基于博弈树搜索的五子棋 AI

17341146 王程钥

1 算法原理

1.1 MinMax 搜索

极大极小值(MinMax)搜索适用于二人零和博弈问题。两个玩家在博弈树上逐层交替行动，两人的利益相互对立，对抗搜索。玩家 AB 均采用最优策略，玩家 A 作为 MAX 玩家希望使得分最大化，玩家 B 作为 MIN 玩家则希望得分最小化。

1.2 Alpha-beta 剪枝

MinMax 搜索必须检查的游戏状态随着博弈的进行呈指数级增长，需要通过高效的剪枝进行优化。一种常用的剪枝是 alpha-beta 剪枝，即剪掉不可能影响决策的分支，尽可能消除部分决策树。

MAX 玩家的估价函数为 alpha 值，MIN 玩家的估价函数为 beta 值。根据 MIN-MAX 搜索的定义，这两个值按照公式(1)(2)的方式更新

$$\text{alpha} = max_{son}(beta_{son}) \qquad (1)$$

$$\text{beta} = min_{son}(alpha_{son}) \qquad (2)$$

如果 MAX 玩家在某个节点上的 alpha 值大于等于它在博弈树上所有祖先的 beta 值的最小值，那么显然该节点的 alpha 值对祖先的 beta 值不会产生贡献。继续对该节点进行搜索只会导致这个节点的 alpha 值变得更大，更大的 alpha 值并不会减小祖先的 beta 值，因此该节点无需继续搜索。对于 MIN 节点同理，如果该节点的 beta 值小于等于所有祖先节点的 alpha 值的最小值，则该节点无需继续搜索。

1.3 估价函数

本次实验我设计了两种不同的估价函数。

第一种估价函数比较简单，枚举棋盘上所有连续段，计算连续段的长度，并根据长度对应的权重计算棋盘的估价值。如果一段棋子是 MIN 玩家下的则在估价值中减去相应长度的估价，如果是 MAX 玩家下的则在估价值中加上相应长度估价。

第二种策略首先考虑所有候选五子连线的落子情况，即棋盘中横、竖、对角线三个方向所有长度为 5 的连续格子的落子情况。统计这些位置的落子数，如果在 5 个位置中游戏双方都有落子则这个位置不可能完成五子连线，该位置贡献为 0。否则统计这 5 个位置中落子的数量，根据落子数加权计算得分。和第一种策略相同，如果一段棋子是 MIN 玩家下的则在估价值中减去相应长度的估价，如果是 MAX 玩家下的则在估价值中加上相应长度估价。

同时在第二种策略中我还加入了一些经典的胜负推断。以下为相关的推断内容(记当前玩家为 X，对手为 O)，若同时发生则按从 1 到 5 顺序选择编号最小的执行，推断 4 和推断 5 的条件可以叠加，比如一个眠四加一个活三也可以导致游戏结束。表 1 为相关条件的部分示例。

1) 棋盘上连续 5 个位置中有 4 个属于 X，剩余一个为空位，此时 X 必胜。
2) 棋盘上出现属于 O 的 4 子相连且两端均为空位(即活四)，此时 O 必胜。
3) 棋盘上连续 4 个位置中有 3 个属于 X，剩余一个为空位，且两端均为空位，此时 X 必胜。
4) 棋盘上出现 2 次或以上出现连续 5 个位置中有 4 个属于 O 另一个为空位的情况(即眠

四)，此时 O 必胜。

5) 棋盘上出现 2 次或以上属于 O 的三子相连且周围 4 子都是空位的情况(即活三)，此时 O 必胜。

编号	局面
1	XXXX. XXX.X
2	.OOOO.
3	.XXX.. .XX.X.
4	OOOO. OO.OO O.OOO
5	..OOO..

（表 1 游戏终结局面）

2 算法流程

Algorithm1 MIN-MAX 搜索

Input: 节点 n, 节点类型 type
Output: 估价函数
`if n is Terminal:`
`    return evaluate(n)`
`if type==MAX do`
`    return max(search(p, type^1) for p in n.Child_List())`
`if type==MIN do`
`    return min(search(p, type^1) for p in n.Child_List())`

算法 1 为 MIN-MAX 搜索的伪代码。

Algorithm2 alpha-beta 剪枝

```
Input: 节点 n, 节点类型 type, alpha, beta
Output: 估价函数
if n is Terminal:
    return evaluate(n)
if type==MAX do
    for p in n.Child_List() do
        alpha = max(alpha, search(p,type^1,alpha,beta))
        if beta <= alpha
            return alpha
if type==MIN do
    for p in n.Child_List() do
        beta = min(beta, search(p,type^1,alpha,beta))
        if beta <= alpha
            return beta
```

算法 2 为 alpha-beta 剪枝的伪代码。

Algorithm2 evaluate

```
Input: 棋盘局面 S
Output: 估价值
```

```
Value=0
If 当前局面先手必胜 do
    Return -inf
If 当前局面先手必败 do
    Return inf
For 棋盘上每一个长度为 5 的连通块 do:
    cnt<-每类棋子的数量
    if cnt[O] && cnt[X]: continue
    Value += len_val(cnt) 统计棋子数量对应估价值
Return Value
```

算法 3 为估价函数的伪代码(使用估价函数 2)。

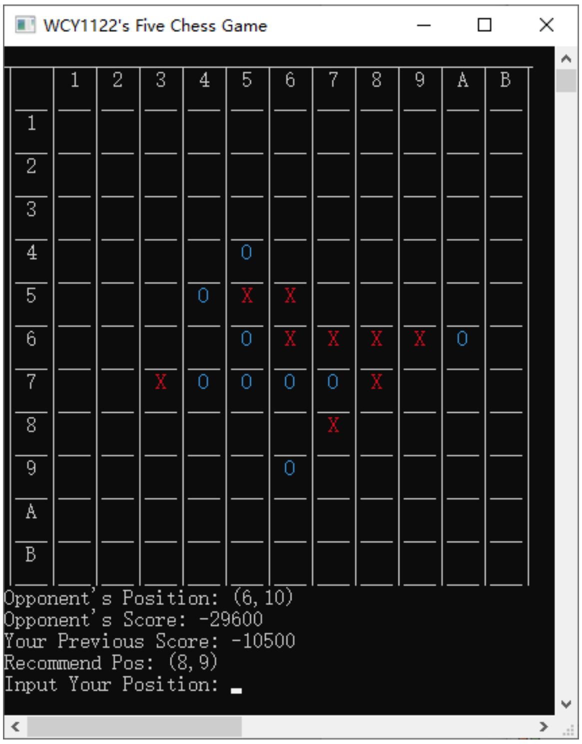
3 实现细节

3.1 界面设计

出于搜索速度的原因，本次实验我选择使用 C++编写代码。如图 1，我使用 C++设计了一个简单的五子棋界面。

上方是一个 11*11 的棋盘，每行有行号。先手玩家下的棋子使用‘X’表示，后手玩家下的棋子使用‘O’表示。每次游戏后相应的落子会在棋盘上直接显示。

棋盘下方包括三行数据，分别是对手上次落子的位置，对手上次落子后他的 alpha-beta 得分，以及本局中你的推荐下棋位置。玩家可以在棋盘下第四行输入下子位置，如果位置非法则会在第五行输出‘Invalid Input’。



(图 1)

界面设计使用了部分 c++控制台函数。

```
void Set_Windows()
```

```
{
  SetConsoleTitle("WCY1122's Five Chess Game");
  SMALL_RECT rc={0,0,50,30};
  SetConsoleWindowInfo(hout,true ,&rc);
  // resize windows
}
```

设置窗口大小和窗口标题。

```
void GetXY(int &x,int &y)
{
  CONSOLE_SCREEN_BUFFER_INFO pBuffer;
  GetConsoleScreenBufferInfo(hout, &pBuffer);
  x=pBuffer.dwCursorPosition.X;
  y=pBuffer.dwCursorPosition.Y;
}
```

获取光标位置。

```
void SetXY(int x,int y)
{
  COORD p;
  p.X=x;p.Y=y;
  SetConsoleCursorPosition(hout,p);
}
```

设置光标位置。

```
void clear(int posx,int posy,int w,int h)
{
  SetXY(posx,posy);
  for(int i=1;i<=w;i++)
  {
    for(int j=1;j<=h;j++)printf(" ");
    printf("\n");
  }
  SetXY(posx,posy);
}
```

区块清空，清除控制台上 x∈[posx, posx+w]，y∈[posy, posy+h]的区间。本质是在这些区间上输出空格。

```
void print_board()
{
  system("cls");
  for(int i=1;i<=4*len+5;i++)printf("_");
  printf("\n");
  printf("|   |");
  for(int i=1;i<=len;i++)
  {
    if(i<10)printf(" %d |",i);
```

```
    else printf(" %c |",'A'+i-10);
  }
  printf("\n");
  printf("|");
  for(int i=1;i<=len+1;i++)printf("___|");
  printf("\n");
  for(int i=1;i<=len;i++)
  {
    if(i<10)printf("| %d |",i);
    else printf("| %c |",'A'+i-10);
    for(int j=1;j<=len;j++)printf("   |");
    printf("\n");
    printf("|");
    for(int j=1;j<=len+1;j++)printf("___|");
    printf("\n");
  }
}
```

输出棋盘界面。

```
void show(int x,int y,int c)
{
  int nowx,nowy;
  if(c==2)SetConsoleTextAttribute(hout,4);
  else SetConsoleTextAttribute(hout,3); // set color
  s[x][y]=c;c=get(c); // num->char
  GetXY(nowx,nowy); // store now pos
  SetXY(y*4+2,x*2+1); // set pos
  printf("%c",c); // output
  SetXY(nowx,nowy); // reset ois
  SetConsoleTextAttribute(hout,7); // reset color
}
```

在(x,y)位置上显示落子。

3.2 博弈树搜索与 alpha-beta 剪枝

在搜索过程中，使用数组 s 记录棋盘局面，0 表示未落子位置，2 表示先手位置(X)，3 表示后手位置(O)。无论是哪位玩家下棋，我统一将博弈树上先手玩家操作的节点置为 MIN 节点，将后手玩家操作的节点置为 MAX 节点。

```
for(int i=1,tag=0;i<=len&&!tag;i++)
    for(int j=1;j<=len;j++)
    {
        if(s[i][j])continue;
        if(!surround(i,j))continue;
        s[i][j]=2; // 落子
        beta=min(beta,dfs(type^1,dep-1,alpha,beta,method)); // 搜索
        s[i][j]=0; // 恢复，回溯
```

```
        if(beta<=alpha){tag=1;break;}
    }
return beta
```

对于 MIN 节点，搜索所有合法落子位置，继续搜索并修改 beta 值，同时根据 alpha 值决定是否继续搜索。对于每层都索索整个棋盘显然没有意义，一般而言五子棋下一步的落子位置会出现在之前落下的棋子的周围。

```
for(int i=1,tag=0;i<=len&&!tag;i++)
    for(int j=1;j<=len;j++)
    {
        if(s[i][j])continue;
        if(!surround(i,j))continue;
        s[i][j]=3; // 落子
        alpha=max(alpha,dfs(type^1,dep-1,alpha,beta,method)); // 搜索
        s[i][j]=0; // 恢复，回溯
        if(beta<=alpha){tag=1;break;}
    }
return alpha;
```

对于 MAX 节点，同理，搜索相邻位置更新 alpha 值。

为了减少不必要的搜索，本次实验采取记忆化搜索的方法，先用一个 hash 函数获取当前棋局状态的编码，后在一个 STL 的 map 中查询这个状态是否出现。若出现过则直接返回其对应的 alpha 值或 beta 值。

```
ll get_hash(int s[][N])
{
  ll sum=0;
  for(int i=1;i<=len;i++)
    for(int j=1;j<=len;j++)
      sum=sum*seed+s[i][j];
  return sum;
}
```

以上为 hash 函数，seed 为一个质数，hash 值允许自然溢出。

```
ll hash_val=get_hash(s);
if(vis.count(hash_val))return vis[hash_val];
```

以上为记忆化部分代码。vis 是一个 STL 的 map。

3.3 估价函数

如 1.3 节，本次实验我使用了两种估价函数。第一种策略的相关代码如下。

```
int len_val[]={0,100,5000,20000,50000};
int res=0;
for(int i=1;i<=len;i++)
    for(int j=1;j<=len;j++)
        for(int k=0;k<4;k++) //四方向
        {
            if(s[i-t1[k]][j-t2[k]]==s[i][j])continue;
```

```
            int l=0,ch=s[i][j];
            for(int x=i,y=j;l<5;l++,x+=t1[k],y+=t2[k])
                if(s[x][y]!=ch)break;
            if(ch==2)res-=len_val[l];
            else res+=len_val[l];
        }
return res;
```

第二种策略的相关代码如下。

```
int tmp[N][N][4]={0};
int len_val[]={0,100,5000,20000,50000};
int res=0;
int count34=0,count3=0,count4=0;
for(int i=1;i<=len;i++)
  for(int j=1;j<=len;j++)
    for(int k=0;k<4;k++) //四方向
    {
      if(cha(i+4*t1[k],j+4*t2[k]))continue;
      int cnt[4]={0};
      for(int l=1,x=i,y=j;l<=5;l++,x+=t1[k],y+=t2[k])
        cnt[s[x][y]]++; // 统计颜色
      if(cnt[2]&&cnt[3])continue; // 双色均出现，无贡献
      int now=max(cnt[2],cnt[3]); // 5 元祖棋子数
      int inv=(cnt[2]?-1:1); // 玩家类型
      int pre=tmp[i-t1[k]][j-t2[k]][k]; // 结合前一个 5 元组

      if(now==4&&cnt[type])return inv*(END+3); // 4+1 type 玩家必胜
      if(now==4&&pre==now)count4++; // 活四
      if(now==4)count34++;// 眠四
      if(now==3&&pre==3)
      {
        if(cnt[type])count3++; // 活三 3+1 下一步变活四
        if(!cha(i-2*t1[k],j-2*t2[k]))
        {
          int prep=tmp[i-2*t1[k]][j-2*t2[k]][k];
          if(prep==3)count34++; // 活三
        }
      }
      res+=inv*len_val[now]; //加权求和
      tmp[i][j][k]=now;
    }
    int inv=((type==2)?-1:1);
    if(count4)return -inv*(END+2); //对手活四, type 玩家输
    if(count3)return inv*(END+1); //type 玩家活三, 赢
    if(count34>=2)return -inv*END; // 对手两个活三 or 眠四, type 玩家输
```

4 实验结果与分析

4.1 问题描述与分析

本次实验是基于棋盘大小为 11*11 的五子棋进行的。五子棋规则参考经典规则，两个玩家对弈，横排、竖排、对角线均可连线，没有禁手，率先形成五子连线的玩家获胜。

因为没有禁手，所以理论上先手必胜。如果在一个策略下后手战胜了先手，那么这个策略显然优于被击败的策略。在本章我将根据不同策略下 AI 相互交战的战绩评估 AI 的水平。

4.2 搜索深度的影响

我尝试枚举不同的搜索深度，同时采用策略 2 作为估价函数，让不同搜索深度的 AI 进行对弈。如表 2 可以看到，大部分棋局下先手都战胜了后手。毕竟本次实验的五子棋是没有禁手的，在这种情况下是先手必胜的。

表 2 中出现了 4 场平局，除了 D=1 后手和 D=5 先手外其它平局先手玩家的搜索深度都小于后手玩家。唯一一次先手输给后手发生在先手搜索深度为 2，后手搜索深度为 5 的情况下。总体而言随着后手玩家搜索深度的增大，先手玩家获得胜利的难度还是越来越大的，获胜步数总体也呈上升趋势。由此可见搜索深度越大搜索的结果也越准确。

通过观察可以发现 D=1 的搜索表现优异，甚至在后手的情况下逼平了 D=5 先手的玩家。这是由于估价函数的设计总体偏防守，包含了几个经典的败局棋谱，导致防守能力强劲的 D=1 玩家多次逼平对手。

	D=1 后手	D=2 后手	D=3 后手	D=4 后手	D=5 后手
D=1 先手	W-14	W-14	W-20	D	D
D=2 先手	W-25	W-8	W-16	W-13	L-8
D=3 先手	W-19	W-20	W-12	W-19	W-12
D=4 先手	W-10	W-7	W-7	W-9	D
D=5 先手	D	W-8	W-14	W-10	W-19

（表 2 W 表示先手获胜，L 表示先手失利，D 表示双方打平，-后接数字表示比赛回合数）

4.3 不同的估价函数

我比较了 3.3 节中提到的两种估价函数，枚举不同深度(1,3,4)和先后手进行对战统计结果。表 3 为策略 1 先手策略 2 后手的结果，表 4 为策略 1 后手策略 2 先手的结果。可以看到使用策略 2 的 AI 被使用策略 1 的 AI 碾压，仅有一次深度较大时后手获胜。采用策略 1 只有在搜索深度增加的情况下才勉强具备一点战斗力。由此可见策略 2 的优越之处，同时可见估价函数对搜索的重要性。

策略 1\策略 2	D=1 后手	D=3 后手	D=4 后手
D=1 先手	L-2	L-3	L-2
D=3 先手	L-2	L-3	L-2
D=4 先手	L-12	L-5	L-21

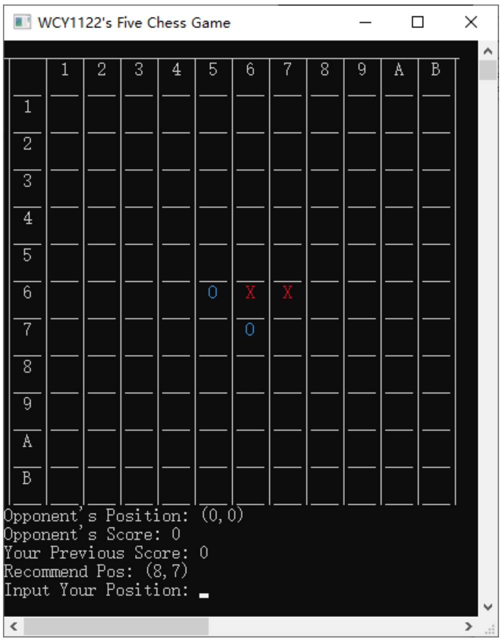
（表 3 先手策略 1，后手策略 2）

策略 1\策略 2	D=1 先手	D=3 先手	D=4 先手
D=1 后手	L-3	L-3	W-11
D=3 后手	L-3	L-3	L-19
D=4 后手	L-3	L-3	L-7

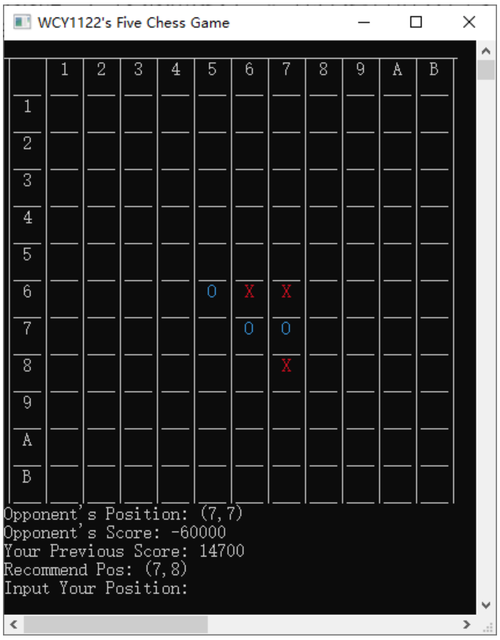
（表 4 先手策略 2，后手策略 1）

4.4 实验结果展示

玩家先手，AI 后手，AI 采用策略 2，搜索深度为 4。程序运行时会同步运行一个为玩家服务的外挂，搜索层数与策略和 AI 相同。图 2~11 为实验结果。



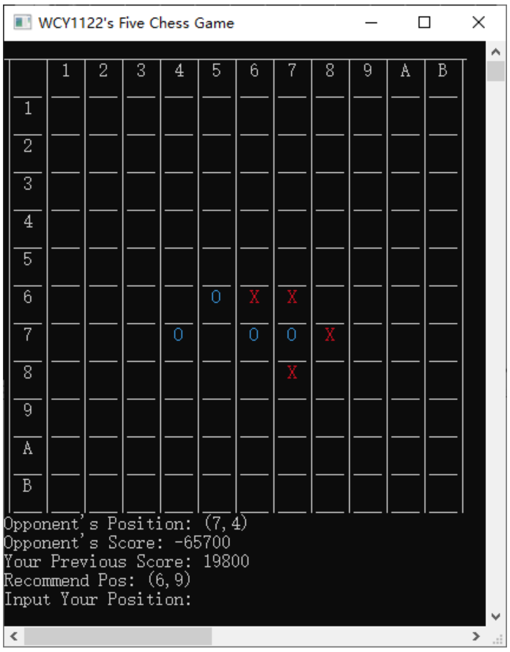
（图 2）



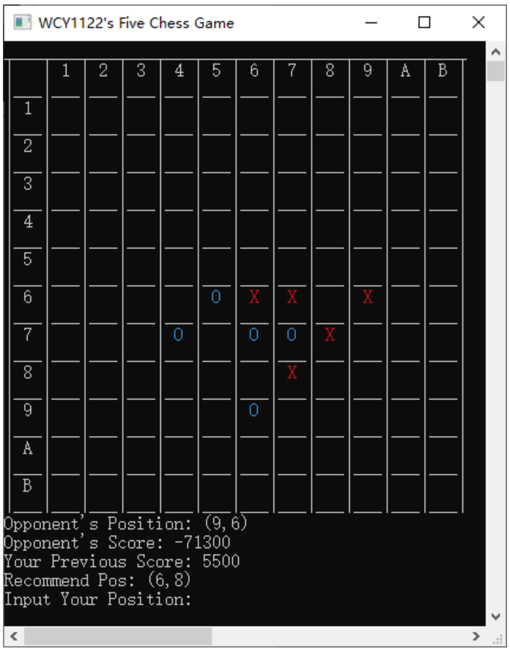
（图 3）

图 2 为初始界面，棋盘上有初始的四个子。

图 3 为第 1 回合后的界面，玩家在(8,7)位置下棋，对手在(7,7)位置下棋。



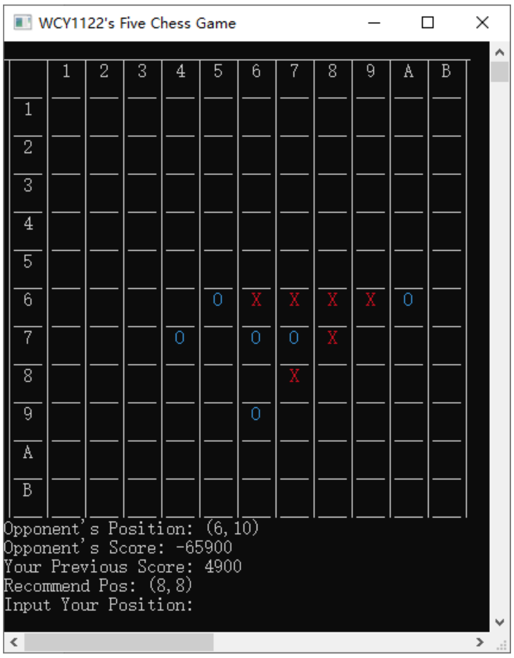
（图 4）



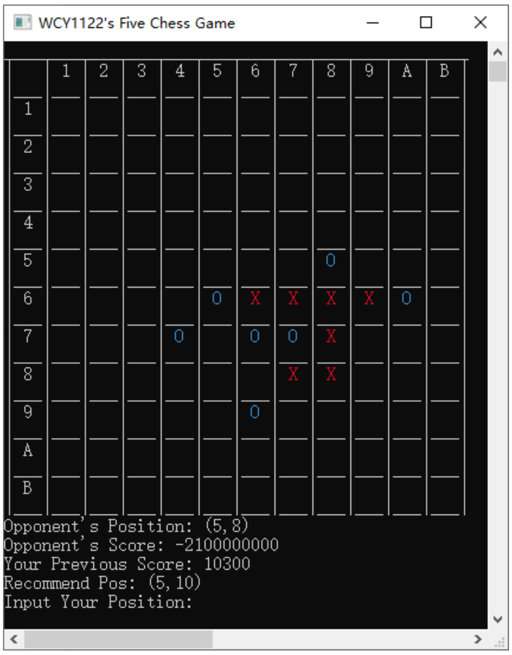
（图 5）

图 4 为第 2 回合后的界面，玩家在(7,8)位置下棋，AI 在(7,4)位置下棋。

图 5 为第 3 回合后的界面，玩家在(6,9)位置下棋，AI 在(9,6)位置下棋。



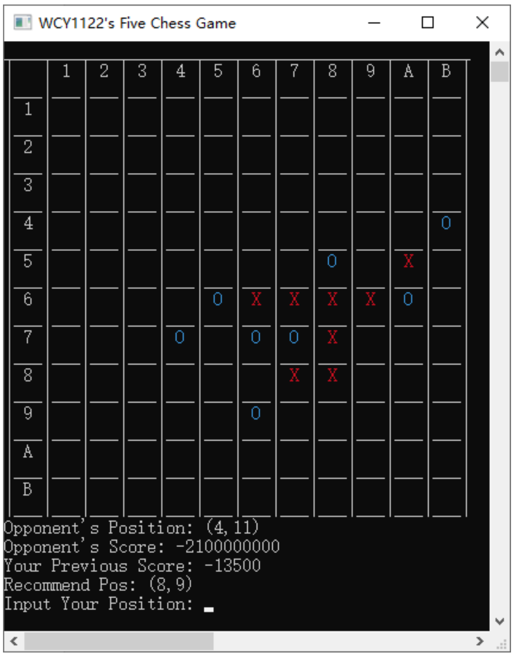
(图 6)



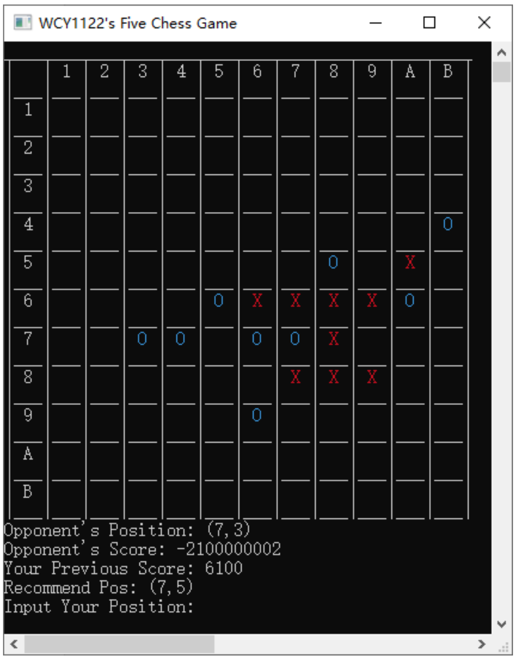
(图 7)

图 6 为第 4 回合后的界面，玩家在(6,8)位置下棋形成眠 4，AI 在(6,9)位置下棋进行防守化解危机。

图 7 为第 5 回合后的界面，玩家在(8,8)位置下棋形成活 3，AI 在(5,8)位置下棋进行防守化解危机。此时估价函数显示先手已经必胜。



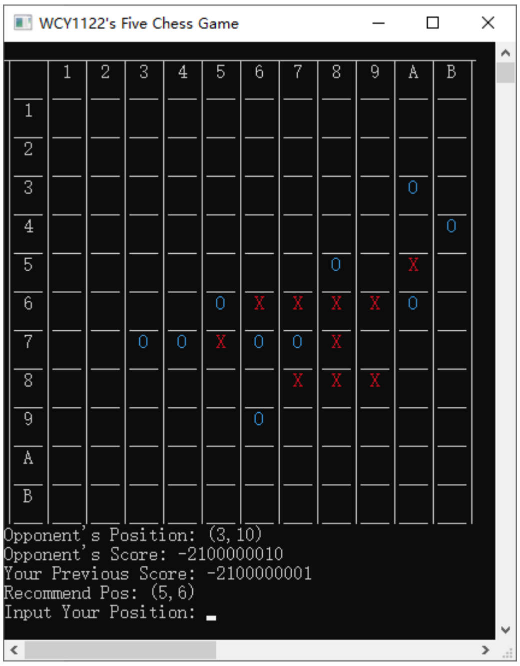
(图 8)



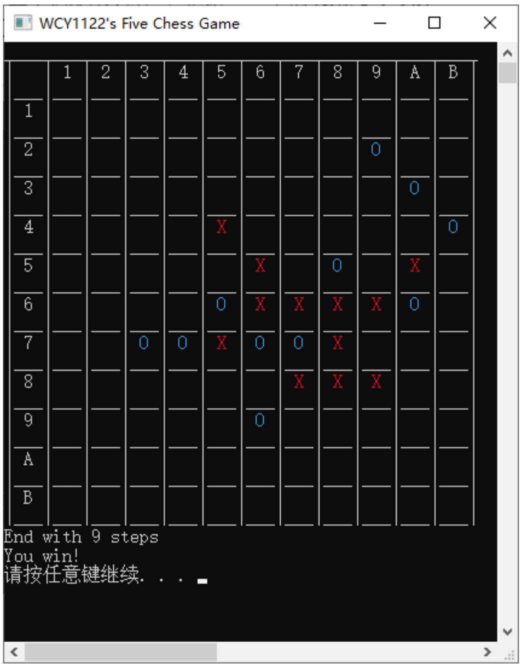
(图 9)

图 8 为第 6 回合后的界面，玩家在(5,10)位置下棋形成眠 4，AI 在(4,11)位置下棋进行防守化解危机。此时估价函数仍然显示先手必胜。

图 9 为第 7 回合后的界面，玩家在(8,9)位置下棋形成双活 3，根据 3.3 节的规则此时可以判定先手必胜，AI 放弃防守最后一搏，在(7,5)位置下棋形成眠 4。



(图 10)



(图 11)

图 10 为第 8 回合后的界面，玩家在(7,5)下棋成功防守住了 AI 的最后一波进攻，AI 见大势已去，放弃治疗在(3,10)位置下棋。

图 10 为最终的结局，玩家在(5,6),(4,5)两个位置连下两颗棋收割比赛获得胜利。

5　总结

本次实验我采用带 alpha-beta 剪枝的极大极小值搜索方法编写了简单的五子棋游戏和五子棋 AI。我设计了简单的界面，并结合五子棋规则设计了简单的估价函数。通过让不同估价策略和不同深度的 AI 相互博弈进行比较，我最终找到了比较出色的估价函数和合适的深度，使得 AI 能够拥有不错的性能。