**实验四：回溯与分支限界算法设计**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 学号 | | 202018526 | 姓名 | 高树林 | 成绩 |  |
| 友情提示 | 1. 算法描述及代码实现与网络或他人雷同者，均按0分计算；  2. 要求算法描述明确、代码清晰、格式美观；  3. 纸质版与电子版同时提交（电子版命名格式：**完整学号-姓名-实验X-实验名称**，其中,不得省略“-”，如：2088166-乔峰-实验1-分治与递归策略）；   1. 电子版中代码需格式化处理，方便查看（http://www.codeinword.com/）。 | | | | | |
| 实验目的 | 1. 掌握回溯法解决问题的一般步骤。 2. 学会使用回溯法解决实际问题。 3. 掌握分支限界法解决问题的基本思想。 4. 学会使用分支限界法解决实际问题。 | | | | | |
| 实验内容 | 1. 骑士游历问题（采用回溯法）：在国际象棋的棋盘（8行×8列）上放置一个马，按照“马走日字”的规则，马要遍历棋盘，即到达棋盘上的每一格，并且每格只到达一次。若给定起始位置,编程探索出一条路径，沿着这条路径马能遍历棋盘上的所有单元格。 2. 行列变换问题（采用分支限界法）：给定两个方格阵列组成的图形和图形，每个方格的颜色为黑色或黄色，如下图所示。行列变换问题的每一步变换可以交换任意2行或2列方格的颜色，或者将某行或某列颠倒。上述每次变换算作一步。试设计一个算法，计算最少需要多少步，才能将图形变换为图形。 | | | | | |
| 算法描述 | 1. 骑士游历问题解题思路或算法思想   在每次选择方向时，不能任意选择，而是要按照一定的策略。可以按照“先苦后甜”的策略，先估算一下当前地点的下一个位置有哪些，再估测这些路径中那些是难走的，将难走的先走，剩下简单走的就留下来了，这样一步就比一步简单。 | | | | | |
| 1. 行列变换问题解题思路或算法思想   在选择结点时，不能任意选择。要先算出当前节点下，他的所有孩子结点的一个函数值，这个函数值代表应该向该孩子结点扩展的程度，以便使搜索向着解空间上有最优解的分支推进，进而更快的找到全局最优解。 | | | | | |
| 程序及运行结果  （附截图） | 1. 骑士游历问题   代码：   1. **def** FindPath(I, J): 2. chess[I][J] = 1 3. posCount = 0 4. posI = [1, 1, 2, 2, -1, -1, -2, -2] 5. posJ = [2, -2, 1, -1, 2, -2, 1, -1] 6. nowI = I 7. nowJ = J 8. nexI = [0] \* 8 9. nexJ = [0] \* 8 10. **for** queuenumber **in** range(2, 65): 11. posCount = 0 12. **for** k **in** range(8): 13. nextI = nowI + posI[k] 14. nextJ = nowJ = posJ[k] 15. **if** nextI >= 8 **or** nextI < 0 **or** nextJ >= 8 **or** nextJ < 0: 16. **continue** 17. **if** chess[nextI][nextJ] == 0: 18. nexI[posCount] = nextI 19. nexJ[posCount] = nextJ 20. posCount += 1 21. **if** posCount == 0 **and** queuenumber < 63: 22. **return** False 23. minPosCounter = 8 24. **for** posNum **in** range(posCount): 25. posCountTemp = 0 26. **for** k **in** range(8): 27. nextnextI = nexI[posNum] + posI[k] 28. nextnextJ = nexJ[posNum] + posJ[k] 29. **if** nextnextI >= 8 **or** nextnextI < 0 **or** nextnextJ >= 8 **or** nextnextJ < 0: 30. **continue** 31. **if** chess[nextnextI][nextnextJ] == 0: 32. posCountTemp += 1 33. **if** minPosCounter > posCountTemp: 34. minPosCounter = posCountTemp 35. nowI = nexI[posNum] 36. nowJ = nexJ[posNum] 37. chess[nowI][nowJ] = queuenumber 38. **return** True  41. **if** \_\_name\_\_ == "\_\_main\_\_": 42. chess = [[0] \* 8 **for** \_ **in** range(8)] 43. a, b = input('请输入起始点坐标（x y):').split(' ') 44. a = int(a) 45. b = int(b) 46. **if** FindPath(a, b): 47. **print**("路径为：") 48. **for** i **in** range(8): 49. **for** j **in** range(8): 50. **print**("%4d" % chess[i][j]) 51. **print**('') 52. **else**: 53. **print**('未找到遍历所有结点的路径！')   截图： | | | | | |
| 1. 行列变换问题   代码：   1. **def** solve(): 2. queue = [sour] 3. **while** len(queue): 4. status\_of\_1 = queue[0] 5. **del** queue[0] 6. **for** i **in** range(4): 7. **for** j **in** range(3): 8. c1 = 1 << (i \* 4 + j) 9. c2 = 1 << (i \* 4 + j + 1) 10. **if** status\_of\_1 & c1 != status\_of\_1 & c2: 11. status\_of\_2 = status\_of\_1 12. status\_of\_2 ^= c1 13. status\_of\_2 ^= c2 14. **if** a[status\_of\_2] == -1: 15. a[status\_of\_2] = a[status\_of\_1] + 1 16. b[status\_of\_2] = (i \* 4 + j) + 1 17. **if** status\_of\_2 == dest: 18. **return** True 19. queue.append(status\_of\_2) 20. **for** i **in** range(3): 21. **for** j **in** range(4): 22. c1 = 1 << (i \* 4 + j) 23. c2 = 1 << (i \* 4 + j + 4) 24. **if** status\_of\_1 & c1 != status\_of\_1 & c2: 25. status\_of\_2 = status\_of\_1 26. status\_of\_2 ^= c1 27. status\_of\_2 ^= c2 28. **if** a[status\_of\_2] == -1: 29. a[status\_of\_2] = a[status\_of\_1] + 1 30. b[status\_of\_2] = - (i \* 4 + j) - 4 31. **if** status\_of\_2 == dest: 32. **return** True 33. queue.append(status\_of\_2) 34. **return** False  37. **def** output(status, moves): 38. **if** status != sour: 39. c1 = c2 = tem\_state = 0 40. **if** b[status] > 0: 41. c1 = 1 << b[status] - 1 42. c2 = 1 << b[status] 43. status\_of\_temp = status 44. status\_of\_temp ^= c1 45. status\_of\_temp ^= c2 46. output(status\_of\_temp, moves - 1) 47. c1 = (b[status] - 1) // 4 48. c2 = (b[status] - 1) % 4 49. **print**("第%d步" % moves) 50. map1[c1][c2], map1[c1][c2 + 1] = map1[c1][c2 + 1], map1[c1][c2] 51. **for** i **in** range(4): 52. **for** j **in** range(4): 53. **print**(map1[i][j], end=' ') 54. **print**('') 55. **else**: 56. b[status] = -b[status] 57. c1 = 1 << (b[status] - 4) 58. c2 = 1 << b[status] 59. status\_of\_temp = status 60. status\_of\_temp ^= c1 61. status\_of\_temp ^= c2 62. output(status\_of\_temp, moves - 1) 63. c1 = (b[status] - 4) // 4 64. c2 = (b[status] - 4) % 4 65. **print**("第%d步" % moves) 66. map1[c1][c2], map1[c1][c2 + 1] = map1[c1][c2 + 1], map1[c1][c2] 67. **for** i **in** range(4): 68. **for** j **in** range(4): 69. **print**(map1[i][j], end='') 70. **print**('') 71. b[status] = -b[status]  74. **if** \_\_name\_\_ == '\_\_main\_\_': 75. map1 = [] 76. Capacity = 1 << 16 77. a = [-1] \* Capacity 78. b = [0] \* Capacity 79. **print**("请输入转换前的图形（0表示黄色方块，1表示黑色方块）：") 80. end = [] 81. s = '' 82. sour = dest = 0 83. **for** i **in** range(4): 84. s += input(' ') 85. map1 = list(s) 86. **print**(map1) 87. **for** i **in** range(16): 88. sour |= int(ord(map1[i]) - ord('0')) << i 89. **print**(sour) 90. **print**("请输入转换后的图形（0表示黄色方块，1表示黑色方块）：") 91. s = '' 92. **for** i **in** range(4): 93. s += input(' ') 94. map2 = list(s) 95. **for** i **in** range(16): 96. sour |= int(ord(map2[i]) - ord('0')) << i 97. Capacity = 2 \*\* 16 98. a = [-1] \* Capacity 99. b = [0] \* Capacity 100. solve() 101. **if** a[dest] != -1: 102. **print**("至少需要%d步" % a[dest])   截图： | | | | | |
| 总结 | 本次实验是四个实验报告中最难的，也是花时间最多的，大概花费了两周时间，通过两周的调代码，看算法，我深刻认识到了回溯法和分支界限法的异同点。相同点在于回溯法和分支界限法都是在梳状的接空间上求解，回溯法找出满足条件的所有解，通过回溯法找出满足条件的所有解，通过约束函数和限界函数可以找到问题的最优解，同时分支限界法主要应用于找满足条件的一个解或者最优解。但是两者在相同中又有很大的不同之处，他们最大的区别在于，回溯法采用的是深度优先遍历搜索的策略，而分支界限法采用的是广度优先遍历的搜索策略，回溯法中一般用栈数据结构实现结点的存储，分支界限法一般多用队列实现结点的存储。回溯法只有在所有子结点被遍历之后才出栈，  而分支界限法中的每个结点只能被访问一次。 | | | | | |