**基于评估函数与Alpha Beta剪枝的五子棋AI算法设计**

姓名： 江淏

学号： 2023080911007

完成日期：2023年12月

**摘要**

不止是五子棋，在任何棋类竞技中，双方对于比赛未来的走向预测会形成一棵巨大的博弈树。博弈树中双方可以选择的路线有很多，我们不可能把每一条路径的结果都计算得出再进行比较，因此需要引入一个评估函数来计算当下局势的相对得分，得分越高表示局势对该方越有利。这样在有限层级的预测里，我们就能了解各个路径的价值高低，最后比较得出AI本次行棋的“最”优解。但是随着层数的递增，博弈树中的枝叶与节点数量会指数级增长，这将不利于我们让AI在有限时间内进行更深层的运算。随后我们引入了Alpha Beta剪枝算法，顾名思义就是直接去除某些不可能被选上的节点，来达到优化运算时间的效果。

**关键词：五子棋，评估函数，Alpha Beta剪枝**

**目录**

基于评估函数与Alpha Beta剪枝的五子棋AI算法设计 ---------------------------------- 1

摘要 ------------------------------------------------------------------------------------------------------ 2

1. 设计目的 ------------------------------------------------------------------------------------ 4

第二章 设计基础 ------------------------------------------------------------------------------------ 5

2.1 Alpha Beta剪枝算法的发展历史 ----------------------------------------------------- 5

2.2 Alpha Beta剪枝算法的优化原理 ----------------------------------------------------- 6

第三章 算法设计 ------------------------------------------------------------------------------------ 7

3.1 AI基础设定 --------------------------------------------------------------------------------- 7

3.1.1 邻居概念 ----------------------------------------------------------------------- 7，8

3.1.2 评估算法 ---------------------------------------------------------------------- 8-10

3.2 Alpha Beta剪枝应用 -------------------------------------------------------------------- 10

3.2.1 AI主函数 --------------------------------------------------------------------- 10-13

3.2.2 Alpha Beta剪枝应用 ----------------------------------------------------------- 13

第四章 运行结果分析 ----------------------------------------------------------------------- 13，14

第五章 总结 ----------------------------------------------------------------------------------------- 15

1. **设计目的**

类似围棋、国际象棋等棋类，五子棋也是一类十分热门的完备信息零和博弈游戏。其中完备信息代表了每时每刻双方对于棋盘信息的共享，零和博弈则指博弈双方的局势不可能同为优势或劣势。这类普及程度仅次于国际象棋的棋类游戏，具有聚集博弈典型意义，容易深入研究，且博弈结果能够直观反映。其实早在上世纪，人类就已经着手于计算机与五子棋的结合，只不过由于计算能力的限制与算法设计的不足，早期AI并没有想象中那么强大。与人类棋手相比，AI具有更强大的计算能力，同时永不疲倦，能够一直保持高水平行棋，0失误对弈。在人机博弈中，双方将回合制进行走棋，己方考虑当自己在所有可行的走法中作出某一特定选择后对方可能会采取的走法并进行对比，从而选择最有利于自己的走法。这种对弈过程构成了博弈树，双方在博弈树中不断搜索，选择对自己最为有利的子节点走棋。在搜索的过程中，我们取极大值的一方称为max，取极小值的一方称为min。max总是会选择价值最大的子节点走棋，而min则相反，这就是极小化极大算法的核心思想。但是极小化极大算法最大的缺点就是会造成数据冗余，而这种冗余有两种情况：极大值冗余和极小值冗余。相对应，Alpha剪枝用来解决极大值冗余问题，Beta剪枝则用来解决极小值冗余问题，这就构成了完整的Alpha Beta剪枝。

本次实验中，我在能力范围内将尝试通过自定评估函数与最基础Alpha Beta剪枝的辅助，尝试编写出具有一定对弈能力的五子棋AI。

1. **设计基础**

**2.1 Alpha Beta剪枝算法的发展历史**

1950年代，剪枝算法在解决棋类游戏中的搜索问题时首次出现。以国际象棋为例，算法需要评估成千上万种可能的走棋组合，剪枝算法通过减少无效或次优路径的考虑，显著提高了搜索效率。1956年，John McCarthy首次提出Alpha Beta剪枝算法，该算法通过确定上下界（Alpha和Beta值），来避免探索对结果不产生影响的分支，这种方法使得计算机在棋类游戏中首次能够与人类选手竞争。1970年代，随着计算能力的提升和算法理论的深入，剪枝技术被进一步优化，特别是在处理更大的搜索树时，引入了如迭代深化搜索（IDS）和最佳优先搜索（Best-First Search）等新方法，这些方法能够更有效地管理资源和时间。1980年代，启发式评估方法的发展为剪枝算法带来了新的应用前景，特别是在复杂的决策环境中，如高级国际象棋程序和复杂的策略游戏。

21世纪，剪枝算法被广泛应用于机器学习、数据挖掘和大数据分析中。在构建决策树和随机森林等模型时，剪枝算法帮助减少过拟合，提高模型的泛化能力。随着数据量和维度的激增，剪枝算法面临着新的挑战，如如何有效处理和分析高维数据集。这引发了对更高效剪枝技术的需求，以及对算法在不同数据规模和复杂性下的适应性研究。

现代剪枝算法开始结合并行计算和云计算技术，以应对大规模数据分析的需求，这不仅提高了数据处理速度，也为算法提供了更大的灵活性和扩展性。

这些事例展示了剪枝算法从早期的基础应用到现代高级数据处理的演变过程，突显了这些算法在解决现代社会和科学问题中的核心作用。

**2.2 Alpha Beta剪枝算法的优化原理**

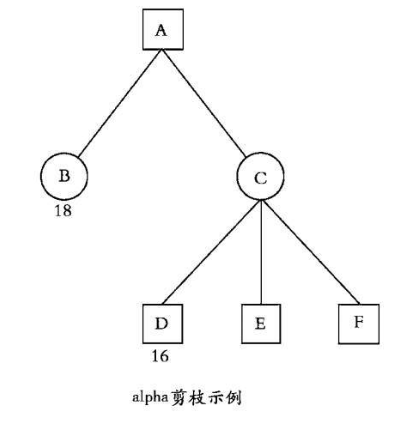
Alpha Beta剪枝是一种搜索算法，用以减少极小化极大算法搜索树的节点数。这是一种对抗性搜索算法，主要应用于机器游玩的二人游戏。当算法评估出某策略的后续走法比之前策略的还差时，就会停止计算该策略的后续发展。该算法和极小化极大算法所得结论相同，但剪去了不影响最终决定的分枝。

Alpha Beta的优点是减少搜索树的分枝，将搜索时间用在“更有希望”的路径上，继而提升搜索深度。该算法和极小化极大算法一样，都是分支限界类算法。若节点搜索顺序达到最佳优化或近似最佳优化，则同样时间内搜索深度可达极小化极大算法的两倍多。

在平均路径数量为b项，搜索深度为d层的情况下，要评估的最大路径数目为O(b)，即和简单极小化极大搜索一样。若始终优先搜索最佳方案，则需要评估的最大路径数目按层数奇偶性，分别约为O(b) O(根号b)。其中层数为偶数时，搜索因子相当于减少了其平方根，等于能以同深度搜索两次。其中对第一名玩家必须搜索全部招法找到最佳招式，但对于它们，只用将第二名玩家的最佳招法截断，因为Alpha Beta确保了无需考虑第二名玩家的其他招法。但在实际运用过程中，由于路径生成顺序随机，实际需要评估的路径数平均约为O(b)。

一般在Alpha Beta中，若招式排序错误，双方的优势会多次切换，每次都让效率下降。随着层数深入，局面数量会呈指数性增长，因此排序早期招式价值很大，尽管改善任意深度的排序，都以能指数性减少总搜索局面。

算法使用两个值Alpha和Beta，分别代表max玩家放心的最高分，以及min玩家放心的最低分。Alpha和Beta的初始值分别为正负无穷大，即双玩家都以可能的最低分开始游戏。在某个特定分枝后，可能发生小分玩家放心的最小分小于大分玩家放心的最大分，即Beta <= Alpha。这种情况下，我们不应选择这个路径，否则在反方向导出的分数会降低，因此该分枝的其他路径也就没有必要继续探索了。

****

**第三章 算法设计**

**3.1 AI基础设定**

**3.1.1 邻居概念**

当AI进行此次行棋的计算时，倘若每次都将棋盘上19×19中的空格子计算一遍，很明显会出现大量的无意义计算。五子棋与围棋大相径庭，并非是在棋盘上某块地方毫无机会时应该尝试新开棋路。与之相反，五子棋中的对抗一般都只会存在一片棋，为了赢而开辟新棋路则至少要多投入5个回合，这显然是不合理的。由此观之，距离“战场”较远的位置就是前面所提的无意义计算点。因此我们引入一个概念——“邻居”。

对于某个坐标为（x0，y0）的棋子A，如果有某个位置B的坐标（x1，y1）与其满足|x1 - x0|+|y1 - y0|<=2，则称B是A的邻居。举个简单的例子，（6，6）就是（5，5）的邻居而不是（5，4）的。

具体代码操作如下：

（其中diatance函数计算了某个位置距离标准位置的绝对值距离，通过后期的排序能更好地优化Alpha Beta剪枝的效果）

|  |
| --- |
| int distance(int a , int b , int c , int d) {  if (a == c && (b + 1 == d || b - 1 == d)) {  return 1;  } else if (b == d && (a + 1 == c || c + 1 == a)) {  return 1;  } else {  return 2;  }  }  int use\_position\_find\_help(int chessboard[15][15] , int i , int j , int f , int i0 , int j0) {  if (chessboard[i][j] != 0 || f == 1 || f == 2) {  return distance(i, j, i0, j0);  } else {  return 0;  } |

|  |
| --- |
| int use\_position\_find(int chessboard[15][15] , int i , int j) {  int f = 0;  if (i >= 1) f = use\_position\_find\_help(chessboard, i - 1, j, f, i, j);  if (i >= 2) f = use\_position\_find\_help(chessboard, i - 2, j, f, i, j);  if (i <= 13) f = use\_position\_find\_help(chessboard, i + 1, j, f, i, j);  if (i <= 12) f = use\_position\_find\_help(chessboard, i + 2, j, f, i, j);  if (j >= 1) f = use\_position\_find\_help(chessboard, i, j - 1, f, i, j);  if (j >= 2) f = use\_position\_find\_help(chessboard, i, j - 2, f, i, j);  if (j <= 13) f = use\_position\_find\_help(chessboard, i, j + 1, f, i, j);  if (j <= 12) f = use\_position\_find\_help(chessboard, i, j + 2, f, i, j);  if (i >= 1 && j >= 1) f = use\_position\_find\_help(chessboard, i - 1, j - 1, f, i, j);  if (i >= 1 && j <= 13) f = use\_position\_find\_help(chessboard, i - 1, j + 1, f, i, j);  if (i <= 13 && j >= 1) f = use\_position\_find\_help(chessboard, i + 1, j - 1, f, i, j);  if (i <= 13 && j <= 13) f = use\_position\_find\_help(chessboard, i + 1, j + 1, f, i, j);  if (chessboard[i][j] != 0) f = 0;  return f;  } |

**3.1.2 评估算法**

该自定义评估算法仅为基础的价值评估，并且不包含禁手的考虑。这里把某个局面下通过评估函数得出的得分称为该局面下的得分，得分越高，则表明该局势对于AI来说越有利。

评估函数主要以五子棋获胜条件为基础，很显然相连同阵营棋子数量越多，对应的局面加分也越多。

以下是各种情况我给出的自定义评分：

五连：10w 活四：1w 活三：1000 活二：100 活一：1

冲四：5000 眠三：100 眠二：10

具体代码操作如下：

|  |
| --- |
| int score\_fomular(int n , int list[15] , int target)  {  int j;  int flag = 0;  int num=0;  for (j = 0; j < n - 1; j++)  {  if(flag == 0) {  if (j <= n - 6 && list[j] + list[j + 1] + list[j + 2] + list[j + 3] + list[j + 4] + list[j + 5]== 6 \* target) {  num += 500000; flag = 6;  }  else if (j <= n - 5 && list[j] + list[j + 1] + list[j + 2] + list[j + 3] + list[j + 4] == 5 \* target) {  num += 100000; flag = 5;  } else if (j <= n - 4 && list[j] + list[j + 1] + list[j + 2] + list[j + 3] == 4 \* target &&  j >= 1 && j <= n - 5 && list[j - 1] == 0 && list[j + 4] == 0) {  num += 50000; flag = 4;  } else if (j <= n - 4 && list[j] + list[j + 1] + list[j + 2] + list[j + 3] == 4 \* target &&  ((j == 0 && list[j + 4] == 0) || (j == n - 4 && list[j - 1] == 0))) {  num += 5000; flag = 4;  } else if (j <= n - 4 && list[j] + list[j + 1] + list[j + 2] + list[j + 3] == 4 \* target &&  j >= 1 && j <= n - 5 && (list[j - 1] + list[j + 4] == -target)) {  num += 5000; flag = 4;  } else if (j <= n - 5 && list[j] == target && list[j + 4] == target &&  list[j] + list[j + 1] + list[j + 2] + list[j + 3] + list[j + 4] == 4 \* target) {  num += 5000; flag = 5;  } else if (j <= n - 3 && list[j] + list[j + 1] + list[j + 2] == 3 \* target &&  ((j == 0 && list[j + 3] == 0 && list[j + 4] == 0) ||  (j == n - 3 && list[j - 1] == 0 && list[j - 2] == 0))) {  num += 100; flag = 3;  } else if (j <= n - 3 && list[j] + list[j + 1] + list[j + 2] == 3 \* target &&  j >= 1 && j <= n - 4 && (list[j - 1] + list[j + 3] == -target)) {  num += 100; flag = 3;  } else if (j <= n - 3 && list[j] + list[j + 1] + list[j + 2] == 3 \* target &&  j >= 1 && j <= n - 4 && list[j - 1] == 0 && list[j + 3] == 0) {  num += 1000; flag = 3;  } else if (j <= n - 4 && list[j] == target && list[j + 3] == target &&  list[j + 1] + list[j + 2] == target &&  ((j == n - 4 && list[j - 1] == 0) || (j == 0 && list[j + 4] == 0))) {  num += 100; flag = 4;  } else if (j <= n - 4 && list[j] == target && list[j + 3] == target &&  list[j + 1] + list[j + 2] == target && j <= n - 5 && j >= 1 &&  list[j - 1] + list[j + 4] == -target) {  num += 100; flag = 4;  } else if (j <= n - 4 && list[j] == target && list[j + 3] == target &&  list[j + 1] + list[j + 2] == target && j <= n - 5 && j >= 1 && list[j - 1] == 0 &&  list[j + 4] == 0) {  num += 1000; flag = 4;  } else if (j <= n - 5 && list[j] + list[j + 4] == 2 \* target &&  list[j + 1] \* list[j + 2] \* list[j + 3] == 0 &&  list[j + 1] + list[j + 2] + list[j + 3] == target) {  num += 100; flag = 4;  } else if (j < n - 2 && j > 0 && list[j] + list[j + 1] == 2 \* target && list[j - 1] == 0 &&  list[j + 2] == 0) {  num += 10; flag = 2;  } else if (j < n - 3 && j > 0 && list[j] + list[j + 2] == 2 \* target && list[j + 1] == 0 &&  list[j - 1] == 0 && list[j + 3] == 0) {  num += 10; flag = 2;  } else if (j == n - 2 && list[j] + list[j + 1] == 2 \* target && list[j - 1] == 0 && list[j - 2] == 0) {  num += 1; flag = 2;  } else if (j >= 1 && j <= n - 2 && list[j - 1] == 0 && list[j + 1] == 0 && list[j] == target) {  num += 1; flag = 1;  }  }  if(flag > 0)  {  flag--;  }  }  return num;  } |

该评估函数只适用于一个已知有效长度的一维数组，因此在源代码中，将棋盘在横竖撇捺四个方向上划分出了许多一维数组，再一一套入评估函数中，把结果全部相加即为该局面下的得分。

除此之外，其中使用到了一个重要的变量flag，我将它称为有效棋子消耗系数，该变量的作用在于防止在计算得分时某些有效棋子会被重复计分。没有该变量的话，譬如在活三计算完后必定会跟上一个眠二的分数，在某几种局势得分接近的条件下，此类情况带来的影响会很大，因此需要避免。

**3.2 Alpha Beta剪枝应用**

**3.2.1 AI主函数**

对于AI代码的主函数action\_AI，我采用的是四层搜索，即预测未来双方各走两步后的结果。在搜索深度推进的期间，只需设立四个独立二维数组存放在各个位置行棋后的局面得分即可，因为计算前后层级的数据不会相互冲突，都是及时处理及时覆盖。随后再反复调用find\_max和find\_min函数来寻找并保留或反馈该二维数组中的最高or最低得分以及其位置坐标。

除此之外，前面提到的邻居也要在此发挥作用，我们默认只有邻居格子才是有效格，即在每层计算时对于需要计算格子的最初始判断。因此，在四层的搜索基础上，有效格子二维数组也需要定义四项，每一项存放了对应该层计算时的有效格子坐标信息。

其中对于邻居数组的建立与排序，具体代码操作如下：

|  |
| --- |
| for (i = 0; i < 15; i++)  {  for (j = 0; j < 15; j++)  {  list\_use\_position[i \* 15 + j] = use\_position\_find(chessboard , i , j) \* 10000 + i \* 100 + j;  }  }  for (i = 224; i > 0; i--)  {  f\_line = 0;  for (j = i; j > 0; j--)  {  if ((list\_use\_position[j] >= 10000 && list\_use\_position[j] < list\_use\_position[j - 1]) || (list\_use\_position[j] >= 10000 && list\_use\_position[j - 1] < 10000))  {  temp = list\_use\_position[j];  list\_use\_position[j] = list\_use\_position[j - 1];  list\_use\_position[j - 1] = temp;  f\_line = 1;  }  }  if(f\_line == 0)  {  break;  }  } |

对于AI主函数中的搜索主体部分，由于源代码重复且繁琐，这里只列举了较深层的两层，部分代码操作如下：

|  |
| --- |
| for (l1 = 0; l1 < 15; l1++) {  for (k1 = 0; k1 < 15; k1++) {  f2 = 1;  if (chessboard[l1][k1] == 0 && list\_use\_position2[l1][k1] != 0) {  chessboard[l1][k1] = -1;  for (i2 = 0; i2 < 15; i2++) {  for (j2 = 0; j2 < 15; j2++) {  list\_use\_position3[i2][j2] = use\_position\_find(chessboard, i2, j2);  }  }  for (p1 = 0; p1 < 15; p1++) {  for (q1 = 0; q1 < 15; q1++) {  if (chessboard[p1][q1] == 0 && list\_use\_position3[p1][q1] != 0) {  chessboard[p1][q1] = 1;  list\_score0[p1][q1] = score\_count\_real(chessboard, -1);  chessboard[p1][q1] = 0;  } else {  list\_score0[p1][q1] = 1000000;  }  if (list\_score0[p1][q1] < max1) {  f2 = 0;  }  if (f2 == 0) break;  }  if (f2 == 0) break;  }  list\_score1[l1][k1] = abs(list\_find\_min(list\_score0)) / 10000;  if (list\_find\_min(list\_score0) < 0) {  list\_score1[l1][k1] \*= -1;  }  chessboard[l1][k1] = 0;  if (list\_score1[l1][k1] > max1) {  max1 = list\_score1[l1][k1];  }  if (list\_score1[l1][k1] > min) {  f1 = 0;  }  } else {  list\_score1[l1][k1] = -1000000;  }  if (f1 == 0) break;  }  if (f1 == 0) break;  }  list\_score2[p][q] = abs(list\_find\_max(list\_score1))/10000;  if (list\_find\_max(list\_score1) < 0) {  list\_score2[p][q] \*= -1;  }  chessboard[p][q] = 0;  if (list\_score2[p][q] < min) {  min = list\_score2[p][q];  }  if (list\_score2[p][q] < max) {  f = 0;  } |

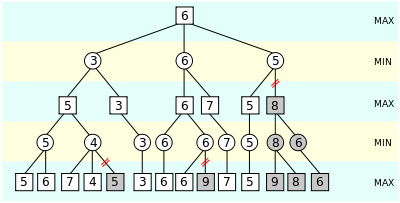
其中每次计算都源于对原棋盘的改变，计算完成后再复原棋盘，min和max1等变量则是每一层的Alpha值或Beta值，而f和f1等变量则用于每一层是否进行剪枝的判断。

**3.2.2 Alpha Beta剪枝应用**

搜索过程中，我们把每次在该max层得到的最高分更新保留，若在其后一个min层中搜索到一个比保留得分还低的得分，那么在max层中的该路径即可全部忽略。同理，把每次在min层得到的最低分更新保留，若在其后一个max层中搜索到一个比其还高的得分，那么该路径也可忽略。

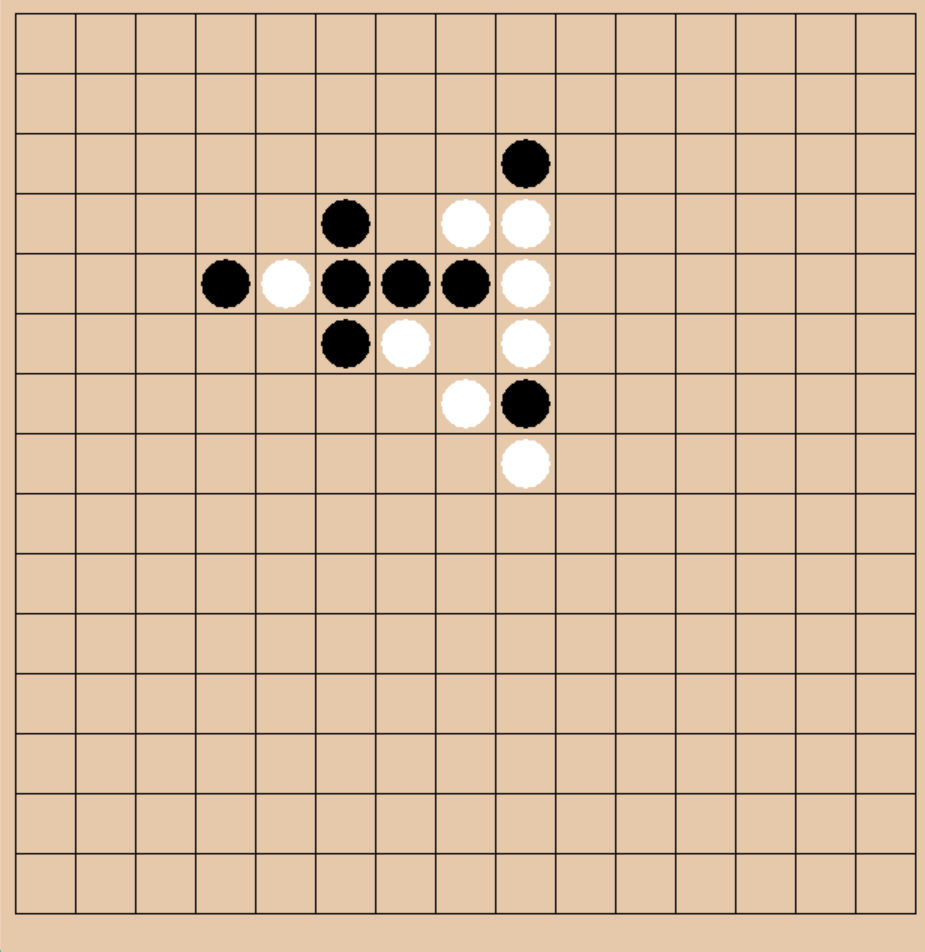
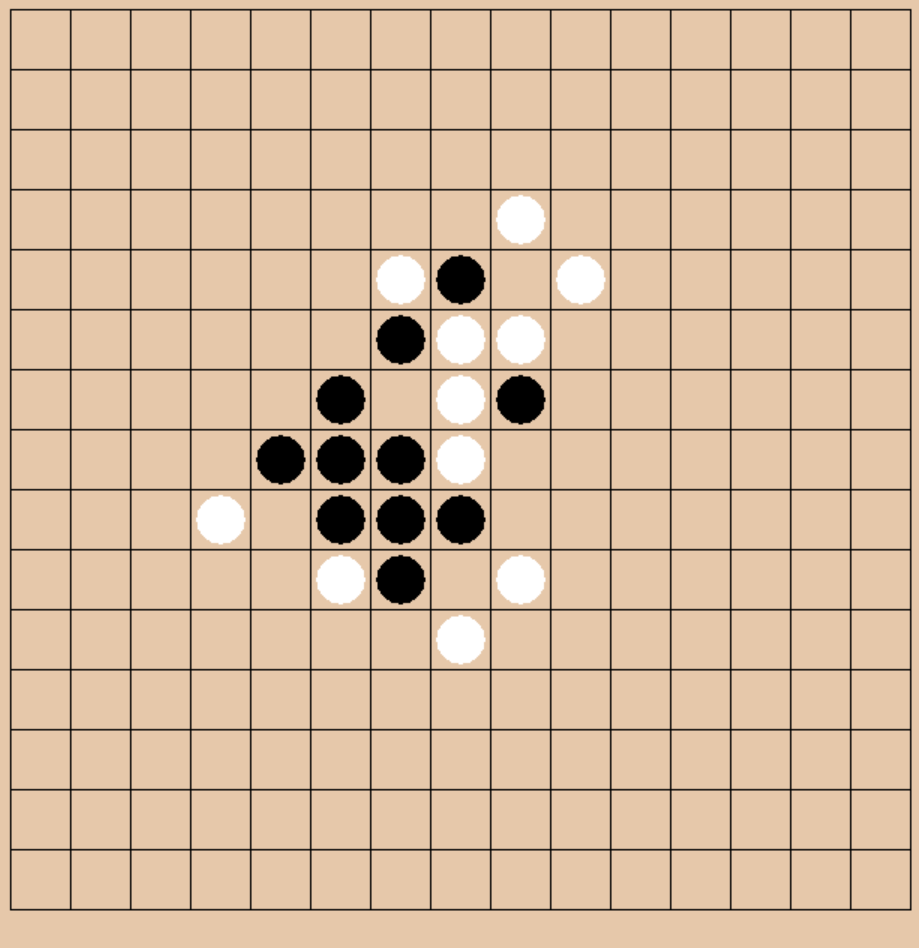
举个例子，比如有max=X，而后在min层有值Y<X，并且min层对上一层的反馈结果K必然满足K<=Y，那么同样也满足K<X，因此在max层的筛选中K必然会落选，即该路径必不可能被选择。

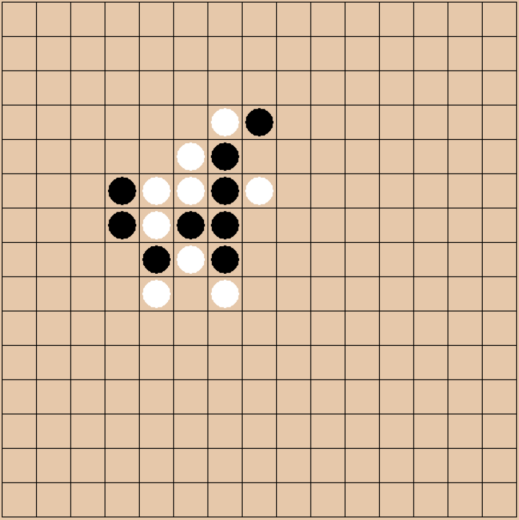
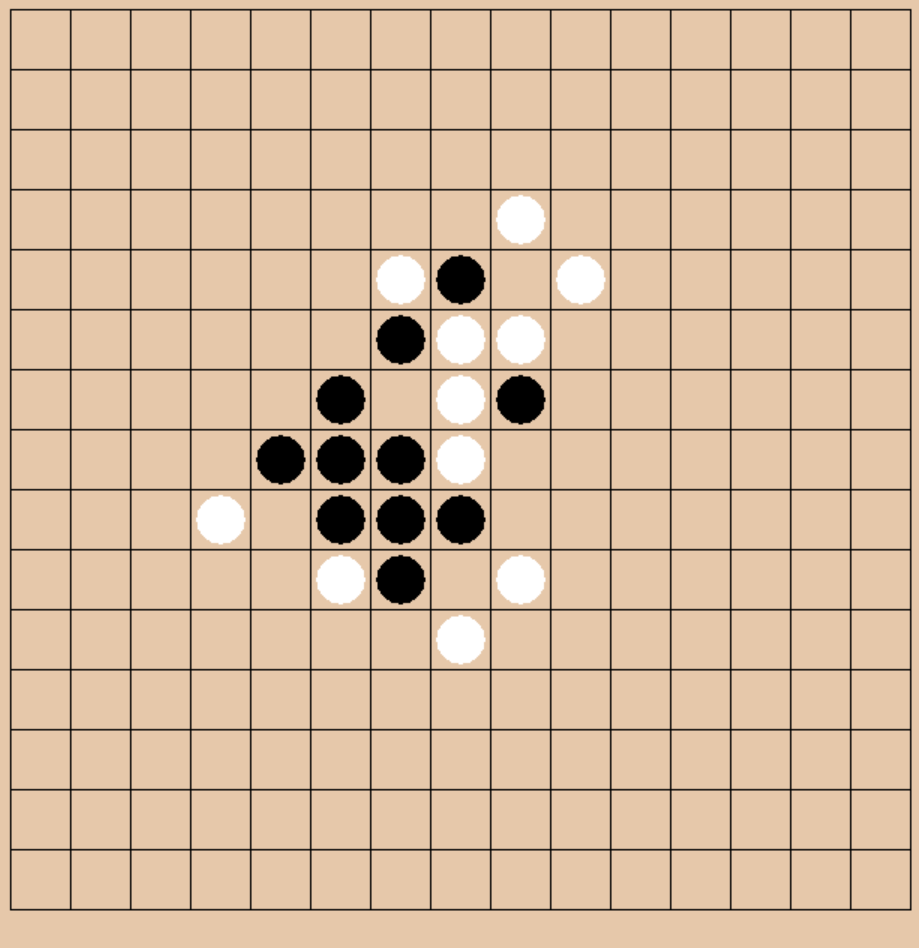
具体图示如下：



**第四章 运行结果分析**

如下二图，作为黑方的AI明显具备了最基础的进攻与防守方式，在白方未去尝试做杀，即价值最高只有眠三的情况下，AI能够判断并选择较为合适的位置去活三，或是去选择价值更高的双三与三四，率先发起进攻。

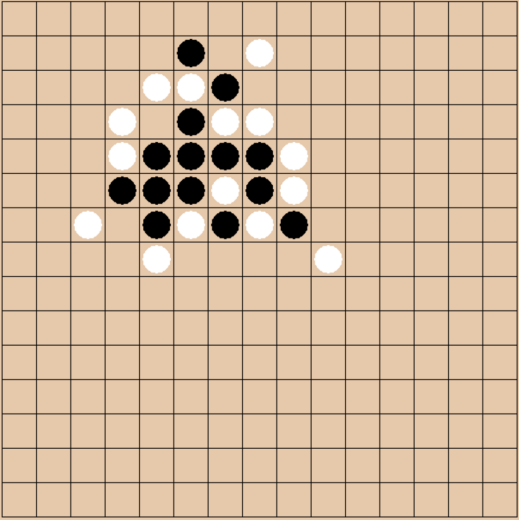
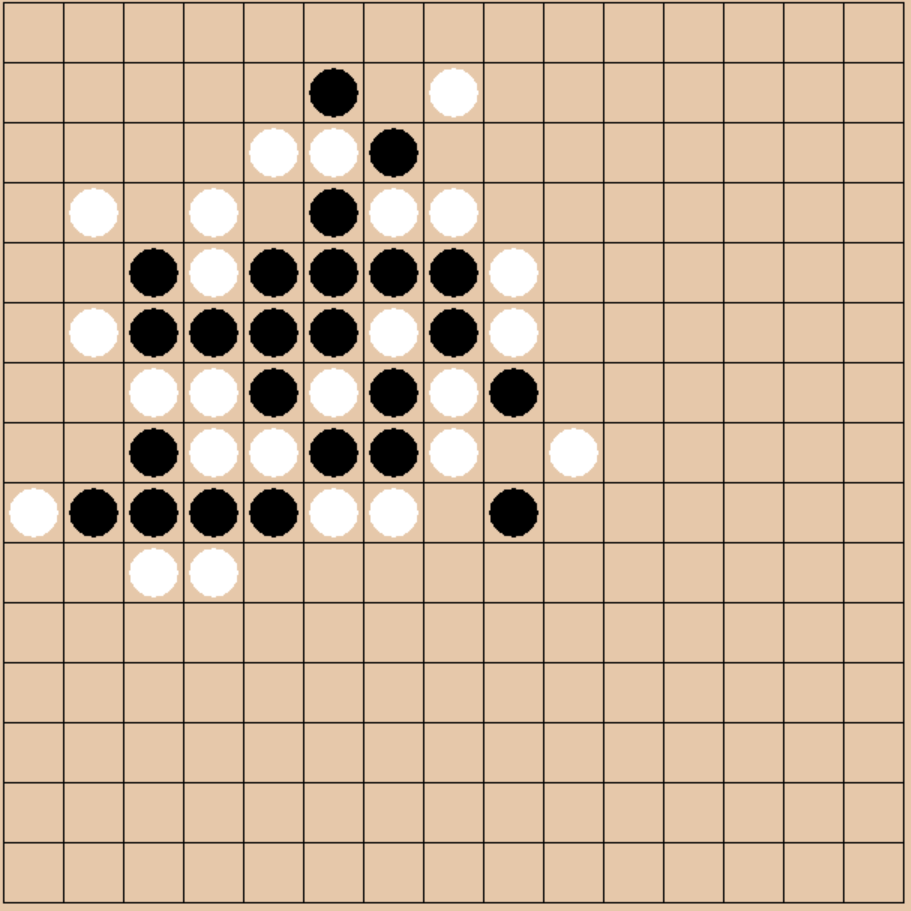


如下二图，与上同理，作为黑方的AI在白方率先进攻时，即在黑方最高只有眠三的情况下已经做出活三、冲四或价值更大的棋面，能够通过高一个等级的价值差来得出防守的结果。

以下是我尝试和AI对弈某局的中期与后期棋局局势，因为我以前自己略微系统地自学过一些五子棋，因此在本局中我放弃了连续高效进攻，以此来模拟普通玩家与AI的对弈真实情况。

而两图说明，该AI已经具备了与从未系统接触五子棋或没有经常下五子棋的玩家对弈的能力，双方都以常规的五子棋规则为基础进行攻防交换，但不会通过某些棋面的略微保留来达到出其不意的效果，这是只有玩家与玩家对弈才会出现的情况。

举个例子，在五子棋中，保留多个眠三是一个常用的手法。由于三四属于必胜局面而双三属于大优局面，若先把所有的眠三都以冲四的代价销毁，则很难保证其中能够成功做出有威胁性的杀棋。这种情况下，在保留了足够多的眠三的基础上，冲四手法就能够产生更多变数，更易做出三四必杀。但是显然本次实验编写的AI缺乏这种能力，也不会有连续进攻手段。



1. **总结**

在本次课程设计中，我们最终编写出了一个基于Alpha Beta剪枝优化的五子棋AI，它能够通过分析不同局势的走向来判断当前较优的落子点，但是其仍旧是比较简单的AI算法，还没有接触到更深层次的逻辑运算。

譬如，我们是否可以改进AI对于局势得分的计算公式，用更复杂的公式来得出更精确的得分。同时，当前它只会死板地通过比较分数来行棋，不会基于攻守转换做出分低但更合理的决策。所以在未来有必要为其增设一个算杀项目，即在安全条件范围内的连续进攻制胜，只有这样才能让AI有更大的威胁性。

本次设计虽然以有限的能力，我只接触到了Alpha Beta剪枝算法，但它们让我更加直观地去面对AI，理解AI的浅层逻辑，为以后的深入学习打下一定的基础。而且因为是自主设计的AI课程，也让曾经对于编程只有考试做题意识的我产生很大的成就感与好奇，希望未来在老师的指导下，能和同学们一同踏进AI的殿堂。