## 智能五子棋

### 1 实验任务

·编写程序实现蒙特卡洛树搜索算法，作为五子棋的下棋算法

·编写程序实现极大极小+α-β剪枝算法，作为五子棋的下棋算法

·将两种算法进行博弈对抗，并实现下棋过程的可视化

### 2 实验原理

#### **2.1 蒙特卡洛树搜索算法**

**2.1.1 算法思想**

蒙特卡洛树搜索算法的核心思想是蒙特卡洛方法，它是一种统计模拟方法，其以概率作为算法基础，通过重复独立实验来对真实值进行估计。假设我们要计算一个不规则形状的面积，我们只需在包含这个不规则形状的矩形内，随机的掷出一个点，每掷出一个点则N+1，如果这个点在不规则图形内则W+1，落入不规则图形的概率即为W/N，于是当掷出足够多的点之后，我们可以认为：不规则图形面积＝矩形面积\*W/N。

在本文的五子棋实验中，双方在某个局面下随机走子，走到终局或者残局为止，随机很多次后计算胜率，那么胜率越高的局面就越好，据此可以选择下一步走子的方位，这就是算法设计的关键所在。

**2.1.2 探索与利用**

探索就是向未知的领域勇敢的进发，如果不进行新的尝试，则永远无法找到比当前更好的方法。利用就是经验万岁，经验有时候虽然不靠谱，但大多数时候还是管用的，而且尝试得越多经验就越靠谱。探索也好利用也罢，怎么去确定他们的分配比例呢？这就是难题所在了，比如我们可以设置一个概率参数p，以p的概率探索，以1-p的概率利用，而在实验中用到的的UCB也是其中的一种，其公式如下所示：

其中，是节点的胜率估计，是节点被访问次数，N是其父节点被访问次数，C则是一个超参数——探索因子。

**2.1.3 基本步骤**

*选择：*从根节点开始，我们可以选择节点的UCB值最大的子结点作为下一个操作的对象，如此向下搜索，直到来到树的底部的叶子节点，等待下一步操作

*扩展：*到达叶子节点后，如果还没有到达终止状态，那么就要对这个节点进行扩展，扩展出一个或多个节点（也就是进行一个可能的action然后进入下一个状态）

*模拟：*基于目前的状态，根据某一种策略进行模拟，直到游戏结束为止，产生结果，比如胜利或者失败

*回溯：*根据模拟的结果，自底向上反向更新所有节点的信息

其完整的流程如图1所示

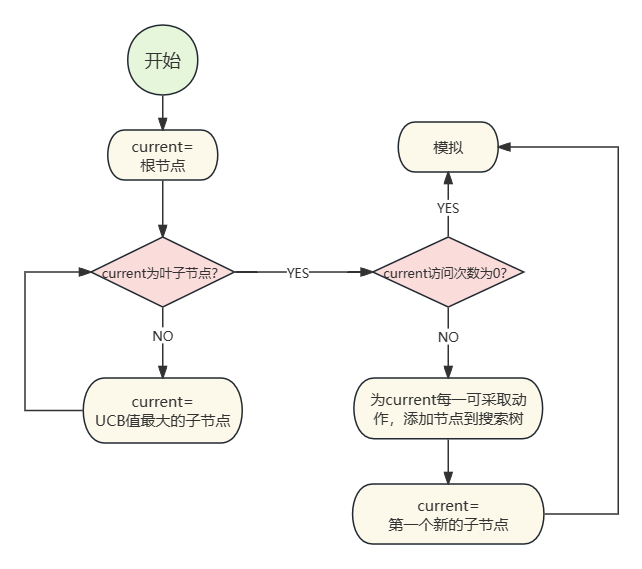


图1

#### 2.2 极大极小+α-β剪枝算法

**2.2.1 算法原理**

在零和博弈中，玩家均会选择将其N步后优势最大化或者令对手优势最小化的做法。将双方决策过程视作一颗决策树，若决策树某一层均为己方决策依据状态，则己方必定会选择使得己方收益最大化的路径，将该层称为MAX层。若决策树某一层均为对手决策依据状态，则对手必定会选择使得己方收益最小化的路径，将该层成为MIN层。由此，一个极大极小决策树将包含max节点（MAX层中的节点）、min节点（MIN层中的节点）和终止节点（博弈终止状态节点或N步时的状态节点），每个节点对应的预期收益成为该节点的minimax值。

**2.2.2 算法过程**

极小化极大算法过程可描述如下：

1）构建决策树

2）将评估函数应用于叶子结点

3）自底向上计算每个结点的minimax值

4）从根结点选择minimax值最大的分支，作为行动策略

minimax计算流程如下：

1）如果节点是终止节点：应用估值函数求值

2）如果节点是max节点：找到每个子节点的值，将其中最大的子节点值作为该节点的值

3）如果节点时min节点：找到每个子节点的值，将其中最小的子节点值作为该节点的值

**2.2.3 估值函数**

前面的蒙特卡洛树搜索算法是通过对当前局面进行多次模拟，根据结果对局面的好坏进行评判，与之不同的是，极大极小法需要我们根据游戏规则自行设计方法评判局面的好坏，这就是所谓的估值函数，其设计的优劣将直接影响到算法的好坏。

在五子棋中，最常见的基本棋型大体有以下几种：连五，活四，冲四，活三，眠三，活二，眠二。

1. 连五：顾名思义，五颗同色棋子连在一起，不需要多讲
2. 活四：如图2中有两个连五点（即有两个点可以形成五），白点即为连五点，活四出现的时候，如果对方单纯过来防守的话，是已经无法阻止自己连五了

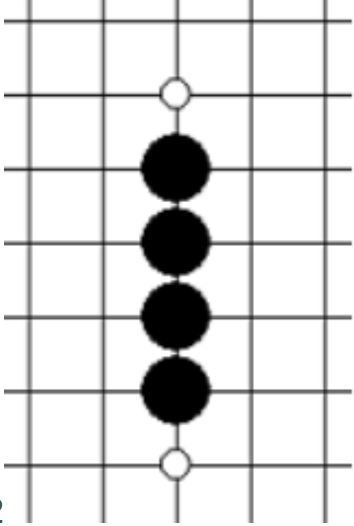


图2 活四

1. 冲四：如图3中有一个连五点则为冲四棋型，对方只要跟着防守在那个唯一的连五点上，冲四就没法形成连五

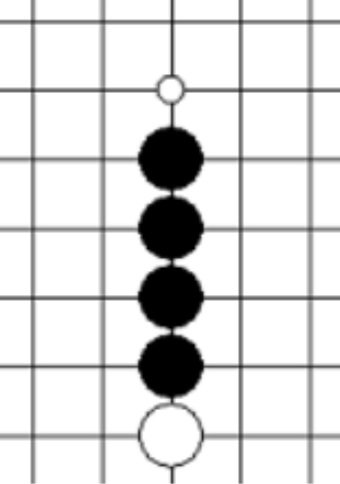
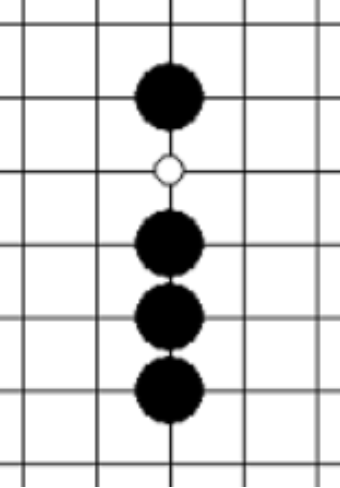
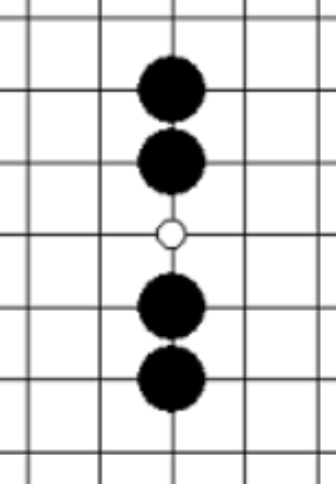
  

图3 冲4

1. 活三：如图4中可以形成活四的三，活三之后，如果对方不以理会，将可以下一手将活三变成活四，而我们知道活四是已经无法单纯防守住了

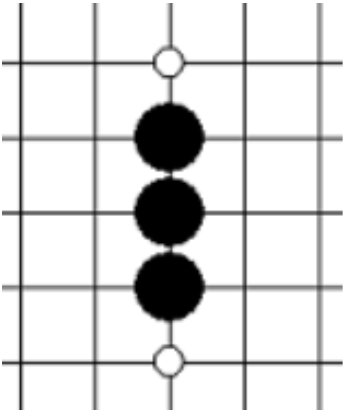
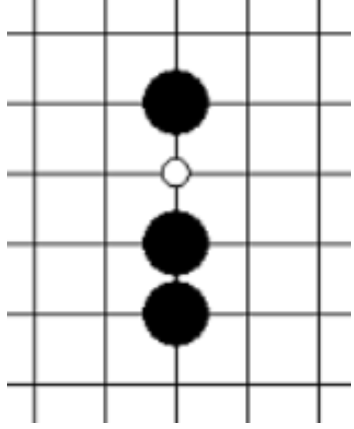
 

图4 活三

1. 眠三：如图5中只能够形成冲四的三，眠三的棋型与活三的棋型相比，危险系数下降不少，因为眠三棋型即使不去防守，下一手它也只能形成冲四，而对于单纯的冲四棋型是可以防守住的

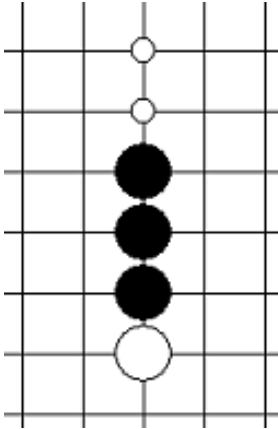
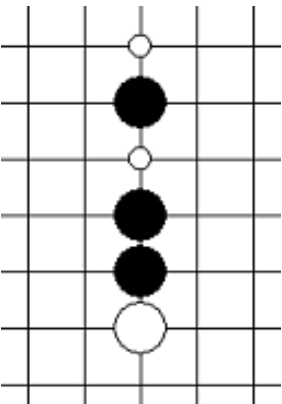
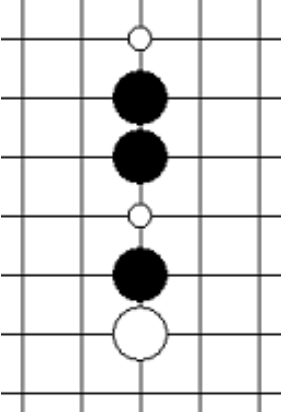
  

图5 眠三

1. 活二：如图6中能够形成活三的二，活二棋型看起来似乎很无害，因为等形成活三再防守也不迟，但其实活二棋型是非常重要的，尤其是在开局阶段，我们形成较多的活二棋型的话，当我们将活二变成活三时，才能够让对手防不胜防

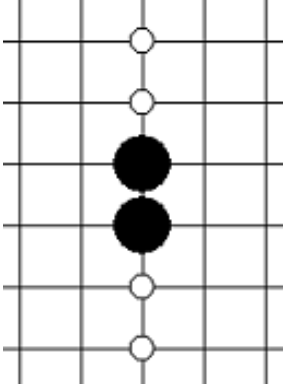
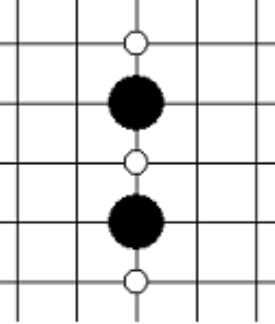
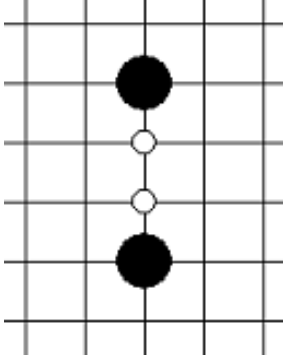
  

图6 活二

1. 眠二：如图7中能够形成眠三的二

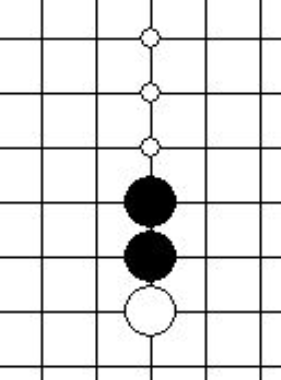
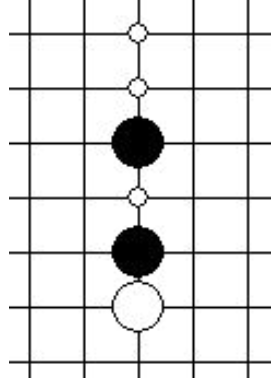
 

图7 眠二

最后，根据棋盘上黑棋和白棋的棋型统计信息，按照一定规则进行评分。假设棋局最后一步是黑棋下的，则可以制定如下规则：

1）黑棋连5，评分为10000

2）白棋连5，评分为 -10000

3）黑棋两个冲四可以当成一个活四

4）白棋有活四，评分为 -9050

5）白棋有冲四，评分为 -9040

6）黑棋有活四，评分为 9030

7）黑棋有冲四和活三，评分为 9020

8）黑棋没有冲四，且白棋有活三，评分为 9010

9）黑棋有2个活三，且白棋没有活三或眠三，评分为 9000

10）针对黑棋或白棋的活三，眠三，活二，眠二的个数依次增加分数，评分为（黑棋得分 - 白棋得分）

其中，前面9条规则为必杀情况，将直接返回评分，具体的评分值也可以进行优化调整。

**2.2.4 α-β剪枝**

剪枝是希望在搜索的时候，根据已搜索的结果，剔除超出最优解的分支，那么意味着这个分支下的所有节点都不需要考虑了，大大降低了搜索的次数。

α−β剪枝的名称来自计算过程中传递的两个边界，这些边界基于已经看到的搜索树部分来限制可能的解决方案集。其中，α表示目前所有可能解中的最大下界，β表示目前所有可能解中的最小上界。

  因此，如果搜索树上的一个节点被考虑作为最优解的路上的节点（或者说是这个节点被认为是有必要进行搜索的节点），那么它一定满足以下条件（N当前节点的估价值）：

α ≤ N ≤ β

  在我们进行求解的过程中，α和β会逐渐逼近。如果对于某一个节点，出现了α > β的情况，那么，说明这个点一定不会产生最优解了，所以，我们就不再对其进行扩展（也就是不再生成子节点），这样就完成了对博弈树的剪枝。

### 3 程序设计

#### **3.1 文件描述**

**gobang.py：**实现棋盘可视化

·draw(ax, chess)：显示当前棋盘

·check(i, j, v, chess)：在(i, j)处落子v形成棋盘chess后，判断v是否赢棋

**mcts.py：**蒙特卡洛树搜索算法

·select(node)：选择

·expand(node)：扩展

·rollout(state, root)：模拟

·backword(node, value)：回溯

·mcts(root, max\_iteration)：算法实现

**max\_min.py：**极大极小法+α-β剪枝

**evaluate.py：**极大极小法中使用的估价函数

**main.py：**两种算法对弈

#### 3.2 关键代码

***蒙特卡洛树搜索算法***

def mcts(root, max\_iteration):

for iteration in range(max\_iteration):

current\_node = root

while current\_node.children != []:

current\_node = select(current\_node)

if current\_node == root or current\_node.visits != 0:

expand(current\_node)

current\_node = current\_node.children[0]

value = rollout(current\_node.state, root)

backword(current\_node, value)

return max(root.children, key=lambda child: child.value).state

***极大极小法+α-β剪枝***

def max\_min(root, max\_deep):

evaluate = Evaluate(10)

v\_min = root.state.v

v\_max = v\_min % 2 + 1

stack = [root]

while stack != []:

current = stack[-1]

if current.points == []:

# 当前结点后代全部探索完

parent = current.parent

if parent:

grand = parent.parent

if grand:

if current.state.v == v\_min and parent.alpha < current.beta:

parent.alpha = current.beta

if parent.alpha >= grand.beta:

parent.points = []

if current.state.v == v\_max and parent.beta > current.alpha:

parent.beta = current.alpha

if parent.beta <= grand.alpha:

parent.points = []

stack.pop()

else:

deep = current.deep + 1

point = current.points.pop()

i, j = point[1], point[2]

v = current.state.v % 2 + 1

chess = copy.deepcopy(current.state.chess)

chess[i][j] = v

state = State2(i, j, v, chess)

node = Node2(deep, state, parent=current)

if deep == max\_deep:

node.points = []

node.alpha = evaluate.evaluate(chess, v\_max)

else:

node.points = get\_points(chess)

current.children.append(node)

stack.append(node)

return max(root.children, key=lambda child: child.alpha).state

### 

### 4 实验结果与分析

#### 4.1 运行示例

如图8所示，蒙特卡洛树搜索算法作为黑方的下棋算法，极大极小+α-β剪枝算法作为白方的下棋算法。

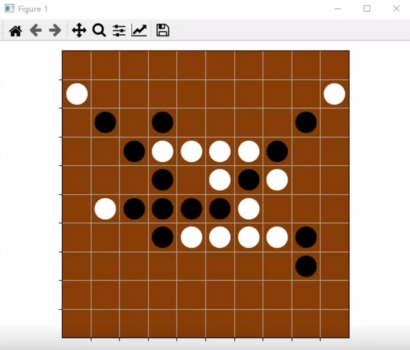
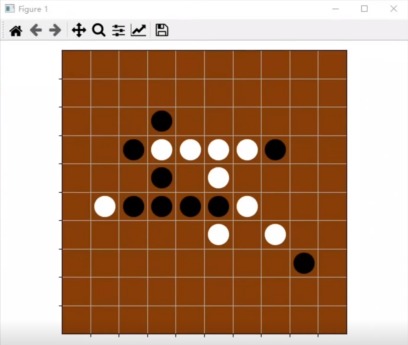
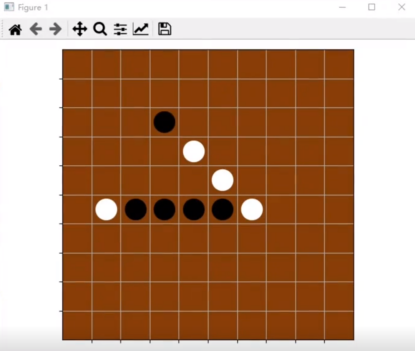


图8

#### 4.2 结果分析

在本次实验中，我已基本实现智能五子棋中的两种算法，并对下棋过程实现可视化，结果较为理想。但较为遗憾的是，程序在运行过程中十分卡顿，当后面下棋步数较多时甚至会出现崩溃的情况。经过分析知道，这是因为两种算法运算量都十分庞大，而本人电脑配置一般算力有限。

为了使程序正常运行，我只能降低蒙特卡洛树搜索的迭代次数max\_iteration和极大极小法中搜索树的深度max\_deep，直至max\_iteration=500以及max\_deep=3，程序才能勉强正常运行，但这也导致两种算法的效果变差，对于蒙特卡洛树搜索算法尤其明显。总之，如何在降低算法运算量的同时增强其效果，将是我后续研究的重点。