人智大作业 1——连连看报告

班级： 自 93

学号： 2019010850

姓名： 王逸钦

完成日期： 2021/11/05

一　UI 设计

1)　UI 使用方法

1　指定参数

启动程序后，在左侧填上 m, n, k, p, z，表示 m 行 n 列 p 种图案共 2k 个再加 z 个阻断。需要保证输入参数合法性，不合法时弹窗提醒重输。注意，由于 pic 文件夹只提供了 40 种图片，因此 p 不应超过 40。若选自动生成则“手动输入”框可以留空；若选手动输入则按下图所示输入，注意此时也要保证参数与输入棋盘一致。

模式 1 只允许两次转向以内的消除动作，模式 2 则允许更多次转向的消除动作(后文算法部分详述)，模式 3 还额外允许设置阻断格数。因此如果想使用阻断功能，不仅需要将 z 设为非 0 值，还需要选到模式 3 方可正常工作。

完成上述指定后，点击“生成棋盘”按钮即在窗体右侧生成棋盘。

图 1：启动后的界面

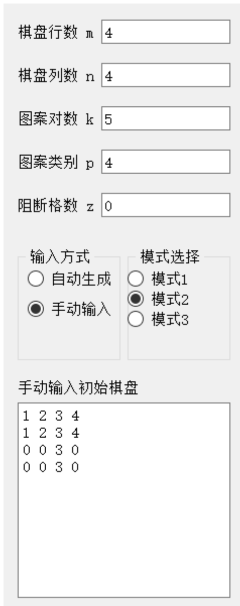

图 2：手动输入

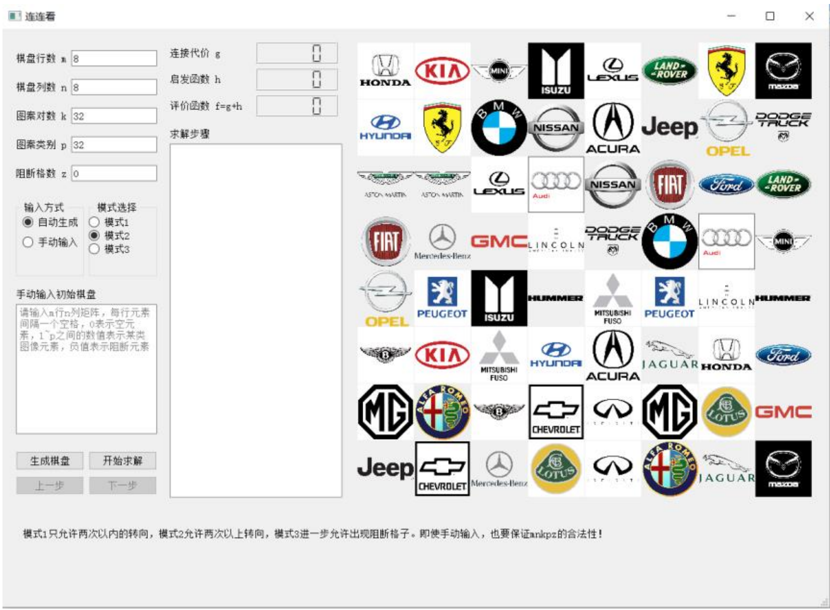

图 3：生成的棋盘

2　*计算*

生成棋盘后，“开始求解”按钮变为可用状态，点击即开始求解该问题。对于不同的图像个数，我采用了不完全相同的搜索方法，在不太大的参数输入下(k<128, p<40, m*n<256)，把运算时间控制在半分钟以内，如图 4。因此，当图像个数少且棋盘尺寸小时，有更大的概率找到最优解（累计的拐弯损失最小）；当图像个数多或棋盘尺寸大时，需要向效率妥协，求得的解可能离最优解存在一定差距。

以求解图 3 中的棋盘为例，求解完毕后弹窗提示如图 5。

图 4：复杂情形

图 5：图 3 棋盘求解结果

3　*查看*

求解完毕后，“上一步”、“下一步”按钮变为可用状态，不断点击“下一步”即可看到图案被一步步消除的过程。此时连接代价 g、启发函数 h、评价函数 f 的值随着消除的进行不断变化，显示在数码管控件中；“求解步骤”框中会打印出每步的 g(连接代价 turn_count)和 f(评价函数 path_cost)。

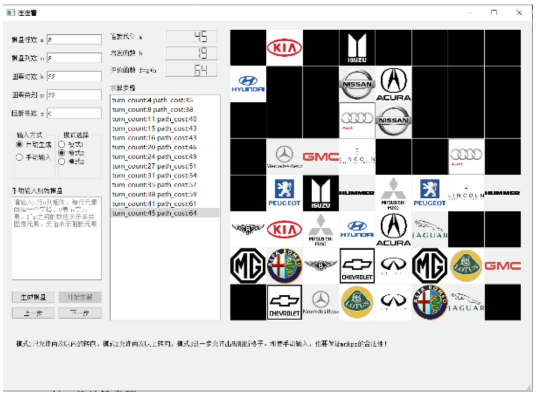

图 6：消除过程(中间)

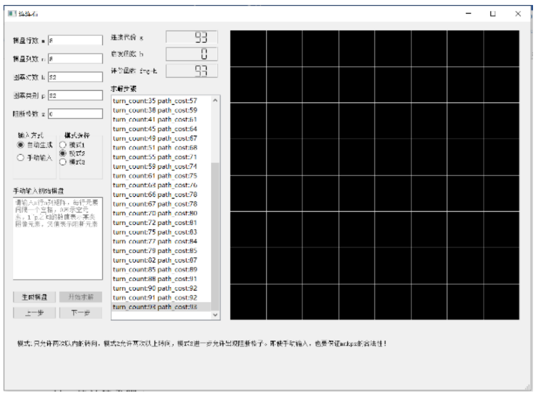

图 7：消除过程(结果)

2)　UI 设计与开发思路

采用 PyQt，使用 Qt Designer 画出图形界面。数值输入使用 LineEdit 控件，多选一使用 GroupBox 套 RadioButton，数值显示使用 LCDNumber，按钮用 PushButton，棋盘显示使用一个大的 Label。界面如图 8。

得到 Qt Designer 的.ui 文件后，利用 pyuic5 工具将其转为可执行的 python 代码，在这个生成的代码之上继续编辑，加入功能和算法即可。

为了让界面更整齐，我将控制按钮和参数显示框均放在窗体左侧，右侧用于显示棋盘；为了让程序更易用，我在下方增加了使用说明，在棋盘的手动输入框内添加了讲述输入规则的预置说明文字。

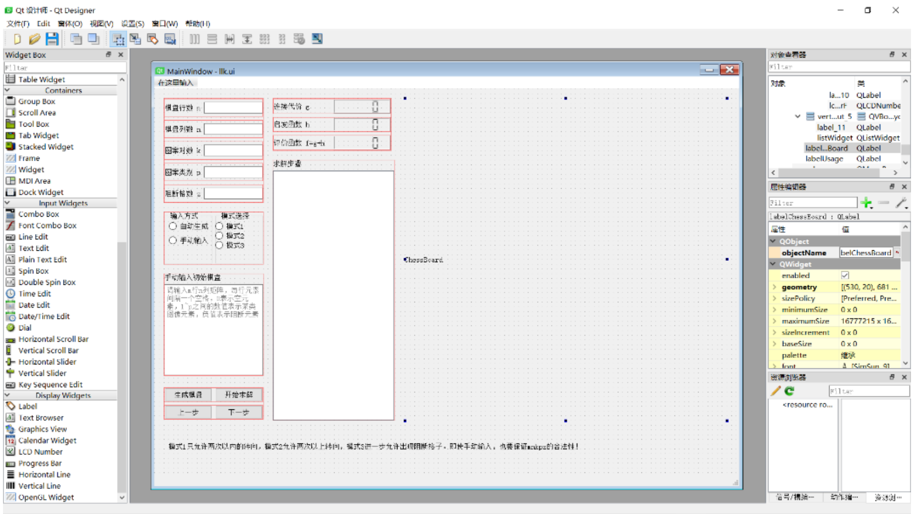

图 8：ui 界面设计

二　类设计与算法

1)　类设计

整个程序由两个.py 文件构成：

llkui.py 由.ui 文件生成，在其之上增加了 4 个槽函数分别用于响应 4 个按钮，以及 1 个主函数，在程序执行时进入主函数，在按钮被按下时进入对应的槽函数。

llkai.py 文件是核心算法，主要由 Node 和 LLKProblem 两个类构成，这两个类的设计思路借鉴了 coding1 作业中助教给的代码框架。

Node 类表示一个节点，在本问题中表示一个棋盘状态，除了以二维 ndarray 保存棋盘以外，还存有当前状态的 g 值和 f 值(便于实施 A*算法)以及父节点(便于搜索结束时反向查找搜索链)。

LLKProblem 类表示一个连连看问题，成员变量包括 m,n,k,p,z。这个类中最核心的成员函数是 actions()，向该函数输入一个棋盘状态，结合该问题实例的固定参数，actions()就可以找到下一步可能的消除方式(找子节点)，将这些消除方式保存在一个 list 中返回。

根据下图，再次梳理整个程序的执行流程：

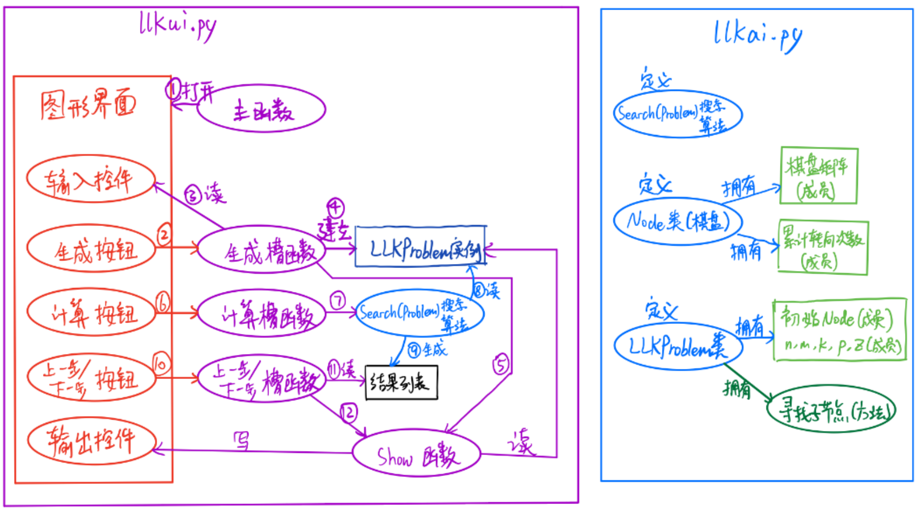

图 9：流程图

llkai.py 定义了 Node 类、LLKProblem 类以及 Search(Problem)函数。

现在考虑 llkui.py 执行过程。图形界面主函数【1.打开】图形界面，用户输入参数之后点击“生成”，【2.调用】生成槽函数，槽函数【3.读取】用户的输入，根据这些参数【4.建立】了一个 LLKProblem 实例，并【5.调用】Show 函数（Show 函数每次被调用时，都读取本类的 Problem 实例并将棋盘显示在输出控件上）。此时，连连看图案出现在窗体右侧。

用户点击“计算”，【6.调用】计算槽函数，槽函数【7.调用】Search(Problem)函数，【8.读取】当前 LLKProblem 实例作为函数参数，通过计算【9.生成】结果列表。此时，计算完毕，弹窗提醒。

用户点击“下一步”或“上一步”，【10.调用】对应的槽函数，槽函数【11.读取】结果列表，并【12.调用】Show 函数显示结果列表中的某个棋盘。此时，连连看图案随点击而变化。

本节没有回答两个问题:

1.如果扩展节点，找到下一步可能的消除方式?

2.如何对连连看搜索树进行搜索?

这两个问题的算法分别对应 actions()函数和 search()函数，在后两小节详述。

2) 扩展节点的方法

给定一个连连看矩阵，现在想要找到其中可以相消的所有“图像对”。以我所使用的汽车车标为例，给出一种直观的思路:

如图 10，对于(0,1)的“阿罗”标，希望找到一条到达另一个“阿罗”标的路径，这个路径沿途不能有其他车标阻挡，也不能有禁止通行的阻挡块，而且希望这个路径拐弯数尽量少。

起初，除了起点(0,1)以外整个棋盘都处于未探索状态，用“-1”标记起点，如图 10(a)。首先从“阿罗”标出发，在同行同列寻找其他“阿罗”，若没碰到则把沿途的空格都标为 0，意思是“这些点拐 0 次弯就可以到达”；如果遇到其他车标或障碍，则停下，并记为“-2”。如图 10(b)。

然后遍历整个图中标有“0”的点，依次从它们出发在同行同列寻找“阿罗”，若没碰到则把沿途的空格都标为 0，意思是“这些点拐 0 次弯就可以到达”；如果遇到其他车标或障碍，则停下，并记为“-2”。如果碰到另一个“阿罗”标且该位置尚未被赋过值，说明新发现了一个“阿罗”，配对成功。本例中，首先从(0,0)位置的 0 出发，向右找到“阿罗”，但标有“-1”说明并非新发现，忽略；向下找到“阿罗”且未被标记过，说明新发现，拐 1 次即可到达。二者消除之后就生成了一个新节点。

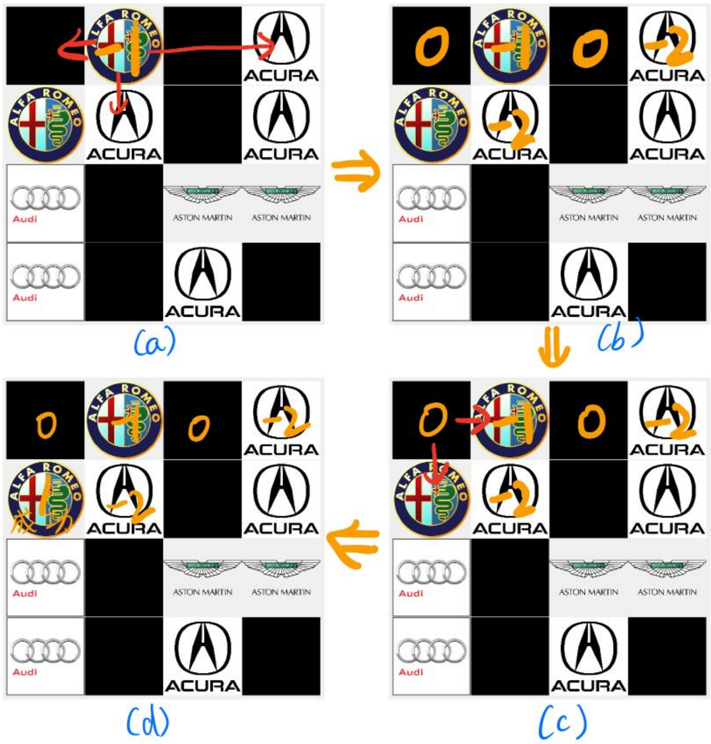

图 10：(0,1)阿尔法罗密欧配对(1,0)

总结上述算法：遍历整个矩阵。对于每个位置，如果是空格或障碍则跳过，如果是图案则以该位置为起点寻找匹配图案。首先以从起点出发向四周“探索”，将沿途空格标为 0，将对他产生阻碍的位置标为-2。然后从图中的每个 0 出发向四周“探索”，将沿途空格标为 1，将对他产生阻碍的位置标为-2。然后从图中的每个 1 出发…依此类推，直到发现一个未被赋值过的同类图案为止。

这种算法保证了：对于每个起始图案，找到的消除方法一定是拐弯数最少的。

对每个图案都找到一种对应的消除方法，将这些消除方法去重之后，每种方法就对应一个子节点。

需要补充的是，在 mode=1 时最多允许两次拐弯的出现，因此“探索”3 轮还没找到的话，就放弃寻找以节点为起点的消除方式。

3)　搜索的方法

使用 A*算法。

目标是寻找一种总拐弯数尽量少的消法，因此不妨定义拐 n 次的消除的代价 g=n+1，即直连代价为 1，拐 1 次代价为 2…

还需要定义启发函数 h，根据树搜索算法的最优要求，h 应该满足可采纳性，也即 h 是对未来的累计消除代价的一种“乐观估计”。此处简单地使用“假设全部直连消除”的代价，也即剩余图案数的一半作为 h。

使用一个优先级队列作为容器，以每个节点的评价函数 f=g+h 作为排序的依据，搜索方法与 coding1 作业中无本质区别。

三　实验与优化

随着图案数量的增多，A*算法的时间复杂度呈指数级上升，在图案较多时很难实施完整的 A*算法。为此，我尝试从两个角度解决此问题：

1.限制搜索树的分支个数为 maxbranch

2.限制每次寻找同类图案时所允许的拐弯次数 maxturn

1.考虑限制搜索树的分支个数，是因为图案较多时，当前棋盘可以有很多消除方案，从而有很多的子节点，将它们全部放入优先级队列可以保证算法的最优性，但从时间维度是不可接受的。事实上，对于连连看这种解法极多的问题，如果只是找到一个 cost 相对较低的解，根本不需要如此多的分支，考虑其中的一部分就足够了，因为很多方案都具有重复性或类似性。因此尝试对最大分支做出限制，每当找到的子节点总数到达 maxbranch 时就不再继续寻找了。

maxbranch 取值极大地影响了算法的时间复杂度。考虑如下 6*6 的初始状态(周围一圈 0 是加一圈空格，使得可以从外侧连线)：

```
<Node [[ 0  0  0  0  0  0  0  0]
 [ 0  8  9  1  2 11  8  0]
 [ 0 11  9  2  1  9 11  0]
 [ 0 12  6  1  2  5  3  0]
 [ 0  5  7  2  1 12  6  0]
 [ 0 11  9  3  4 10 11  0]
 [ 0 10 11  4  7 11 11  0]
 [ 0  0  0  0  0  0  0  0]](path_cost=18,turn_count=0)>
```

图 11：一种 m=n=6，k=18，p=11 的初始状态

如果使用原本的 A*算法，不限制最大分支数，则笔记本电脑在 10 分钟以内无法算出结果。设置 maxbranch 分别为 1 和 2 之后，就可快速求得结果，分别耗时 0.15s 和 27s。观察二者的 cost 可发现，maxbranch 为 2 时明显更小，这也说明分支越多虽然运算慢，但 cost 更贴近理论最小值。

```
<Node [[0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]](cost=46,turns=46)>

Running time: 0.154 Seconds
```

图 12(a)：maxbranch=1, maxturn=3

```
<Node [[0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]](cost=36,turns=36)>

Running time: 26.99 Seconds
```

图 12(b)：maxbranch=2, maxturn=3

2.考虑限制搜索树的最大拐弯次数，是因为拐 3 次弯已经足以找到绝大部分可以配对的节点了，如果拐 3 次还无法配对，那这种配对方式代价太大，不应在当前回合执行。因此我在 mode=1 时按规则规定了 maxturn=2，其他情形令 maxturn=3。

为验证 maxturn=3 的可用性，我构造 16*16 棋盘，放置 64 种共 128 个图案，经实验证明，即使对于一个如此大规模的连连看问题，maxturn=3 也并不影响算法进行到底，也即不存在某一时刻棋盘上所有的配对方式都需要拐 4 次或更多。这说明 4 次及以上拐弯的消法没有存在的价值。

```
使用A*求解LLK:
<Node [[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0. 55.  6.  3. 42. 15. 54. 38. 46.  4. 42. 28. 62. 10. 23. 18. 11.  0.]
 [ 0.  4. 38. 40.  5. 26. 20. 13. 41. 11. 23. 24. 55. 17. 40. 41. 39.  0.]
 [ 0. 35. 61. 55. 52. 24. 16.  2. 38. 49.  2. 47. 14. 62.  1. 49. 21.  0.]
 [ 0. 61. 18. 15. 11. 45. 41. 31. 38. 12. 54. 22. 32. 36. 21. 23. 58.  0.]
 [ 0. 58. 63.  8. 30. 56. 52. 31. 64. 44.  7. 34. 44. 12. 10. 45. 18.  0.]
 [ 0. 48. 41. 50.  1. 31. 64. 60. 19. 62. 59. 57. 59. 17.  6. 36. 19.  0.]
 [ 0. 32. 57. 51. 31. 57.  8. 59.  6. 39. 29. 48. 28. 53.  9. 43. 42.  0.]
 [ 0. 21. 12. 47. 64. 56. 33. 22. 45. 27. 25.  6. 24. 34. 13. 13. 42.  0.]
 [ 0. 37.  9. 28. 26. 15. 10. 16. 35.  5. 45. 37.  5. 46. 17. 60. 29.  0.]
 [ 0. 51.  7. 46. 32. 17. 46. 47.  8. 49.  7. 60. 53. 22. 56. 36. 22.  0.]
 [ 0. 61. 54. 34. 11. 14. 62. 52. 54. 61. 58. 43. 56. 47. 43. 14. 53.  0.]
 [ 0. 20.  9. 48. 51. 23. 18. 13. 15. 51. 21. 44. 35. 59. 19.  2. 25.  0.]
 [ 0. 30. 39. 19. 37. 43.  2.  4.  1. 16. 48.  3.  1.  9. 27. 58. 24.  0.]
 [ 0. 30. 33.  3. 50.  3. 26. 12. 60. 33.  8. 63. 55. 44. 30. 50. 25.  0.]
 [ 0. 10. 57. 36. 34. 32. 40. 33. 28. 25. 26. 63. 64. 20. 63. 27. 53.  0.]
 [ 0. 27. 14.  5. 40. 37. 20. 29. 52. 29. 49. 50. 16. 35.  7.  4. 39.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]](cost=128,turns=0)>

<Node [[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]](cost=415,turns=415)>

Running time: 25.09 Seconds
```

图 13：maxbranch=1, maxturn=3，可以找到解

是否可以将 maxturn 设为 2，在任何条件下忽略拐 3 次的消除方法？答案是否定的。仍然考虑图 11 给出的初始棋盘，前文已给出 maxbranch=2, maxturn=3 时问题的时间复杂度是可控的，本次我尝试 maxbranch=3, maxturn=2，经过约 10 秒的运算，程序中止，并未全部消除，如图 14 所示。仔细观察可以发现，此时任何两个配对图案都至少要拐 3 次才能消除，而本例仅为 6*6 棋盘，尚且不大。这说明 maxturn 设为 2 虽然能进一步提高速度，但可能导致本能搜索到底的问题提前终止。

```
<Node [[ 0  0  0  0  0  0  0  0]
 [ 0  0  0  1  2  0  0  0]
 [ 0  0  0  2  1  9  0  0]
 [ 0 12  6  1  2  5  3  0]
 [ 0  5  7  2  1 12  6  0]
 [ 0  0  9  3  4  0  0  0]
 [ 0  0  0  4  7  0  0  0]
 [ 0  0  0  0  0  0  0  0]](cost=30,turns=19)>

Running time: 10.75 Seconds
```

图 14：maxbranch=3, maxturn=2，搜索提前终止

基于上述思路，我又对不同规模的问题进行了多次实验，发现问题的规模主要取决于图案数量 2k，同时也跟棋盘大小 m*n 存在一定相关性。为保证任何规模的问题都能在半分钟内给出一个解，应该采用小规模多分支、大规模少分支的策略。经实验，按下述方式根据 k,m,n,mode 的值来确定 maxbranch 和 maxturn 的方法是有效的。

```
if k>16:
    self.maxbranch=1
elif k>11:
    self.maxbranch=2
elif k>8 or (m*n)>40:
    self.maxbranch=3
else:
    self.maxbranch=4

self.maxturn = 2 if (mode==1) else 3
```

图 15：根据问题规模自动确定 maxbranch

四　亮点、困难与解决方案、可改进之处、收获

1) 亮点

- 类设计清晰直观，函数调用关系清晰直观，图形界面控件命名规范(采用范老师 oop 课上所学命名法，控件类+所存变量)，代码注释完备，易读性好
- 使用控制分支的方法控制时间复杂度，对于不同规模的问题可以自适应地调节算法策略，在效率和优化目标之前取得平衡
- 可以自动生成初始图案分布，不必每次都手动输入，提高易用性
- 可以逐步展示消除过程，并显示对应时刻的 cost 等参数值

2) 困难与解决方案

-

　　在生成子节点列表时可能存在重复(两个图案相互找到了彼此，其实是一种消法)。考虑到生成带有重复的列表后再做消除比较低效，我直接采用集合来存储消除方法，每个消除方法都是集合中的一个元素。

　　一个消除方法是由两个坐标以及它们的“拐弯数”三者共同构成的，其中两个坐标是无序的，因此我首先想到用形如[{ (x1,y1), (x2,y2) } , turns]的方式来表示一个消除方法，两坐标也放在一个 set 中，具有无序性。这带来一个严重的问题：可变的数据结构是不可哈希的，但 set 的底层是通过哈希表实现的，因此上述形式的消除方法不可做为集合的元素！

　　因此我全部改用不可变数据结构：(turns, (x1,y1), (x2,y2))，规定一种坐标比较法，让两个坐标以固定的顺序存进 tuple，这样有重复消法时就会被 set 判定为重复而不再加入集合。

- 生成子节点时，要在原棋盘基础上修改，将两个消除的位置变为 0。我简单赋值之后修改，没使用 deepcopy，使原棋盘也被修改了，改用 deepcopy 解决
- 图案自己消失，单步调试发现自己在跟自己消，观察代码发现没有排除从起点出发找回起点的可能性。完善分支逻辑，只有发现跟自己相同且之前未被发现过的图案时，才算匹配，问题解决。

3) 可改进之处

- 界面有美化空间
- 可以给运算单开一个线程，避免在图形界面类中运算，避免图形界面卡住

上述两点无关算法和类设计，和课程目标不相关，在有限的时间中属于低优先级事项，因此没有做。

4) 收获

- 首次使用 PyQt 开发图形界面，使用体验比 C++的 Qt 好很多，这主要归功于 python 类具有较大的灵活性，可以比 C++少考虑很多传参的问题，且脚本语言的开发流程也比编译式的更友好。
- 基础的 A*算法并不存在很大难度，算法方面的难度在于如何对搜索方法进行改进。本作业没有对最优性提出明确要求，这让我们可以自由地平衡效率和搜索目标。
- 本次作业的最大收获其实并非来自算法，而是来自类设计。coding1 作业中我就仔细研究了助教给出的 Node 类和 Problem 类，本次将其转化为自己所用，对 python 类的理解更加到位了。