P1 miniAlphaGo for Lines of Action

1 背景介绍

集结棋(Lines of Action)，简称 LOA，是 Claude Soucie 在 1969 年推出的棋类。为奥林匹亚电脑游戏程式竞赛的指定棋类之一。棋盘可使用国际象棋棋盘。棋子各方有十二颗，以两色各代表一方。

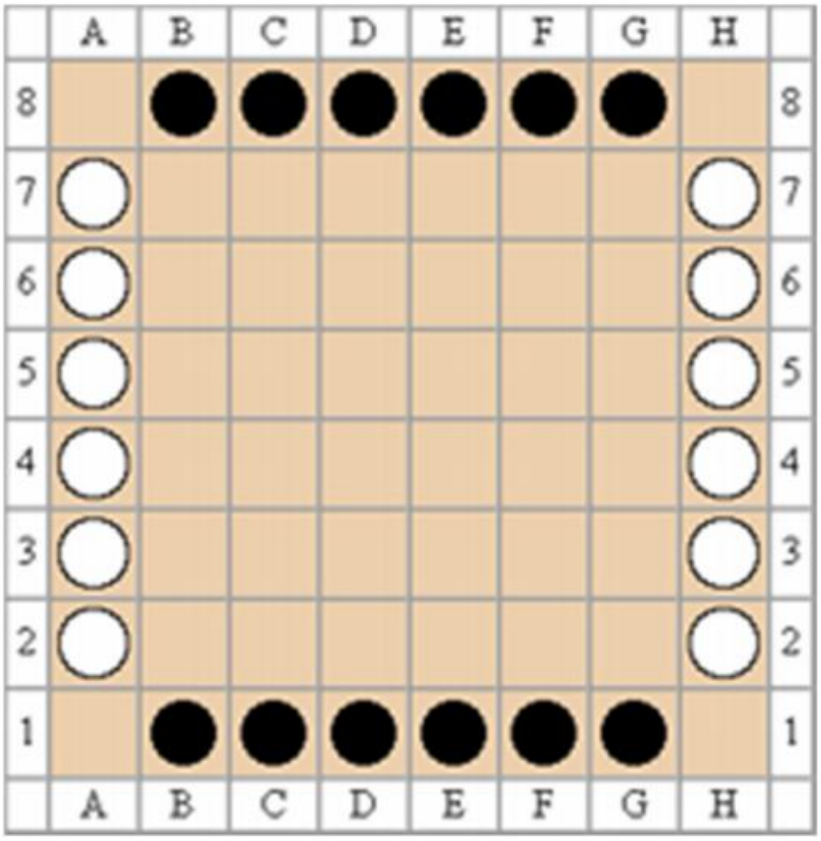

图 1 集结棋起始局面

游戏规则：

- 棋局开始时一方棋子摆于棋盘的最底与最上横行，角落不放棋。另一方放于最右与最左纵列，角落也不放棋。
- 黑方先行，双方交替下棋。
- 一步合法的棋步包括：可沿横、纵、斜线做直线移动至敌棋或空位处，途中可以穿过己棋，其移动步数必须正好等于该线上所有棋子的总数。
- 若至敌棋处，将后者移出游戏不再使用。
- 如果一方至少有一步合法棋步可下，他就必须落子，不得弃权。

- 棋局持续下去，直到某一方到获胜条件或者双方都无合法棋步可下。
- 如果某一方落子时间超过 1 分钟，则判该方失败。
- 先把棋盘上所有的己棋集合在一起即获胜，不拘形状，斜线相连也算。或者把对方吃掉只剩一枚敌棋，也即获胜。

2 项目意义

集结棋的走法空间比黑白棋稍大，胜利条件判断较为复杂。因此，在 MCTS 方法中，集结棋对走法空间、快速走子以及胜负判断等各个模块的时间复杂度有更高的要求。集结棋的胜利条件判断是一个图的连通问题，MCTS 算法总体运行的时间可以作为连通图算法的性能指标之一。

此外，集结棋在棋局后期时，随机落子策略表现很差，经常需要大量步数才能最终将棋子聚集起来，造成了运算量的浪费。如果减少随机落子的次数，又经常会陷入平局的局面，很难在小的样本量下模拟出贴近实际的结果。因此，集结棋需要采用一定的启发式策略作为指引。启发式策略的研究和分析也能为我们研究其他棋类算法提供一定的参考。

3 问题描述

- 使用 MCTS 算法实现 miniAlphaGo for Lines of Action
- 提供图形化界面用于人机对战
- 提供每一步花费的时间以及总时间显示

4 解决方案设计

4.1 MCTS 算法设计

4.1.1 选择模块设计

选择模块负责从蒙特卡洛树中选取权值最大的未扩展完全的非根节点进行扩展操作。通常采用 UCB 算法进行节点的权值计算，UCB 算法综合考虑了节点的模拟胜率以及节点的访问次数，实现了深度和广度的兼顾。

$$value = \frac{child.wins}{child.visits} + \sqrt{\frac{C \times \log(self.visits)}{child.visits}}$$

图 2 UCB 公式

如上式，权重的左半部分是子节点的胜率，右半部分根据访问次数增加而减少。常量参数 C 越大，蒙特卡洛树搜索就越偏向于广度优先；越小就越偏向深度优先。C 的值需要在测试中进行调整，比对多组测试结果后选取合适的值，以平衡深度和广度。

```
# Select the best evaluated node
while node.untriedMoves == [] and node.child != []:  # node is fully expanded and non-terminal
    node = node.select_child()
    state.do_move(node.move)
```

图 3 选择模块代码

如上图所示，大部分方法封装于节点类中，故选择过程十分简洁。整个节点类的设计参考了 MCTS Research Hub 提供的样例。[1]

untriedMoves 方法调用了 state 类的 get_moves 方法。首先通过遍历整个棋盘找到当前选手的所有棋子。然后对每颗棋子调用 get_end_loc 方法，找到每颗棋子对应的所有合法移动结果，从而组成 moves。moves 中每个 move 都包含了两个坐标，分别表示起点位置和终点位置。此外，这种表示方法也能够方便地与 GUI 进行互动。

select_child 方法根据 UCB 公式对所有子节点进行排序，从而获得估值最高的子节点作为选择结果。全局对象 state 中保存的是当前的棋盘局面，因此选择节点后需要进行移动完成棋局的更新。

4.1.2 扩展模块设计

扩展模块中，针对选择模块选中的节点，随机选择一种未尝试的走法进行棋局更新，并将生成的新节点保存为被选择节点的子节点。

```
# Expand a child node from a random chosen move
if node.untriedMoves:
    move = random.choice(node.untriedMoves)
    state.do_move(move)
    node = node.add_child(move, state)  # add child and descend tree and remove the move
```

图 4 扩展模块代码

如上图所示，若被选择节点有未尝试的走法，则随机选择一种走法进行扩展。

add_child 方法保存了新节点的棋局状态以及获得该状态所采用的走法。保存下来的走法在选择模块中被用来访问下一层的节点。

4.1.3 快速走子模块设计

快速走子可以说是 MCTS 的核心模块，MCTS 通过大量随机的局面模拟估计出一个较为贴近实际的胜率值，其中随机的局面模拟就是通过快速走子来实现的。通常情况下，为了效率起见，我们会选择随机走法进行模拟，以求效率。但如果能在保证效率的情况下加入一定的启发式策略，那么模拟结果也会更准确。可惜的是，尝试了许多种不同的走法后，我们发现不能够兼顾效率和棋力，因此还是选用了随机策略。

与黑白棋，围棋等下满结束的棋类不同，集结棋的规则是同色相聚为胜。这导致了残局情况下，随机走法可能会使得棋子变得越加分散，需要数倍的步数才能够随机到聚合的情况。因此，我们加入了限制步数的参数，到达步数后仍未分出胜负的话直接判为平局。（优化后不再直接判为平局，而是通过棋局评估函数给出一个估计值，在优化设计部分会提及）

```
# Roll out to the game end
move_count = 0
while state.win_check() == 0 and move_count <= rollout_num:
    move = state.quick_move()
    state.do_move(move)
    move_count += 1
```

图 5 快速走子模块代码

如上图所示，快速走子模块通过胜负判断和步数控制来决定是否到达终点。而 win_check 方法在每一步走子后都需要被执行一次，因此降低 win_check 方法的时间复杂度能够有效提高 MCTS 搜索效率。此外，quick_move 的复杂度也非常重要，这两个方法的优化会在优化设计中详细描述。

4.1.4 反向传播模块设计

模拟对局的结果需要反向传播回根节点，多次叠加后就可以得出较为可靠的胜率。这部分的代码较为简单，只需要从叶子节点开始循环遍历直至更新到根节点即可。唯一需要注意的地方就是每返回一层，都需要将胜负结果翻转一次。

```
# Back propagate from the expanded node and work back to the root node
if move_count > rollout_num:
    game_res = state.evaluate_state(-1)
else:
    game_res = state.win_check()    # black -1 white 1 draw 0
while node:
    node.update((game_res * -node.round + 1) / 2)
    node = node.parent
```

图 6 反向传播模块代码

4.2 GUI 设计

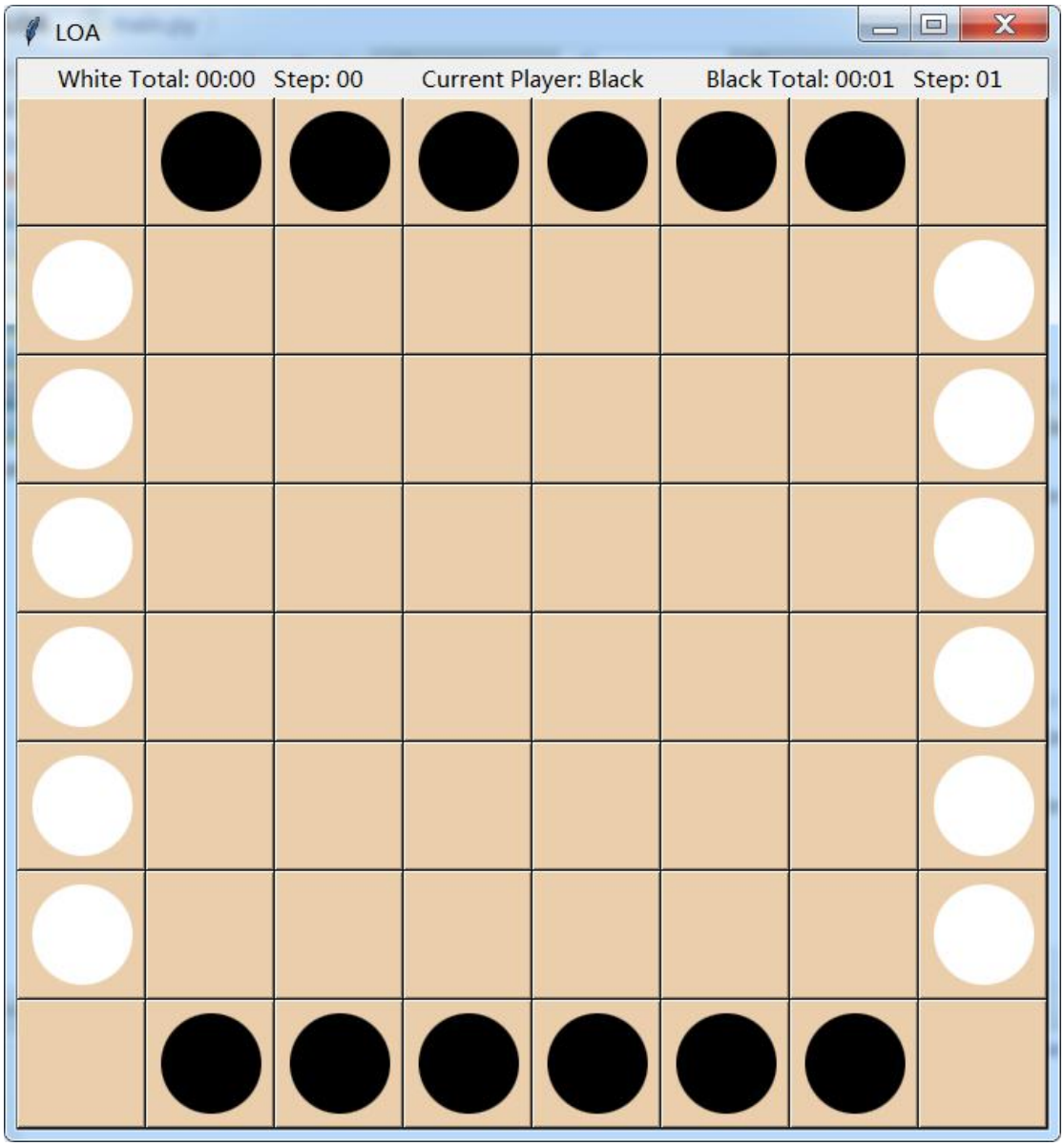

图 7 GUI

GUI 部分主要包括了对应每一个格子的按钮以及计时板。总共有 8*8 的按钮

矩阵，与 state 类中 8*8 的棋盘格一一对应。因此，调用 MCTS 算法返回的坐标值可以直接对应到相应坐标的按钮的触发事件上，可以方便的实现人机博弈。上方的计时板分别显示了黑白双方的总耗时和单步耗时，单步耗时超过 60s 就判负。

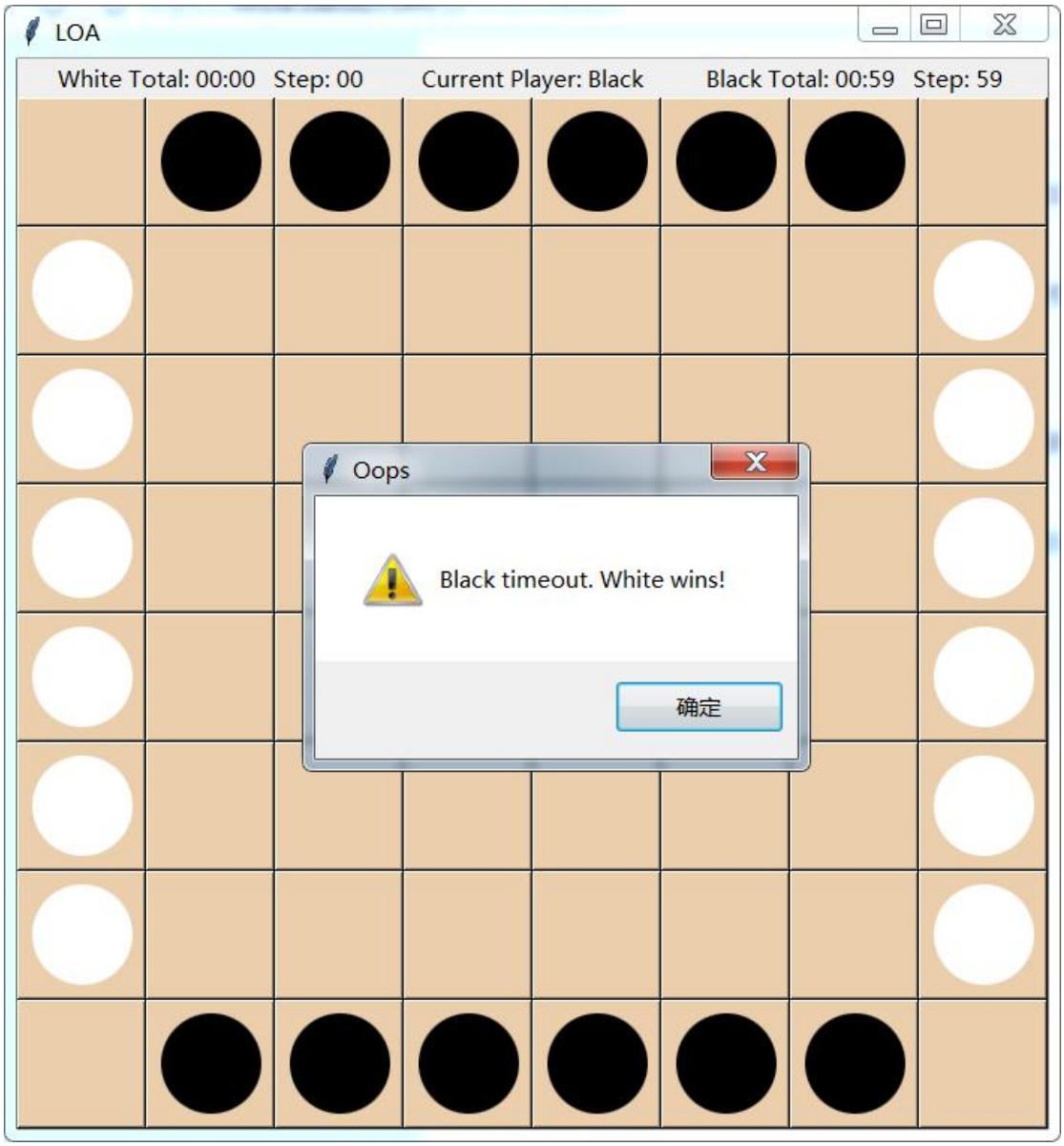

图 8 黑方超时提示，判白方胜

此外，还增加了较为便利的提示功能。被选中的棋子会被红光标示出来，可行的移动目标会被黄光表示出来，减少了下棋时的思考量。同时，如果选择了非法的走法，程序会给出提醒，确认后可以正常继续下棋。

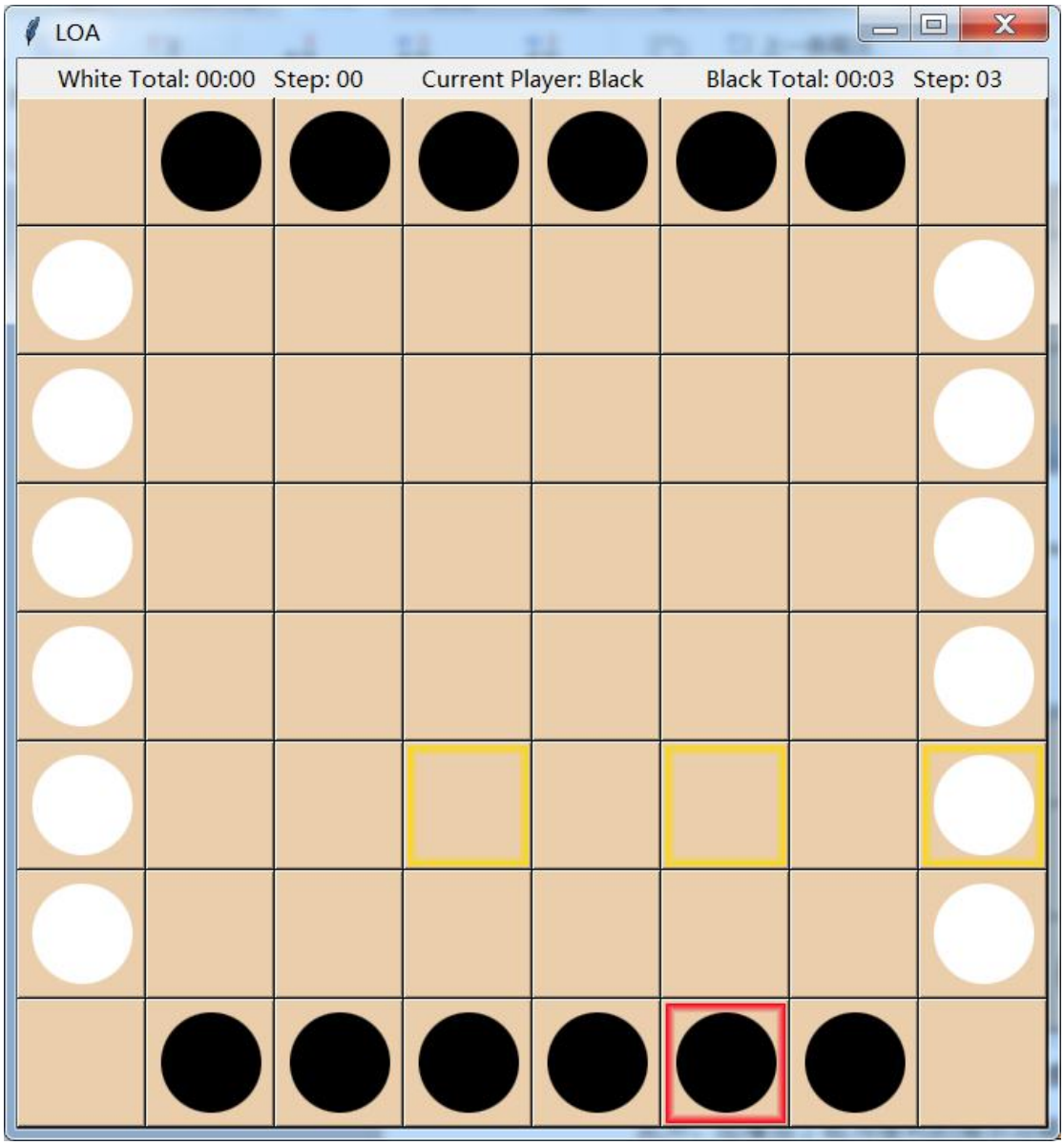

图 9 选中棋子后的提示效果

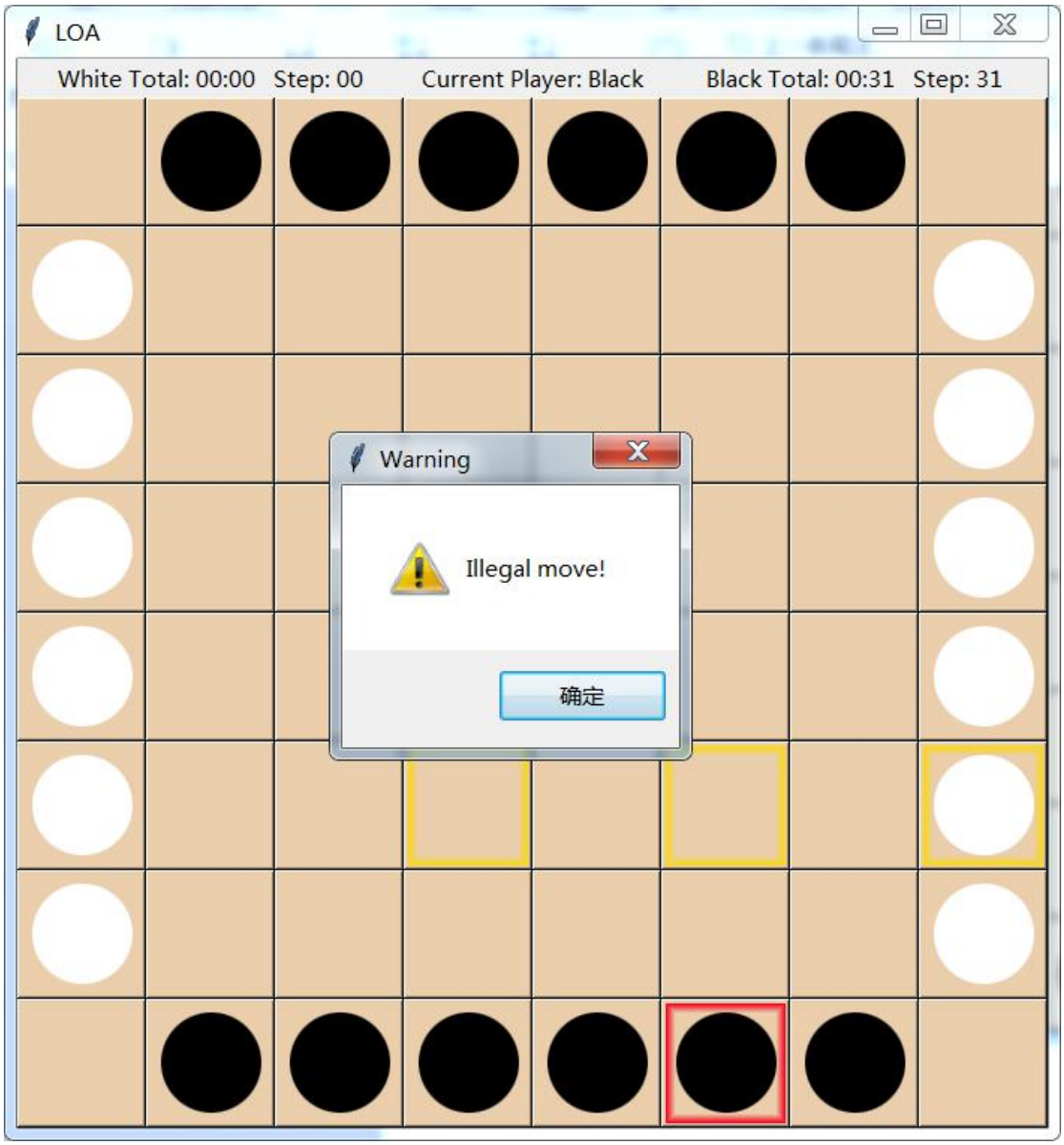

图 10 点击合法移动目标以外的格子后的提示

4.3 优化设计

4.3.1 UCB 公式优化

根据 Winands 论文[2]中提到的对 UCB 公式的优化，我们在 UCB 公式中加入了 score 项用于局面的评估。优化后的 UCB 公式如下图所示。

$$value = \frac{child.wins}{child.visits} + \sqrt{\frac{C \times \log(self.visits)}{child.visits}} + score$$

图 11 优化后的 UCB 公式

为了获取 score 值，我们必须设计一个较为合理准确的局面评估函数。为此，我们参阅了 Winands 先前设计的 MIA 集结棋 AI 论文[3]，并根据他的思路设计了两个较为容易实现的局面特征——连通度和区域价值作为评估标准。

- 连通度即每个棋子周围 8 格同色棋子的平均值，这个值越高，棋子越集中。
- 区域价值被设定为中心 4*4 区域价值为 1，其余区域价值为 0。总价值越高，棋子越集中在棋盘中心。

实际使用时都采用两种棋子对应参数的差值作为评估标准，虽然两个特征都不甚准确，但将整个模型的棋力提高了许多。

4.3.2 快速走子模块优化

快速走子模块是整个 MCTS 中调用次数最多的部分，因此，这个模块的性能很大程度上决定了整个 MCTS 算法的性能。我们对这个模块做了以下优化：

- 简化胜负判定 win_check：

通常情况下，棋局以一方棋子连通为终结。但少数情况下会有双方都无合法走法的终局。由于判断合法走法的算法需要遍历每一颗棋子，耗时较长，因此我们放弃了这一部分的精度，每次只执行连通性的胜负判定。

- 舍弃跳步判定：

在一方没有合法走法时，另一方可以连续行动两次。同样的，合法走法的判定过于耗时，所以舍弃了跳步判定。(只是在模拟对局中舍弃了这个判定，实际对局中还是会有跳步出现）

- 限制步数：

在步数超过限制，且棋局尚未结束时，直接跳出循环，用 4.3.1 中所述的棋局评估函数直接获得评估结果并作为胜负结果返回。需要注意的是，评估结果和胜负结果的值域相同，但评估的结果可能有许多位小数。值域的相同确保了估值结果和胜负结果的权重一致。

- 改进随机走法 quick_move：

不同于一开始的获取所有走法后随机取一个的方法。优化后，我们采用了随机取一颗棋子，并从这颗棋子的所有合法走法中随机取一个的方法。这个优化是所有优化中效果最为显著的，至少将算法的性能提升了 10 倍（准确的说是提升了 12 倍左右，因为原先需要遍历 12 颗棋子，现在只需 1 颗）。但同时也会有选取的棋子没有合法走法的风险。于是我们加入了一个判断，在没有合法走法时，重新随机选取一颗棋子进行走法的选取。

5 实验结果

首先我们让 AI 与随机走法进行对战，测试了十数局，AI 全胜。

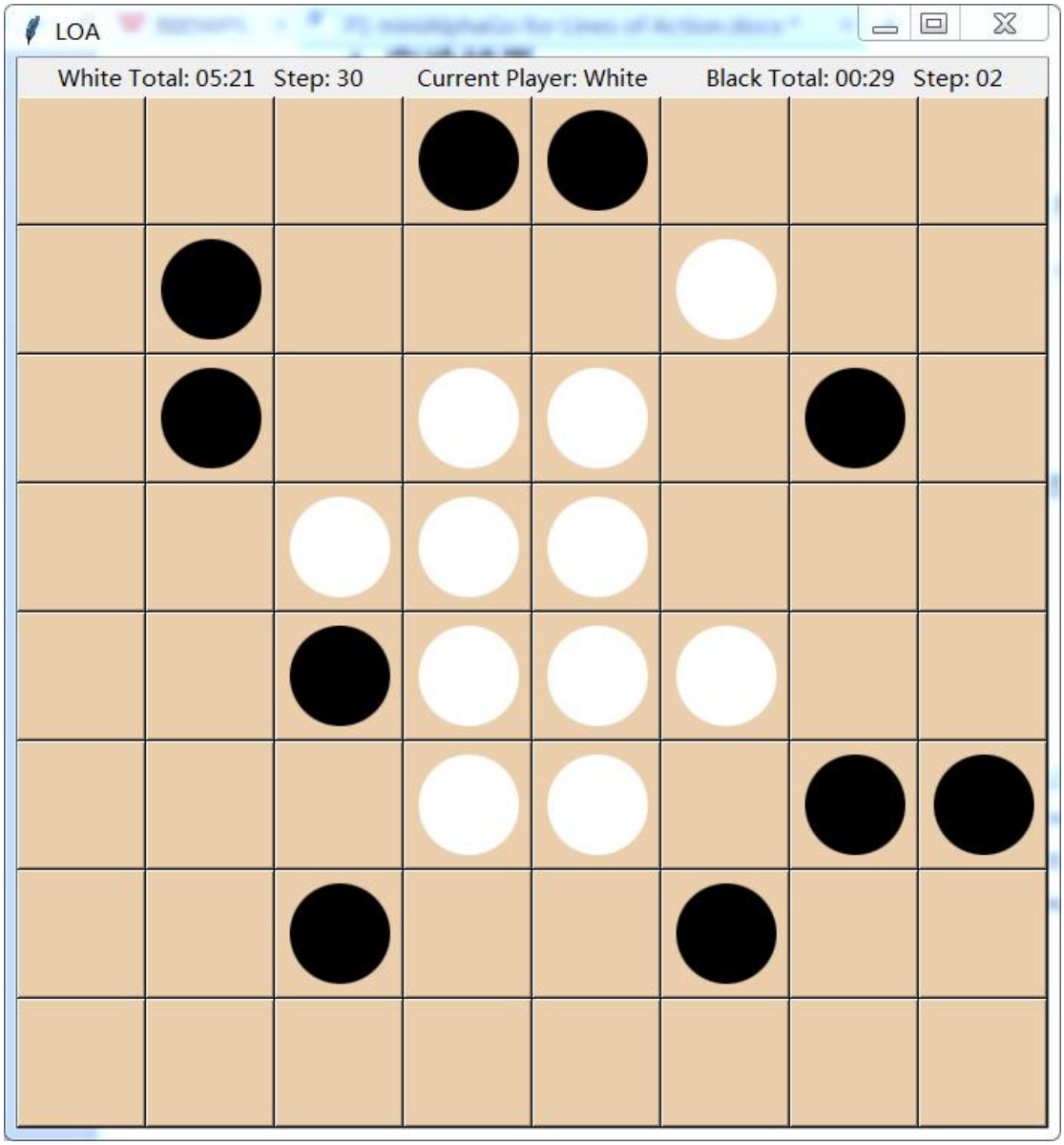

图 12 AI 执白战胜随机走法

随后，我们邀请了另一位同学临时学习集结棋的规则并与 AI 对战，AI 以 2:0 战胜了这位同学。随后，该同学表示找到了一些技巧，并成功在第三局打败了 AI。总比分为 2:1。

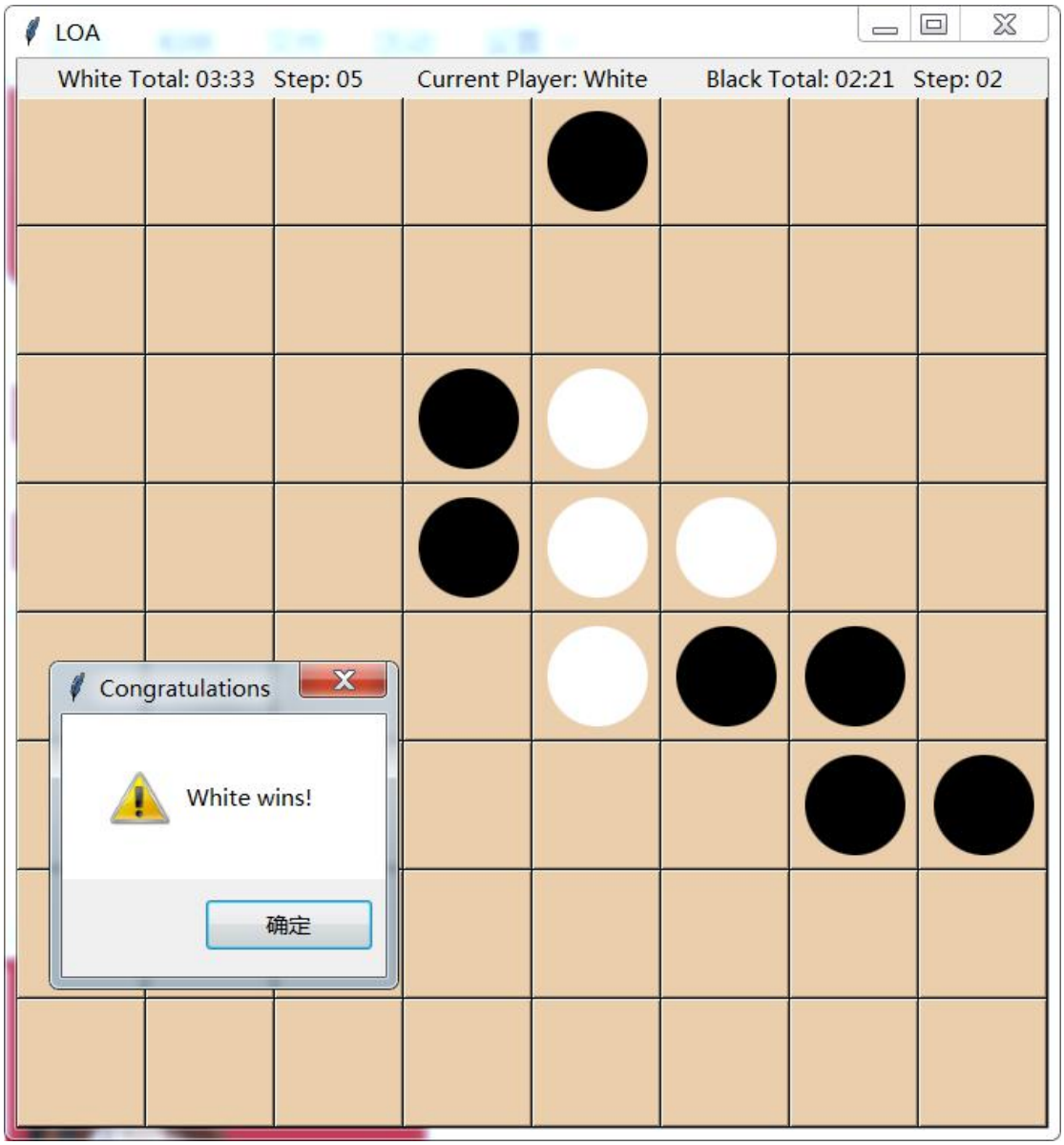

图 13 AI 执白战胜初学者

6 结论与反思

这次实验我们仅仅完成了一个能和初学者对战的初级 AI，但我们的估值函数也只是对局面出于直觉的估计。MIA 的设计论文中提及了 10 种之多的特征，把这些特征纳入考虑的话，AI 性能应该会获得飞跃。

参考文献：

[1] Monte Carlo Tree Search - Python Code. http://mcts.ai/code/python.html, 2018/6/20

[2] Winands M H M, Bjornsson Y, Saito J T. Monte Carlo Tree Search in Lines of Action[J]. IEEE Transactions on Computational Intelligence & Ai in Games, 2011, 2(4):239-250.

[3] Winands M H M, Herik H J V D. MIA: A World Champion LOA Program[C]// The, Game Programming Workshop in Japan. 2006.