山东财经大学

本科毕业论文(设计)

题目：基于 Monte Carlo 方法的五子棋算法设计与实现

学　　院　　计算机科学与技术学院

专　　业　　金融信息化

班　　级　　金融信息化 1 班

学　　号　　201618866148

姓　　名　　于松黎

指导教师　　赵志崑

山东财经大学教务处制

二〇　20　年 5　月

山东财经大学学士学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出重要贡献的个人和集体，均已在论文中作了明确的说明并表示了谢意。本声明的法律结果由本人承担。

学位论文作者签名：________________

________年____月____日

山东财经大学关于论文使用授权的说明

本人完全了解山东财经大学有关保留、使用学士学位论文的规定，即：学校有权保留、送交论文的复印件，允许论文被查阅，学校可以公布论文的全部或部分内容，可以采用影印或其他复制手段保存论文。

指导教师签名：_____________　　　　论文作者签名：________________

_____年____月____日　　　　_____年_____月_____日

基于 Monte Carlo 方法的五子棋算法设计与实现

摘 要

本文选择五子棋为研究课题，在对大量相关文献进行分析研究的基础上，使用 Monte Carlo Tree Search 算法的原理设计了五子棋博弈系统的模型。所做的工作主要在以下方面：1.研究了五子棋在计算机中的表示问题，确定五子棋棋盘在计算机中的存储与表示方式。 2.研究棋类游戏人机博弈方法中的权值法、MiniMax 方法，并总结了两种算法之间的联系与区别。 3.在人机博弈基本方法的基础上，详细说明了 Monte Carlo Tree Search 算法的构造。在上述工作的基础上，本文的创新性研究主要包括以下方面： 1.针对 Monte Carlo Tree Search 的理论知识，以 2 * 2 棋盘为模型进行算法步骤演示，说明了不同步骤在整个算法中起到的作用。 2.构建五子棋游戏的 Monte Carlo Tree Search 算法，并根据五子棋的特点，提取局势特征，优化棋局的判断条件。在算法实际设计中，采用不同数据结构存储节点信息。

关键词：蒙特卡洛；博弈；五子棋；人工智能；搜索算法

FA Design and implementation of gobang algorithm based on Monte Carlo method

ABSTRACT

In this paper, gobang is selected as the research topic. Based on the analysis and research of a large number of relevant literatures, the game system model of gobang is designed by using the principle of Monte Carlo Tree Search algorithm. The work done is mainly in the following aspects:

Studied the representation of gobang in the computer, and determined the storage and representation of gobang board in the computer.

Studied the weight method and MiniMax method in the man-machine game of board games, and summarized the connection and difference between the two algorithms.

Based on the basic man-machine game method, the structure of Monte Carlo Tree Search algorithm is explained in detail.

On the basis of the above work, the innovative research of this paper mainly includes the following aspects:

Based on the theoretical knowledge of Monte Carlo Tree Search, the 2 * 2 checkerboard model was used to demonstrate the algorithm steps, and the role of different steps in the whole algorithm was illustrated.

Build the Monte Carlo Tree Search algorithm for the game of gobang, extract the characteristics of the situation according to the characteristics of gobang, and optimize the judgment conditions of the game. In the actual design of the algorithm, different data structures are used to store node information.

Keywords: Monte Carlo; Game; Gobang. Artificial intelligence; Search algorithm

目　录

一、绪论

人类对于人机博弈的遐想最早可追溯到1769年。彼时，匈牙利工程师Baron Wolfgang von Kempelen为奥地利皇后做了一台会下国际象棋的机器来消遣。而这台机器实际上是由一名象棋高手藏在机器中实现的。虽然当时的科学技术不足以完成人机博弈，但是却体现了人类对于科学技术的渴望。

1952年，阿伦·图灵(Alan Turing)写出了第一个棋弈程序。早期的博弈程序由于算法与硬件的限制，效率十分底下。后来John C. Harsanyi， John F. Nash及Reinhard Selten等人在冯·诺伊曼(John Von Neumann)提出的广义极大极小（MiniMax）算法的基础上继续进行对抗性博弈中的均衡问题研究，产生了棋类博弈中的MiniMax算法。这种算法基于遍历博弈树的方式，而博弈树节点数量庞大，难以在有限的时间遍历完成。

1958年，匹兹堡大学的科学家Allen Newell,，Shawn，Herbert A. Simon提出了alpha-beta剪枝方法，以减少搜索树的节点数。1997年，IBM公司的“深蓝”利用MiniMax算法战胜了国际象棋世界冠军卡斯帕罗夫，而围棋由于局面的特殊性，迟迟没有较大突破。直到2006年Remi Coulom等人提出Monte Carlo Tree Search算法，并在2016年由Google Deepmind的AlphaGo程序采用此算法战胜了围棋冠军李世石后，围棋人机博弈才真正被攻克。

本文从使用广泛的权值法、MiniMax算法出发，通过列举权值法与MiniMax算法构造，比较算法的异同点，以说明Monte Carlo算法的优势，并针对五子棋游戏进行Monte Cralo算法设计。

二、棋类游戏人机博弈的基本方法

对于大多数棋类博弈游戏，如五子棋、围棋、象棋等，都属于有限全息回合制零和博弈。它们满足的条件有：

①博弈要求必须有多方参与，对于上述棋类则是双人博弈。

②有限要求玩家只存在有限互动方式。比如只能通过合法走步改变当前局面状态。

③全息要求局面必须是透明的，即双方玩家获得的信息相同。

④回合制表示参与游戏方必须轮流行动，当一方未做出下一步之前，另一方只能等待。

⑤零和博弈表示参与游戏方的总收益为0，当一方取得胜利时，必然存在失败方。

针对上述特点，一个人机对弈的程序，应具备如下几个部分：

①全息的特点要求程序能够将可视化的局面通过代码描述为计算机能够识别的模式。

②有限的特点要求计算机能够通过局面产生符合规则要求的走法，这使博弈过程变得公平且透明。计算机也可以通过这种方式，判断玩家是否进行不符合规则的走法。

③零和的特点要求计算机能够从所有由当前局面构成的走法集合中挑选出一个最优走法。

④能够对当前局面进行合理评估，并配合挑选走法做出自动化选择。

⑤除此之外，还需要一个承载程序的界面。

（一）棋盘表示与走法产生

棋盘表示方法有两种：通过二维数组表示或通过比特棋盘表示。用二维数组的下标表示横坐标与纵坐标，这种表示方式比较直观，在程序开发过程中，可读性较强。比特棋盘则将棋盘上每个坐标都用一个64位数字表示，通过位运算，改变棋盘的数字，来表示当前坐标下棋子的变化情况。由于位运算比遍历数组更快，因此效率更高。但是缺点则是不够直观。

在棋局规则下，棋盘上存在合理位置，也存在不合理位置，需要通过走法产生器的在不违背规则的前提下，产生所有合理走步。这对于不同的棋类是不一样的，比如五子棋，所有棋盘上的空点就是合法走步集合。对于象棋则需要针对不同的棋子做不同规则限制，而此时有可能造成运算时间相当长。针对这种情况，需要设计良好数据结构，以保障走法的快速产生。一种广泛使用的做法是将走法全部写入开局库，当需要产生走法时，直接从开局库中读取，如果开局库中没有，则通过走法生成器生成合理位置。

（二）搜索算法

1. 权值法

对于五子棋来说，由于胜利的标志是某一方向至少包含5个连续相同颜色的棋子。基于这种规则下，产生了权值法。

给定一个棋盘，将其拆分成5个位置为一组，称为五元组。例如在15 ＊ 15的五子棋棋盘下，产生的五元组的个数为572个。针对某一个五元组中黑棋与白棋的个数（可以不考虑相对于棋盘的位置），给该五元组评分，每一个位置的所有五元组的得分就是这个位置的最终得分。电脑可以通过得分的高低，判断出每一个位置是否值得尝试，最终从整个棋盘中挑出得分最高的位置作为最佳走步。以下是一种评分表的计分方法：

```
// tuple is empty
Blank,
// tuple contains a black chess
B,
// tuple contains two black chesses
BB,
// tuple contains three black chesses
BBB,
// tuple contains four black chesses
BBBB,
// tuple contains a white chess
W,
```

```
// tuple contains two white chesses
WW,
// tuple contains three white chesses
WWW,
// tuple contains four white chesses
WWWW,
// tuple does not exist
Virtual,
// tuple contains at least one black and at least one white
Polluted
tupleScoreTable[0] = 7;
tupleScoreTable[1] = 35;
tupleScoreTable[2] = 800;
tupleScoreTable[3] = 15000;
tupleScoreTable[4] = 800000;
tupleScoreTable[5] = 15;
tupleScoreTable[6] = 400;
tupleScoreTable[7] = 1800;
tupleScoreTable[8] = 100000;
tupleScoreTable[9] = 0;
tupleScoreTable[10] = 0;
```

权值法的优点在于：①实现相对简单，只要根据自己的经验，列举五元组中所有可能的情况，并且给情况全部打分即可。②不需要遍历棋盘上的所有位置，只需判断最后一步棋的八个方向。因此运行速度较快。

但权值法也存在很多明显缺陷：①只能用于特定的棋类，一旦换成其它棋类，这种方法就失效了。②在给五元组打分的过程中，程序员必须依靠自己对于棋类的知识进行判断。如果程序员对于棋的规则或者棋局不精通，产生的评分表可能不会带来很好的效果。③由于只能给当前局面进行打分，而忽略了几步之后胜率可能会变得很高的点或者几步之后可能会变得被动的潜在位置，因此很可能陷入局部最优的状况。

2. MiniMax算法与改良

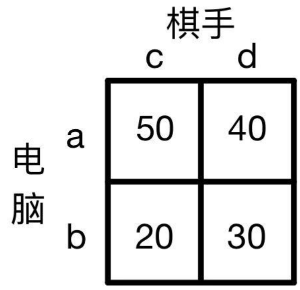
为了解决上述局部最优的问题，极小极大（MiniMax）算法被研究出来。MiniMax算法能够根据当前局面进行更深层次的探索，并将每一个局面以及当前局面的后续局面包装成节点，节点之间通过局面的产生关系连接成为博弈树。通过遍历博弈树，计算机可以预测当前局面后若干步。

(1)MiniMax算法组成

MiniMax算法包括以下部分：搜索、剪枝、估值函数。

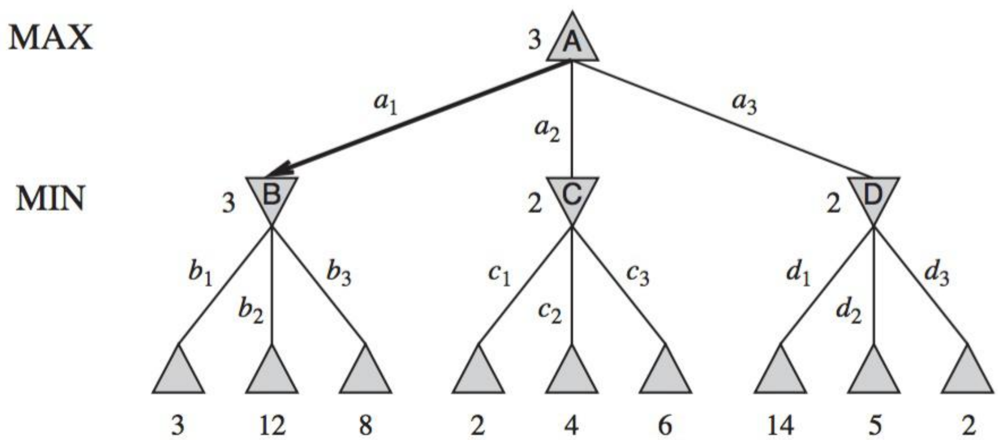
a.搜索与剪枝

由于MiniMax基于对手与计算机具有相同棋力的假设，因此通过MiniMax算法计算后，程序最终选择的是最差结果中的最好走法。以一个简单的模型举例：



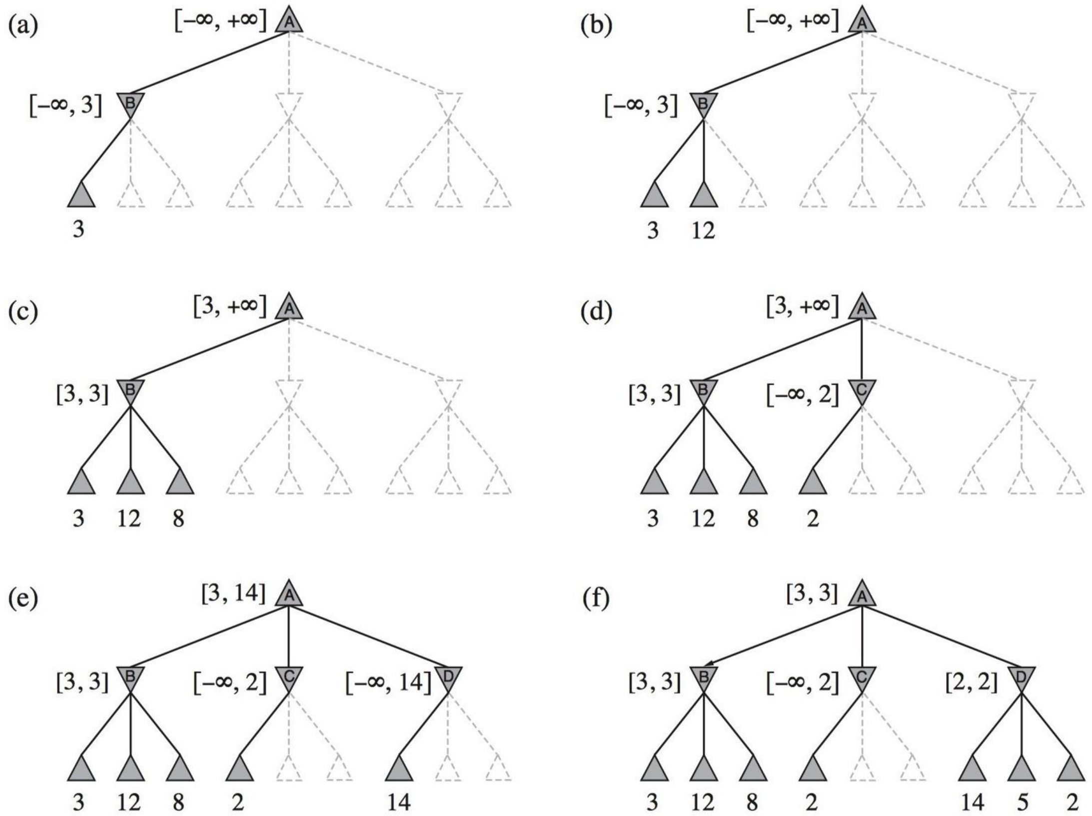
一个棋局的最终博弈结果是由电脑和玩家共同决定。现假设电脑只允许在a与b中选择某一行，棋手只允许在c与d中选择某一列，最终行列相交的数字表示一个局面的评分，评分越高，对当前下棋方越有利，且电脑与棋手水平相当。那么对于电脑来说，如果选择a行，棋手必定选择c列，此时的局面为50，反而对棋手更有利。因此电脑正确做法是选择b行，而棋手为了使自己收益最大，必定会选择d列，最终评分为30，博弈达到均衡状态。

在MiniMax中，下棋双方同样需要相似地考虑自己的落子会对对方产生的收益。在博弈过程中，如果甲选择最大化收益，那么乙必定会做出让甲的收益最小的行动。因此在博弈开始时，将甲的收益置为最小，乙的收益最大化（相对于甲而言），然后通过不断遍历博弈树，甲的收益值逐渐升高，乙的收益值逐渐降低，最终到达均衡状态，



这就构成了搜索技术的基本手法——完全展开博弈树。而实际上完全展开的博弈树将非常庞大，根本无法完全遍历，也无法产生最优走步。因此在搜索过程中应采用深度优先搜索。深度优先的好处在于搜索完成一个分支的时候，可以将此分支删除，这就保证了博弈树不会变得非常大而使内存溢出。但是造成的长时间搜索问题并没有得到解决。

alpha-beta剪枝方法可以部分解决搜索时间长的问题。在MiniMax的过程中，会存在一定的数据冗余。如果某一个节点轮到甲走棋，而甲向下搜索节点时发现第一个子节点就可以取得胜利（节点值为最大值），则剩下的节点就无需再搜索了，甲的值就是第一个子节点的值。这个过程就可以将大量冗余的（不影响结果的）节点抛弃。alpha-beta剪枝的效果受节点的产生顺序影响非常大，在最差情况下，不会产生剪枝。



b.估值函数

估值函数根据不同的棋类会有所不同，需要人为设置每种棋型的权重，包括基本棋型的影响与棋型间附加的影响。此函数与权值法类似，都需要程序员具备棋类的知识。估值函数的优劣将直接影响整个算法的计算时间与精确度。

(2)MiniMax算法优化

在1975年，由Knuth和Moore在MiniMax算法的基础上提出了负极大值（Negamax）算法，即每次遍历完成后，都将估值取负值向上带回，这样做与极小极大算法的结果是一样的，但是简化了代码，并且消除了两方的差别。但此算法要求估值函数能够通过正负值反映出是哪一方在下棋。

上述无论是极小极大算法还是负极大值算法，窗口都是负无穷到正无穷，这种做法是将窗口不断缩小，直到缩到正确的值上。而另一种做法是事先划定一个小窗口，采用alpha - beta剪枝，如果不在窗口内，就扩大窗口重新搜索，这个思路是渴望搜索（Aspiration Search）。如果更极端一点，将窗口无限小，那么算法搜索开始时必定无法落在窗口中，需要进一步扩大窗口，最终的结果是窗口恰好满足搜索需求。基于这种思路，Judea Pearl与Alexander Reinefeld研究出极小窗口搜索（Minimal Window Search，又称PVS搜索或NegaScout搜索）。

除了对于搜索算法的优化，还有针对开局或残局大量数据的开局库或残局库、针对重复数据的置换表与针对alpha-beta的剪枝顺序的历史启发等更多优化方式。

(3)MiniMax算法与权值法的异同点

相比于权值法，MiniMax算法通过构建博弈树并深度遍历树节点的方式，解决了权值法的局部最优问题，使计算机具备了人类的预测走法功能。

但MiniMax算法需要依赖于特定的棋类知识。其中的估值函数虽然模式固定，但是直接影响到程序判断走法的优劣性。因此在绝大多数程序都使用了相同的优化手法的前提下，估值函数成为了评判一个程序的最重要的指标。并且在围棋等局面更复杂的情况下，使用MiniMax无法评判一个局面的好坏。

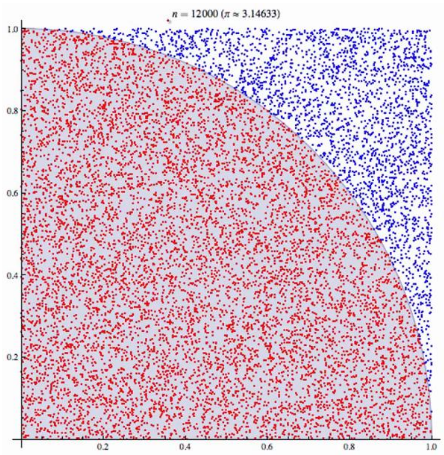
三、Monte Carlo Tree Search算法概述

蒙特卡洛树搜索（Monte Carlo Tree Search，简称MCTS）算法则是应用于树结构的一种启发式搜索算法。MCTS算法继承了MiniMax的博弈树概念，基于Monte Carlo类似多次随机模拟实验的方法。与Monte Carlo方法不同的是，由于只能在树结构中使用，因此MCTS算法的应用场景更少。在分析MCTS算法之前，需要对Monte Carlo方法进行说明。

（一）Monte Carlo方法

Monte Carlo方法与Monte Carlo Tree Search算法在思路上有继承性，但在具体实现中不同。Monte Carlo方法实际上是一种基于随机抽样的统计学模拟方法，是二十世纪四十年代中期由于科学技术的发展和电子计算机的发明，而被Bruce Abramson提出的一种以概率统计理论为指导的一类数值计算方法。这种方法旨在通过使用随机数来解决计算问题。

比如求圆周率Π时，在正方形中随机布点。当布点的次数足够多时，这些点将均匀地分布在正方形范围内，这时只需计算出圆内与圆外点的个数，即可估算出Π的值：



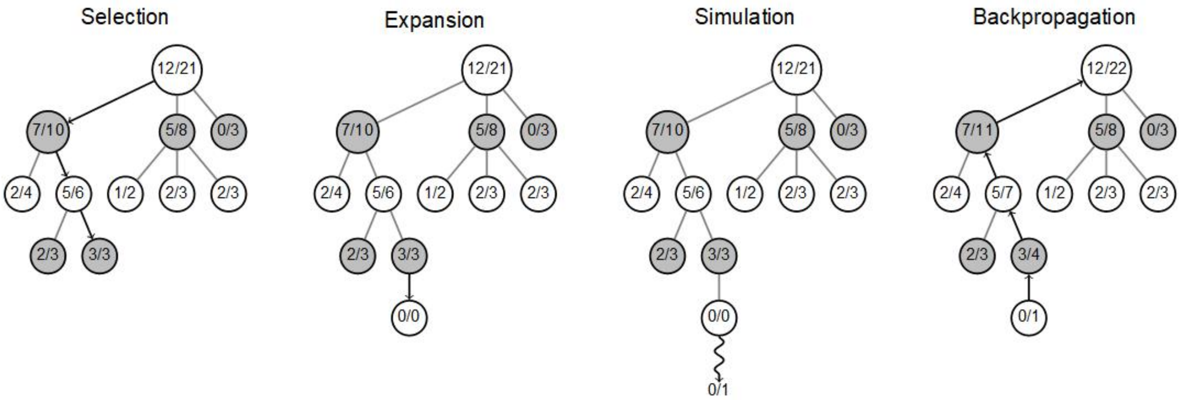
Monte Carlo方法的意义在于，让电脑帮助人们进行多次模拟重复计算，可以解决大量手工模拟的繁琐。这种方法具有普适性，不仅在棋类游戏中可以使用，在数据分析时也可以使用这种多次模拟且具有随机性的实验方法。

（二）Monte Carlo Tree Search 算法

通过不断的模拟得到大部分节点的价值，然后下次模拟的时候根据价值有策略地选择值得利用和值得探索的节点继续模拟，在搜索空间巨大并且计算能力有限的情况下，这种启发式搜索能更集中地、更大概率找到一些更好的节点。、

1．Monte Carlo Tree Search算法基本原理

MCTS算法的步骤是：选择（Selecttion）与扩展（Expantion）、模拟（Simulation)、反向传播（Backpropagation）。选择与扩展步骤属于Tree Policy，模拟步骤属于Rollout Policy。在MCTS进行过程中，这些步骤会重复多次，直到算力到达极限才停止。



上述步骤都是针对节点进行，而每一个节点，都至少包括两个属性和一种节点状态。

两个属性：总模拟奖励 Q，用于反映节点最终的结局。总访问次数 N，用于反映此节点被访问的次数。通过 UCT 函数反映一个节点最终是否值得探索或被选择。UCT 值越大，节点越有价值，在选择的时候就会更被关注。公式中第一项为胜率，第二项为探索分量。胜率表征当前节点在模拟中的胜利次数比例，探索分量表征此节点是否值得下一次被探索。胜率与探索分量之间通过参数 c 控制选择的偏好。c 越大，在选取的时候会越偏向未访问过的节点。c 越小，在选取的时候会越偏向胜率高的节点。

$$UCT = \frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}}$$

UCT函数有很多变体形式，在实际使用中，我们采用相对更合理的UCB1函数：

$$UCB1 = \frac{Q(\text{当前节点})}{N(\text{当前节点})} + c\sqrt{\frac{2\ln N(\text{父节点})}{N(\text{当前节点})}}$$

节点状态分为：已访问节点、已完全展开节点、未完全展开节点、终端节点。

①已访问的节点：指在MCTS搜索过程中，节点的信息被更新过。这个信息更新是由后述的反向传播步骤完成。

②已完全展开的节点：指当前节点下的所有子节点都至少被访问过一次。

③未完全展开的节点：指当前节点仍存在没有被访问过的子节点，或者仍存在未产生的子节点。未产生的子节点一定是未被访问的节点。

④终端节点（叶节点）：指当前节点对应的局面是确定的，且无法继续下棋。确定的状态包括：电脑胜利、玩家胜利、平局。对于五子棋，只要棋盘上仍存在空位置，就不是终端节点。

(1)选择与扩展（Tree Policy）

选择与扩展步骤的目的是：从根节点出发，深度遍历整棵博弈树，直到获得一个值得探索的子节点。由于 MCTS 会多次进行模拟，TreePolicy 也会被执行多次。但是每一次执行，博弈树都会产生一些新的子节点。

在算法进行过程中，根节点会存在以下几种状态：①MCTS 开始时的根节点：其子节点还未产生，因此需要随机选取一个子局面作为子节点展开。②已经至少进行了一次 MCTS 的根节点；a.根节点没有完全展开：虽然已经存在部分子节点，但是仍有节点没有展开。此时应优先展开子节点，而不是在目前的已经展开的子节点中选择。b.根节点已经完全展开：此时所有子节点都是已访问状态。在这种状况下，需要通过 UCT 函数（后叙）计算得出一个值得探索的子节点。对于一个节点的展开方法：查找所有子局面，并随机一个子局面生成子节点。

(2)模拟(Rollout Policy)

模拟步骤的目的是评估当前节点的价值。作用类似于 MiniMax 的估值函数。但是模拟的评估是随机评估，如果只进行一次，结果将非常不准确，这也是 MCTS 需要进行多次模拟的原因。随着执行次数增加，博弈树会变得越来越庞大，算法的可信度就越高。

实现模拟的步骤是对上一步产生的节点的棋盘进行随机落子，直到把整个棋盘填满，或者分出了胜负，最后返回模拟下棋的结果。在进行下棋之前，需要先判断一下当前节点是否是终端节点，如果是终端节点，就不进行模拟下棋，直接返回真实结果。此步骤体现了 Monte Carlo 方法的随机性。模拟步骤没有改变节点的定位.在模拟结束后，当前节点依然是通过 TreePolicy 产生的那个节点。

在执行落子之前，需要重新拷贝一份当前的棋盘，否则模拟后将会对目前的棋盘产生影响，使下一次模拟结果出错。

(3)反向传播

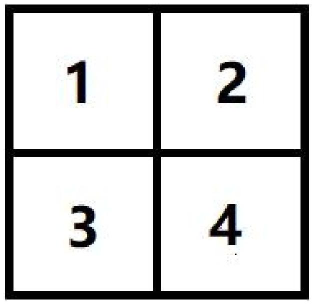
反向传播的目的是将当前节点的模拟结果传达到父节点，并让父节点继续向上传播，直到传播到根节点。通过反向传播，根节点可以得知子节点在若干步后的情况，并根据情况判断这个子节点是否值得继续探索。在反向传播过程中，节点的Q值与N值会被更新，每进行一次反向传播，节点的N值就会加1。而Q值则反映的是模拟的输赢情况。如果模拟胜

利方是当前节点对应的那一方，则Q加1，反之则减1。

(4)选择下一走法

在算力达到极限时，MCTS会停止节点的探索，并以博弈树节点的Q值与N值作为依据，选出最终的选择。在选择时有两种方案：一种方案是选择访问次数N值最高的节点，另一种方案是选择最大胜率的节点。在实际测试过程中，我们发现大部分情况下，两种方式选择的是同一节点，因此在本设计中，我们采用最大胜率作为最终的选择依据。

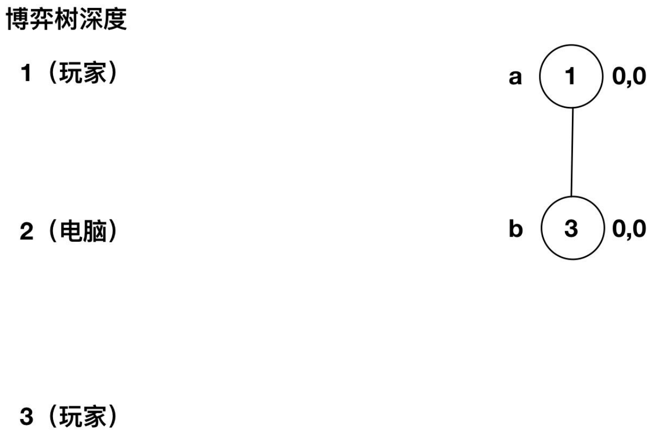
2．Monte Carlo Tree Search算法过程示例（以2 * 2棋盘为例）



为了能够清晰地描述整个算法的工作过程，我们假设在2＊2棋盘的简化模型中，整个算法在算力足够大，能够遍历所有博弈树的前提下进行计算。对于此模型中涉及到随机的部分，采用程序模拟的方式给出随机值。最终的胜负条件为1/2概率随机得出。在简化模型中，取UCB1公式超参数的C值为1.414，且计算出的UCB1值采用截断的方式保留小数点后两位。

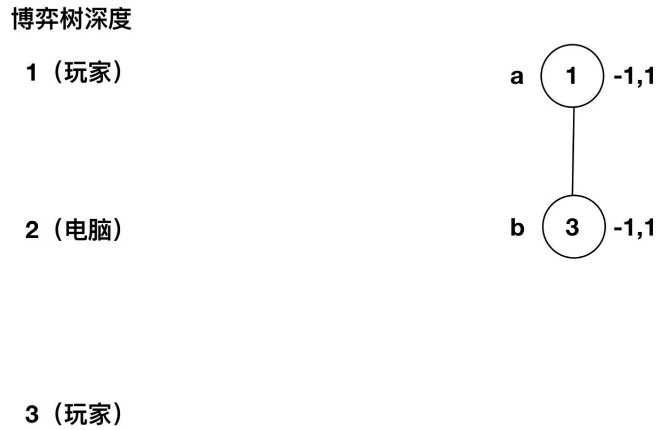
第一次模拟：

假设棋手先下棋，此时博弈树根节点a产生。程序执行MCTS算法：在Tree Policy中发现根节点未完全展开，因此随机展开一个子节点b：



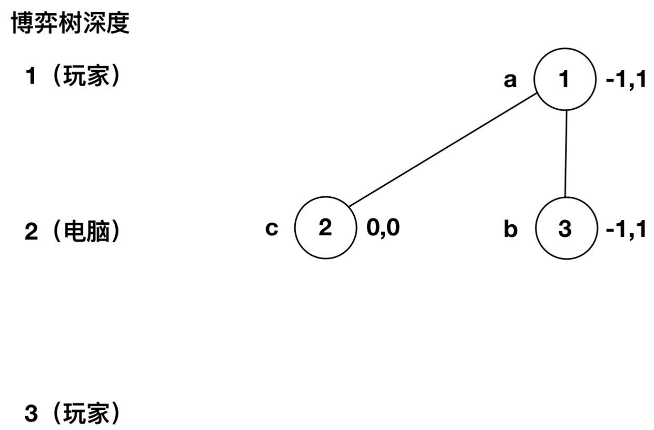
对展开的节点进行Rollout Policy，模拟结果为人类棋手赢。 节点b进行反向传播。由于电脑需要在自身的角度考虑局面，相对于电脑来说，这是一个不利局面，因此3号坐标对应的节点b的模拟奖励Q值-1，访问次数N值+1。此时节点状态变为已访问。所有父节

点的状态被更新：

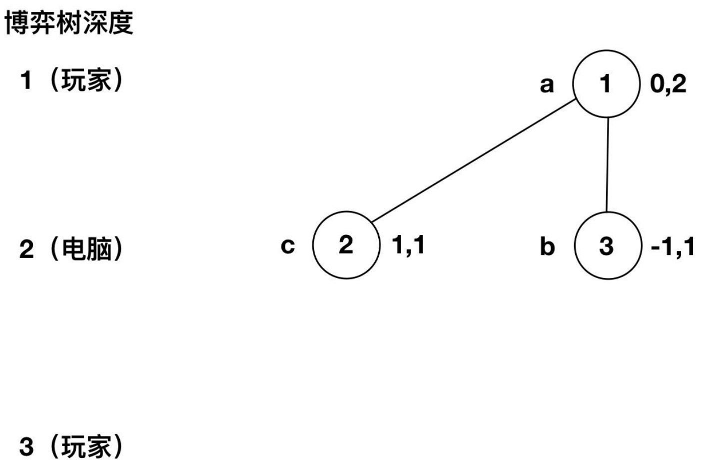


第二次模拟：

对根节点进行 Tree Policy，发现仍存在未展开的子节点。采用随机选择的方式，选择节点 c 展开：

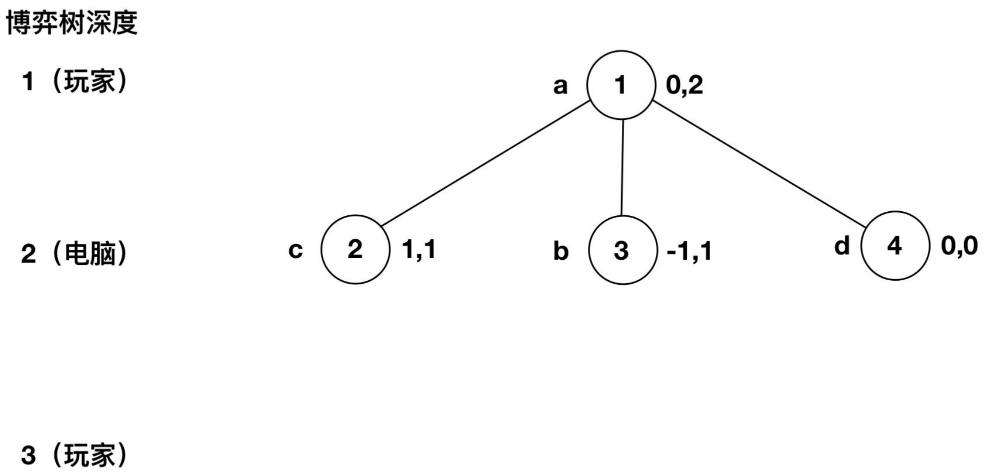


对展开的节点c进行Rollout Policy，模拟结果为电脑赢。 节点c进行反向传播。相对于电脑方，此位置对应节点的模拟奖励Q值+1，访问次数N值+1，此时节点c状态变为已访问。所有父节点的状态被更新：

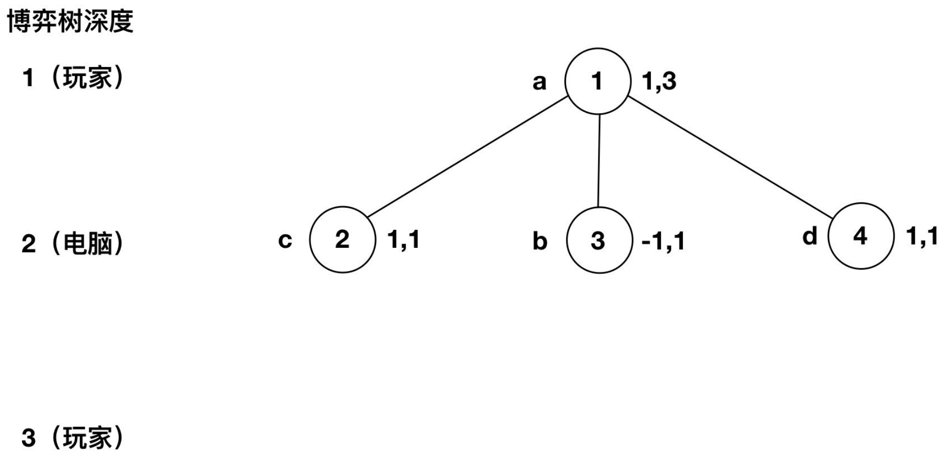


第三次模拟：

对根节点 a 进行 Tree Policy，发现仍存在未展开的子节点，依然采用随机策略，选择节点 d 展开：



对展开的节点d进行Rollout Policy，模拟出结果为电脑赢。 节点d进行反向传播。相对于电脑，此位置对应节点的模拟奖励Q值+1，访问次数N值+1。此时节点d状态变为已访问。所有父节点的状态被更新：



第四次模拟：

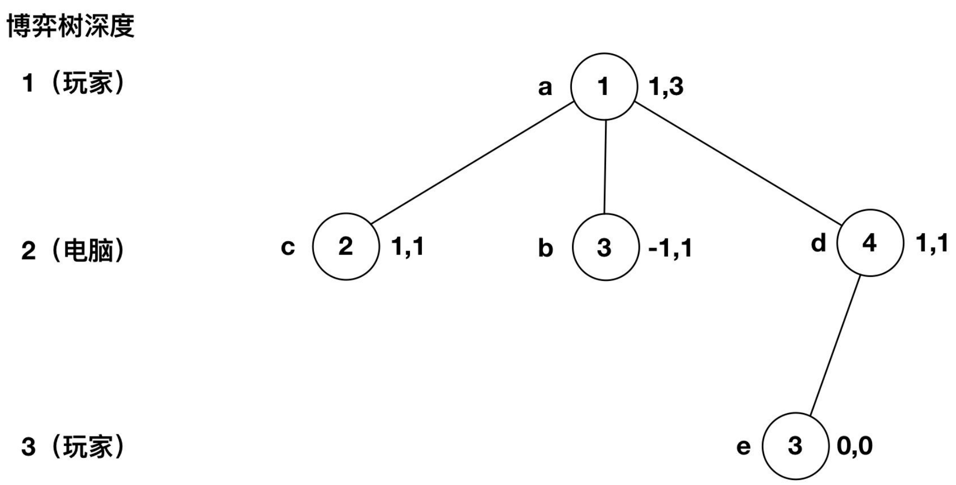
对根节点 a 进行 Tree Policy，发现所有子节点全部访问，此时根节点 a 完全展开，采用 UCB1 函数计算每个子节点，得到下一步要搜索的节点。分别将节点属性进行计算：

UCB1[节点 c] = 3.09

UCB1[节点 b] = 1.09

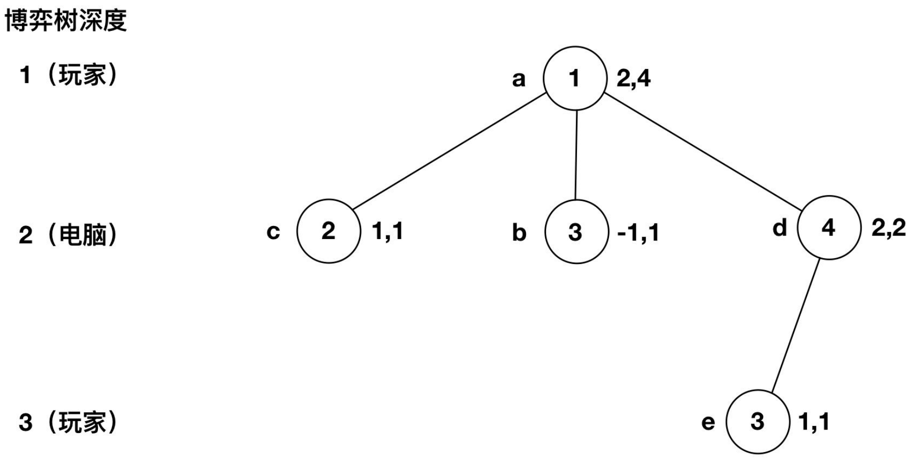
UCB1[节点 d] = 3.09

最终选择 UCB 最高的节点 d，发现此节点没有完全展开，因此采用随机选择的方式展开一个子节点 e：



对节点 e 进行 Rollout Policy，模拟出结果为电脑赢，

节点 e 进行反向传播。相对于电脑，此位置对应节点的模拟奖励 Q 值+1，访问次数 N 值+1。此时节点 e 状态变为已访问。所有父节点的状态被更新。



第五次模拟：

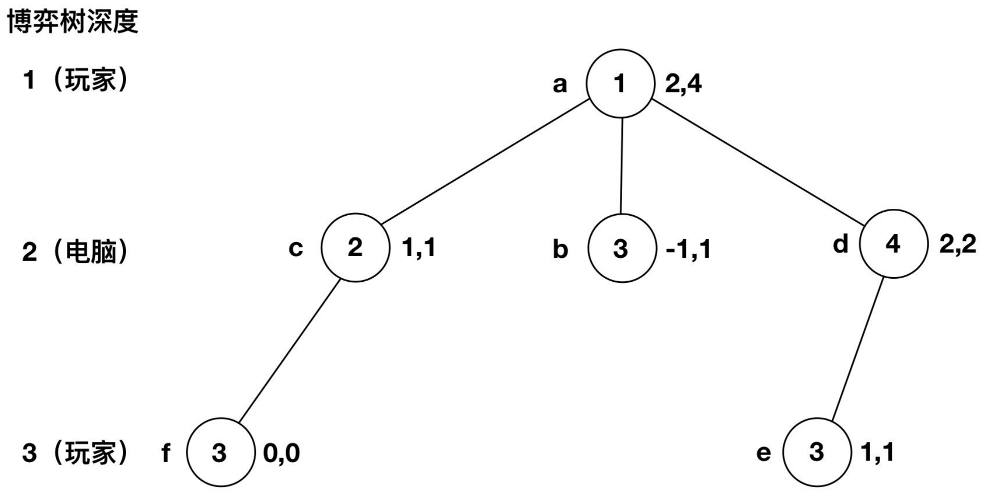
对根节点进行 Tree Policy，发现所有子节点全部访问，此时根节点完全展开，应采用 UCB1 函数计算每个子节点，得到下一步要搜索的节点。分别计算子节点的 UCB1 值：

$$UCB1[节点 c] = 3.35$$

$$UCB1[节点 b] = 1.35$$

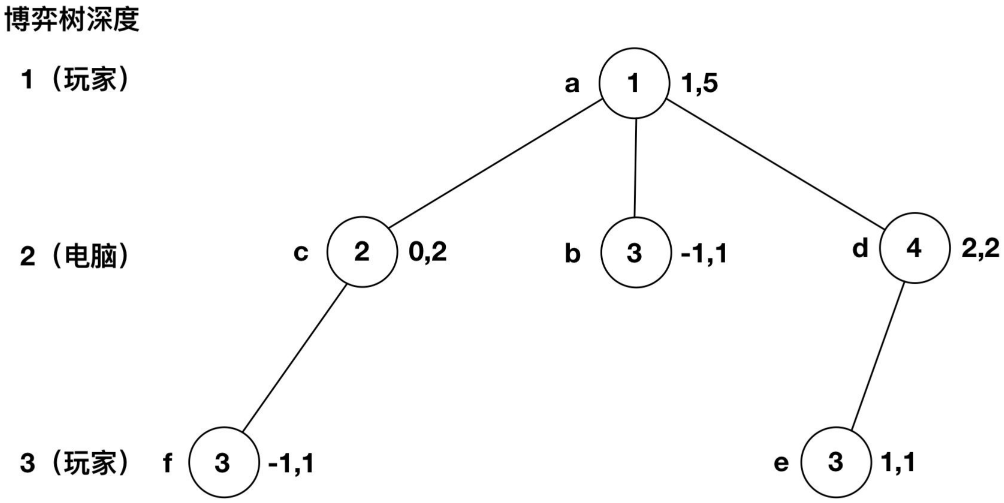
$$UCB1[节点 d] = 2.66$$

最终选择 UCB1 值最高的节点 c，发现此节点没有完全展开，因此采用随机策略展开一个子节点 f：



对节点 f 进行 Rollout Policy，模拟出结果为人类棋手赢。

节点 f 进行反向传播。虽然这是人类棋手的回合，但是相对于电脑来说这是一个不利局面，因此此位置对应节点的模拟奖励 Q 值-1，访问次数 N 值+1。此时节点 f 状态变为已访问。所有父节点的状态被更新。



第六次模拟：

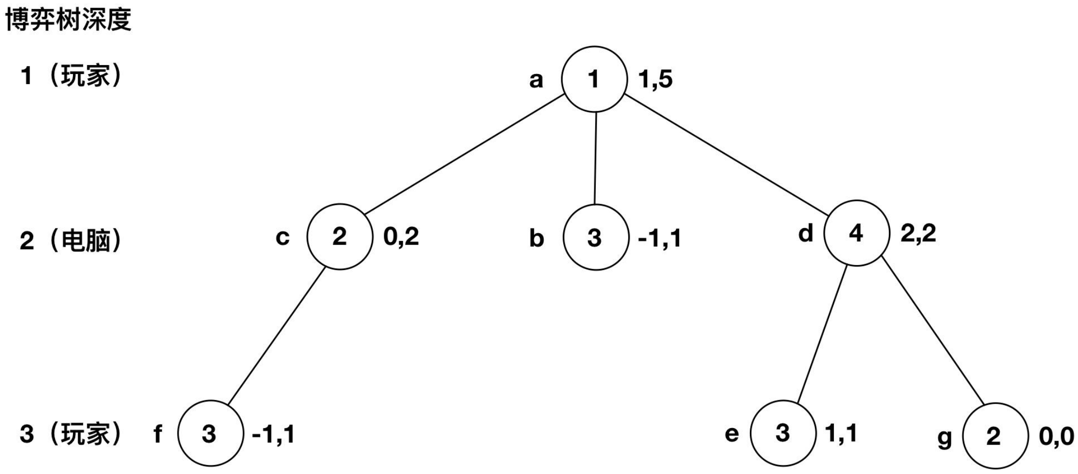
对根节点进行 Tree Policy，发现所有子节点全部访问，此时根节点完全展开，应采用 UCB1 函数计算每个子节点，得到下一步要搜索的节点。分别对节点进行 UCB1 值计算：

UCB1[节点 c] = 1.79

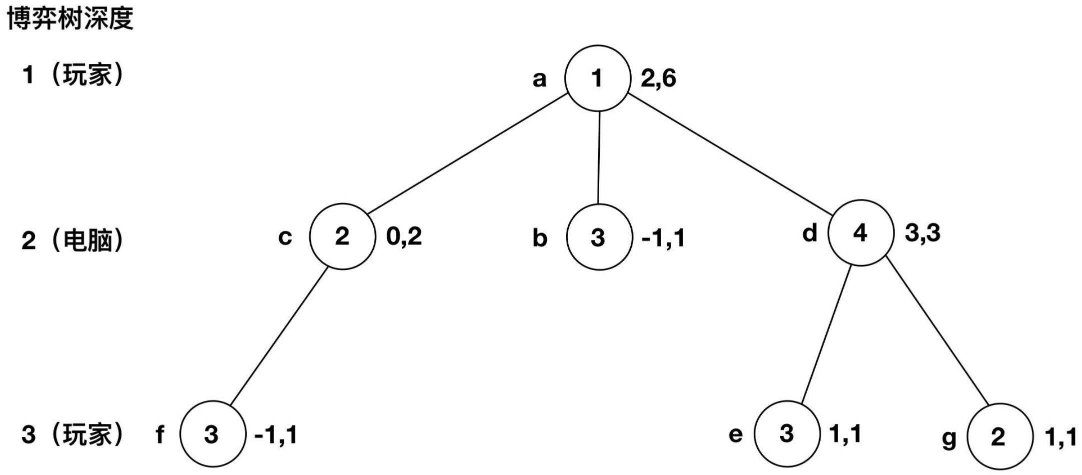
UCB1[节点 b] = 1.53

UCB1[节点 d] = 2.79

最终选择 UCB 最高的节点 d，发现此节点没有完全展开，因此采用随机策略展开一个子节点 g：



对节点g进行Rollout Policy，模拟出结果为电脑赢， 节点g进行反向传播。相对于电脑来说，此位置对应节点的模拟奖励Q值+1，访问次数N值+1。此时节点g状态变为已访问。所有父节点的状态被更新。



第七次模拟：

对根节点进行 Tree Policy，发现所有子节点全部访问，此时根节点完全展开，应采用 UCB1 函数计算每个子节点，得到下一步要搜索的节点。分别将节点属性进行计算：

UCB1[节点 c] = 1.89

UCB1[节点 b] = 1.67

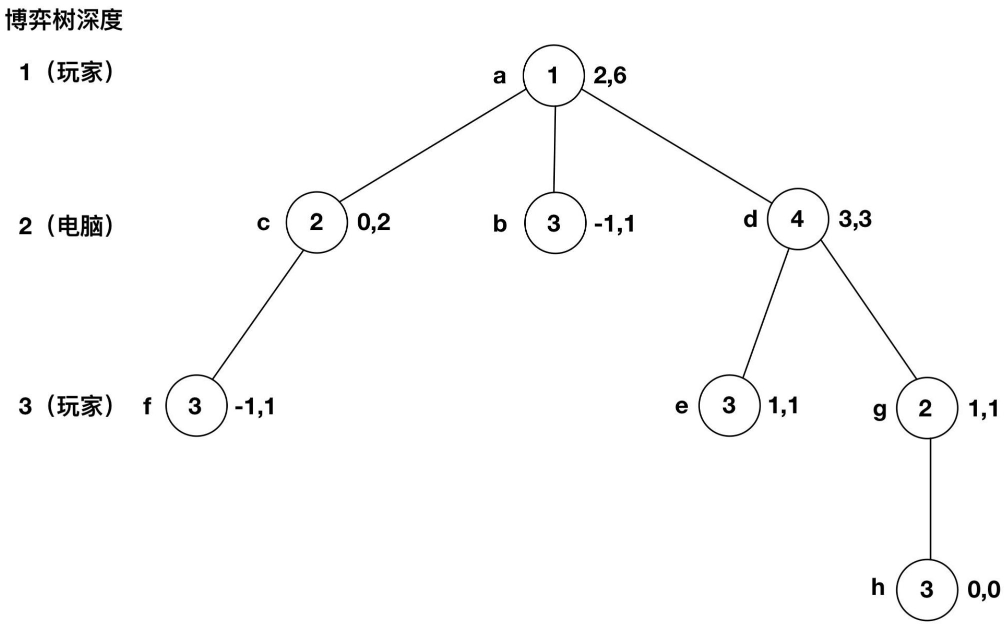
UCB1[节点 d] = 2.54

最终选择 UCB 最高的节点 d，发现此节点已经完全展开，因此继续使用 UCB1 函数计算每个子节点，得到下一步要搜索的节点。分别将节点属性进行计算：

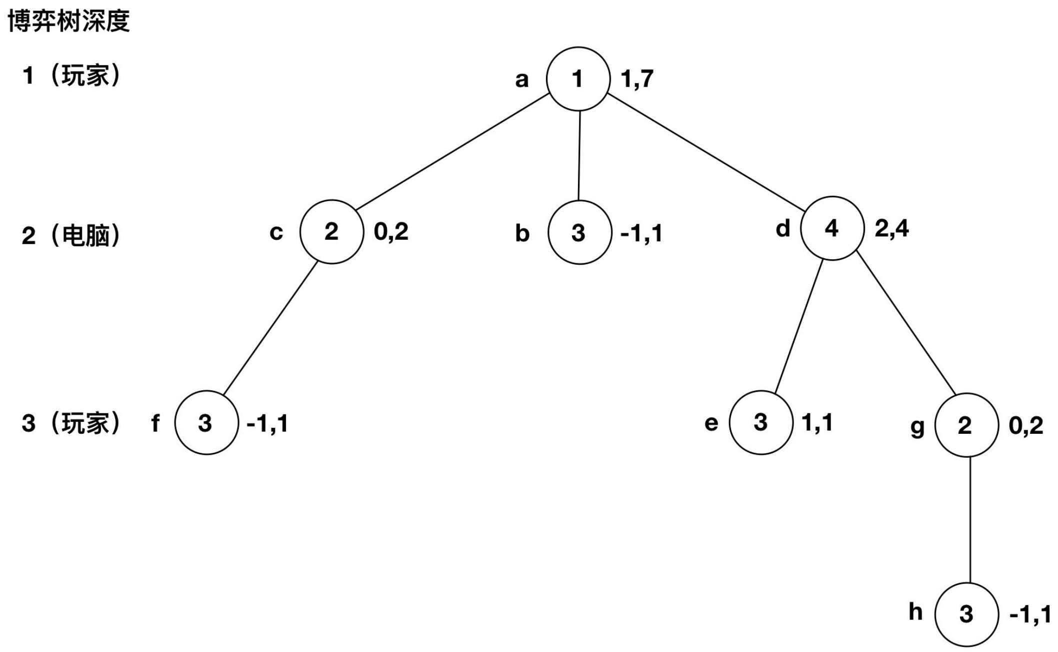
UCB1[节点 e] = 3.09

UCB1[节点 g] = 3.09

因此选择 UCB1 值最高的节点 g，此节点没有完全展开，因此采用随机选择的方式展开一个子节点 h：



此节点为终端节点，因此不进行模拟，直接将确定的局面反向传播（由于此简化模型的胜负条件是1/2概率，因此此步骤依然存在随机性。这里假设“确定的局面”为人类棋手赢）。相对于电脑，此位置对应节点的模拟奖励Q值-1，访问次数N值+1。此时节点h状态变为已访问。所有父节点的状态被更新。



如此循环往复，在算力足够大的情形下，程序会按照UCB1的启发式搜索顺序，搜索到全部博弈树的节点。但实际情况下，算力不可能是无限的，因此我们假设在进行上述七次模拟后，算力达到极限。此时程序将从节点c，节点b，节点d中选择胜率E最高的节点作为下一步走法：

$$E[\text{节点c}] = 0.00$$

$$E[\text{节点b}] = -1.00$$

$$E[\text{节点d}] = 0.50$$

上述节点中，节点d的胜率最高，因此选择节点d作为下一步的走法。

3．Monte Carlo Tree Search算法与其他搜索算法的异同点

MCTS 算法与 MiniMax 算法的相同点：两种方式都采用深度遍历。并且都通过某种评估手段缩小博弈树。

MCTS 算法与 MiniMax 算法的不同点： ①局面评估方式不同。MiniMax 算法的评估是通过估值函数实现，而 MCTS 算法的评估是通过 Rollout Policy 实现。 ②剪枝方式不同：MCTS 算法通过 UCB1 函数的判断实现自动化剪枝。MIniMax 算法通过 alpha-beta 剪枝。③ MCTS 不依赖特定的棋类知识，只要在 rollout 的方法中加入棋类规则即可。而 MiniMax 需要程序员对棋类了解，并且人工设置某种局势的优劣。

四、五子棋程序实现

（一）Monte Carlo Tree Search 算法的伪代码实现

1. 状态的预定义与MCTS基本步骤

首先定义出判断局面的基本单位：Action 类，用于表示一个坐标或者一个动作。由于动作是最小单位，为了保证数据安全，将横坐标 x 与纵坐标 y 定义成无法修改。

```
class Action {
    public final int x;    //横坐标
    public final int y;    //纵坐标

    public Action(int x, int y) { //Action 类的初始化方法
      初始化 x 的值;
      初始化 y 的值;
    }
}
```

还需要将当前棋盘 board 与当前玩家 player 包装成一个局面。为了防止棋盘状态被误更改，需要针对传入的棋盘深拷贝，以确保产生的新棋盘的局面与原棋盘相同，但是地址不同。最后用 State 类来描述：

```
class State {
    public int[][] board;  //一个棋盘，用二维数组表示
    public int player  //当前回合下的玩家

  public State(int[][] board, int player) { //State 类的初始化方法
        深拷贝当前棋盘状态;
        初始化当前玩家;
    }
}
```

由于算法会针对博弈树进行操作，因此需要定义博弈树的节点结构 Node 类。

一个节点应由当前的局面（state）和父节点产生，并包含所有可能的子节点。对于一个节点用 untriedAction 属性来描述所有的可供选择的下一步集合。由于可供选择的下一步个数是不固定的，且每当产生一个子节点，相应的动作应从“可供选择的下一步”中删除，因此应使用链表结构。

为了最后能够找到节点相对应的动作，每一个子节点都需要和相应的动作绑定。并且在查找子节点的过程中，我们并不要求顺序一致，而是更关心查找的速度，因此应采用底层为哈希表的键值对映射结构 HashMap，并通过 Action=Node 的键值方式映射。

每一个节点都需要记录模拟奖励与访问次数。由于节点分为玩家的回合与电脑的回

合，我们将其设置为针对电脑方计算奖励。如果当前回合是电脑，且电脑胜利，加 1 分。如果当前回合是人类，且人类胜利，减 1 分。其他情况类同，在后述的 bestChild()方法中进行分类。

```
class Node {
    public State state;    //当前局面
    public Node parent;    //父节点
    public HashMap<Action, Node> children = new HashMap<Action, Node>();
  //针对子节点键值对集合，定义一个空的映射。
    public LinkedList<Action> untriedActions;   //可供选择的行动集合
    public int Q_FOR_COMPUTER = 0;    //当前节点的模拟奖励（对于电脑方）
  public int N = 0; //当前节点的访问次数

    public Node(State state, Node parent) { //Node 类的初始化方法
      this.state = state;
      this.parent = parent;//如果当前节点是根节点，此值为null
         this.untriedActions = this.state.getUntriedActions();
    }
}
```

由于可供选择的下一步集合是由当前局面产生，因此在 state 类中补充方法 getUntriedActions()用于通过一个棋盘获得所有可供选择的动作集合：

```
State:
//通过一个棋盘获得所有可供选择的动作集合
public LinkedList<Action> getUntriedActions() {
    定义一个空链表 untriedActions;
    for(遍历棋盘获得每个坐标) {
        if (当前坐标没有棋) {
            根据当前棋盘的 x 与 y 产生一个 Action;
            在链表 untriedActions 中添加此 Action;
        }
    }
    打乱列表顺序;    //保证可选择的行动集是无序随机的
    return untriedActions;
}
```

采用自顶向下的代码编写方式，定义 MCTS 类，用于算法步骤。

```
class MCTS {
    public MCTS() { //MCTS 类的初始化方法
    }
}
```

MCTS 类中，需要定义一个主方法 UCTSearch()。此方法的作用是，按照 MCTS 算法的步骤，通过传入一个棋盘与当前下棋方完成搜索操作，并最终返回一个坐标 Action，用于电脑方判断落子位置。

```
MCTS:
//主方法
public Action UCTSearch (int[][] board,int player) {
    通过传入的棋盘 board 与玩家 player，产生一个局面 state;
    通过局面 state 产生根节点 Node;
    //算法搜索步骤:
    for (只要算力在允许的范围内) {
        Node node = treePolicy(root);  //选择与扩展，找出一个值得进行模拟的节点
        int winner = node.rollout();   //模拟此节点，返回一个确定的游戏结果
        node.backup(winner); //反向传播模拟结果
    }
    Action action = root.bestChild(0.0).getKey();//选择胜率最大的子节点对应的动作
    return action
}
```

2. 选择与扩展：treePolicy()方法

上述 MCTS 算法的基本步骤中，选择与模拟通过 treePolicy()完成。如果一个节点是终端节点，将直接返回。如果不是终端节点，且已经完全展开，将从所有子节点中选择一个节点继续上述判断，直到来到一个未完全展开的节点，通过探索此未完全展开的节点，得到一个新的节点。在代码表示中，C_Param 是一个超参数，用于权衡胜率与探索的比例。

```
MCTS:
//用于选择与扩展的方法
public Node treePolicy(Node node) {
    while (!node.isTerminalNode()) {  //如果节点不是终端节点
        if (!node.isFullyExpanded()) { //如果子节点没有完全展开
            return node.expand();  //对此节点进行探索，得到一个新的值得模拟的节点
        } else {  //如果子节点已经完全展开
            node = node.bestChild(C_Param).getValue();  //从子节点中挑选一个最优节点
        }
    }
    return node;
}
```

在此方法中，存在一些未定义的方法，因此优先将它们定义：isTerminalNode()，isFullyExpanded()，expand()，bestChild()。

Node 类中的 isTerminal()方法：用于判断一个节点是否是终端节点。如果一个节点的游戏状态已经确定，则说明是终端节点。由于游戏状态与 State 类有关，为了保证类的功能明确，让 state 类中的 isOver()方法检查游戏状态，并将当前游戏状态返回给 isTerminalNode()方法，由 isTerminalNode()方法判断游戏是否结束。

```
Node:
//用于判断是否是终端节点
public boolean isTermanalNode() {
    //根据 State 类提供的游戏状态，判断当前节点是否是终端节点
    return this.state.isOver() == NOT_WIN;
}
```

在 State 类中相应定义 isOver()方法，判断游戏状态：

```
State:
//判断游戏状态：HUMAN_WIN，COMPUTER_WIN，ALL_FILLED，NOT_WIN
public int isOver() {
    for(遍历棋盘) {
      for (从四个方向判断) {
          if (某一方向存在五子连珠) {
                return 当前位置的颜色;
             }
          }
    }
  if (棋盘下满了) {
      return ALL_FILLED;   //此值表示棋盘全部下满
  }
    return NOT_WIN; //没有获取到任何结束状态，说明棋局未定
}
```

Node 类中的 isFullyExpanded()方法：判断当前节点是否已经完全展开。完全展开指的是所有子节点至少被访问过一次，这说明可选择的动作集合中不存在任何动作（所有动作已经生成了子节点），并且所有已经产生的子节点的 N 值都大于 0。在代码实现中，我们首先考虑可选择的动作集合是否仍存在动作，如果存在动作，说明存在未产生的子节点，即当前节点不是完全展开节点。如果不存在动作，进一步检查所有子节点的 N 值是否都大于 0，如果存在 N 值为 0 的子节点，说明当前节点不是完全展开节点。只有两步检查都通过时，才说明当前节点是完全展开的。

```
Node:
//判断当前节点是否已经完全展开
public boolean isFullyExpanded() {
    if (可选择的动作集合的长度 != 0 || 子节点映射集合的长度 == 0) {
```

```
        return false;
    }
    for (遍历子节点集合) {
        if (子节点的N值 == 0) {
            return false;
        }
    }
    return true;
}
```

Node 类中的 expand()方法：用于未完全展开节点的扩展。由于在 treePolicy()方法中,已经通过 isTerminalNode()和 isFullyExpanded()方法将终端节点与完全展开节点排除，因此如果一个节点调用了此方法，说明此节点必定是非终端节点和非完全展开节点，即此节点中的 untriedActions 存在元素,或者是此节点的子节点中存在 N 值为 0 的节点。扩展思路是优先在 untriedActions 中寻找满足条件的 Action，如果 untriedActions 是空列表，再从 children 子节点映射中寻找。因此对于扩展操作，同样需要分类讨论：

①对于 untriedActions 存在元素的状况,需要选出一个 Action 并创建相应的新节点，然后将 Action 从 untriedActions 中删掉并将 Action 和新节点在子节点列表中注册，最终返回子节点。

②对于 untriedActions 不存在元素，但是子节点中存在 N 值为 0 的节点的状况，直接返回此子节点即可。

③对于 untriedActions 不存在元素，且子节点中不存在 N 值为 0 的节点的状况，将无法获取任何子节点，因此最终将返回错误的 null。

```
Node:
public Node expand() {
    if (可选择的行动集的长度 != 0) {
        选出一个 action，并在 untriedActions 中删除此 action;
        根据 action 与调用此方法的 node，得到下一步状态 state;
        根据 state 与下一步玩家建立一个新的节点 newChild;
        将 action 与 newChild 组成的键值对注册到 children;
        return newChild;
    } else if (子节点映射 children 的长度 > 0) {
        for (遍历 children) {
            if (存在 N 值为 0 的子节点) {
                return 子节点;
            }
        }
    }
    return null;
}
```

Node 类中的 bestChild()方法：当所有子节点都至少被访问过一次后，无法再继续扩展，只能通过此方法得到下一步值得进行模拟的节点。为了能够灵活的选取键或值，此方法最终返回的是键值对对象。

```
Node:
    public Map.Entry<Action, Node> bestChild(double c) {
      for (遍历子节点映射) {
            获得一个节点 child;
            计算 child 的 UCB 值;
        }
        获得以 UCB 最大的子节点为值的键值对对象 bestChildObject;
        return bestChildObject;       //返回键值对对象
    }
```

在上述代码中，“计算 child 的 UCB 值”过程如下：

```
int left = child.Q_FOR_COMPUTER / child.N;
int right = c * Math.sqrt((2 * Math.log(child.parent.N)) / child.N);
UCB = left + right;
```

3．模拟：rollout()方法

通过 treePolicy()方法得到的节点有三种可能性：

①节点是终端节点。对于这种情况，应在 rollout 中首先判断出来，不进行模拟并直接返回游戏结果。

②节点是通过扩展得到的。由于在扩展的时候已经排除了终端节点和完全展开的节点，因此节点中必定存在未展开的子节点或者访问次数为 0 的子节点。这种情况需要对当前节点进行模拟，直到模拟到游戏结束。为了防止在模拟时改变节点的棋盘状态，需要对节点的 state 属性进行深拷贝，保证得到的新棋盘与当前状态棋盘完全隔离。

③节点是通过 bestChild()方法计算得到的。由于在执行此方法之前，已经将终端节点排除，因此这种状态同样不是终端节点，需要对节点进行模拟。模拟方法与②相同。

```
Node:
//模拟
public int rollout() {
    if (当前节点胜负已分) {
        return 游戏结果;
    }
    State currentState = this.state.stateDeepCopy(); //拷贝当前局面
    while (胜负未分) {
        通过 getUntriedActions()方法获取 curerntState 的所有可落子位置集合;
        随机选出一个 Action;
        通过 Action 改变 currentState 的棋盘状态;
```

```
        currentState = 产生的下一个局面;
    }
    return 游戏结果;
}
```

4. 反向传播：backup()方法

此方法只能更改调用者的状态，并通过递归的方式“提醒”父节点需要更改自己的状态，而不是直接改变父节点的状态。在传播的过程中，节点需要改变自身的 Q 值与 N 值。由于 N 值反映的是访问次数，因此每进行一次反向传播，节点的 N 值就会增加一次。但 Q 值是用来描述电脑方的胜算，因此需要根据当前的下棋方与模拟结果分类讨论：

```
Node:
//反向传播
public void backup(int winner) {
    this.N++;
    if (当前回合是电脑) {
        if (电脑胜利) {
            this.Q++;
        } else {
            this.Q--;
        }
    } else {
        if (玩家胜利) {
            this.Q--;
        } else {
            this.Q++;
        }
    }
    if (父节点存在) {
        父节点.backup(winner);
    }
}
```

5. 选择下一走法

最终选择胜率最大的子节点。胜率最大意味着完全抛弃探索分量，因此最终让 UCB1 方法中的超参数 c = 0，即可得到胜率最大的点。

（二）五子棋界面的实现

目前程序分为三个类与一个接口：Board 类、BoardConfig 接口、Computer 类、Main 类。

1.Board类

五子棋的面板类，继承自 JPanel，实现下棋功能。此类中定义了两个成员内部类：按钮监控类与面板监控类，分别用于按钮的动作监听与面板的鼠标监听。棋局的状态通过面板上一个带滑动条的文本域显示。此文本域默认位置为跟随光标的位置。此面板被 JFrame 包裹。

①当棋手下一步棋时，会另开一个 Computer 类线程，完成电脑的判断，并通过改变面板类的二维数组，实现电脑落子。

②判断五子棋的输赢情况。当存在五子相连时，程序会进行胜负判断显示在文本域上，并不再允许落子以及不允许电脑线程启动，并将胜负标记设置为 true（分出胜负）。如果棋局未分出胜负，将不做任何处理。

③实现重置棋盘功能。当点击重置棋盘按钮时，胜负标记会被重置为 false（未分胜负），存储棋子的二维数组被清空，先手标记会重置为先手方（先手方参数可在 BoardConfig 接口设置，可选项为 BLACK 或 WHITE）并且如果电脑线程已经启动，将会停止电脑线程。最后清空文本域，并在文本域输出已重置信息。

④解决落子冲突。当电脑进程在启动过程中，如果检测到棋手点击棋盘范围，将不产生落子动作。并在文本域上显示提示信息。

2．BoardConfig接口

设置五子棋的参数。可调整的参数为棋盘大小、棋盘的单位格子大小、先手颜色、棋手与电脑执棋颜色。除此之外，其他参数都以相对长度计算。调整参数的目的是在编程时调试程序，实际运行过程中，将不调整任何参数。（其他参数包括：面板大小与位置、按钮大小与位置、窗体大小与位置、棋盘的背景颜色）。

3.Computer类

设置五子棋的参数。可调整的参数为棋盘大小、棋盘的单位格子大小、先手颜色、棋手与电脑执棋颜色。除此之外，其他参数都以相对长度计算。调整参数的目的是在编程时调试程序，实际运行过程中，将不调整任何参数（其他参数包括：面板大小与位置、按钮大小与位置、窗体大小与位置、棋盘的背景颜色）。

4.Main

程序的入口，创建 Board 类对象。

五、展望

Monte Carlo Tree Search 在棋局庞大的局面下，尤其是围棋博弈中，起到了非常显著的作用。在 AlphaGo 程序中，研究者对常规的 MCTS 算法做了大量优化：

AlphaGO 主要由四个神经网络构成：快速走子网络（判断准确度约 20%）、监督学习（SL）策略网络（判断准确度约 55%）、强化学习（RL）策略网络、价值网络。每一个神

经网络都可以当作是一个独立的程序，但是棋力会下降。由于目标是人类顶级水准，因此 Deepmind 采用四个网络协同工作的方式。

①快速走子网络结构简单，判断速度比 SL 策略网络快，因此用于 Rollout Policy。

②SL 策略网络则存储了大量的人类对弈记录，用于训练棋力。

③RL 策略网络用于复制 SL 中的数据后进行自我博弈，博弈完成后将结果传回 SL 策略网络。因此只要服务器一直开启，AlphaGO 的棋力会不断变强。但 RL 策略网络的作用不仅用于自我博弈，还将结果提供给价值网络，用于在 MCTS 中做出判断。

④价值网络的功能类似于 UCT 函数，即给一个局面评分。此网络中并没有直接采用 SL 策略网络的人类记录，而是 RL 网络自己模拟出的记录。原因是人类的对弈数据大致相同，有很强的自相关性，这会导致过度拟合使判断失误。

在 MCTS 算法过程中，当扩展到一定程度时，通过快速走子网络模拟出剩余步数的胜率。虽然在模拟过程中会存在误差，但是由于模拟的次数很多，误差之间可以相互抵消，因此最终的结果依然可以反映出真正的胜率。然后用价值网络通过权衡奖励和访问次数来确定这个未完全展开点的胜率，最后反向传播回根节点，让系统做出走子判断。

在五子棋设计中，也可以借鉴 AlphaGo 的方式，采用神经网络进行自我博弈。并且在 Rollout Policy 中，可以进行多次模拟实验，相对于单次模拟实验，这种做法更加准确。同时也可以设计成多线程程序，将不同节点信息隔离，并分配给不同线程同时判断，可增加 MCTS 的效率。

参考文献

[1]董红安. 计算机五子棋博奕系统的研究与实现[D].山东师范大学,2005.
[2]董慧颖,王杨.多种搜索算法的五子棋博弈算法研究[J].沈阳理工大学学报,2017,36(02):39-43+83.
[3]王杨. 基于计算机博弈的五子棋算法研究[D].沈阳理工大学,2017.
[4]林云川. 基于深度学习和蒙特卡洛树搜索的围棋博弈研究[D].哈尔滨工业大学,2018.
[5]曾小宁.五子棋中的博弈问题[J].广东教育学院学报,2003(02):96-100.
[6]肖齐英,王正志.博弈树搜索与静态估值函数[J].计算机应用研究,1997(04):76-78.
[7]刘瑞. 五子棋人工智能算法设计与实现[D].华南理工大学,2012.
[8]Silver David,Schrittwieser Julian,Simonyan Karen,Antonoglou Ioannis,Huang Aja,Guez Arthur,Hubert Thomas,Baker Lucas,Lai Matthew,Bolton Adrian,Chen Yutian,Lillicrap Timothy,Hui Fan,Sifre Laurent,van den Driessche George,Graepel Thore,Hassabis Demis. Mastering the game of Go without human knowledge.[J]. Nature,2017,550(7676).
[9]Browne C B , Powley E , Whitehouse D , et al. A Survey of Monte Carlo Tree Search Methods[J]. IEEE Transactions on Computational Intelligence & Ai in Games, 2012, 4(1):1-43.
[10]Wang, Junru & Huang, Lan. (2014). Evolving Gomoku solver by genetic algorithm.[J].Proceedings - 2014 IEEE Workshop on Advanced Research and Technology in Industry Applications, WARTIA 2014. 1064-1067. 10.1109/WARTIA.2014.6976460.

致 谢

我的大学学习生涯即将结束，在本科论文完成之际，我要向所有给过我帮助的老师、家人与同学们致谢。

我要感谢我的导师赵志崑教授。赵老师曾教过我的 JavaEE 课程，在课堂学习期间，赵老师亲切风趣的师德与专业严谨的治学作风让我在钻研求学与待人接物中受益终身。在论文指导过程中，老师独到的学术见解拓展和丰富了我的知识视野与专业水平，在此向老师致以最诚挚的谢意。

我要感谢计算机科学与技术学院的所有老师。感谢老师们四年的辛勤栽培，在教学的同时更多的是传授我们做人的道理，感谢老师们孜孜不倦的教诲。

我要感谢我的父母在学习与生活中给我的关心与照顾。在写论文期间，他们为了让我专注于研究，尽量不进房间打扰我学习与写作，而我也因没能照顾好他们而深感内疚。十分感谢父母的理解与支持，祝愿父母身体健康，每天开心！

我要感谢我的同学杨帅，在程序设计与论文写作过程中给予了我帮助。当我遇到瓶颈时，通过沟通我的疑惑便豁然开朗。

最后我要感谢学校对我四年的培养。时光飞逝，转眼间我就毕业了。回首四年，虽然平淡，但我却在不停地成长。从一个孩子变成一个有了责任感的大人，我相信是学校的学术氛围让我对学习产生了浓厚的兴趣。

我十分享受学习与总结的过程，相比无所事事的状态，专注研究这件事虽然很累，但十分有收获。我想这也是这篇论文的初衷。五子棋程序中出现过各种各样的失误，而我也在调试与总结中逐渐学会了耐心、细心、坚持等品质。所以我想感谢自己，在以后的生活与学习中，我也会保持这种对生活充满动力的状态，去迎接可能会出现的各种挑战。