**简易五子棋AI**

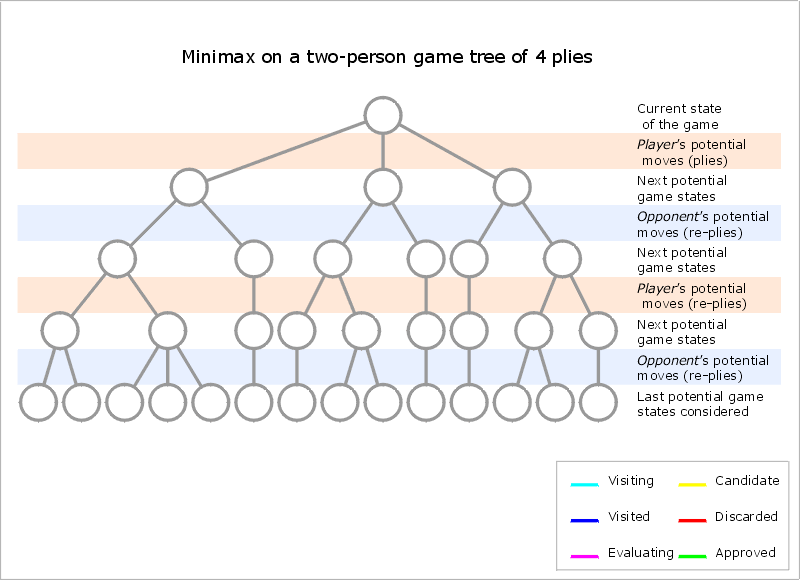
**前言**

对于五子棋AI的课程设计原本是想用蒙特卡洛树搜索算法的，但由于本人在进入大学前C语言零基础加上这大一上半学期过得有些浑浑噩噩，导致最后只能退而求其次使用极小化极大值搜索+α-β剪枝算法来完成了，我选择的算法搜索深度是6层，处于一个不高不低的位置，最后得到的结果虽不算优秀，但是与没有五子棋基础的普通人对战胜率还是比较高的。使用了EasyX库进行了简单的图形化。

**核心博弈算法1——极小化极大值搜索**

Minimax算法是一种递归算法，是一种消极的算法，他会假定对手的每一步都会走在对自己最有利的位置，也是一种深度优先的算法，他会先探索一个分支直至最底层，这也为Alpha-Beta剪枝提供了条件。极小化极大值搜索仅要求计算最底层叶节点的估值，而对于枝节点不做要求，为后面的alpha-beta剪枝提供了先决条件。我的五子棋AI中，极小化极大值搜索深度为6层，通过估值函数计算所有得到第六层情况下的返回值，方便后面进行剪枝。

极小化极大值搜索将博弈树所有层分为两类，MAX层和MIN层。对于MAX层的节点而言，它的赋值应是它所有子节点中最大的；对于MIN层的节点而言，它的赋值应应当是它所有子节点中最小的。MAX层和MIN层的决策建立在一个原则上：当下棋双方不出现失误的情况下，我方会选择对自己最有利的情况，而敌方会选择对我方最不利的情况。由于五子棋中敌我双方交替下棋，所以MAX层与MIN层也应交替出现，根节点为当前盘局，由于棋权在AI方，所以为MAX层，为考虑到敌方对AI所下棋子的对应手段，搜索底层应为MIN层，所以极小化极大值搜索的层数为偶数最佳，同时考虑到电脑CPU的计算能力，我选择将搜索深度定为6层。



维基百科上关于Minimax算法博弈树的动态图

附上Minimax函数伪代码：

function minimax( node, depth, maximizingPlayer ) is

if depth = 0 or node is a terminal node then

return the heuristic value of node

if maximizingPlayer then

value := −∞

for each child of node do

value := max( value, minimax( child, depth − 1, FALSE ) )

return value

else (\* minimizing player \*)

value := +∞

for each child of node do

value := min( value, minimax( child, depth − 1, TRUE ) )

return value

**核心博弈算法2——alpha-beta剪枝**

如果只是单纯使用Minimax算法，带来的计算量明显是个人电脑无法承担的，为了减少运算量，我们可以在Minimax算法的基础上使用alpha-beta剪枝进行优化。通俗地讲，alpha-beta剪枝建立在一个基础上：Alpha(α)表示目前所有可能解中的最大下界，Beta(β)表示目前所有可能解中的最小上界，如果出现最大下界大于最小上界的情况，说明解必定为零，则可以停止计算所有达成此条件的节点，以达到减少计算量的目的。换言之，任何被考虑在内可能为最优解的节点都必须满足一个先前条件（N为此节点的估值）：

α≤N≤β

在计算过程中α和β不断接近，最终出现α>β时，任何节点都没有可能成为最有解，则返回至仍满足α<β的枝节点并跳过其α>β的子节点及其所有叶节点。

反映至代码实现中，当我们发现α>β情况发生后，直接回溯该节点就可以解决问题：

if (tmp->Beta<=tmp->Alpha) Minimax(tmp);

alpha-beta剪枝伪代码实现：

function alphabeta(node, depth, α, β, maximizingPlayer) is

if depth = 0 or node is a terminal node then

return the heuristic value of node

if maximizingPlayer then

value := −∞

for each child of node do

value := max(value, alphabeta(child, depth − 1, α, β, FALSE))

α := max(α, value)

if value ≥ β then

break (\* β cutoff \*)

return value

else

value := +∞

for each child of node do

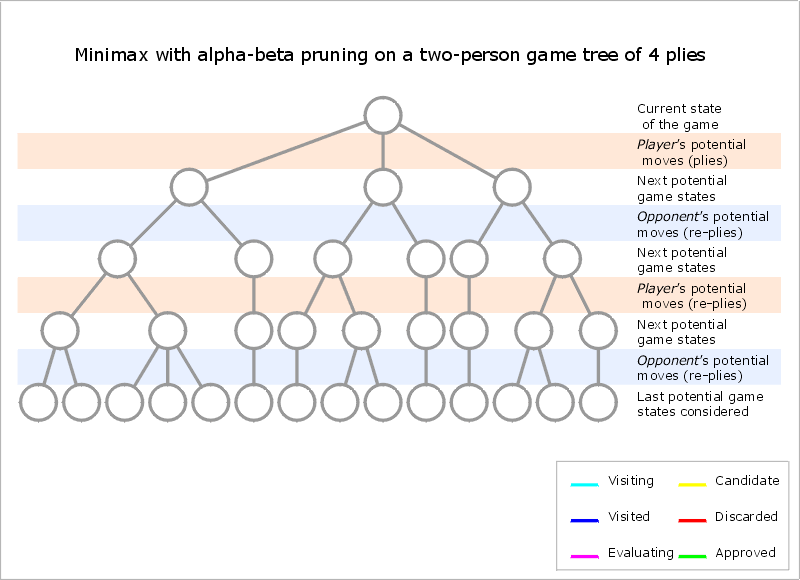
value := min(value, alphabeta(child, depth − 1, α, β, TRUE))

β := min(β, value)

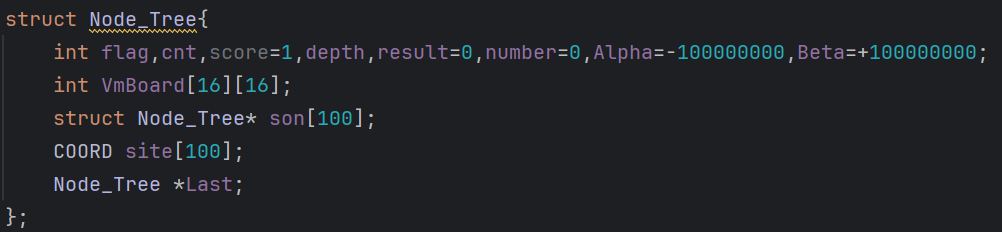
if value ≤ α then

break (\* α cutoff \*)

return value



维基百科上关于Minimax算法博弈树的动态图



AI对于剪枝节点的定义

其中flag代表AI方执子类型，cnt储存下一层节点的探索索引，depth储存深度，result储存节点棋盘是否已有胜负，number储存下一层总节点数量，Alpha和Beta储存当前节点的最大最小值用以剪枝，VmBoard[16][16]储存模拟棋盘，struct Node\_Tree\* son[216]储存下一节点的指针，COORD site[216]储存下一层节点的落子点，Node\_Tree \*Last指向上一节点。



加入alpha-beta剪枝的Minimax函数

我的代码中博弈树生成由三个函数构成：

1. createroot()基于当前局势产生的根节点的函数
2. createleaf()由某一节点向下生成子节点的函数
3. createlist()整个博弈树生成函数

**局势评估算法**

Minimax函数所有底层节点的估值是基于局势评估函数实现的。我对于评估函数的算法进行了一次迭代，在V1版本中，我的AI中仅有全局局势评估函数，且其对于跳棋这类走法可谓是一窍不通。V1版本中评估函数由横向函数、纵向函数、左斜向函数和右斜向函数共同构成，直接基于对棋盘状态数组的分析，通过分析任一我方连续棋子的第一粒的棋子状态，通过对其前一以及向后延申的终止进行判断。这也带来一个缺陷，即其对部分冲棋的判断不准确。下面举一个简单的例子：

棋面情况：空 黑 黑 空 黑 空

该情况下黑棋实际上构成了一个“冲三”，而对于V1版本的评估函数而言，它实际上由活二+活一构成，而在一般情况下，此二者的价值和远低于单个“冲三”的价值，使得AI对于棋局实际情况产生误判，因此发展了V2版本。

V2版本的评估函数在改进了全局评估函数的同时加入了启发式评估函数。在全局评估函数上发生的变化主要体现从直接对盘面进行分析转向使用字符数组复制某行或列或斜的棋面情况，并使用find函数将其与提前规定的不同棋面情况下的估值进行匹配，最后求和以获得相应评估值。由于与程序文件中的盘面情况直接对应，该方法可以更加准确的进行分析，下面是不同情况下的盘面打分原则（2为我方，1为空位，0为敌方）：

string five[10] = {"22222", "1222211", "1122221"};//活五活四

string four[10] = {"022221", "122220", "22212", "22122", "21222"};//冲四

string three[10] = {"122211", "112221", "121221", "122121"};//活三

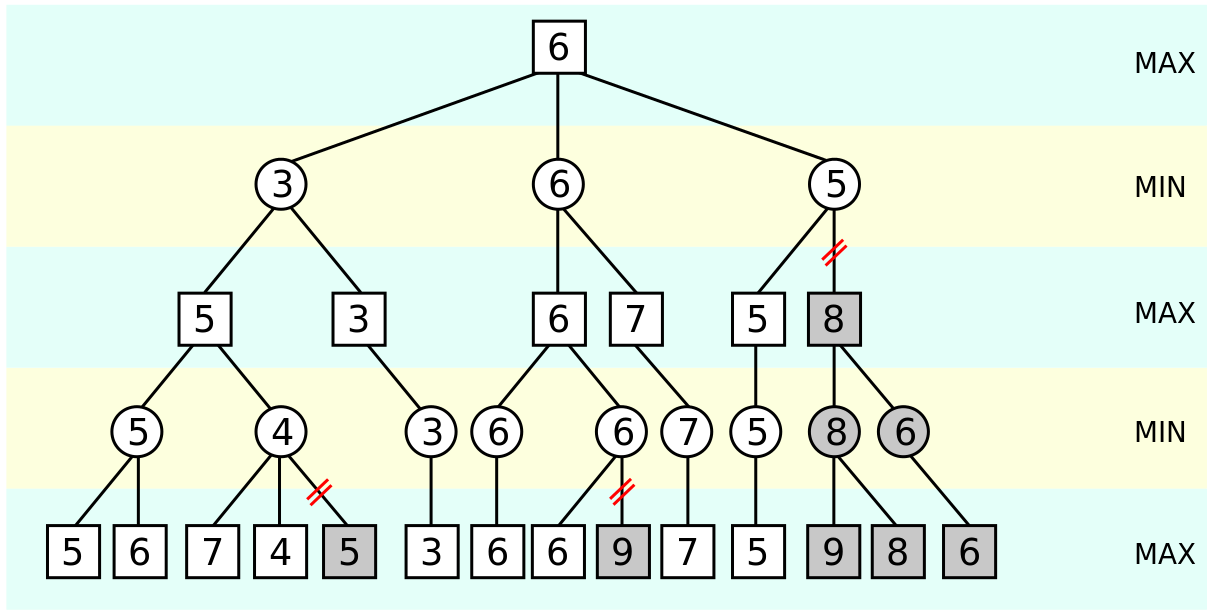
string LowRank[10] = {"022211", "112220", "022121", "121220", "021221", "122120", "1211221", "1221121", "1212121",

"0122210"};//眠三

string LiveTwo[10] = {"112211", "12121", "121121"};//活二

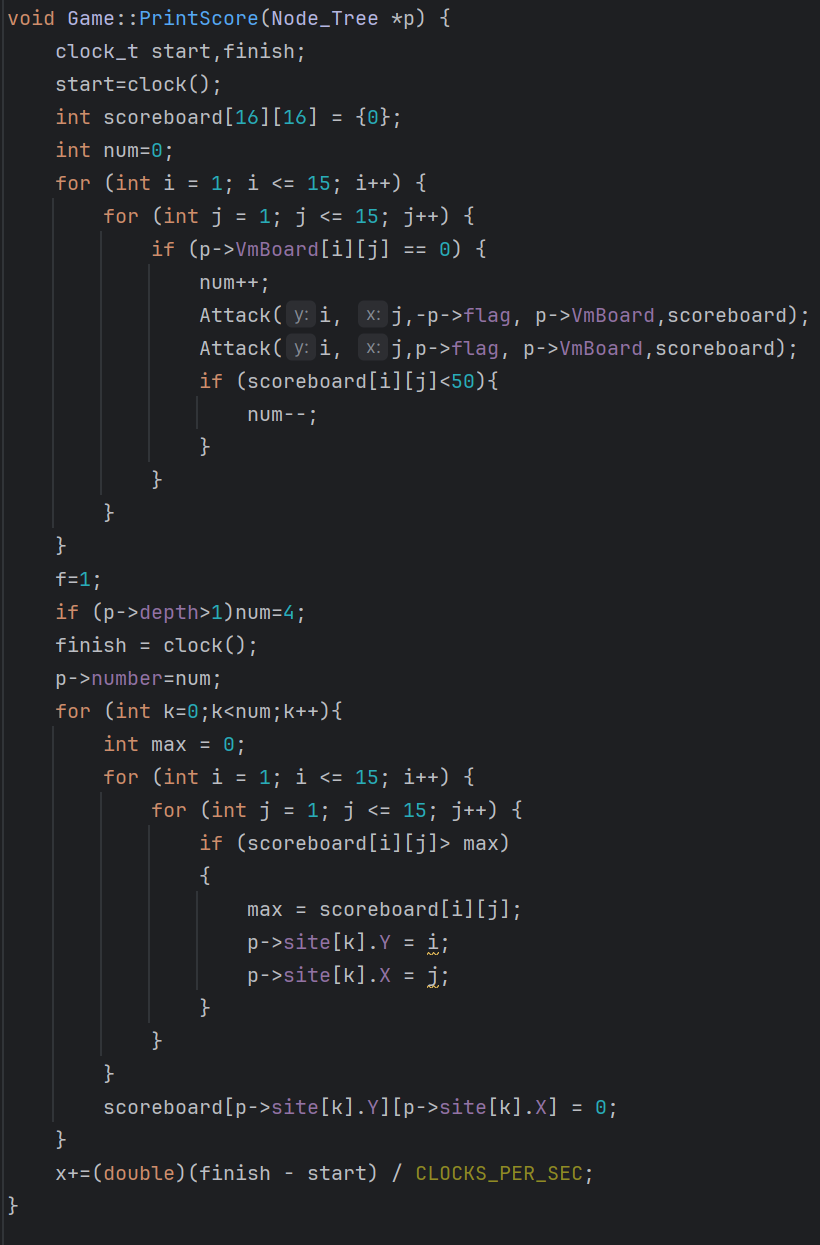
string SleepTwo[10] = {"022111", "111220", "021211", "112120", "021121", "121120", "21112"};//眠二

使用全局估值函数分别对我方和敌方进行评判，并使用我方估值减去敌方估值得到最终节点的估值。

启发式估值函数针对的是当前盘面上所有空位的估值函数，其存在价值是提高alpha-beta剪枝的效率。alpha-beta剪枝的效率很大程度上取决于子节点的排序，以下面的图像为例：

上面可以看到在第二层中，第一个节点的值是3，第二个是6。因为3比较小，而这一层的最大值会被选中，所以第二个节点也需要完整计算所有子节点。如果3和6调换一下顺序，6在前，3在后。那么当第二个节点计算出第一个子节点5的时候就没有必要计算之后的子节点了。也就是，Alpha-Beta 剪枝的效率和节点排序有很大关系，如果最优的节点能排在前面，则能大幅提升剪枝效率。对于Max层节点同样适用。

所以设置一个启发式评估函数PrintScore()对所有待搜索的位置进行打分，并根据分数由高到低进行排序，与进行全局估值的Allscore()函数原理相似，不过根据这个位置是否能成五，活四，活三等来进行打分。首个方案是取空位周围米字形的8个方向从中间延申取4个点位，最后对得到的4个数组进行与Allscore()函数类似的匹配打分机制，得到每个节点的分数。但是由于其耗时过多，所以使用优化改进方案，通过确定几个参数， empty, count, block, able。其中empty对应相同棋子间的空位，count对应相同棋子的数量，block对应棋子两端与棋局边缘或不同颜色棋子相连的数量，able对应四周可落子的数量。仅当满足able + count + empty >=5这个条件才能使连五成为可能。对于不同情况的分析则在Attack()函数和ScoreGet()函数中。



PrintScore()函数，其中Attack()是对某一点进行米字型的分析函数

**总结与展望**

最后将以上所有函数全部整合到Robot()函数中就结束了，我已将一些核心重要函数的内容放在报告中，对于其他重要性不高或是过于冗长的函数的具体实现可以查看项目文件中”Game.cpp”。

在已有的五子棋AI程序基础上，有其他同学向我建议可以使用蒙特卡洛策略梯度再次优化alpha-beta剪枝过程。具体实现过程是在当前PrintScore()函数的基础上再次增加一个权重值，通过对已有的Allscore()得到的评分相同或接近的盘面进行进一步的胜率分析，对得到的盘面进行随机落子直至出现胜负，规定一定深度后进行胜率计算，得到更优解。