同济大学计算机系

人工智能课程设计实验报告

学　　号 1951247

姓　　名 钟伊凡

专　　业 计算机科学与技术

授课老师 赵才荣

一.负责内容

我们第 15 组的成员是我、刘云帆、陈朝琛、赵艺博。我负责的内容和对小组人工智能象棋项目做出的贡献包括以下方面：

- 所有**搜索算法模块**的学习、研究、实现和改进
- 第二版象棋程序的**估值模块**
- 两版象棋中搜索算法模块和 UCCI 通信模块、估值模块、开局库残局库模块的**接口设计和实现**
- **维护本组代码的 Github 仓库**，解决 git push, git merge 中发生的冲突。
- 作为组长，代表整组向老师助教**询问进展中的疑惑**
- **协调组员分工**
- 在象棋 AI 表现不理想时继续仔细研究、发现问题，**精神上鼓励队友们**不屈不挠地改进象棋 AI
- **做出学术诚信的表率**，所有内容完全自己实现。

二.准备工作

本学期初，这个大作业一布置下来我立刻带领我们组的同学开始研究。我们发现象棋百科全书官网上有关于中国象棋 AI 非常全面的知识介绍，于是定好计划开展了学习。在那个时期，我们主要完成的工作是学习 UCCI 通信协议，学习基础的象棋博弈算法。

没过几周，助教又与我们分享了前届学长的冠军代码和实验报告以及汇报 PPT。我们又是如获至宝，认真阅读了学长们的代码，并对他们的代码结构进行了分析。从他们的报告中，我们理解了他们的实现思路，并对他们的制胜算法进行总结，排出了我们自己的实现顺序。同时，我们吸取他们优胜的经验教训，努力在自己的项目进展中进行避免。

三.第一版象棋程序

我们在四月初就开始着手实现我们组的象棋程序。第一版的计划是由赵艺博同学设计估值模块，我来实现相适应的搜索模块。我们的棋盘为 256 位一维数组，基类型为棋子种类的枚举。赵艺博设计的估值模块构思巧妙周到，通过当前局面的棋子种类、棋子数目、棋子位置、棋子相对位置、棋子灵活度和攻击性等多个方面给出局面的静态估值，能比较准确地给出棋局估值。但它的缺点是，时间复杂度和维护难度无法同时降低。如果想维护简单，则每次估值从头计算即可，但是时间复杂度为 O(n^2)；如果时间复杂度为 O(n)，则维护时需要时刻记录每个棋子的位置，对每方相同种类的几个棋子（如五个兵）还要进行分辨。于是，我们选择后者，实时维护两方每个棋子的位置，每次估值的时候遍历局面上这些棋子来完成估值。同时，由于 O(n)对于需要拼搜索深度的象棋博弈来说还是太长了，我们还实时更新棋盘的 Zobrist 哈希值，将计算过的所有棋盘局面存到置换表中，这样下次遇到相同局面就可以直接得出结果。也就是说，在这一版程序中，我的算法部分在搜索的过程中需要维护好棋盘的状态、当前双方每个棋子的位置状态、棋盘的哈希值、当前棋盘的估值……我们当时还觉得中国象棋有黑白对称和左右对称的哲学，因此还试图维护一个镜像棋盘哈希。

实现之后在调试中发现，由于带 alpha-beta 的 minimax 在很多个地方都会对棋局做出修改（搜索时、走棋时，等等），尽管我已经在所有可能的地方做出了我们认为完全合理正确的维护，但花费两周时间都未能真正正确维护好我们搜索类中的棋盘。其实从上面的叙述中已经可以感受到，要维护的实在有点多，赵艺博的估值、我的搜索、刘云帆的 Zobrist 哈希之间严重相互依赖。尽管我们做了大量的沟通，但是很可能仍有一些细小的地方我们对对方的实现有误解，从而导致维护上出了问题。

工作了比较长的时间，程序仍然出现大量 bug，全队心理上都有些失落。我仔细思考，认为当前代码就算把 bug 都查出来解决掉，表现出的智能也只是很初级的。而在此基础上再做优化恐怕是非常繁琐痛苦的事情。究其原因，主要在于估值函数考虑太全面，计算复杂、计算要求高。直接放弃原有代码对设计它的组员也是打击。于是，我请三位组员在原来的基础上继续 debug，我则先一个人重写搜索和估值，不加置换表，从头开始、从简单开始，实现一个有智能的象棋 AI，探索一条新的道路。后来发现比第一版可行很多，我们四人方才全部转到第二版上共同开发。

四.第二版象棋程序

1. 高内聚低耦合的设计

我们反思第一版失败的原因，认为主要在于各个模块之间耦合度太高。于是我在探索第二版时，努力做到不和其他模块之间有多余的联系。估值函数，就给估值。搜索完成搜索功能，估值地方尽量直接调用，不专门维护。置换表也仅仅是在搜索开始的时候搜一下看看有没有历史记录，搜索完成后进行一下更新。

有了这样的思路，这一版从设计上就比第一版要更加成功。

2. 估值函数

这一版，我们的估值函数是将棋子乘以它在该位置的权值相加作为总估值。
在设置初始棋盘的时候计算估值，在每一步的走法中只需根据移动的棋子进行对应的修改即可，O(1)时间能计算出来新的估值。排序时按照新局面的估值排序。

$$redValue = \sum pieceValue[pieceType[pos]]$$

$$blackValue = \sum pieceValue[pieceType[pos]]$$

$$value = redValue - blackValue$$

100	100	96	91	90	91	96	100	100
98	98	96	92	89	92	96	98	98
97	97	96	91	92	91	96	97	97
96	99	99	98	100	98	99	99	96
96	96	96	96	100	96	96	96	96
95	96	99	96	100	96	99	96	95
96	96	96	96	96	96	96	96	96
97	96	100	99	101	99	100	96	97
96	97	98	98	98	98	98	97	96
96	96	97	99	99	99	97	96	96

炮静态子力

206	208	207	213	214	213	207	208	206
206	212	209	216	233	216	209	212	206
206	208	207	214	216	214	207	208	206
206	213	213	216	216	216	213	213	206
208	211	211	214	215	214	211	211	208
208	212	212	214	215	214	212	212	208
204	209	204	212	214	212	204	209	204
198	208	204	212	212	212	204	208	198
200	208	206	212	200	212	206	208	200
194	206	204	212	200	212	204	206	194

车静态子力

9	9	9	11	13	11	9	9	9
19	24	34	42	44	42	34	24	19
19	24	32	37	37	37	32	24	19
19	23	27	29	30	29	27	23	19
14	18	20	27	29	27	20	18	14
7	0	13	0	16	0	13	0	7
7	0	7	0	15	0	7	0	7
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

卒静态子力

90	90	90	96	90	96	90	90	90
90	96	103	97	94	97	103	96	90
92	98	99	103	99	103	99	98	92
93	108	100	107	100	107	100	108	93
90	100	99	103	104	103	99	100	90
90	98	101	102	103	102	101	98	90
92	94	98	95	98	95	98	94	92
93	92	94	95	92	95	94	92	93
85	90	92	93	78	93	92	90	85
88	85	90	88	90	88	90	85	88

马静态子力

3. 着法生成

其思想是遍历当前走子方的每个棋子，根据不同棋子动作数组生成下一步可行走法，存到数据结构中（我使用了 vector），最后进行排序。

动作数组示例：

```
484  const int kingMove[4] = {-16, -1, 16, 1};
485  const int shiMove[4] = {17, -17, 15, -15};
486  const int shiMove1[8] = { -16, 17, 1, -17, 16, 15, -1, -15 };
487  const int xiangMove[4] = {34, -34, 30, -30};
488  const int maMove[8] = {-14, -18, 33, 31, 14, 18, -33, -31};
489  const int maMove_illegal[8] = {1, -1, 16, 16, -1, 1, -16, -16};
490  const int maMove_illegal_reversal[8] = {-15, -17, 17, 15, 15, 17, -17, -15};
```

4. 搜索算法，从简单开始，alpha-beta

第一版程序的开局库、搜索、估值、历史状态查找等部分耦合度太高，难以维护，组员一起调试很久仍然有不少问题。于是，我们另起炉灶，从简单开始，一步一步增添新内容，不断优化。

本着使象棋程序能跑起来的目标，我先实现了朴素的 alpha-beta 剪枝算法。

实现流程相对简单：着法生成、着法排序、带 Alpha-beta 剪枝的 minimax 搜索、最后以第一层的所有步骤中搜索后返回值最高的作为当前走法。这样写出来的能和傻瓜打一打。

5. 优化搜索，主要变例搜索算法（PVS）

对 alpha-beta 的进一步优化普遍选择 PVS，因此我马上研究了这个算法。

这个算法的前提是着法排序做的比较准，这样在求解了第一个着法的估值之后，就认为后面的都不如这个着法好。具体来说，就是把调用下一层搜索的 alpha 和 beta 的窗口区间设为 0 长度，增大剪枝的可能。由于证明一个着法坏（剪枝）比证明一个着法好（更新）一般要快不少，使用 PVS 就能提升不少效率。不过，如果新的着法并不坏，则需要对新的着法

以正常的窗口重新搜索。但一般重新搜索需要的时间比节省的时间要少，也就节约了时间。Alpha-beta 转 PVS 还是比较容易的，主要就加一个是否已搜到主要变例节点的判断，如果有的话使用零宽窗口搜索，没有或者搜出来的结果证明当前走法比主要变例走法更好，则使用正常的窗口宽度进行下一步的搜索。在代码实现中也就多加六行左右即可。

实践证明，这样反而变慢了许多。我们输出中间过程，发现经常重新搜索，也就意味着我们的着法排序做得不够好。故下一步选择优化着法排序。

6. 优化排序，吃子优先并适应性地调整走法估值

当前估值函数存在的问题：有些棋子的位置对权值的影响比吃子影响更大。

吃子走法一般是相对较优的，故我们将吃子走法排在前面，并按照吃的子种类先排序，再按权值排序。

表现好了一些，但还没完全变好，因为遇到了下面这个问题：能吃不吃，被吃不逃，走法很笨。

我分析了一下，原因如下。将吃子走法排到前面，仅仅优化了一点点排序，稍微增加了剪枝，但是对估值没有影响，因此没有完全解决之前说的“吃对方棋子权值上不如自己棋子好位置”的问题。那么相应的解决方法就是，调整估值函数计算来优化走棋。

红方，原估值：new_evaluation = original_evaluation – piece_value[chessboard[src]][src] + piece_value[chessboard[src]][dst] + piece_value[chessboard[dst]][dst]

红方，现估值：new_evaluation = original_evaluation – piece_value[chessboard[src]][src] + piece_value[chessboard[src]][dst] + piece_value[chessboard[dst]][dst] *** weight**

也就是说，加一个 weight 权重来使得吃子变得“受重视”。

经测试，weight 为 4 时表现不错，因此我们后面一直延续给吃子 4 倍权值。

能明显感觉到我们的象棋 AI 走得更加智能了。此时我们已经可以和菜鸟打一打了。

7. 优化排序，历史启发

历史启发是比较好的启发式排序方法。学期初我们学习了 A*搜索算法，它里面也用到了启发式排序方法。启发式一般不完全正确，但是可以比较好地反映事实，同时由于它计算简单，常常起到优化的效果。

历史启发的思想很简单。不管是什么棋子，只要历史上我方搜出来过这步，就认为这步不错，在表中这步棋对应的位置做相应记录。这个表称为历史表。以后生成走法之后，根据历史表中的记录先一步排序。

看起来，这个启发式很鲁莽，不讲道理，但仔细想是有一定道理的。历史表中的走法其实能比较好地区分不同棋子（比如有些走法只可能是马走出来），也符合下棋的实际情况。

8. 优化排序，杀手启发

在之前的搜索中产生过剪枝的走法称为杀手。由于这个走法能完成剪枝，它很可能是比较优的，因此杀手启发将这样的走法排在前面的位置上。

综合上面的多重排序方法，我将生成的所有走法按照下面的顺序安排（以红方为例）：长打长将着法放在最后，而且不选；首先，杀手着法优先；其次，历史启发表中积累的值较高的优先；再次，吃子而且吃的子较有威力的优先；最后，估值较高的优先。

实现出来测试表明，这样的排序方法表现较好，能把比较优的走法都排在比较靠前的位

置上，从而增加剪枝的出现频率。到这时，搜索速度还算比较快（五层搜索时间在两三秒），而且能确保较好地走法总是比较靠前地被找到，这时可以和象棋巫师软件的新手、入门有来有回了。

9. 优化算法，迭代加深

为了进一步增加搜索深度，我们进一步使用迭代加深搜索。

我们采用这个搜索算法的原因在于当时，我们搜 5 层，60 秒绰绰有余；搜 6 层，60 秒不够用。那么我们如果能确保浅层先搜出来结果，利用多余的时间进一步搜索更深层的话，时间利用比较充分，同时搜索结果也不会表现差。这就使我们确定要使用迭代加深算法进行优化。具体来说，先用几十毫秒搜三层，然后按照搜索出来的结果，重新排序进行更深一层的搜索。它的优点主要有两个方面：

- 前一深度搜索过后进行排序，能使得排序效果更好，更容易剪枝，搜索更深。而且很多之前搜过得到的结果可以直接重用。
- 解决之前经常遇到的“搜索 5 层时间富裕，搜索 6 层有时搜不完”的问题。能确保有 5 层搜索的结果，如果搜得更深就来更新，对时间利用率更高。

我们在搜索过程中输出层数，和象棋巫师的业余打，可以从中间输出中看到，浅则搜索 6 层，深则搜索到 10 层，优化效果不错。

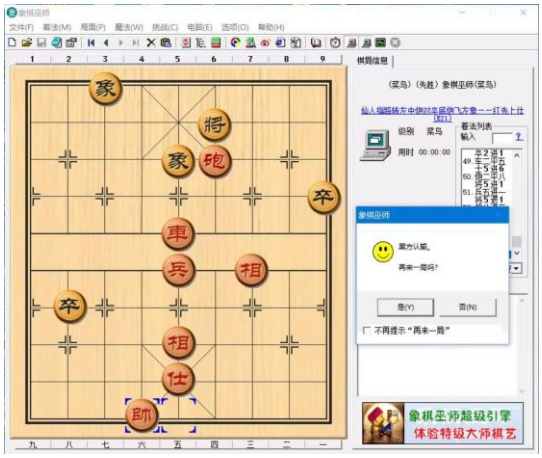
10.算法部分总结

对象棋百科全书上的所有算法，我们都进行了学习讨论。上面的算法，我都学习并掌握了。根据我们的数据结构和逻辑，我们实现了 alpha-beta，主要变例搜索，迭代加深，历史启发，杀手启发，置换表；没有应用静态搜索和空着裁剪。

没有实现的两个算法都很巧妙聪明，也符合下象棋的常识，没有加入我们的象棋中比较可惜。

我通过实现一个个算法，更进一步地理解了它们的思路，增强了算法的设计和理解能力。而且，我们在实现的基础上，还根据我们的实际需求、实际问题、实际数据结构进行了适应性地调整，对我们的象棋 AI 程序确实起到了增强棋力的作用，总体上非常有收获。

11.赛况记录

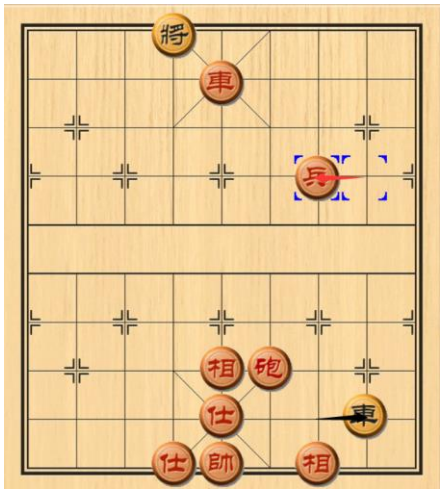


五.开发环境

Windows 10 操作系统，Visual Studio 2019 x86 Debug 和 Release。

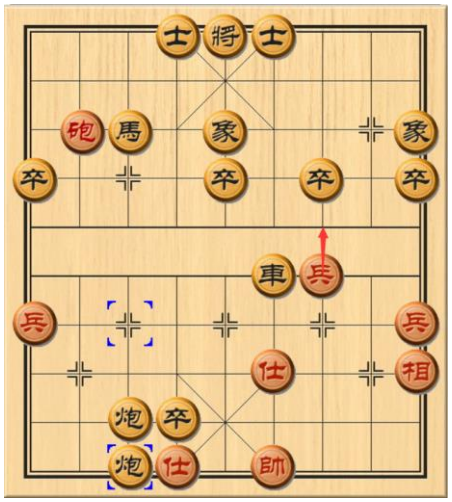
六.遇到的问题和解决方案

1. 长打长将



缺乏长打判断时，出现在自身优势时由于长打判负的情况，增加长打判断后即解决了该项问题。

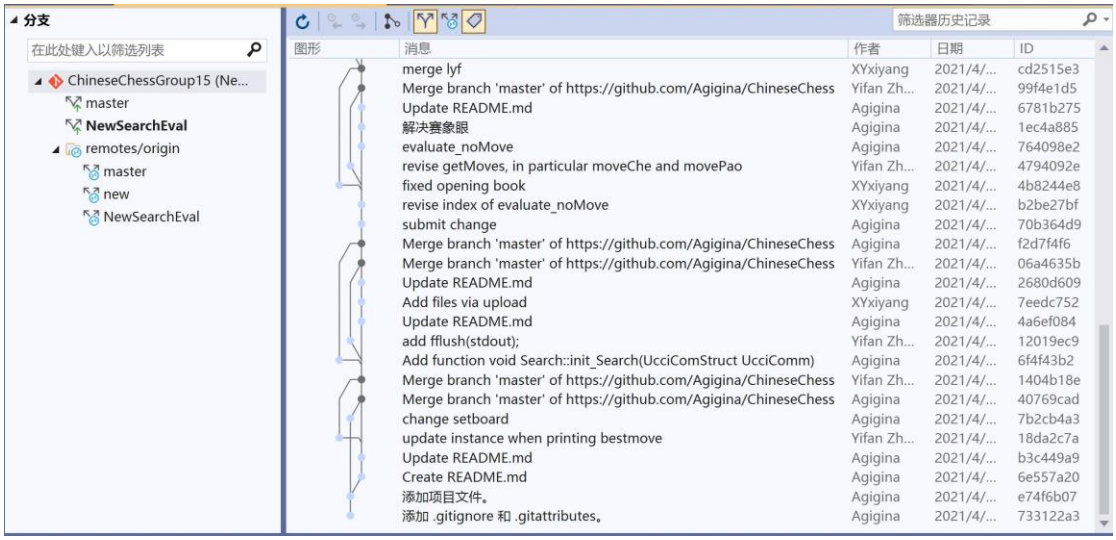
2. 不解将



```
position startpos moves g3g4 h7g7 b2e2 g9e7
e0f0 c9e7 e1f2 h2h3 e2d2 h3f3 f2e1 h9h0 f0f1 f
e3f3 f2e1 f3c3 e1f2 c2c1 f1f0 c3c0
go nodes 46656
info timeNeeded: 454
info judge time: 2
info getMoves time: 278
bestmove g4g5
```

实验过程中，出现了自己即将被将军，但我方引擎未解将的情况，出现该情况的原因在于经过多层搜索，判断解将操作最终也将导致失败，因此解将与非解将最终得到的搜索价值相等。

七.项目过程展示



前后共有

- 189 commits
- 4 branches
- 6 releases
- ……

我们还经常一起开发到很晚，在图书馆、电信楼、全家，都留下过比较难忘的团队开发记忆。

八.心得体会

这是一个令人难忘的大作业。开始做之前，感觉难度很大，无从下手。我们小组一起开发，齐心协力，集思广益，很快打开局面，取得了一定的进展。发现第一版的思路不好之后，我们立即退回起点带着教训重新出发，让我们达到了新的高度。

本次作业，我们小组四个人，共同完成了一个完整的象棋 AI 程序。这个程序能表现出来一定的智能，能击败象棋巫师的新手甚至业余水平的对手，算是对我们付出的三个月心血的回报。最最重要的是，我们组的两千多行代码，都是我们自己思考后原创而成的，因此感觉非常踏实，而且确实是很有收获。

我们组的象棋程序在算法上还可以再改进，比如加入静态搜索和空着裁剪，应该在预判对手走棋上会表现更好。此外，我们还可以多花时间进行调试，发现问题后针对性地进行优化；调整数据结构，使得存储效率和时间效率都达到最高。循环进行这些优化，棋力再上一两个等级应该是没有问题的。可惜这次虽然我们很早开始，也付出了大量的时间精力，但是这些还是没来得及完成，有些许遗憾。不过对比我们已经完成的努力，还是感觉基本上比较满意了。感谢这次作业，我们在 AI 博弈这个方向还会继续探索下去。