数据结构与算法课程实习作业报告

韩泽尧* 李煦东 杨佳宇 刘金禹 王之昱

摘要：主要思想是为局面设置一个估值函数，使用 minmax 算法搜索最优策略，使用 αβ 剪枝增加搜索层数。此外，使用在对方空格较多的时候只在己方落子的启发式规则；用当前剩余时间和回合数决定搜索层数，在不超时的前提下充分利用思考时间；对落子和移动操作使用不同的估值函数等。

关键字：minmax 算法、αβ 剪枝、决策树

1 算法思想

1.1 总体思路

首先为棋局设计一个估值函数，用来评价现在的我方局面。估值函数值越大，说明我方的局面优势越大。我方进行落子、移动的决策时，总是倾向于最大化估值函数；但是在一步操作后保证了估值函数最大化，不一定能够保证数步后估值函数依然是最大的，于是在决策时需要多考虑几步。

每次我方的决策都尽可能的最大化估值函数，并假定每次对手的决策都尽可能的最小化估值函数，依此在每次决策的时候都考虑若干步以后的局面，再进行决策，以保证若干步之后的估值函数是最大的。

此外，在搜索过程中，使用 αβ 剪枝来尽可能减少不必要的搜索过程，再结合一些启发

式规则，尽可能节约时间，来保证足够深的搜索深度。同时，再针对剩余回合数和时间来灵活设置搜索深度来避免超时或是时间的不充分利用。

1.2 算法流程图

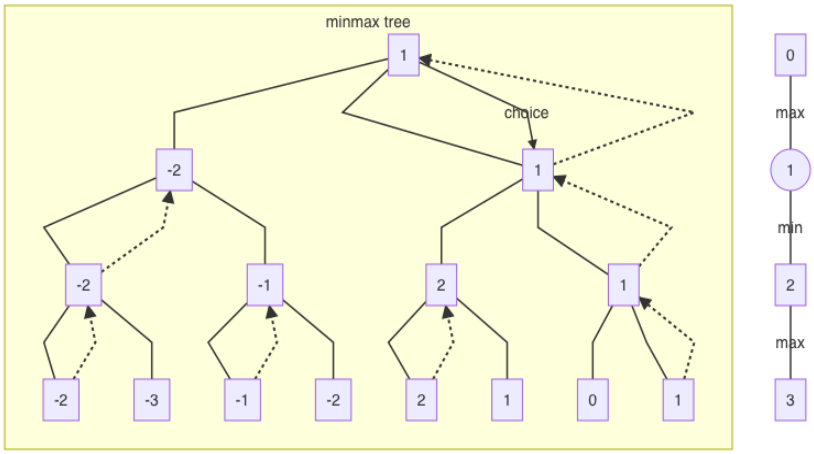

图 1：minmax 树示意图

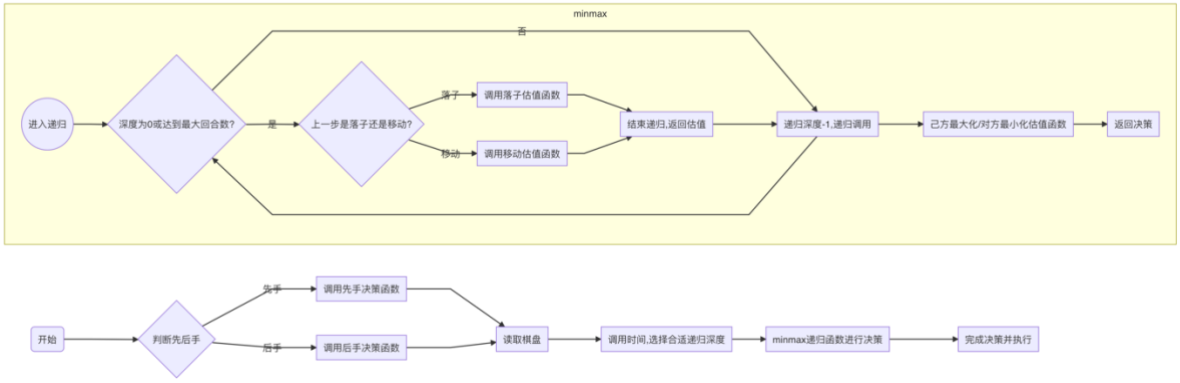

图 2:算法流程图

1.3 算法运行时间复杂度分析

主要复杂度在于 minmax 决策中的递归搜索。在棋盘上有 n 个可落子位置时，进行搜索搜索要考虑 4 种可能性；在移动的决策中一般会有 4 种可能的情况。如果搜索 h 层，大致会有 h/2 次是对落子操作的搜索，有 h/2 次是对移动操作的搜索，并假定棋盘上可落子数目无显著改变，时间复杂度大约为$O[(4n)^{h/2}]$.

我们虽然采取了 αβ 剪枝，减少每层不必要的搜索，使时间复杂度大幅减少，但是对于搜索深度 h 而言，仍然是指数级别增长。

在对局中，如果对方空格多于 4 格，我们就不考虑在对方棋盘上落子，这启发式规则可以极大的限制 n 的大小，复杂度大致减少到$O(4^h)$.

可见在该算法中搜索深度 h 会显著的影响决策用时。虽然搜索深度越深，决策的效果越好，但是，实际战斗中思考时间有限，要尽量充分利用时间，但同时也不能超时。所以我们采取了对搜索层数进行了实时的调节，根据剩余每回合的平均时间来调整每次决策的搜索深度。在这样的时间管理策略下，整个对局既不会轻易超时，也能充分利用 5s 的决策时间。

2 程序代码说明

2.1 数据结构说明

程序并未直接使用各种数据结构，但在代码中运用了一些数据结构的思想。在 minmax 决策中使用了递归函数调用，实际上是对线性结构栈的应用。在搜索中实际上应用了树结构的思想，不同的决策作为树的不同子节点，最后再从叶节点逐层决策返回到根节点。

2.2 函数说明

初始化：传入先后手，棋子权重，最大局数，最长思考时间，开启时间管理的回合数阈值。
getActions：传入棋盘、当前回合、当前玩家，返回该玩家该回合所有可能的落子位置，其中在对方落子的所有位置返回为一个列表。
getDepth：传入当前回合和剩余时间，返回决策的递归深度，用于时间管理。
weightSum：传入棋子分数列表，返回他们的权重之和，用于局面估值。
scoreMove：传入棋盘，返回移动操作后的局面估值。
score：传入棋盘和当前回合数，返回落子操作后的局面估值，用于决策。
_minMaxRecur：递归调用，返回若干次递归后 minmax 决策下的局面估值，用于决策。
minmaxDecisionF：先手决策函数，传入棋盘、当前回合和模式，返回最终决策。
minmaxDecisionNF：后手决策函数，传入棋盘、当前回合和模式，返回最终决策。
output：根据我方先后手分别调用各自决策函数，返回最终决策。

其中核心递归函数代码如下：

```
def _minMaxRecur(self, board, depth, phase, currentRound, alpha=-(2E9), beta=+(2E9)):# 返回值为局面估值
# phase 0 1 2 3，分别对应每一回合的先手落子、后手落子、先手移动、后手移动四次操作
    if depth <= 0 or currentRound >= self.maxRounds:
        if phase == 1 or phase == 2: # 是落子操作的下一步
            return self.score(board, currentRound)
        else:  # 是 move 之后的下一步
```

```
                return self.scoreMove(board)
        peer = not bool(phase % 2)  # peer 表示当前操作的一方
        flag = bool(peer == self.isFirst)  # flag = True 时为自己回合，False 时为先手方回合
        if phase == 2 or phase == 3:
            if phase == 3:
                currentRound += 1
            for d in [0, 1, 2, 3]:  # 可以改顺序 方便剪枝
                newBoard = board.copy()
                if newBoard.move(peer, d):
                    curScore = self._minMaxRecur(newBoard, depth - 1, (phase + 1) % 4, currentRound,
alpha, beta)
                    if not flag and (curScore < beta):
                        beta = curScore
                        if alpha >= beta:
                            return alpha
                    elif flag and (curScore > alpha):
                        alpha = curScore
                        if alpha >= beta:
                            return beta
        else:
            nextMove, posLst = Player.getActions(currentRound, board, peer)
            if nextMove is not None:
```

```
                posLst.insert(0, nextMove)  # 插在最前面，方便 alpha-beta 剪枝
            if len(posLst) < 5:  # 对方棋盘上可以下的空比较少时才考虑向对方棋盘上落子
                for pos in posLst:
                    newBoard = board.copy()
                    newBoard.add(peer, pos)
                    curScore = self._minMaxRecur(newBoard, depth-1, phase + 1, currentRound, alpha,
beta)
                    if not flag and (curScore < beta):  # alpha-beta 剪枝
                        beta = curScore
                        if alpha >= beta:
                            return alpha
                    elif flag and (curScore > alpha):
                        alpha = curScore
                        if alpha >= beta:
                            return beta
            else:
                newBoard = board.copy()
                newBoard.add(peer, posLst[0])  # 空格比较多时，能下自己这儿下自己这儿，下不了自己
这儿下第一个对手空格处
                # 这个设计很容易导致自己被卡死（即容易被对手一连塞爆几十回合无法脱身），但经过对
比，多搜几层带来的好处更多
                curScore = self._minMaxRecur(newBoard, depth-1, phase + 1, currentRound, alpha, beta)
                return curScore
```

```
        return alpha if flag else beta
```

另外两个估值函数的函数代码如下：

```
def score(self, board, currentRound):   # 用于落子的估值函数
    myScoreLst = board.getScore(self.isFirst)
    if board.getNext(self.isFirst, currentRound) is None:
        total = -self.weight[myScoreLst[-1]] / 3.0   # 无空可走的情况酌情扣分
    else:
        total = 0
    RivalScoreLst = board.getScore(not self.isFirst)
    total += self.weightSum(myScoreLst) - self.weightSum(RivalScoreLst)
    return total   # 估值函数考虑了己方棋子分数、对方棋子分数和己方是否被卡死



def scoreMove(self, board):   # 用于 move 操作的估值
    myScoreLst = board.getScore(self.isFirst)
    RivalScoreLst = board.getScore(not self.isFirst)
    total = self.weightSum(myScoreLst) - self.weightSum(RivalScoreLst)
    # move 操作的估值函数中所有空格有额外两分
    total += (len(board.getNone(self.isFirst)) - len(board.getNone(not self.isFirst))) *2.0   # t
    return total
```

为避免占用篇幅过长，只给出比较核心的代码。更多代码详见另附的程序文件，或者通过文章后面的 GitHub 链接访问。

2.3 程序限制

观察对局，发现三个明显缺陷：
1. 如果已经通过吃子进入对方棋盘，而后一步为了使分数最大又再一次深入对方阵地，就会导致这个棋子看起来属于自己，实际上无法回来，在数个回合之后一定会被对方吞并，实际上造成分数损失（这个缺陷应该各种算法都有可能存在，除非进行过针对性的优化，而且某些情况下可以达到干扰对方的作用，但是整体上弊大于利）。
2. 决策时在对方空格较多的时候直接选择在己方棋盘上落子，有时候会很容易被对方卡死，从而无法取胜，甚至在卡死之后被对方胁迫，不得不给对方吃子。但这样的策略也有好处，因为更多的时候在对方空位较多时往往都会下在己方，因此可以减少很多没有必要的搜索，大大地节约时间，保证其它操作有更深的搜索深度。
3. 没有考虑盘面平滑性，有些时候在随机序列并不理想时，会出现几个比较大的子相互隔开无法进一步合并的情形。但棋子权重的设置略微弥补了这一点，不过总体上还是会出现这样的无法合并的情形。

3 实验结果

3.1 测试数据

前期代码的编写和参数的调整由组员各自在自己电脑上进行，在主要代码成形之后，每个人在各自调整参数后都在本地和之前的代码进行内战，选取出最优的参数。

为了贴合实战，中后期所有的测试全部在天梯平台上用个人账号进行，各自调整代码或者参数后在天梯上和旧版本、其他同学的代码进行实战测验，根据实战效果来决定是否对代码进行这样的调整。

因为各个组员各自都参与了修改代码、测试自己的想法的过程，故在小组天梯赛正式开始前进行了大量的尝试，测试后效果良好的代码改动会上传到 GitHub 上，因此可以从 GitHub 的 commit 历史中看到一步步测试完善的过程。在小组天梯赛开始后代码只有细微的改动，通过每天的系统自动发起的比赛结果来对改动作出评判调整。

下面截取本组在三个时间点的对战情况：

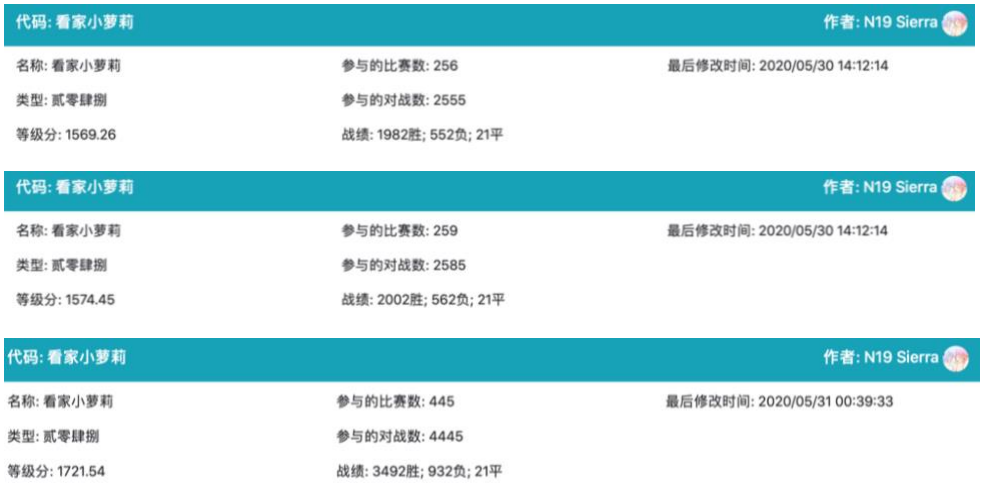

图 3:小组赛战绩

在之后的天梯赛和淘汰赛中，本组代码均表现良好，最终获得 N19 赛区冠军，在友谊赛中输给了 F19 第一名。

3.2 结果分析

在我们的算法使用了如下策略：

策略	效果
相同棋子合成高一级棋子后分数变高（1+1>2）	鼓励合并棋子，不容易卡死，容易获得高级棋子
对落子时对是否卡死额外扣分，在移动时对空格数目额外计算分数	防止卡死，容易发现卡死对方的操作
根据剩余时间控制递归深度	有效利用时间，不至于浪费时间或者超时
递归中进行 αβ 剪枝	排除明显劣势策略，减少运算时间，加大搜索深度，得到更优策略
落子时如果对方空格较多且能下在己方则不用搜索，直接下在己方位置	避免不必要的搜索，节省时间，从而加大其它操作决策的搜索深度

主要时间开销在于 minmax 递归，具有指数复杂度。通过根据剩余时间控制迭代深度进行时间管理，我们基本上不会超时而且一局总使用时间基本在 4s 以上，最大化利用已有时间，在不超时的前提下尽可能利用 5 秒的时间，来搜索更多的层数，提高胜率。

可以看出，我们的代码在自己分组的竞技场中表现良好，偶尔会出现与对方僵持不下的状况，推测是因为随机序列导致。总体上超时和报错较少，且发挥稳定，。

4 实习过程总结

4.1 分工与合作

韩泽尧：编写 minmax 算法主体部分

李煦东：添加 αβ 剪枝

杨佳宇：调节时间管理参数，协助撰写报告

刘金禹：调节空格分数列表，撰写报告

王之昱：调节棋子分数，协助撰写报告

我们组很早就开了第一次会，进行分工，编程能力较强的同学主要负责为程序进行结构上的改进（如撰写主题，增加剪枝、时间管理函数等），其余的同学主要负责调整参数，观察对局来提出建议。这里的分工并不是严谨的，大家都对本组代码的最终形成做出了自己的贡献，并且都参与了调参的工作，上面只是列出了较为重要的贡献。

我们组的交流方式为集中的线上小组会议与平时的微信群交流相结合。在几个重要时间点利

用腾讯会议来总结当前成果，交流并制定目标，统筹分配下一步工作。在平时则在微信群里实时交流每天的进展，并在 GitHub 上同步更新。

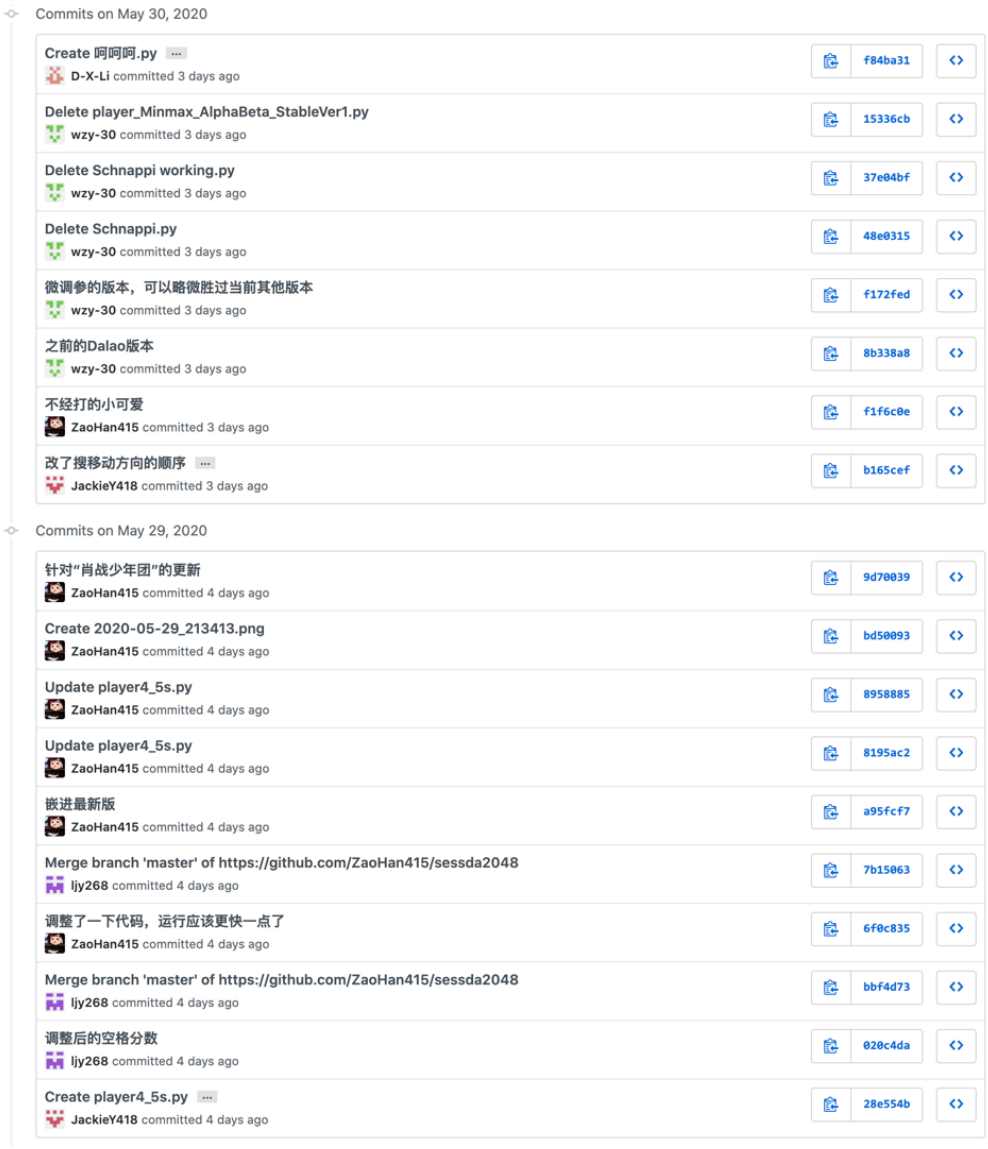

图 4：Github 上 commits history 部分截图

附本组 GitHub 项目网址：https://github.com/ZaoHan415/sessda2048

另附本组第一次和最后一次组会的合影：

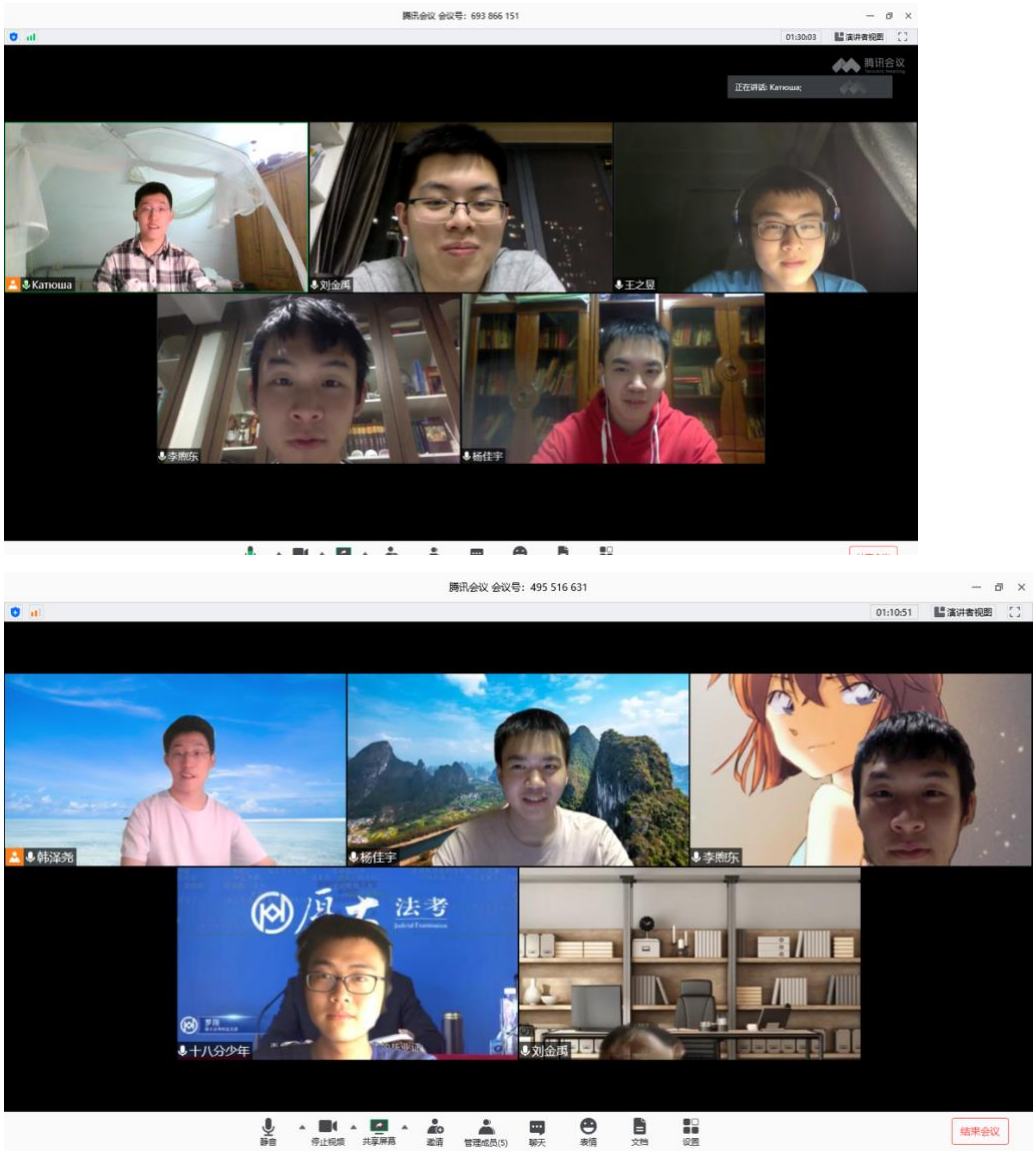

图 5:小组会议合影

4.2 经验与教训

本组在这次实习作业中的得意之处在于开工较早，早在五一假期 minmax 算法和 αβ 剪枝就已经基本成型，已经可以完成正常的比赛。这使得我们的时间更为充裕，有更多的时间去测试并优化我们的代码。

另外我们组的整体思路是让每个人都可以尝试自己的思路和想法，没有规定只有某一两个人编写代码，这就使得在一些参数的调整上每个人都参与进来，相比一两个人参与算力提高了不少，在中后期我们在代码结构、启发式规则、估值函数的选取等方面思路也因此非常开阔，对各种情况的利弊也都能够有一个大致的了解。

此外，我们所有人都在密切关注天梯动态。出现哪个组上传了特别强的代码就立马去和对方对战，观察棋局，寻找问题，这使得我们的代码能够一直跟着天梯版本走，在 5 月中旬曾一度进入天体前三，仅次于 RBQ 系列代码和古明地恋系列代码。

本组成立前期由于个人能力不同、分工不够明确、某些前期方案（如蒙特卡洛算法，存储搜索结果等）被否定，导致了在开发出初代代码后有一段效率比较低迷的时期。所幸我们

开工较早，后面通过交流，以及各位组员的包容理解，大家都在组里更加明确自己该干什么，找到了属于自己的工作和位置。或许，如果从一开始就交流充分的话，我们的效率和分工可以更加合理，也能让最后的结果更加令人满意。

使用一般的 minmax 算法时，关键在于调整估值函数，因此有大量的参数需要调整。由于不太会在程序中和网页交互，我们并没有选择使用程序进行代码测试，搜索最优参数的过程一直是人工完成的，导致非常耗费时间精力。由于本地的运行环境与网络上有较大差异，导致必要的调整得出的结果只有上传到网络平台才能得知其最终性能，这导致了大量的时间成本的浪费（等待匹配的过程中）。也许使用程序测试代码可以大幅提高中后期优化代码的效率，这一点是值得我们改进的。

代码方面，在与 F19 第一名的对战中，我们的代码以较大的劣势输掉。通过技术组同学在课堂上的透露，得知对方的估值函数十分复杂。事实上在估值函数的选取上我们虽然有各种不同的尝试，但是依然没有去尝试如平滑性等因素，可能是这样的原因让我们最终不敌对方（展示了 F19 同学的真正实力）。如果我们能够花更多的时间钻研估值函数和调整参数的话，或许可以取得更好的效果。

4.3 建议与设想

这次实习作业在竞技场基础设施、竞赛等方面技术组的同学、老师和助教们都已经做得很好了。

对于竞赛赛制我们有一点小建议，就是希望在比赛时加快进度，同时加入复活赛或者改为双败淘汰赛，使得比赛中的一些随机性因素的影响更小。这个随机性不光指比赛的随机序列，也指在淘汰赛中匹配的随机性。当然这只是一个美好的期待，毕竟一节课的时间很有限，也可以理解在现实条件下的赛制已经很合理了。

另外，此次代码天梯上在后期出现了这样一种现象：上传代码，和别人的代码测试对战，打完就删掉原来的代码的情况。这样固然可以防止自己的代码被对手针对，进行专门优化击败自己的代码，但是也影响了比赛的环境和良好的竞争氛围，使得在后期天梯上很难找到太多其他组的最新版小组代码来进行切磋测试。当然这也是小组的一种策略，但还是感觉这给后期代码的优化产生了一定的阻碍。希望明年老师能提出并改良一下这个问题，让大家都可以公平快乐的享受到这次比赛！

在这里也希望明年的学弟学妹可以享受这次大作业。不必将这看作一种任务或者是负担，几个人一起，想办法把一个游戏玩好，这难道不也是一件很嗨皮的事情吗？虽然在写代码、debug 的过程中会感到有一些枯燥，但是在你拥有了属于自己的 AI 之后，会觉得一切努力都是值得的。

5 致谢

感谢技术组的张宇昊等同学以及各位助教，耐心地解答我们关于规则的疑问，不断地完善竞技场代码，为我们带来了更好的对局体验。

感谢各位排行榜上的大佬们，你们的存在是我们进步的方向和动力。

感谢陈斌老师，您在本学期的课程的教授以及在直播课上对 minmax 算法的讲解，都对本组作品的产生和完善产生了十分重要的促进作用。

感谢这次大作业，让我们五个人有机会成为组员，共同完成这一有趣、艰巨，同时富有

意义的任务。

6 参考文献

minmax 算法：
http://web.cs.ucla.edu/~rosen/161/notes/minimax.html

αβ 剪枝：

https://www.hackerearth.com/blog/developers/minimax-algorithm-alpha-beta-pruning/
http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html/