chessAI：六子棋人工智能程序设计报告

学号：20112020063, 20212020106

姓名：李铭涛，李嘉祥

邮箱：20112020063@fudan.edu.cn

20212020106@fudan.edu.cn

目录

1. 六子棋问题描述与形式化表示

按照题目设定，六子棋的游戏规则为，黑方先下一子，之后白黑双方轮流每次各下两子，先连成六子者获胜。棋盘大小为 15 乘 15。想要设计一个六子棋 AI 程序，首先需要将实际游戏中的各种概念转化为数学表述，并将其分解为明确的编程问题，如下所示。

棋盘：可以转换为一个 15 乘 15 的矩阵，矩阵中的每个元素位置代表一个棋盘格。为了表示每个棋盘格上的落子情况，我们规定元素为 0 代表没有落子、元素为 1 代表落黑子、元素为 2 代表落白子。因此每次一方落子之后，都可以视为对原本的棋盘矩阵进行了一次刷新。

棋型：指棋子在棋盘上可能组合成的各种情况。对于六子棋而言，根据互联网上的参考资料[1]，我们将其分为 15 种情况（六连，七连，活五，眠五，死五，活四，眠四，死四，活三，朦胧三，眠三，死三，活二，眠二，死二）。在上述的这些术语中，“活”指对方需要两手棋才能挡住，“眠”指对方用一手棋就可以挡住，“死”指对方已经挡住。而“朦胧三”作为一种特殊情况，指的是我方再下一手棋就能形成眠四，再下两手棋就能形成活五。真实的六子棋棋局中，横竖撇捺四个方向都可能出现上述棋型，由此可见，每种棋型实际上可以转换为一个特定的向量，以列表的形式存储。列表中的元素排列，对应的就是一种棋型的具体落子情况。

敌我方的最优决策：每一个棋局都分为我方和敌方两大阵营。对于我方而言，每下一步棋都需要依据此时的棋盘矩阵情况，作出最优的决策。所谓最优决策，往往由**进攻**和**防守**两部分构成。进攻指的是尽可能让我方的棋子更具威胁，更容易连出六子；而防守指的是尽可能降低敌方的棋子带来的威胁，不让敌方连出六子。不同的棋盘矩阵情况，对应着不同的最佳策略，更偏向于进攻或防守要视实际情况而定，并无定论。在本程序中，我们假定敌方选取最优策略的思路与我方是一致的（如果针对某个特定对手，也可以选择通过训练学习对方的下棋风格）。

算棋：指下棋时预测推演接下来几步的发展情况，从而帮助棋手在此刻作出最优决策。在人类下棋时，算棋是一项十分重要的技巧，强大的算棋能力是老手和菜鸟最显著的区别。理论上讲，算棋能力越强的一方，越容易获得最终的胜利。而想要在程序中实现算棋，就意味着该程序不能简单的采用贪心算法，而是需要考虑在下完当前这步棋后，对手针对新的棋局情况会怎样进一步落子，并结合推演中的未来棋局形势倒推当前这步棋的最优选择。

除了以上这些与下棋直接相关的部分，题目还对交互模式提出了要求。该程序需要能仿真一名真正的棋手，在确定自己执棋的颜色后，完整下完整个棋局。其中每下一步棋（落两子），该程序需要读取敌方的 csv 文件（存储了敌方最新下的棋），并输出我方的 csv 文件（存储了我方最新下的棋）和 log 文件（存储了对于棋局的日志说明）。为了提防对手出现篡改、超时等违规行为，该程序还需要能判断违规和胜负。最后，题目还要求本程序能实现自我对弈，这又对很多交互上的细节实现提出了更高的要求。

综上，本次题目所要求实现的六子棋人工智能程序设计，可以分为以下三类编程问题。

（1）**流程实现部分。**包括但不仅限于：如何将 csv 文件中的数据读取到程序中，并输出 csv 文件和 log 文件；如何针对棋盘矩阵打印出整个棋局，并对棋盘矩阵进行刷新；如何判断胜负和违规情况；如何让主程序只在对方下出新一步棋后才运行下一次落子程序，等等。

（2）**判断棋局情况部分**。该程序必须能够以量化的方式，判断当前局势下自己的得分、自己下了某一步棋后的增益情况、对手下了某一步棋后的减损情况等等。这一部分程序设计

的血肉在于能够遍历整个棋盘的“横竖撇捺”四个方向上的各种棋型情况，并计算出累加得分；但灵魂在于对于不同棋型对应的分数权重设置。后续实验证明，不同的棋型分数权重设置方案，将很大程度影响最终程序的下棋策略选择。

(3) **生成最优解部分**。该程序需要能基于当前的棋盘矩阵，给出我方的最优解，即最优的落子方案。这部分程序设计不仅要能实现贪心算法，即根据当前棋盘矩阵找出所有落子方案中增益最大的局部最优解；还要能实现算棋功能，即搜索找到敌我交替延展几轮后我方的最优局势，并以此情况倒推当前的真正最优解。此部分不仅需要在精度和成本之间做好取舍，还需要避免在推演未来的过程中忽视了当下的决胜步或致命步。因此这部分的算法与程序设计，是整个程序设计中最为复杂的部分。

2. 程序设计流程和实现方法

本程序采用模块化设计思路，整体代码由一个主程序与 7 个子模块组成，它们各自的功能概述如表 1 所示。本章节中，我们会详细介绍前 5 个模块，因为这些模块涉及该程序核心功能的实现；而后 3 个模块内容较为简单，不在此展开，可在附件程序中查看。

模块名	实现功能概述
1. chessAI	主程序，规定了程序以怎样的流程下完一局棋。
2. Evaluator	对于不同的棋盘情况，或不同的落子情况，给出打分评价。
3. Generator	基于当前的棋盘矩阵，使用贪心算法找到我方当前最优的两步棋。
4. Branch	使用基于博弈树极大极小值的 alpha-beta 剪枝搜索算法，在给定搜索宽度与深度后，求出最优的两步棋。
5. FoulAndWin	判断犯规情况和胜负情况。
6. CSVtransform	将CSV文件中存储的时间、步数、落子位置等数据，读取到程序中。
7. UpdateChess	在给定落子位置后，对棋盘矩阵进行刷新。
8. PrintBoard	在给定棋盘矩阵后，打印出对应的棋盘。

表 1: chessAI 程序各模块功能概述

2.1. chessAI 程序设计与实现方法

chessAI 模块作为主程序，其流程如图 1 所示。使用 Haskell 完成此流程的设计时，出现了以下几个技术问题，我们逐一将其解决。

(1) **实现指令的顺序执行。**Haskell 作为一门函数式编程语言，从编程思路上是不支持像命令式语言一样顺序执行指令的。不过在 main 函数中，在 do block 内可以实现多个指令的顺序执行。

(2) **实现部分指令的循环执行。**本程序中，根据当前棋局和对未来的推演，决定此刻的最优落子方案，这一步骤会频繁的执行，因此需要循环此部分的代码块。然而 Haskell 中并没有一般语言常见的 for, while 循环，函数内的循环往往通过递归来实现。我们通过在 main 函数中调用一个 mainplay 函数（此函数是需要被循环的代码块），并在 mainplay 函数顺序执行到末尾时重复调用它自己，完成了循环执行。

(3) **在违规或获胜时退出循环，结束程序。**每次循环刚开始时读取敌方的 csv 文件中的数据，调用 FoulAndWin 模块中的函数判断对方是否出现了超时、篡改等违规情况，若出现的话会直接以 error 的形式结束主程序的运行。此外，每次循环到末尾时可以判断是否有

某方连出了六子，如果有的话以 return ()的形式退出循环程序的运行。

（4）**时刻访问敌方的 csv 文件，只在其发生变化时真正执行循环程序。**本程序需要在对手落子后执行 mainplay 程序，但对手什么时候落子是未知的，因此程序必须能时刻访问敌方的 csv 文件，并只在该文件发生变化时（即对手落子时）执行 mainplay 程序。我们的解决方案是给 mainplay 函数添加一个存储上一时刻敌方 csv 文件内容的输入变量 oppoContent，同时每次循环程序开始时会读取当前时刻敌方 csv 文件的内容并记录为 oppoContentNew。若 oppoContent=oppoContentNew，则无需执行后续内容，直接进入下一次循环；否则意味着敌方有新落子，此时再执行后续内容。

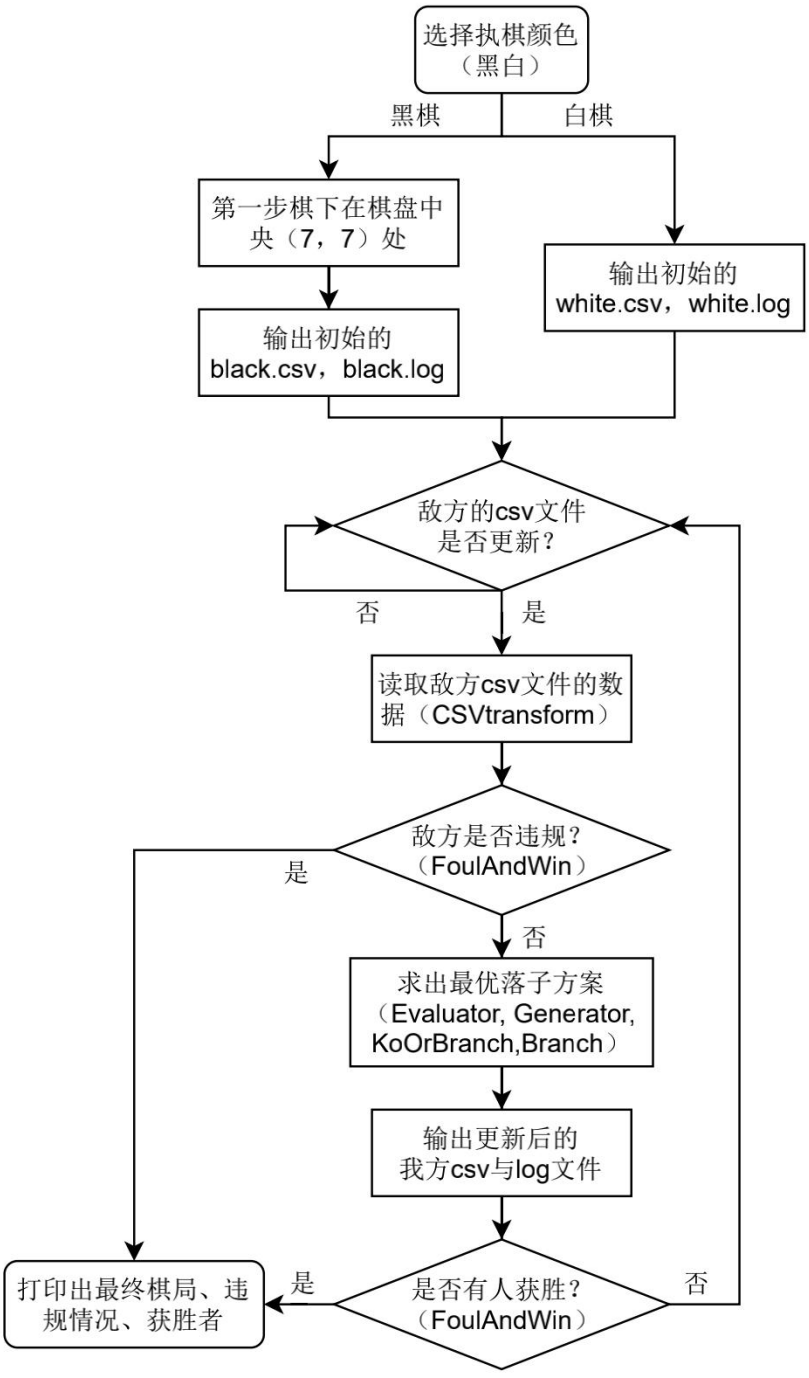


图 1: chessAI 主程序流程图

2.2. Evaluator 程序设计与实现方法

Evaluator 模块中定义的两个关键函数是 **evaluateBoard 函数**与 **evaluate1Move 函数**。顾名思义, evaluateBoard 函数在输入一个棋盘矩阵时, 能给出此时黑棋/白棋的得分; evaluate1Move 函数在输入一个棋盘矩阵和一步落子时, 能给出这一步落子的得分。这两个评估函数在求最优解的过程中会被频繁调用, 因此它们性能优越度将直接决定整个程序的性能好坏。

本小节我们将会从棋型分数设置, matachListWithType 和 evaluate1Row, 四方向遍历三个部分展开介绍。

2.2.1. 棋型分数设置

根据第一章的介绍可知, 六子棋中的每一种棋型, 都可以对应一个列表。例如, 黑棋的六连对应[1,1,1,1,1,1,1], 白棋的死四对应[1,2,2,2,2,1]。每种棋型对应的情况可能十分复杂, 就不在此处一一列举, 细节设计可以在附件程序中的 Evaluator.hs 中查看。

想要获得一个准确客观的评估函数, 从而指导程序对于不同棋局的好坏判断, 其根基在于对 15 种棋型的分数设置。我们经过大量实验调整, 最终的棋型分数设置方案如表 2 所示。

棋型	分数	棋型	分数
六连	60000000	活三	2000
七连	60000000	朦胧三	500
活五	80000	眠三	100
眠五	70000	死三	0 或-1000
死五	0 或-10000000	活二	10
活四	70000	眠二	5
眠四	65000	死二	0 或-10
死四	0 或-10000000		

表 2: 棋型分数设置方案

由表 2 可见, 随着棋型变长, 分数的量级也会显著提升。值得注意的是, 对于死五、死四、死三、死二这几种情况, 分数出现了 0 和负数两种可能。这样设计的原因是, 在我们的多次实验中, 我们发现对于敌方的死棋的情况添加负增益, 可以显著提升我方下棋时的防守意愿（把对面堵死的意愿）。然而对于我方的死棋, 应当视作增益为 0（不会对棋局产生后续影响）, 不然会对棋局的整体判断带来许多干扰。因此我们分数设置时规定, 对于四种死棋的情况, 如果是敌方的死棋, 视为对方的负增益（我方的正增益）; 如果是我方的死棋, 则应该视为零增益。

本次程序设计中, 分数的设置都是根据多次试验手工调节的, 肯定并非最优分数设置策略。如果未来有更多精力和能力的话, 可以尝试用一些机器学习算法对分数权重进行优化, 这样应该会进一步提高评估函数的性能。

2.2.2. matachListWithType 和 evaluate1Row

对不同棋型设置好分数后, 我们就可以开始评估棋盘中的一行 (横竖撇捺任意方向都可) 的得分是多少, 这就是 evaluate1Row 函数要实现的功能。该功能的实现思路是, 将棋盘中的这一行与 15 种棋型分别进行匹配, 将出现的棋型分数累加即可。由此可见, 此处程序设计

的核心在于，如何将棋盘中的一行与棋型进行匹配。

我们设计了 matachListWithType 函数解决了棋型匹配问题。举例说明，活五的一种可能性为[0,1,1,0,1,1,1,0]，此棋型由 8 个元素构成。而假定我们取横向某一行进行棋型匹配，我们需要先提取该行的前 8 个元素组成列表，与[0,1,1,0,1,1,1,0]进行匹配，如果匹配成功则计数加 1。然后我们从该行的第 2 个元素开始依次提取 8 个元素组成列表，再进行匹配，以此类推。最终该函数可以输出在选定的这一行中，存在几个特定的棋型。

evaluate1Row 函数可以将棋盘中的一行，与 15 种棋型分别进行匹配，然后将对应的分数进行累加，即可得到这一行的评估分数。然而这种累加方式将引入大量冗余计算，例如，对于只有 3 个黑子的一行，根本不需要对四子及以上的情况进行匹配。为了提高计算效率，我们设计的 evaluate1Row 函数会先对输入的列表进行指定颜色的落子数量统计。例如，在评估某一行黑棋的得分时，如果该行只有 2 个黑棋，则三子及以上的情况无需判断；若有 3 个黑棋，则四子及以上的情况无需判断，以此类推。这样微小的设计调整，在 evaluate1Row 函数被反复调用的情况下，能极大程度优化整体程序的运行速度，并且不带来任何精度损失。

2.2.3. 四方向遍历

从 evaluate1Row 函数对一行进行评估，到 evaluateBoard 函数对整个棋盘进行评估，需要做的是对棋盘的“横竖撇捺”四个方向进行遍历以及分数累加。由于 Haskell 中并没有直接提供数组和矩阵等数据结构，本程序在对棋盘矩阵进行存储时采用的是[[Int]]数据类型，这为我们四个方向的遍历带来了一定难度。

其中横向遍历最简单，因为直接提取第 i 个元素就代表着提取了第 i+1 行；纵向遍历也较为简单，可以通过列表生成式直接提取。撇和捺这两个方向的遍历比较复杂，我们的解决思路是设计专门的递归函数，能在给定某初始位置后，依次取出“撇”方向（横坐标+1，纵坐标+1）和“捺”方向（横坐标-1，纵坐标+1）的所有元素，将其整合为一个列表。

实现四方向遍历后，对于 evaluateBoard 函数来说，需要对 15 乘 15 的棋盘矩阵的所有方向进行遍历。易算得，横方向 15 个列表，纵方向 15 个列表，撇方向 29 个列表，捺方向 29 个列表，共计 88 个列表需要被 evaluate1Row 函数打分，而分数累加后就是输入的棋盘矩阵的最终得分。

而 evaluate1Move 函数只需要评估某一步棋带来的增益，此增益虽然可以用下棋前后的棋盘得分相减而直接得出，但这意味着评估一步需要对 176 个列表进行打分，十分耗时。而实际上，每一步棋只会对落子所在位置的横竖撇捺四个方向产生影响，因此我们设计的 evaluate1Move 函数只需要计算落子所在位置的横竖撇捺四个列表的前后增益即可，如图 2 所示。通过这种设计，评估一步棋的增益只需要对 8 个列表（下棋前的 4 个方向+下棋后的 4 个方向）进行打分，而不是 176 个。

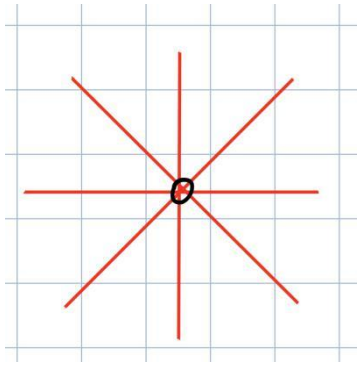


图 2：落子所在位置的“横竖撇捺”四方向

2.3. Generator 程序设计与实现

Generator 模块的核心在于构建一个 generator 函数。该函数需要实现的功能是，输入一个棋盘矩阵以及我方的执棋颜色，能够输出贪心算法下当前的最优落子方案，即为我方带来最大增益的两步棋。显然，generator 函数会大量调用上述的 evaluate1Move 函数。

本小节我们将从算法描述，程序实现两个部分展开介绍。

2.3.1. 算法描述

若我们要解决的是五子棋问题，那么寻找当前最优落子只需要两步：

（1）根据当前的棋子分布，确定下一步棋的可选落子空间；

（2）尝试可选落子空间中的所有情况，用 evaluate1Move 函数计算每个落子位置的增益，选择增益最大的位置作为最优落子位置。

然而本次要解决的六子棋问题的特殊性在于，每次需要落两子。如果依旧像五子棋的解决思路一样，先选定一个可选落子空间，然后尝试其中所有的两步组合，那么计算量会随着棋局的进展显著增加。例如，若可选空间的落子位置数为 10，则两步组合的数量为 45；若可选空间的落子位置数为 30，则两步组合的数量为 435。这样的计算开销，在棋局后期是无法承担的。

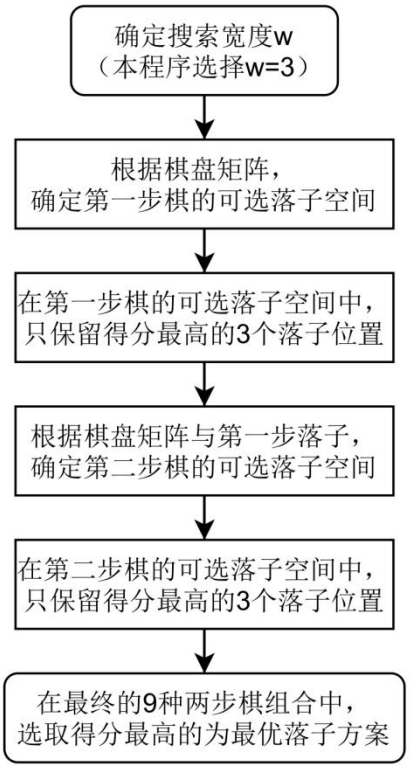


图 3: generator 求解最优落子方案算法流程

我们最终采取的算法流程如图 3 所示。此算法可以简单理解为，重复了两次五子棋中的最优落子寻找过程。第一步找出了 3 个最优解，即得分前三的 3 个落子位置；第二步基于第一步的 3 个最优解，分别再去寻找第二步的 3 个最优解。因此，最终得到 3×3=9 种两步组

合，如图 4 所示。而从这 9 种组合中挑选增益最大的一种组合，可以作为最优落子方案。

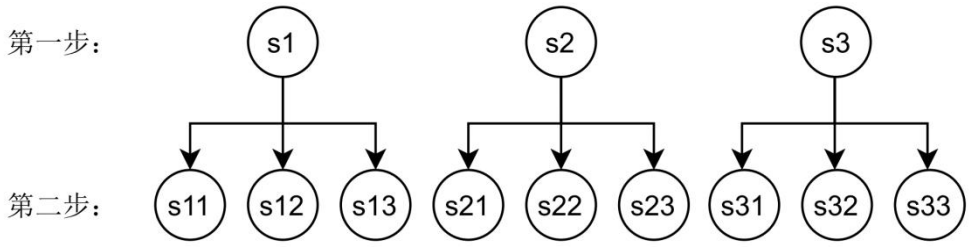


图 4: generator 算法生成的 9 种组合

此算法中的两个核心步骤分别是，寻找可选落子空间和在空间内寻找最优解。我们设定当前棋局中所有棋子相邻的空格为可选的落子空间，设定前 w 个最优解为得分最高的 w 步，从而实现算法的核心步骤。

2.3.2. generator 程序实现

在实现 generator 函数时，出现了以下几个技术问题，我们逐一将其解决。

（1）**构建可选落子空间时，要避免出现棋盘越界情况。**在我们的初版代码调试时，总会在棋局进展到后期时出现“Index too large”或“Index negative”的报错，后来经排查发现是在构建可选落子空间时，我们设定当前所有棋子位置相邻处都可以作为落子空间，但如果棋子本身处于棋盘边缘，那么就会出现索引访问错误。为了解决此问题，我们对于初始求得的落子空间用 filter 函数过滤掉了坐标值为负或坐标值大于 15 的位置，从而有效避免了棋盘越界情况的发生。

（2）**构建第二步的可选落子空间时，棋盘矩阵需要进行更新。**第二步的可选落子空间基本与第一步一致，唯一的区别在于第一步最终的落子位置不再是空位，而且该位置周边可能拓展出一圈可选落子位置，此处细节需要在程序设计时注意。

（3）如何选取前 w 个最优落子位置。我们让程序先计算出所有可选落子位置的得分，然后将位置和对应的得分捆绑为一个 tuple，再对 tuple 组成的列表进行得分从小到大的排序。这样构建出的列表的最后 w 个元素，就是 w 个最优落子位置。

2.4. Branch 程序设计与实现

Branch 模块的核心在于，使用基于博弈树极大极小值的 alpha-beta 剪枝搜索算法，通过决策树这一数据结构，以算棋的方式推理当下的最优落子方案。在此过程中，将会调用上述的 generator 函数与 evaluateBoard 函数。

本小节我们将从算法描述，程序实现，koOrBranch 三部分展开介绍。

2.4.1. 算法描述

本程序构建了六子棋博弈树图来对未来棋局情况进行推演，如图 5 所示。该决策树部分主要运用了两种算法：最大最小值算法和 Alpha-Beta 剪枝算法。下面将先阐述算法的基本思想，之后分别阐述两种算法的含义和作用。

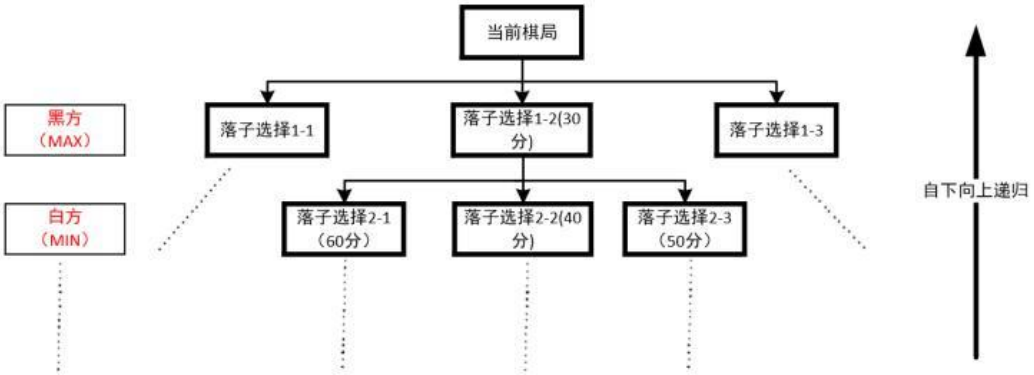


图 5: 六子棋博弈树图

整个博弈树是对未来棋局的一种形象化表达，每一个节点的值代表了到目前为止，当前棋局的得分，对于黑方（先手方）来说，我们取正向得分，也就是目前棋局得分越高，那么对黑方越有利，对于白方来说，我们取负向得分，也就是目前棋局得分越低，对白方越有利，每个节点的得分是根据最大最小值算法进行决定的，该算法隐形地表示了整个算法是一个深度优先的算法，最终决定的落子位置应该是迭代到顶层的 N 个节点的中得分最高的那一步。

（1）最大最小值算法

该算法的含义是指：对于黑方来说，永远取同层 N 个节点的最大值，这样对于黑方最有利。对于白方来说，永远取同层 N 个节点的最小值，这样对白方最有利，从某种程度上说，该算法是以一个最保险思想进行的思考，因为无论对于黑白来说，他们考虑的永远都是对方的下一步会下出最优的落子。

（2）Alpha-Beta 剪枝算法

该算法的目的是为了缩减搜索空间，提高搜索效率，同时该算法是一个安全算法，对最终选出来的结果没有任何影响。Alpha-Beta 算法的含义是指，对于黑方来说每一层永远取的是最大值，假设目前得到了一个最大值 X，那么在对下一个相邻节点 B 的下一层深度也就是白方进行节点值的计算时，如果找到了一个节点的值为 C 小于 X，那么后面的节点对应的分支就不用进行计算了，因为白方永远取得是最小值，因此 B 节点的值只会小于 C，也一定会小于 X，因此后面的分支计算就变得没有意义，因此应当舍弃。

2.4.2. Branch 程序实现

（1） calculateScore 函数

作用：算出选取最好的一步棋的在考虑了之后的 4 步后的得分，如图

伪代码：

```
calculateScore:
if deep ==4 =目前的棋盘分数
else if deep /= 4 &&  目前为黑棋  =  选取下面一步的三种选择的最大分数
else if deep /= 4 &&  目前为白棋  =  选取下面一步的三种选择的最小分数
```

（2）DG2 函数

作用：算出来待选择的三步最好棋的各自的分数

伪代码：

```
if a < 3 =  本次得分 ：DG2New
else = [ ]
```

(3) findIndex 函数

作用：返回最好的一步棋的坐标

伪代码：

```
A =  最好的一步棋的分数
B =  待选择的三步棋的分数
C = A  在  B  中的索引
D =  最好的三步棋的坐标
Return D ！！  C
```

(4) alpha beta 剪枝函数

由于剪枝是一个优化算法，比较简短，就直接包含在了递归函数的代码中，因此这里没有放伪代码只给作用：

Alpha-beta 剪枝主要是为了减少函数递归搜索的空间，从而减少决策树的生成时间。

2.4.3. koOrBranch 功能与程序实现

原理上讲，chessAI 程序每一次落子都应该调用 Branch 模块，并以该函数的输出作为最优落子方案。然而在一些情况下，程序的落子方案是固定的，必须这么走没有其他选择，我们将这种情况称为 ko 步。ko 步分为两种：第一种是进攻上的 ko 步，即下完后我方可以连为六子；第二种是防守上的 ko 步，即不这样下的话敌方下一步会连成六子。

而我们在大量实验中发现，在该走 ko 步的时候，Branch 模块输出的最优落子方案却未必是 ko 步，这就会导致我方错过绝杀机会，或惨遭敌方绝杀。为了避免此情况发生，我们创新性地设计了 koOrBranch 函数。该函数会判断 generator 函数输出的最优解的分数值是否高于 5000000。由于在表 2 的分数设置方案中，只有让我方形成六连七连，或让敌方形成死五死四，才可能让分数大于 5000000，因此在这些情况下 chessAI 将不会进行后续 Branch 模块的调用，而是会直接以 ko 步作为最优方案。而在分数低于 5000000 时，才会以 Branch 模块的输出作为最优落子方案。

因此，结合 koOrBranch 函数，chessAI 求解最优落子方案的整体流程如图 6 所示。

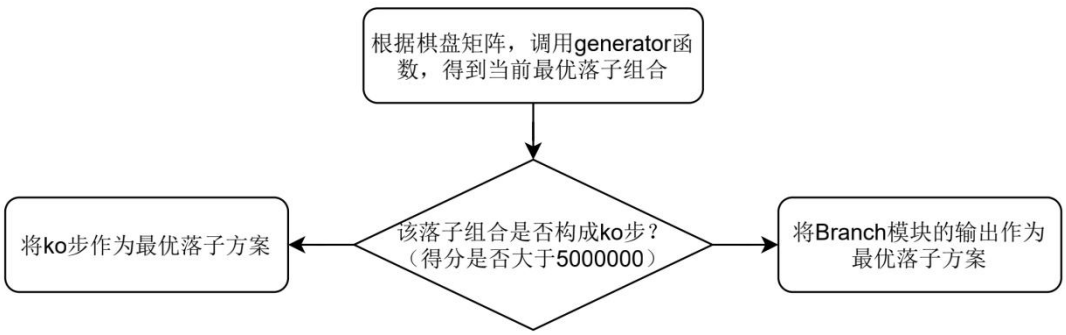


图 6: chessAI 求解最优落子方案的整体流程

2.5. FoulAndWin 程序设计与实现

FoulAndWin 模块需要实现判断胜负和判断犯规两类功能。判断是否有某方获胜，只需要判断黑棋和白棋是否出现连成六子或七子的情况；而判断犯规时，一方面需要判断对手是否修改棋盘，另一方面需要判断对手是否犯了多下一子或者超时这类常规错误。

下面是对一些关键函数的解释。

2.5.1. 判断胜负

(1) bl×××函数

作用：用来遍历最后落子之处 8 上下左右，右上，右下，左上，左下 8 个方位连续有子的数量。

伪代码：

```
blXXX:
（x,y）的 XX 方向下一个地方如果有子的话=1+blXXX（x±1，y±1）;
（x,y）的 XX 方向下一个地方如果没有子的话=0;
```

(2) Judgement 函数

作用：用来判断某个子的上下左右，右上，右下，左上，左下 8 个方位是否有连续 6 个子

伪代码：

```
Judgement:
If (x,y)的横向有 6 个子‖（x,y）的纵向有 6 个子‖（x,y）的右斜向有 6 个子‖（x,y）的左斜向有 6 个子 =“you are winner”
else =” you are loser”
```

2.5.2. 判断犯规

(1) findFault 函数

作用：判断对手是否修改棋盘

伪代码：

```
findFault:
A = 旧棋盘序列 – 新棋盘序列
If 其中不为 0 的个数不等于 2
    返回 1—说明对手下棋违规
Else
    返回 0—说明对手下棋没有违规
```

(2) judgementForgetOneHandB 函数

作用：用来判断黑子在走棋中少下一子

伪代码：

```
judgementForgetOneHandB:
if a==1 && length black_pos[a] == 1=0+ judgementForgetOneHandB(a+1)
if a==1 && length black_pos[a] != 1 = 1+ judgementForgetOneHandB(a+1)
if a!=1 && length black_pos[a] == 2 =0+ judgementForgetOneHandB(a+1)
if a!=1 && length black_pos[a] != 2 =1+ judgementForgetOneHandB(a+1)
else = 0
```

(2) judgementForgetOneHandW 函数

作用：用来判断白子在走棋中少下一子

伪代码：

```
judgementForgetOneHandW:
if   length white_pos[a] == 2 =0+ judgementForgetOneHandW(a+1)
if   length white_pos[a] != 2 =1+ judgementForgetOneHandW(a+1)
else = 0
```

(3) udgementFirstHand 函数

作用：用来判断白棋是否抢了先手

伪代码：

```
judgementFirstHand:
if time_white[0] <= time_black[0] =1
else =0
```

(4) judgementWinStep 函数

作用：用来判断棋手用了几步赢了

伪代码：

```
judgementWinStep
if judgement pos a ==  “you are loser”  = 1+ judgementWinStep a+1
else =0
```

(5) judgementUtterlyDiscomfitedW 函数

作用：用来判断白方是否在黑方已经胜利的情况下仍然在下棋

伪代码：

```
judgementUtterlyDiscomfitedW
if length time_white >= judgementWinStep black= 1
else =0
```

(6) judgementUtterlyDiscomfitedB 函数

作用：用来判断黑方是否在白方已经胜利的情况下仍然在下棋

伪代码：

```
judgementUtterlyDiscomfitedB
if length time_white >= judgementWinStep white = 1
else =0
```

(7) udgementThinkTooLongW 函数

作用：用来判断白方是否下棋超时

伪代码：

```
judgementThinkTooLongW
if a < length time_white && time_white[a] – time_black[a] >5 …
=1+ judgementThinkTooLongW a+1
If a>length time_white && time_white[a] – time_blackj <=5…
=0+ judgementThinkTooLongW a+1
else = 0
```

(8) judgementThinkTooLongB 函数

作用：用来判断白方是否下棋超时

伪代码：

```
judgementThinkTooLongB
if a < length time_black && time_black[a] – time_white[a-1] >5 …
=1+ judgementThinkTooLongB a+1
If a>length time_black && time_black[a] – time_white[a-1] <=5…
=0+ judgementThinkTooLongB a+1
else = 0
```

3. 功能测试与鲁棒性检测

3.1. 功能测试

本程序的测试环境为 mac os 的 terminal。运行 chessAI 的方法为，先在 terminal 中输入“ghc --make chessAI”，对所有代码进行编译；然后输入“./chessAI”，运行程序。

我们采取的测试方法为，在包含 chessAI 源代码的路径下，同时打开两个 terminal 分别运行 chessAI 程序，设置一方为黑棋，另一方为白棋。此时一方输出的 csv 文件，会成为另一方的输入，从而实现程序与自己的交互对弈。该过程详见附件中的“chessAI 运行视频演示”。

测试结果显示，该程序可以实现自我对弈，在 8 个回合后黑棋会获得胜利。

3.2. 鲁棒性检测

本文先针对以下几种常见的违规情况进行了检测。

(1) 先手违规
(2) 黑方多掷一子
(3) 白方多掷一子
(4) 黑方无视白方胜利
(5) 白方无视黑方胜利
(6) 黑方思考超时
(7) 白方思考超时

以上 7 类鲁棒性测试的结果如图 7.1 到图 7.7 所示，测试成功。

```
chessGame.hs: white steal the first move
CallStack (from HasCallStack):
  error, called at ./FoulAndWin.hs:135:44 in main:FoulAndWin

[Done] exited with code=1 in 1.216 seconds
```

图 7.1. 鲁棒性测试（1）

```
chessGame.hs: Too much chess for one step!!!
CallStack (from HasCallStack):
  error, called at ./CSVtransform.hs:54:26 in main:CSVtransform

[Done] exited with code=1 in 0.971 seconds
```

图 7.2. 鲁棒性测试（2）

```
chessGame.hs: Too much chess for one step!!!
CallStack (from HasCallStack):
  error, called at ./CSVtransform.hs:54:26 in main:CSVtransform

[Done] exited with code=1 in 0.971 seconds
```

图 7.3. 鲁棒性测试（3）

```
chessGame.hs: black? you lose your mind
CallStack (from HasCallStack):
  error, called at ./FoulAndWin.hs:139:53 in main:FoulAndWin

[Done] exited with code=1 in 1.081 seconds
```

图 7.4. 鲁棒性测试（4）

```
chessGame.hs: white? you lose your mind
CallStack (from HasCallStack):
  error, called at ./FoulAndWin.hs:138:53 in main:FoulAndWin

[Done] exited with code=1 in 1.074 seconds
```

图 7.5. 鲁棒性测试（5）

```
chessGame.hs: black, you have used out of time
CallStack (from HasCallStack):
  error, called at ./FoulAndWin.hs:141:50 in main:FoulAndWin

[Done] exited with code=1 in 1.072 seconds
```

图 7.6. 鲁棒性测试（6）

```
chessGame.hs: white, you have used out of time
CallStack (from HasCallStack):
  error, called at ./FoulAndWin.hs:140:50 in main:FoulAndWin

[Done] exited with code=1 in 1.09 seconds
```

图 7.7. 鲁棒性测试（7）

此外，还有一类特殊的犯规情况就是对手篡改了棋盘，此时需要调用 findFaultIO 函数对原本的棋盘和新棋盘进行核对，如果不一致则会报错，如图 7.8 所示，测试成功。

```
*FoulAndWin> findFaultIO [[0,1],[2,0]] [[1,1],[2,0]]
"*** Exception: opponent break the rule!
CallStack (from HasCallStack):
  error, called at FoulAndWin.hs:149:30 in main:FoulAndWin
```

图 7.8. 鲁棒性测试（8）

4. chessAI 创新点与 Haskell 编程心得

本次大作业不仅让我们熟练掌握了如何使用 Haskell 语言编程解决实际问题，还让我们对于六子棋人工智能问题有了更深的了解，可谓受益匪浅。

本小节我们将首先回顾本程序设计中的一些创新点，然后梳理总结下完成作业过程中对于 Haskell 编程的一些心得。

4.1. chessAI 创新点

本次程序设计中，我们使用的核心算法是基于博弈树极大极小值的 alpha-beta 剪枝搜索算法，与网络上的大部分参考资料给出的实现方案基本类似。然而在具体程序设计过程中，我们的 chessAI 程序具有以下几个创新点。

（1）**对死棋情况的打分进行分类处理。**在评估函数中，如果落子导致敌方出现死棋，则敌方得负分，也就是我方的正收益；而对于我方来说，已有的死棋视为 0 分，不会对局势产生影响。通过这样的设计，既能让程序更主动地去防守从而减少被对方绝杀的概率，又能

避免在评估己方局势时引入巨大负分干扰正常判断。

（2）**evaluate1Row 函数减少了冗余的计算量。**在求解最优落子方案时，将频繁调用 evaluate1Row 函数对某一行进行打分，因此这一函数必须做到足够精炼，没有冗余计算。我们先统计该行中目标颜色棋子的数量，然后只计算小于等于此棋子数量的各种棋型，从而大幅度减少了冗余计算量。

（3）**koOrBranch 函数保证程序能下出 ko 步。**在一些情况下，程序必须下在固定的位置，因为这些位置能让我们直接胜利，或者避免让对方直接胜利。我们将这些位置称为 ko 步。而为了保证程序一定能下出 ko 步，我们设计了 koOrBranch 函数。该函数会对 generator 生成的当前最优解进行判断，若其分数超过临界线，则将其作为 ko 步输出；若没超过，再调用 Branch 模块求最优解。这样既提高了胜率，又节省了计算开销。

（4）**Branch 模块中的交互递归调用。**用 Haskell 语言去写出一个博弈树比较困难，而我们创新性地通过两个函数 F1 和 F2 的相互调用迭代完成博弈树的建立。F1 函数主要选择待选择三步的最大值和最小值，F2 进行待选择三步棋的分数计算，F1 需要调用 F2 去获得三步棋的分数，同时 F2 需要去调用 F1 知道在从下层向上递归的规则中，某一结点下层的三步棋的分数的最大值或最小值。

4.2. Haskell 编程心得

李铭涛：

Haskell 是我第一次接触函数式编程，与此前使用的 c++，python 等命令式编程语言有着天壤之别。虽然一开始使用时会感觉被一些缺少的东西束手束脚的（比如没有 for 循环，需要用递归思想代替等等），但习惯后也体会到了函数式编程独特的美感。不需要东写一行，西写一行，一切需要实现的功能都可以由一个个函数优雅的封装好，而不同函数堆积木一般就可以最终实现一个极其复杂的功能，比如这次的 chessAI。

虽然未来的绝大多数工作都不会用 Haskell 语言完成，但是学会这门语言就像掌握了一个优雅有趣的玩具，希望未来在接触某个需要函数式编程的任务时可以再次见到它!

李嘉祥：

1. 了解了一门拥有新颖的设计思路的语言，了解并初步体会了面向函数的语言的魅力和程序写作方法。
2. 通过该课程的各类作业，我初步了解了机器博弈中重要的博弈树的基本理论，并通过编写程序，提升了自身的 coding 能力，和 debug 能力，同时也是第一次初步了解一个合格的商用化软件的搭建思路以及对软件的鲁棒性有着很严格的要求
3. 该课程虽然与自己本身模拟方向无关，但是一门好的课程应该是让自己快乐的收获了某些知识，而非狭隘的仅仅限制在一个领域，而 Haskell 就是这样一门课程。

感谢老师付出和陪伴!

六子棋机器博弈技术综述

摘要：台湾交通大学教授于 2005 年提出了 6 子棋的概念，相较于五子棋在两种主流规则版本下存在的先手方必胜的情况，六子棋的公正性目前还没有被证伪，同时六子棋由于一次可以下两颗子，其状态数是五子棋的 10^{72} 倍，所以其搜索节点数大大增加，极具挑战性。本文将针对六子棋的机器博弈过程中运用的技术进行分析，从走法生成，博弈树与搜索算法两方面进行阐述，，分析各种算法的优缺点，从而为新方法的产生提供灵感。

六子棋也被称为 Connect(19,19,6,2,1)游戏，其规则是黑白双方在 19*19 的棋盘上进去博弈，除了黑方的第一手下一个子以外，其余都每次下两颗子，先连成 6 子的一方获胜。从 2006 年开始，六子棋赛事在全球各地如雨后春笋般冒出来，同时六子棋也成为了计算机奥林匹克竞赛中的一项游戏，充满无限可能的下法的它疯狂的吸引着全球玩家，也使得它成为了计算机博弈领域的新宠。本文将从走法生成和博弈树以及搜索算法的生成两方面对六子棋博弈进行介绍，并分析各类算法的优缺点。

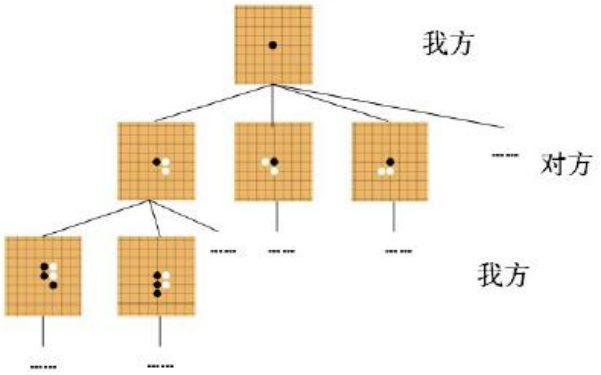
1 走法生成

走法生成及将下一步所下的棋的所有可能位置罗列出来。枚举走法的过程往往依赖于搜索技术，因此一个好的走法生成策略会大大提升计算机博弈的效率和能力。常见的走法生成策略有：预置表，棋盘扫描，NULL-Move 启发以及他们的各类结合。预置表策略[2]指将所有情形的所有可能的走法存储在一个表格中，之后调用该步骤即可，但是由于六子棋棋盘很大并且一次下两枚子，它拥有和围棋一样多的状态数，想要罗列各类情形基本不可能，只能罗列典型的几类清醒，这就导致该方法显得笨重，不够灵活，但是它的优点也显而易见，就是速度快。期盼扫描策略即依照六子棋的规则，对可落子区域进行遍历，最终给出落子位置，该方法清晰易懂，也是很多六子棋算法的基本出发点。NULL-Move 启发也称为空着启发，是指假设一方先不动，让另一方落子，而另一方落子的目的就是获胜，因此只要己方占住这些区域（称为 R-Zone）即可，这种策略会大大减少搜索空间，提升落子效率，但是显然是完全防守策略，在旁人来看只是一味挨打，最好的结果就是平局，但是最终也可能因为防不胜防而导致失败，因此该策略往往会和其它策略结合使用，而非单独使用。

2 博弈树和搜索算法

博弈树是指某一方对之后双方所下棋局的猜想，也是去评判某一步落子的好坏，如果博弈策略正确且博弈深度足够深，那么该方往往能够通过预知未来而获胜。在博弈的想象中，自己落子的回合和对方落子的回合构成的树成为与或树，己方回合落子成为与，因为自己需要考虑所有可能落子的情况，各个情况之间是与的关系，而对方回合的落子成为或，因为对于对方的落子情形，只需要挑一种出来即可[3]。

在博弈的过程中，算法需要对各个节点进行评估，整体的评估方式如下，我方回合的落子取下一步对手所有落子情况的最小值，也就是对手可能取得的最高分，可以发现分数是从底层向上滚动产生的，也就是只有先算出来最底层的局面评估函数后，才能得到上层的分值。下面将分别说明评估函数，极大极小值算法，Alpha-Beta 剪枝，MCTS 树搜索。



图一　与或树的形象表达

2.1 评估函数

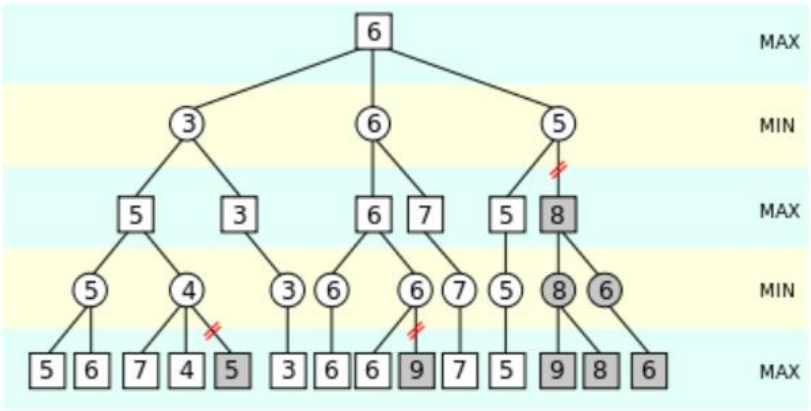
评估函数是对目前棋盘局面的一个数值化估算，其估算方式就是将黑白两方的所有活 X，眠 X，死 X 的棋形进行加权求和，从而得到目前黑白两方的各自分数，之后进行相减得到评估函数，此后各个节点的取值便是基于估值函数得到的。

2.2 极大极小值算法

该算法的含义是指己方在针对某一层的所有落子进行分析时往往取得所有节点中的最大值，从而保证自身利益的最大值，而对方在针对某一层的所有落子进行分析时往往取得所有节点中的最小值，从而保证自身利益的最大化。

2.3　Alpha-Beta 剪枝

Alpha-Beta 剪枝的基本依据时：棋手不会做出对自己不利的选择，依据这个前提，如果一个节点是明显不利于自己的节点，那么就可以直接剪掉这个节点。其原理是，在与层的时候，往往会取最大值，如果发现假设已经搜索到了一个极大值 X，下一个节点的下一层会产生一个比 X 很小的值，那么就会直接剪掉此节点。运用该算法可以有效提升搜索的效率。



图二　Alpha-Beta 剪枝

2.4　MCTS 树搜索

MCTS 搜索即蒙特卡洛树搜索[4]。虽然该算法也是通过剪枝来缩小搜索空间，但是不同的是，它对节点的评价并不是以人为制定的规则来评判的，而是通过蒙特卡洛模拟进行的，随着层数的提升，那么越接近最优点，收敛较快。该搜索算法主要有四个部分：选择，扩展，

模拟以及回溯更新。

选择阶段，从起点开始递归地对评价最优的点进行搜索，评价采用 UCT 策略，该策略在搜索时采用 Upper Confidence Bounds，其思想是：先对起点所有的子节点都搜索一遍，按照下列公式计算分数，之后选择一个分数最大的点继续进行搜索。

$$Score_s = \frac{Q(s)}{N(s)} + c\sqrt{\frac{2\ln N(p)}{N(s)}}$$

s 是指当前的节点，p 代表父亲节点 Score，表示当前节点的分数，Q（s）表示当前节点的累计分数，N 表示节点的访问次数，c 是一个自定义的常数。

扩展阶段，选中一个节点对其进行扩展子节点。

模拟阶段，根据蒙特卡洛模拟方法对拓展出的子节点进行模拟，即随机选择一个可落子位置作为其子节点，之后继续对子节点进行模拟，直到结束。

3 结束语

目前六子棋的主流算法还是基于博弈树的生成，但是如何进行博弈树的搜索就会引出多种算法，比如蒙特卡洛搜索，Alpha-Beta 剪枝等，同时对于评估函数的生成也有多种方式，全局值的求解也仅是一种方式，虽然随着 alpha-Go 的产生，六子棋算法的更新变慢了，但不失为一个值得机器博弈学术研究的领域。

参考文献

[1] https://wenku.baidu.com/view/1a23f5ed7c1cfad6195fa7e3.html

[2] 何轩,洪迎伟,王开译,彭耶萍.机器博弈主要技术分析——以六子棋为例[J].电脑知识与技术,2019,15(33):172-173.

[3] 何轩,洪迎伟,王开译,彭耶萍.机器博弈中搜索策略和估值函数的设计——以六子棋为例[J].电脑知识与技术,2019,15(34):53-54+61.

[4] S. Yen and J. Yang, "Two-Stage Monte Carlo Tree Search for Connect6," in IEEE Transactions on Computational Intelligence and AI in Games, vol. 3, no. 2, pp. 100-118, June 2011, doi: 10.1109/TCIAIG.2011.2134097.

组员分工

李嘉祥：

（1）完成了 Branch.hs, FoulAndWin.hs 的程序设计；

（2）完成了报告中 2.4.1, 2.4.2, 2.5 的撰写，并完成了算法综述部分的撰写；

（3）完成了 PPT 中对应部分的制作。

李铭涛：

（1）完成了 chessAI.hs, Evaluator.hs, Generator.hs, CSVtransform.hs, UpdateChess.hs, PrintBoard.hs，koOrBranch 函数的程序设计；完成了程序的运行测试；

（2）完成了报告中剩余部分的撰写；

（3）完成了 PPT 中对应部分的制作。