基于极小化极大值搜索的五子棋 AI 算法设计

姓名：刘浩楠

学号：2023080906008

摘 要

随着人工智能技术的不断进步，棋类人工智能的智能化程度将越来越高。五子棋 AI 算法属于人工智能领域的研究范畴。通过对五子棋 AI 算法的研究，可以推动人工智能相关技术的发展，如强化学习、搜索算法、神经网络等。本模型基于传统搜索算法，以极小化极大值搜索为核心，构建博弈树来模拟对弈过程，优先进行深度搜索，利用评估函数为棋局赋分，并通过 Alpha-Beta 剪枝，启发式搜索函数等技巧简化算法，提高运算速度，加深搜索层数，以获得最佳落子位置。

关键词：五子棋；极小化极大值搜索；博弈树；Alpha-Beta 剪枝；启发式搜索函数

目　录

第 1 章 引言

1.1 背景介绍

1997 年，美国 IBM 公司研发的超级国际象棋 AI “深蓝”战胜了当时的国际象棋之王卡斯帕罗夫。这是电脑第一次在棋类中战胜人类顶尖选手，让世人见证了 AI 的威力。棋类游戏 AI 自此迅速发展，其博弈算法不断得到改进，形成了较为成熟的高级算法体系。2016 年，AlphaGo 利用深度学习，击败了人类围棋的巅峰选手李世石。不久后，AlphaGo Zero 通过自我对弈和学习人类棋谱，大幅提升了棋力，击败了少年围棋天才柯洁。

从初期的简单算法到现在的深度学习技术，棋类人工智能的发展历程是一个不断探索和创新的过程，人们不断挑战着计算机在棋类游戏方面的能力极限。通过不断挑战和解决棋类问题，可以推动人工智能技术的发展和创新，为其他领域的应用提供借鉴和启示。总之，随着技术的不断进步和应用场景的不断拓展，棋类人工智能的未来发展将充满无限可能性和机遇。我们期待着未来更多的突破和创新。

1.2 设计目的

五子棋（又称为连珠或五目棋）是一种两人对弈的策略棋类游戏。五子棋的博弈算法具有以下特点：

1.搜索空间大：五子棋的搜索空间相对较大，因为棋盘大小和可能的棋步组合都很多。因此，需要高效的搜索算法来处理这种庞大的搜索空间。

2.零和性：五子棋的博弈具有零和性质，即一方的胜利意味着另一方的失败。这要求算法能够快速评估局面并做出最佳选择。

3.信息完全：五子棋是信息完全的博弈，即双方拥有相同的棋盘信息和对手的行动信息。这使得博弈算法可以通过对当前局面的分析来做出决策。

4.连续性：五子棋的博弈是连续的，每一步都可能影响后续的行动。因此，博弈算法需要考虑每一步的长期影响，并做出最优的选择。

5.对抗性：五子棋的博弈具有很强的对抗性，因为双方都在寻找对方的弱点并试图获得优势。因此，博弈算法需要能够快速应对对手的行动，并制定出有效的反击策略。

6.策略多样性：五子棋的策略非常多样，不同的策略可能导致不同的结果。因此，博弈算法需要能够根据不同的局面制定和调整不同的策略。

五子棋 AI 算法属于人工智能领域的研究范畴。通过对五子棋 AI 算法的研究，可以推动人工智能相关技术的发展，如强化学习、搜索算法、神经网络等。五子棋 AI 算法的研究不仅限于五子棋本身，还可以扩展到其他类似的博弈和决策问题中。例如，在金融、经济、军事等领域中，一些决策问题也可以通过类似的算法来解决。研究五子棋 AI 算法不仅有助于提高五子棋的竞技水平，还可以促进人工智能领域的发展和推广，为其他领域提供新的解决方案，并提升大众对人工智能的认识和理解。

第 2 章相关工作

2.1 理论介绍

2.1.1 极小化极大值搜索的原理

极小化极大值搜索是一种决策策略，它的原理是在有限的深度范围内，使用深度优先搜索（DFS）算法，利用递归回溯从可能的走法中选择对自己最有利的走法，即让自己的收益最大、对手的收益最小。

具体来说，极小化极大值搜索会首先构建决策树，并自底向上计算每个节点的 minimax 值。在计算过程中，该策略会遍历决策树的所有节点，求取每个 minimax 值。在决策树的构建与搜索过程中，该策略会避免展开不必要搜索的节点，以节省搜索时间。

当一个零和博弈双方每一步可选动作数量较多时，决策树会变得非常庞大，因此构造决策树，并对其进行遍历，求取每个 minimax 值将会非常耗时。而极小化极大值搜索则通过在有限的深度范围内进行搜索，避免了这个问题。

在选择行动策略时，极小化极大值搜索从根结点选择 minimax 值最大的分支。这样做的目的是为了最大化自己的收益，同时最小化对手的收益。

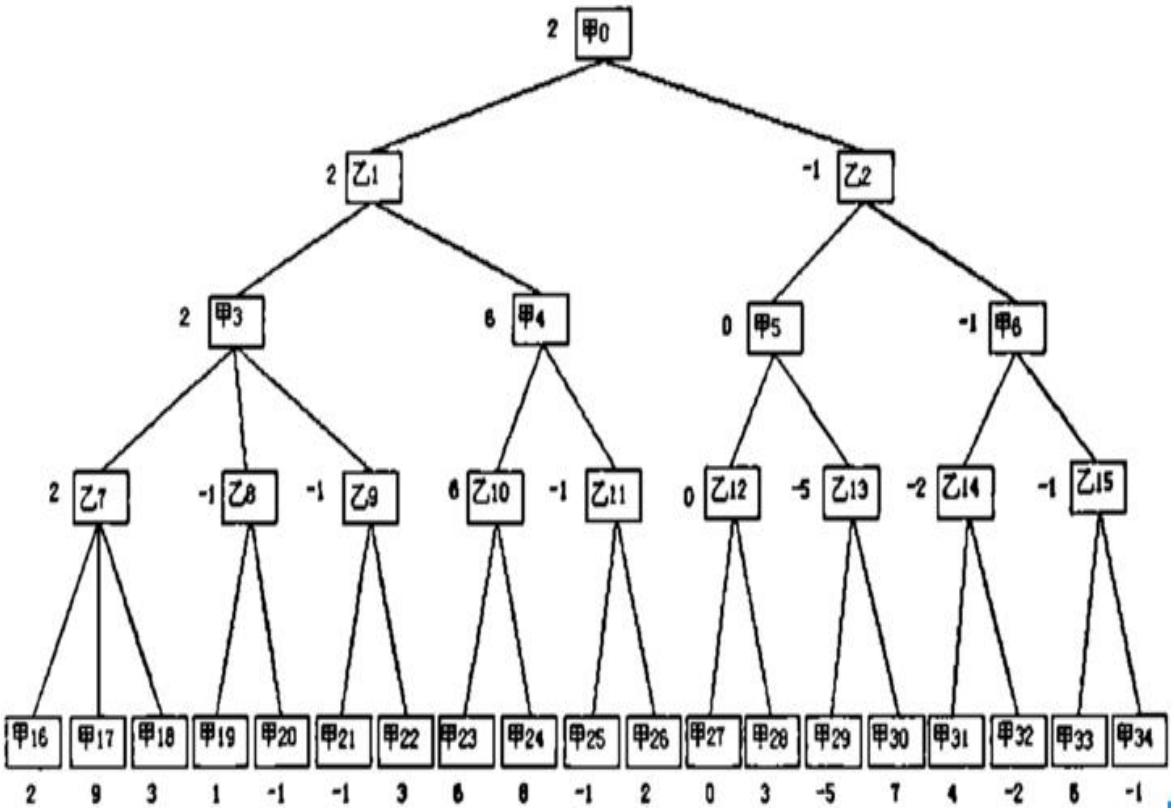
2.1.2 博弈树的逻辑

五子棋看起来有各种各样的走法，而实际上把每一步的走法展开，就是一颗巨大的博弈树。在这个树中，从根节点为 0 开始，奇数层表示电脑可能的走法，偶数层表示玩家可能的走法。假设电脑先手，那么第一层就是电脑的所有可能的走法，第二层就是玩家的所有可能走法，以此类推。

电脑走棋的层我们称为 MAX 层，这一层电脑要保证自己利益最大化，那么就需要选分最高的节点。

玩家走棋的层我们称为 MIN 层，这一层玩家要保证自己的利益最大化，那么就会选分最低的节点。

这也就是极大极小值搜索算法的名称由来。



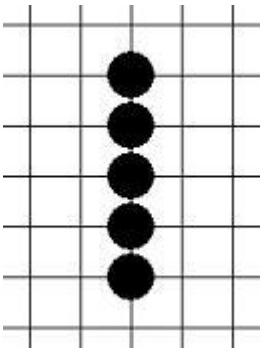
此图中甲是电脑，乙是玩家，那么在甲层的时候，总是选其中值最大的节点，乙层的时候，总是选其中最小的节点。而每一个节点的分数，都是由子节点决定的，因此我们对博弈树只能进行深度优先搜索而无法进行广度优先搜索。深度优先搜索用递归非常容易实现，然后主要工作其实是完成一个评估函数，这个函数需要对当前局势给出一个比较准确的评分。

2.1.3 评估函数

有了搜索策略，我们还需要进行局势的评估。我们简单的用一个整数表示当前局势，分数越大，则自己优势越大，分数越小，则对方优势越大，分数为 0 是表示双方局势相当。

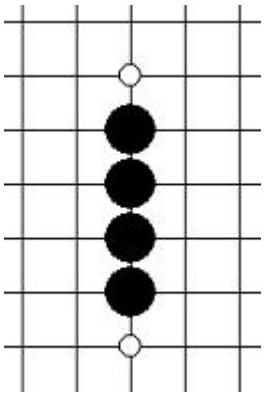
我们对五子棋的评分是简单的把棋盘上的各种连子的分值加起来得到的，最常见的基本棋型大体有以下几种：**连五，活四，冲四，活三，眠三，活二，眠二**。

①连五：顾名思义，五颗同色棋子连在一起，不需要多讲。

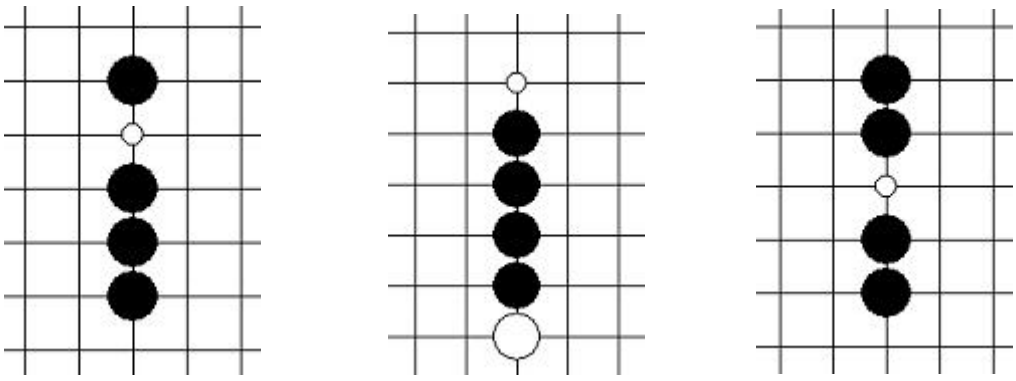


②活四：有两个连五点（即有两个点可以形成五），图中白点即为连五点。稍微思考一下就能发现活四出现的时候，如果对

方单纯过来防守的话，是已经无法阻止自己连五了。

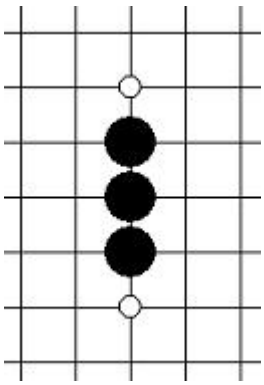
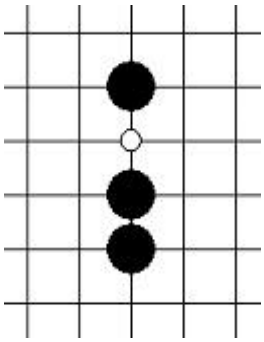


③冲四：有一个连五点，如下面三图，均为冲四棋型。图中白点为连五点。相对比活四来说，冲四的威胁性就小了很多，因为这个时候，对方只要跟着防守在那个唯一的连五点上，冲四就没法形成连五。

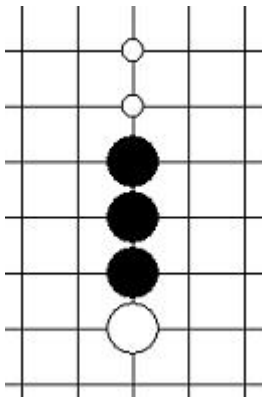
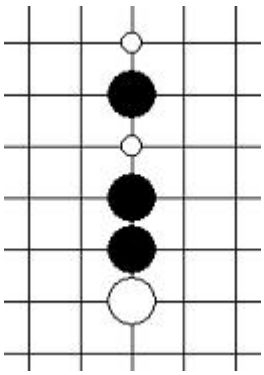
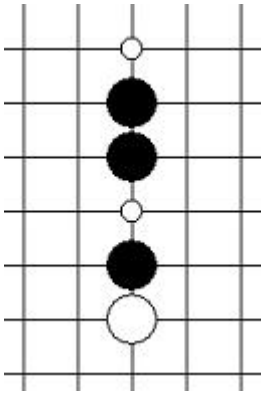
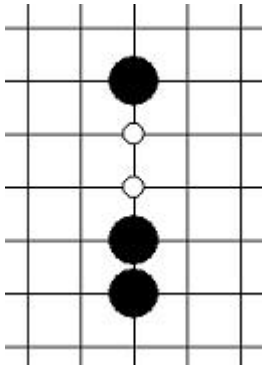
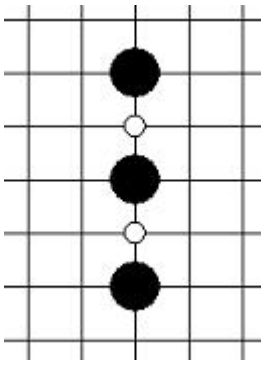
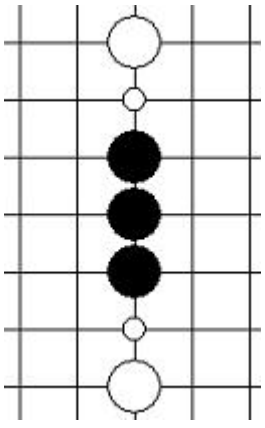


④活三：可以形成活四的三，如下图，代表两种最基本的活三棋型。图中白点为活四点。

活三棋型是我们进攻中最常见的一种，因为活三之后，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。所以，当我们面对活三的时候，需要非常谨慎对待。在自己没有更好的进攻手段的情况下，需要对其进行防守，以防止其形成可怕的活四棋型。

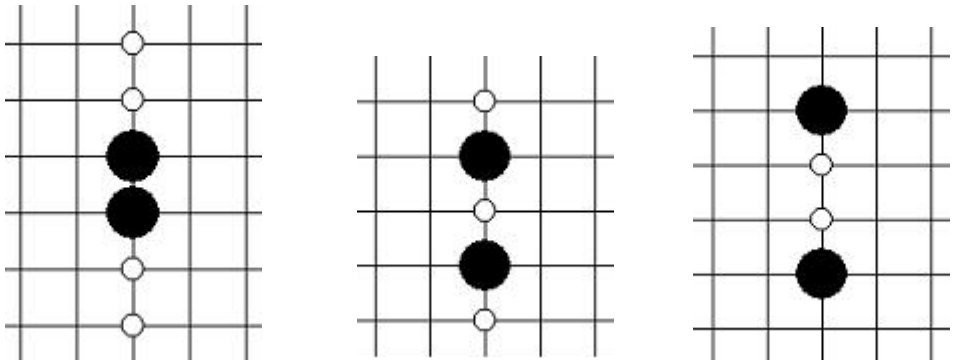
⑤眠三：只能够形成冲四的三，如下各图，分别代表最基础的六种眠三形状。图中白点代表冲四点。眠三的棋型与活三的棋型相比，危险系数下降不少，因为眠三棋型即使不去防守，下一手它也只能形成冲四，而对于单纯的冲四棋型，我们知道，是可以防守住的。

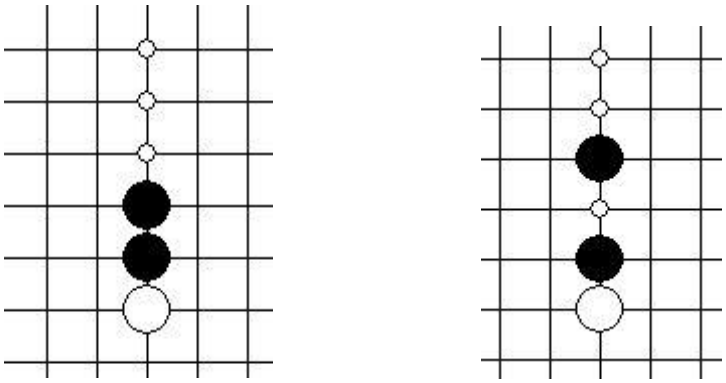
如上所示，眠三的形状是很丰富的。对于初学者，在下棋过程中，很容易忽略不常见的眠三形状。

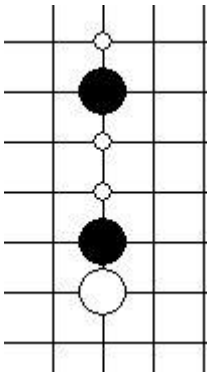
⑥活二：能够形成活三的二，如下图，是三种基本的活二棋型。图中白点为活三点。

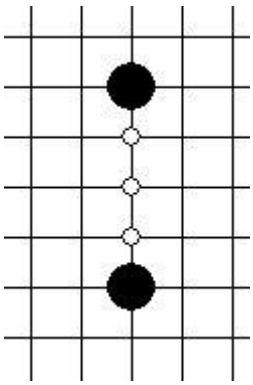
活二棋型看起来似乎很无害，因为他下一手棋才能形成活三，等形成活三，我们再防守也不迟。但其实活二棋型是非常重要的，尤其是在开局阶段，我们形成较多的活二棋型的话，当我们将活二变成活三时，才能够令自己的活三绵绵不绝微风里，让对手防不胜防。



⑦眠二：能够形成眠三的二。图中四个为最基本的眠二棋型，图中白点为眠三点。







可见，五子棋的棋形复杂多变。因此，评估函数应尽量全面而精准。一个好的评估函数能较好反映出局势优劣，从而为 AI 落子提供良好思路，显著提升 AI 棋力。

2.2 Raylib 绘图库

Raylib 是一个功能强大、易于使用、跨平台的图形库。利用其丰富的绘图资源，我们可以制作出一个精简的五子棋棋盘 UI，便于呈现对局过程。同时，Raylib 提供了简便的工具链，通过其内置的各种参数功能，可以将棋局信息实时传递到 c 源文件中，同时能反馈玩家的落子情况，实现了前后端的的链接。

第 3 章相关工作

3.1 基本框架

3.1.1 UI 棋盘界面

我们首先在 Cmakelist 中引入 Raylib 绘图库,并将其作为头文件引用。

```
include_directories(gobang)

find_package(raylib)

target_link_libraries(chessboard raylib)
```

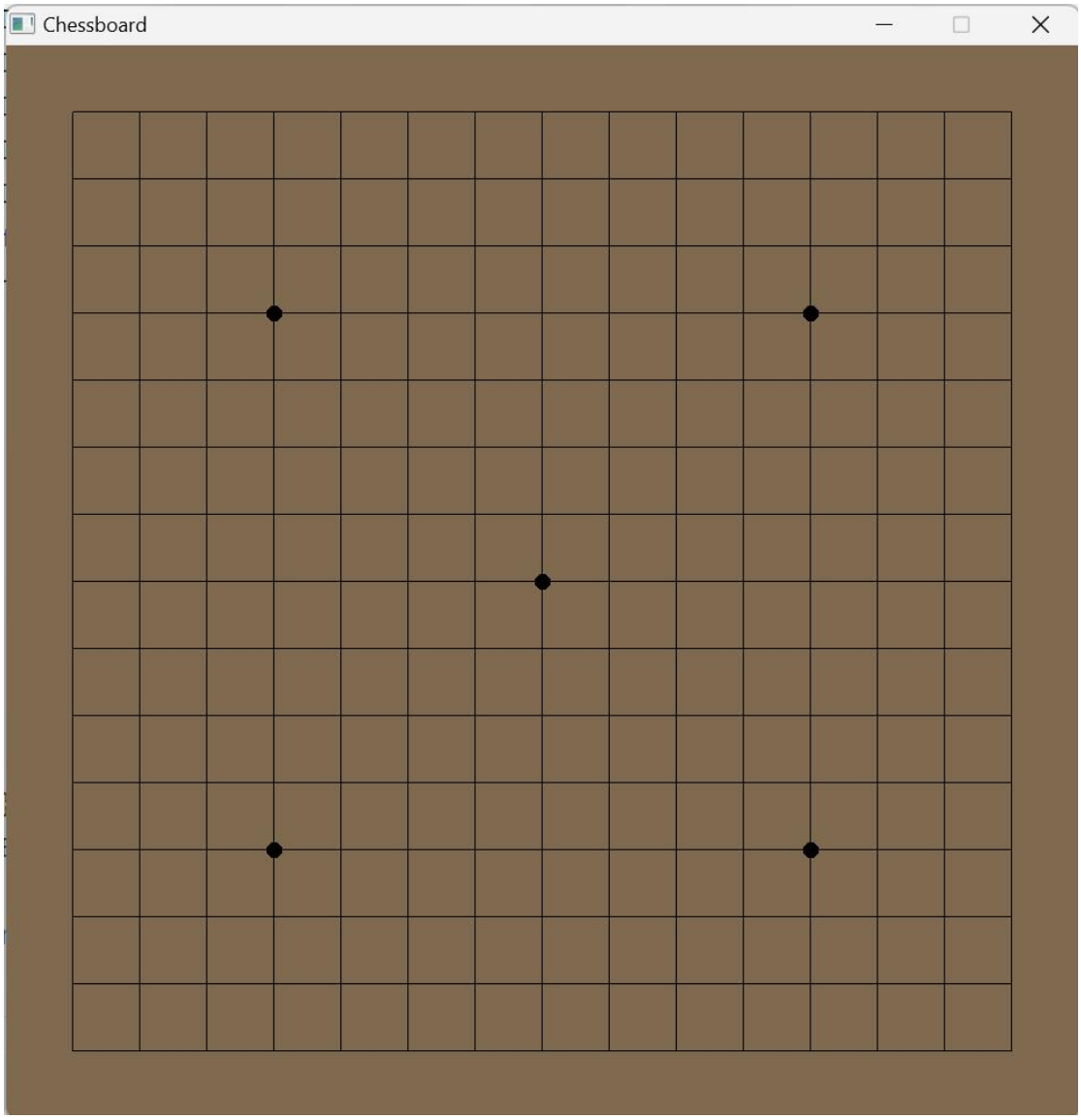
下面是创建 UI 棋盘界面的代码呈现：

```
//初始化棋盘
int map[15][15]={};
InitWindow(960,960,"Chessboard");
//创建UI 棋盘界面
while (!WindowShouldClose())
{
    //检查玩家落子情况
    if (IsMouseButtonPressed(MOUSE_LEFT_BUTTON))
```

```
{
    int m = (int) ((GetMouseX() - 30) / 60);
    int n = (int) ((GetMouseY() - 30) / 60);
    if (m >= 0 && m <= 14 && n >= 0 && n <= 14 && map[m][n]!=-1)
    {
        map[m][n] = 1;
    }
    //玩家落子后，AI 进行思考并落子
    Root* pt = Create_Root(map);
    int score=pt->score;
    int x= pt->x;
    int y= pt->y;
    map[x][y]=-1;
    printf("%d x=%d y=%d\n",score,x,y);
}
//反复渲染棋盘
BeginDrawing();
ClearBackground(BROWN);
for (int i = 60; i <= 900; i += 60)
{
    DrawLine(i, 60, i, 900, BLACK);
}
for (int i = 60; i <= 900; i += 60)
```

```
{
    DrawLine(60, i, 900, i, BLACK);
}
DrawCircle(240,240,7,BLACK);
DrawCircle(240,720,7,BLACK);
DrawCircle(720,240,7,BLACK);
DrawCircle(720,720,7,BLACK);
DrawCircle(480,480,7,BLACK);
for (int i = 0; i <= 14; i++)
{
    for (int j = 0; j <= 14; j++)
    {
        if (map[i][j] == 1)
        {
            DrawCircle(60 * (i + 1), 60 * (j + 1), 25, BLACK);
        }
        if (map[i][j] == -1)
        {
            DrawCircle(60 * (i + 1), 60 * (j + 1), 25, WHITE);
        }
    }
}
EndDrawing();
```

创建好的 UI 棋盘界面如下：



3.1.2 评估函数的设计

对于棋盘上的排成直线的一串棋子，有横、竖、左上到右下，左下到右上四种方向，对于某一个方向上的排成一条直线的棋子，我称其为**棋链**。评估函数的算法概括起来讲，是找到棋盘上的点位，然后计算这个位置四个方向上的**连五，活四，冲四，活三，眠三，活二，眠二**的个数，并根据这些棋型由高到低进行赋分。当为所有点位赋完分后，将所有分值全部相加，得到一个当前局面的总分数值。

下面对算法的细节进行更为详细的介绍：

在获取点位四个方向的棋子时不但要注意获得的棋链长度，还要注意该点位在棋链中的位置，拿一个长度为 5 的连 5 棋链来说，该点位可能位于第 1 或 2 或…第 5 个位置，也就是说，判断一个点位在四个方向、五个位置是否为连 5，要取得 4X5 共 20 个棋链，，而一条棋链最长长度为 7，也就是说,获取一个点位的棋型数，最坏一共要取得 4X7=28 条棋链。

下面我通过代码形象地展示全过程。

```
//创建向量以储存棋链
int *vector = malloc(5*sizeof(int));
int count=0;
//依次改变空位在棋链中的位置
for(int k=0;k<5;k++)
{
    //边界检查
    if(k-i<=0 && k-i+10>=0)
    {
        //将棋链存入向量
        for(int t=0;t<5;t++)
        {
            vector[t]=map[i+t-k][j];
        }
        //与特殊棋形对照
        if(vector[0]==1 && vector[1]==1 && vector[2]==1 &&
vector[3]==1 && vector[4]==1)
        {
            count--; //若黑棋成特殊棋形，分数减少
        }
        else if(vector[0]==-1 && vector[1]==-1 && vector[2]==-1 &&
vector[3]==-1 && vector[4]==-1)
        {
            count++; //若白棋成特殊棋形，分数增加
        }
    }
}
```

上图以**连五**的评估为例，展示了**横**方向上棋链的识别，判断和计数。值得一提的是，评估函数是以 AI 为主体设计的，换言之，局面越有利于 AI，分值越高；局面越有利于玩家，分值越低。而 AI 默认为后手执白棋，因此若白棋形成特殊棋形，则增加计数；若黑棋形成特殊棋形，则减少计数。这种机制较好的反映了局面的优劣，体现出五子棋“零和博弈”的特点。

接着，我们需逐个统计某一点特殊棋形的个数，并根据这些棋型由高到低进行赋分。如下所示：

```
//给某一点赋值
int GiveValue(int i,int j,int map[][15])
{
    int value;
    value=10000 * Count_linkfive(i,j,map)
           +1000 * Count_wakefour(i,j,map)
            +100 * Count_rushfour(i,j,map)
             +100 * Count_wakethree(i,j,map)
              +10 * Count_sleepthree(i,j,map)
               +5 * Count_waketwo(i,j,map)
                +1 * Count_sleeptwo(i,j,map);
    return value;
}
```

最后，我们将所有点位的分值相加，得到局面总分值：

```
//评估当前局面总分数值
int Evaluate(int map[][15])
{
    int score=0;
    for(int i=2;i<13;i++)
    {
        for(int j=2;j<13;j++)
        {
            if(Is_GiveValue(i,j,map))
            {
                score+=GiveValue(i,j,map);
            }
        }
    }
    return score;
}
```

3.1.3 博弈树的构建

有了对博弈树的基本认识，我们就可以用递归来遍历这一棵树。首先定义根节点和子节点的结构：

```
typedef struct Root
{
    bool IsMax;//判断是否为Max 节点
    int depth;//节点深度
    int pre;//前驱alpha 或beta 值，便于后续剪枝
    int score;//节点最终得分
    int x;
    int y;
}Root;
typedef struct Node
{
    bool IsMax;
    int depth;
    int alpha;
    int beta;
    int pre;
    int score;
}Node;
```

然后创建根节点：

```
//创建根节点
Root* Create_Root(int map[][15])
{
    //开辟根节点空间
    Root* root = (Root*) malloc(sizeof(Root));
    //设置根节点深度和Max 类别
    root->depth=1;
    root->IsMax=1;
    root->score=-100000;
    root->pre=-100000;
    //调用启发式函数生成搜索顺序表
    pos* pt = Search_list_white(map);
    for(int k=0;pt[k].x!=0 || pt[k].y!=0;k++)
    {
        int i = pt[k].x;
        int j = pt[k].y;
        map[i][j]=-1;
        //若有连五，直接返回
        if(Is_over(i,j,map))
        {
            root->score=999999;
            root->x=i;
            root->y=j;
            return root;
        }
        int feedback =
Create_children(root->IsMax,root->depth,root->pre,map);
        if(feedback>=root->score)
        {
            root->score=feedback;
            root->pre=feedback;
            root->x=i;
            root->y=j;
        }
        map[i][j]=0;
    }
    return root;
}
```

其中，Create_children 是一个生成子节点的函数。它继承了父节点的信息，并递归调用自身，实现深度搜索，直到达到最大搜索深度为止。如下所示：

```
int Create_children(bool Is_Father_Max,int
Father_depth,int pre,int map[][15])
{
    //开辟节点空间
    Node *node = (Node *) malloc(sizeof(Node));
    //判断是否为Max 节点
    if (Is_Father_Max)
    {
        node->IsMax = 0;
    }
    else
    {
        node->IsMax = 1;
    }
//初始化α和β的值
node->alpha = -100000;
node->beta = 100000;
```

```
//若为Max 节点
if (node->IsMax)
{
    //调用启发式函数生成搜索顺序表
    pos *pt = Search_list_white(map);
    //判断是否为叶节点
    node->depth = Father_depth + 1;
    //若为叶节点
    if (node->depth == 4)
    {
        node->score = -100000;
        for (int k = 0; pt[k].x != 0 || pt[k].y != 0; k++)
        {
            int i = pt[k].x;
            int j = pt[k].y;
            map[i][j] = -1;
            //若有连五，直接返回
            if(Is_over(i,j,map))
            {
                int score = 999999;
                free(node);
                map[i][j] = 0;
                return score;
            }
            int feedback = Evaluate(map);
            map[i][j] = 0;
            //剪枝
            if (feedback > pre)
            {
                int score = 100000;
                free(node);
                return score;
            }
            node->score = (feedback >= node->score ? feedback :
node->score);
        }
    }
```

```
    //若不是叶节点
    else
    {
        node->pre = -100000;
        for (int k = 0; pt[k].x != 0 || pt[k].y != 0; k++)
        {
            int i = pt[k].x;
            int j = pt[k].y;
            map[i][j] = -1;
            //若有连五，直接返回
            if(Is_over(i,j,map))
            {
                int score = 999999;
                free(node);
                map[i][j] = 0;
                return score;
            }
            int feedback = Create_children(node->IsMax, node->depth,
node->pre, map);
            map[i][j] = 0;
            node->pre = (feedback > node->pre ? feedback : node->pre);
            //剪枝
            if (feedback > pre)
            {
                int score = 100000;
                free(node);
                return score;
            }
            node->alpha = (feedback >= node->alpha ? feedback :
node->alpha);
        }
        node->score = node->alpha;
    }
}
```

```
//若为Min 节点
else
{
    //调用启发式函数生成搜索顺序表
    pos *pt = Search_list_black(map);
    //判断是否为叶节点
    node->depth = Father_depth + 1;
    //若为叶节点
    if (node->depth == 4)
    {
        node->score = 100000;
        for (int k = 0; pt[k].x != 0 || pt[k].y != 0; k++)
        {
            int i = pt[k].x;
            int j = pt[k].y;
            map[i][j] = 1;
            //若有连五，直接返回
            if(Is_over(i,j,map))
            {
                int score = -999999;
                free(node);
                map[i][j] = 0;
                return score;
            }
            int feedback = Evaluate(map);
            map[i][j] = 0;
            //剪枝
            if (feedback < pre)
            {
                int score = -100000;
                free(node);
                return score;
            }
            node->score = (feedback <= node->score ? feedback :
node->score);
        }
    }
```

```
        //若不是叶节点
        else
        {
            node->pre = 100000;
            for (int k = 0; pt[k].x != 0 || pt[k].y != 0; k++)
            {
                int i = pt[k].x;
                int j = pt[k].y;
                map[i][j] = 1;
                //若有连五，直接返回
                if(Is_over(i,j,map))
                {
                    int score = -999999;
                    free(node);
                    map[i][j] = 0;
                    return score;
                }
                int feedback = Create_children(node->IsMax,
node->depth, node->pre, map);
                map[i][j] = 0;
                node->pre = (feedback < node->pre ? feedback :
node->pre);
                //剪枝
                if (feedback < pre)
                {
                    int score = -100000;
                    free(node);
                    return score;
                }
                node->beta = (feedback <= node->beta ? feedback :
node->beta);
            }
            node->score = node->beta;
        }
    }
    int score = node->score;
    free(node);
    return score;
}
```

以上为博弈树的构建过程。整棵树枝干庞杂，尤其是递归函数的调用，会大大增加运算量。在进行深度搜索时，每增加一层搜索深度，运算量都会提升几十倍。若不进行任何的算法优化，搜索达到两层就已经非常缓慢，而要想棋力能与人类玩家抗衡，搜索层数至少为四层。因此，我们需采用各种优化算法来提升运算速度，从而加深搜索层数，使 AI 的棋力大幅提升。

3.2 性能优化

3.2.1 Alpha-Beta 剪枝

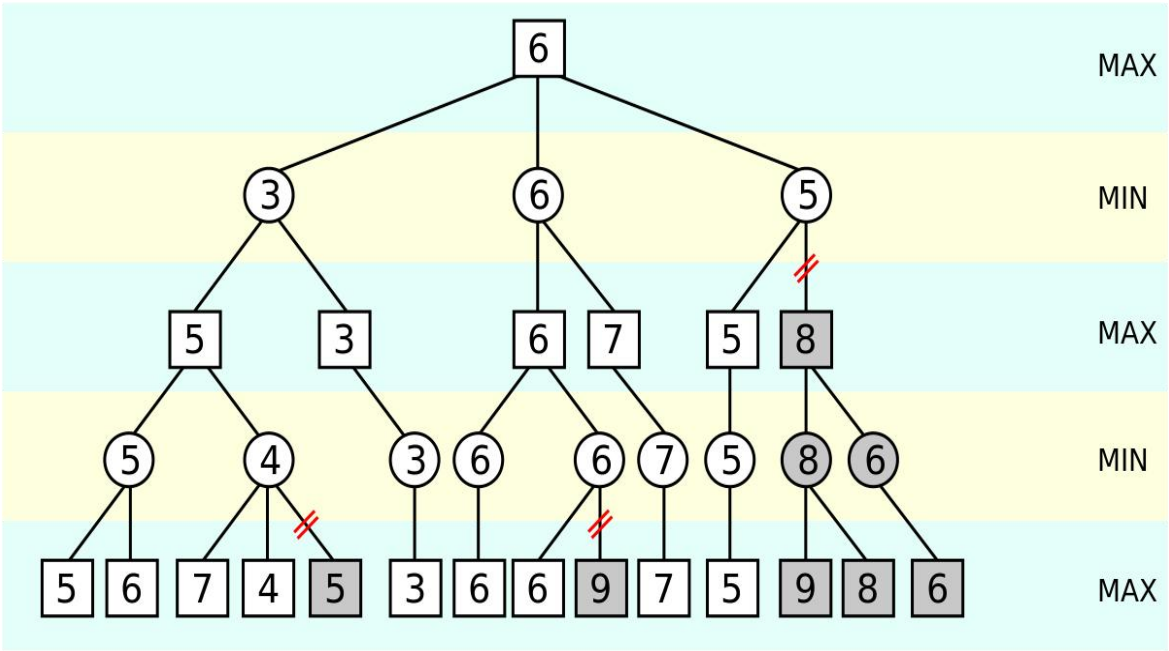
Alpha-Beta 剪枝算法是一种安全的剪枝策略，也就是不会对棋力产生任何负面影响。它的基本依据是：玩家不会做出对自己不利的选择。依据这个前提，如果一个节点明显是不利于自己的节点，那么就可以直接剪掉这个节点。

前面讲到过，AI 会在 MAX 层选择最大节点，而玩家会在 MIN 层选择最小节点。那么剪枝策略如下：

1. 在 MAX 层，假设当前层已经搜索到一个最大值 X，如果发现下一个节点的下一层（也就是 MIN 层）会产生一个比 X 还小的值，那么就直接剪掉此节点。

2. 在 MIN 层，假设当前层已经搜索到一个最小值 Y， 如果发现下一个节点的下一层（也就是 MAX 层）会产生一个比 Y 还大的值，那么就直接剪掉此节点。

下面图解说明：



如上图所示，在第二层，也就是 MIN 层，当计算到第二层第三个节点的时候，已知前面有一个 3 和一个 6，最大值至少是 6。 在计算第三个节点的时候，发现它的第一个孩子的结果是 5，因为当前是 MIN 节点，会选择孩子中的最小值，所以此节点值不会大于 5。而第二层已经有一个 6 了，第二层第三个节点肯定不会被选择。因此此节点的后序孩子就没有必要计算了。这是 MAX 节点的剪枝，MIN 节点的剪枝也是同样的道理。

代码实现如下：

```
//更新前驱值
node->pre = (feedback < node->pre ? feedback :
node->pre);
//剪枝
if (feedback < pre)
{
    int score = -100000;
    free(node);
    return score;
}
```

以 Min 节点为例，其前驱值 pre 由其之前的 Min 节点给出。若搜索到子节点的反馈分小于前驱值，则可以停止搜索，直接返回。同时，node->pre 也需要不断更新，便于递归后子节点的剪枝操作。

另一方面，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。那么如何排序呢？就是给所有待搜索的位置进行打分，按照分数的高低来排序。因此，我们需要一个启发式搜索函数来生成搜索顺序，从而提高剪枝效率。

3.2.2 启发式搜索函数

首先，我们想一下。什么样的点，容易出现上述的“极大值”（或者“极小值”）？显然，那些对胜负至关重要的点，相比那些不重要的点，更容易出现极值。那么，我们在遍历下一层之前，先交换一下遍历的顺序，让重要的点先进行遍历，就更加容易出现“运气好”的情况—更容易很快出现极值。那么我们首先需要一个函数，来给每个点一个评估，它到底对胜负是否非常重要，这个函数就被称为“启发式搜索”函数。

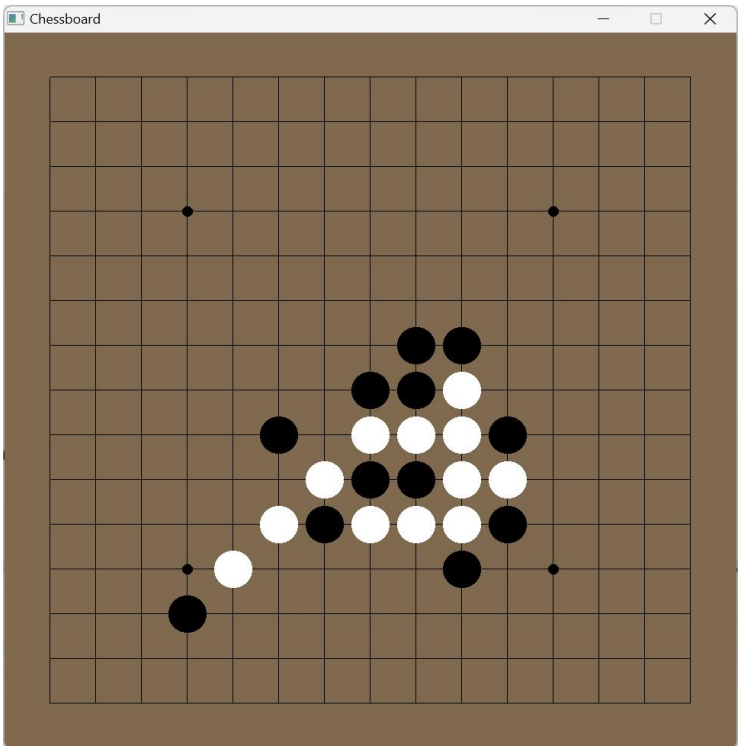
具体实现的时候，是根据这个位置能否形成**连五，活四，活三**的顺序来排序的。

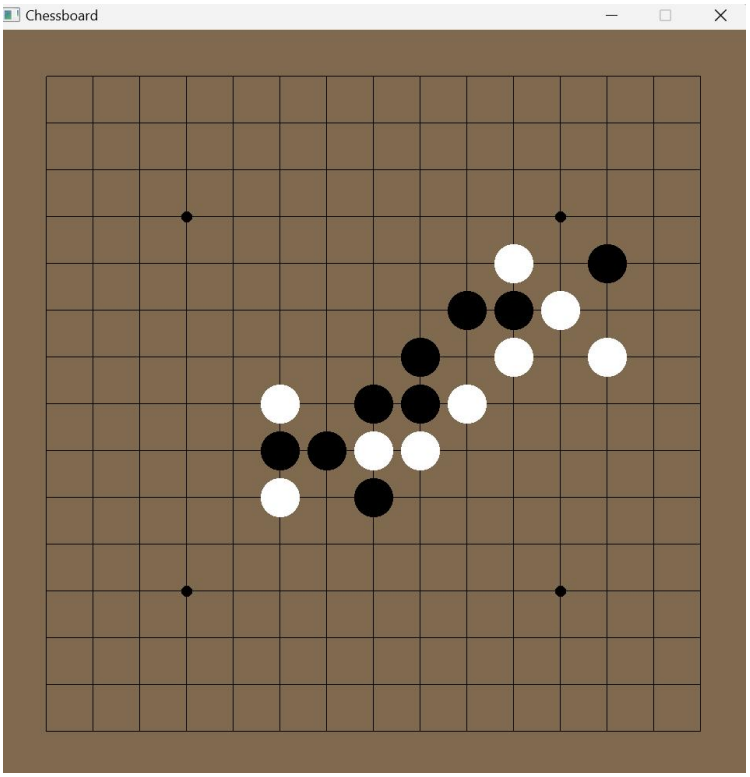
```
for(int i=0;i<15;i++)
{
    for(int j=0;j<15;j++)
    {
        if(Is_drop(i,j,map) && Is_linkfive_white(i,j,map))
        {
            list[n].x=i;
            list[n].y=j;
            n++;
        }
        else if(Is_drop(i,j,map) && Is_wakefour_white(i,j,map))
        {
            list[n].x=i;
            list[n].y=j;
            n++;
        }
        else if(Is_drop(i,j,map) && Is_wakethree_white(i,j,map))
        {
            list[n].x=i;
            list[n].y=j;
            n++;
        }
    }
}
```

尽管启发式搜索函数的设计比较粗糙，但只要我们对待搜索的节点进行一个大致的排序，就能极大的提高剪枝效率，省略大量不必要的运算。理论上来说，最大优化效果应该达到 1/2 次方。也就是如果你本来需要计算 10000 个节点，那么最好的效果是，你只需要计算 100 个点就够了。这是建立在所有的节点排序都是完美的假设上的。因为我们不可能完美排序，所以我们的优化效果达不到那么好。但是依然可以达到约 3/4 次方 的优化效果。

3.2 稳定性测试

我们默认玩家先手执黑，AI 后手执白，搜索层数为 4 层。





棋力方面：AI 能一定程度上兼顾进攻和防守，能与玩家周旋，很少下出俗手，也几乎不会迅速落败。尽管搜索层数并不深，但得益于评估函数较为全面而准确（几乎考虑了所有可能的棋形），AI 做出的决策往往十分沉稳，难以露出破绽。因此该 AI 的棋力甚至可能接近于 6 层搜索深度，对战普通玩家能够立于不败之地。

性能方面：AI 每步棋的思考时间基本能控制在 5 秒以内。若局势较为复杂或棋子太多时，AI 思考时间会明显增加，但最多不会超过 30 秒。

不足之处：

1. 接近边界时，AI 的判断时常出现低级错误。此问题源于评估函数对边界条件的处理不够妥当。
2. 当 AI 发现败局已定时，会出现“摆烂”行为，即索性胡乱走棋，而不是适当地“挣扎”一下。
3. AI 很少会迅速取胜，除非玩家犯错，换言之，AI 缺少有效的、致命的进攻策略，攻击性不足。若加入“算杀”功能模块，AI 的棋力还能更上一层。

第 4 章 总结与感想

在本课程设计中，我们成功实现了一个基于极小化极大值搜索的五子棋 AI。在设计各种模块功能的过程中，我们综合运用了多种数据结构与算法，进一步加深了对 c 语言的理解，如递归函数、动态内存空间的分配等等。通过逐步摸索，我积累了一些设计大型项目的经验，如 Cmakelist 的编辑，头文件的封装，外部库和工具链的运用，模块化设计。

在项目的进展中，我深深感受到人工智能的巨大潜力。计算机强大的算力使得博弈过程成为暴力的搜索，这是人类难以触及的高度。倘若再加入机器学习，AI 的性能将进一步提升，在某些复杂领域，能够做出人类无法想象的高级决策。我的五子棋 AI 只不过是一个极其简陋的计算模块，甚至不能称之为“人工智能”。但该项目激发了我对人工智能的好奇心与求知欲，也让我获得了极大的成就感。我希望能在日后进一步积累知识和经验，培养创新思维，继续在计算机领域开拓，探索和成长。

参考资料

https://github.com/raysan5/raylib

https://github.com/lihongxun945/myblog

https://blog.csdn.net/qq_44732921/article/details/104068832

https://blog.csdn.net/weixin_44062380/article/details/105881036

https://blog.csdn.net/qq_44671353/article/details/87893598