **return** **false**; 75. j--; 76. } 77. **return** **true**; 78. } 80. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 81. \* 函数描述： 回溯递归函数，进行深度优先搜索 82. \* 参数描述： Chess[N][N]——棋盘数组 83. \*           n——递归到第n层 84. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 85. **void** backTrack(**int** Chess[N][N], **int** n) 86. { 87. // 如果第(N+1)行就结束，表示N个皇后已经安放完毕 88. **if** (n == N) 89. { 90. cout << ++num << ":" << endl; 91. printChessboard(Chess); 92. cout << "--------------------------" << endl; 93. **return**; 94. } 95. **else** 96. { 97. **for** (**int** j = 0; j < N; j++) 98. { 99. **if** (isValid(Chess, n, j)) 100. { 101. Chess[n][j] = 1;          //放皇后 102. backTrack(Chess, n + 1); //递归 103. Chess[n][j] = 0;          //回溯，把皇后拿掉 104. } 105. } 106. } 107. } 109. **int** main() 110. { 111. **int** a[N][N]{0}; 112. backTrack(a, 0); // 回溯 113. cout << "\nTotal:" << num << endl; 114. **return** 0; 115. } 116. 分析时间复杂度   n皇后问题如果不进行剪枝的情况下，具有种排列方式，它的解空间树是一棵n叉子集树，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。经过剪枝后的算法时间复杂度为，空间复杂度为。 | | | | | |
| 1. 测试数据和执行结果   下图展示了当的运算结果，因为八皇后问题一共有92种解，输出过长，因此在这里我只对开头和结尾进行了截图展示。图中“q”代表放置皇后的位置，“\_”代表其它位置。    …… | | | | | |
| **旅行售货员问题回溯算法**   1. 数据结构   在本次实验中，我选用二维数组对问题进行表示，二维数组存储了图的邻接矩阵。在运算求解的时候，是使用深度优先搜索的解空间树对问题进行递归回溯求解，其最优解的储存为一个一维数组。   1. 回溯临界条件   回溯的临界条件是当前的点无法继续访问其它未访问的点，则进行回溯，如果该点是最后一个点，则它无法与第一个点连接，则进行回溯。回溯的终止条件是，到达了解空间树的叶子节点，即形成了一个哈密顿回路。   1. 解空间树      1. 旅行售货员问题回溯算法的伪码算法      1. 旅行售货员问题回溯算法C++源代码 2. #include <iostream> 3. #include <cstring> 4. #include <math.h> 5. #define NoEdge -1 6. #define N 200      // 可执行的最大的顶点个数 7. **using** **namespace** std; 8. **int** n;                   // 图的顶点个数 9. **int** adjMatrix[N][N];     // 图的邻接矩阵 10. **int** x[N];                // 当前解 11. **int** bestX[N];            // 当前最优解 12. **int** bestC;               // 当前最优值 13. **int** cc;                  // 当前费用，形成环的时候与bestC比较看能不能更新bestC 15. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 16. \* 函数描述： 数据输入以及内存的初始化 17. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 18. **void** input() 19. { 20. cin >> n;   // 输入顶点个数 21. **int** k; 22. memset(adjMatrix, NoEdge, **sizeof**(adjMatrix));   // 邻接矩阵的内存初始化 23. cin >> k;   // 输入边的个数; 24. **int** p, q, len; 25. // 初始化邻接矩阵 26. **for** (**int** i = 1; i <= k; ++i) 27. { 28. cin >> p >> q >> len; 29. adjMatrix[p][q] = len; 30. adjMatrix[q][p] = len; 31. } 32. // 初始化最优解和当前解 33. **for** (**int** i = 1; i <= n; ++i) 34. { 35. bestX[i] = i; 36. x[i] = i; 37. } 38. bestC = NoEdge; 39. cc = 0; 40. } 42. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 43. \* 函数描述： 格式化打印结果 44. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 45. **void** printTravel() 46. { 47. cout << "\n===================\n最短路径为：" << bestC << endl; 48. **for** (**int** i = 1; i <= n; i++) 49. cout << bestX[i] << " ---> "; 50. cout << bestX[1]; 51. } 53. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 54. \* 函数描述： 回溯递归函数，进行深度优先搜索 55. \* 参数描述： i——递归到第i层 56. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 57. **void** Traveling(**int** i) 58. { 59. // 访问到第n个节点，判断是否形成回路 60. **if** (i == n) 61. { 62. // 形成回路：x[n-1]与x[n]连通，且x[n]与x[1]连通 63. **if** (adjMatrix[x[n - 1]][x[n]] != NoEdge && adjMatrix[x[n]][1] != NoEdge) 64. { 65. // 如果当前值优于最优值，更新最优值和最优解。或初始化最优解 66. **if** (cc + adjMatrix[x[n - 1]][x[n]] + adjMatrix[x[n]][1] < bestC || bestC == NoEdge) 67. { 68. **for** (**int** k = 2; k <= n; ++k) 69. bestX[k] = x[k]; 70. bestC = cc + adjMatrix[x[n - 1]][x[n]] + adjMatrix[x[n]][1]; //更新最优值 71. } 72. } 73. **return**; 74. } 75. // 当前在第i层，需要继续寻找 76. **else** 77. { 78. **for** (**int** j = i; j <= n; ++j) 79. { 80. // 判断是否可以进入x[j]子树， 81. // x[i-1]与x[j]连通使得1-i层连成一条路径且累计花费优于目前最优值， 82. // 如果可以，则交换x[i]与x[j],进入i+1层， 83. // 现在的解是x[1],x[2]...x[i]...x[j]...x[n] 84. **if** (adjMatrix[x[i - 1]][x[j]] != NoEdge && cc + adjMatrix[x[i - 1]][x[j]] < bestC || bestC == NoEdge) 85. { 86. swap(x[i], x[j]);                // 交换之后，现在的解是x[1],x[2]...x[j]...x[i]...x[n] 87. cc += adjMatrix[x[i - 1]][x[i]]; // 更新路径的长度，进入i+1层 88. Traveling(i + 1); 89. cc -= adjMatrix[x[i - 1]][x[i]]; // 还原路径的长度，比较x[j+1]子树 90. swap(x[i], x[j]);                // 还原之前的解，现在的解是x[1],x[2]...x[i]...x[j]...x[n] 91. } 92. } 93. } 94. **return**; 95. } 97. **int** main() 98. { 99. input(); 100. Traveling(2); // 出发点已知 101. printTravel(); 102. } 103. 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. 测试数据和执行结果   输入的第一行是节点数，第二行是边数，之后的行是边的信息，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。下图中给出了一个四个节点的测试用例，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。 | | | | | |
| 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，我对回溯算法有了更深入的了解。我认为回溯算法的核心在于构建解空间树，并确定约束函数和限界函数。  在实验时，起初总是得不到正确的结果，后来发现是我在写递归的时候，进入下一层解空间树在回溯的时候没有进行还原操作。递归后的还原操作是十分重要的，而且比较容易遗忘。  在本次回溯算法的实验中，我选取了“n皇后问题”和“旅行售货员问题”，这两个都是经典的回溯问题。“n皇后问题”是解空间树为子集树，而“旅行售货员问题”的解空间树是排列数。他们的本质都是回溯，但是在一些细节上还是不同的。子集树在回溯的时候是**做标记**后进入下一层，而排列树在回溯时是进行元素的**交换**后进入下一层；子集树的组合情况为种，而排列树的组合情况为种，排列树的时间复杂度也会比子集树的更小一些。 | | | | | |
| 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： | | | | |