人工智能实验报告——黑白棋AI

学号	姓名	联系方式
16337303	张家豪	994328597@qq.com

实验内容

1. 实现 8×8 黑白棋翻转的人机对战：
    - 实现博弈树后MinMax算法
    - 使用Alpha-Beta剪枝
    - 优化评价函数

算法原理

1. 博弈树和MinMax算法

    针对某一赛局 s 建造博弈树的过程是指我们从该赛局，将其所有可能的发展赛局加入其子节点中，再对其每个子节点中的赛局进行这个迭代的过程，直到赛局结束（分出胜负）。Minmax算法就是从所有可能的结束状态出发，其父节点节点赛局总能在子节点赛局中找到最优行动，不断迭代到起始的目标赛局 s，这样处于赛局 s 的玩家就能做出最优的选额。

```
function minimax(node, depth)
    if node is a terminal node or depth = 0
        return the heuristic value of node
    if the adversary is to play at node
        let α := +∞
        foreach child of node
            α := min(α, minimax(child, depth-1))
    else {we are to play at node}
        let α := -∞
        foreach child of node
            α := max(α, minimax(child, depth-1))
    return α
```

——伪代码摘自维基百科

2. 评价函数设计和Alpha-Beta剪枝

    在博弈树和MinMax算法的实现中，由于算法的时间复杂度是指数级别，实际情况中要探索的层数过多，导致要探索到的子节点也过多。这时候就需要进行剪枝，一般来说我们可以控制博弈树的深度，对当前局面作出效益值的评估来替代为最终结果的预测。

为了进一步加快搜索速度，这时候我们需要进行剪枝，减少搜索时探索的分支，便能将搜索时间充分利用在有可能产生最优解的子树上。

从某一赛局 s 出发，为其初始化变量 $\alpha=-\infty$, $\beta=+\infty$ ，将这两个变量向其子赛局节点传递，其子赛局节点再分别将其 α 和 β 向下不断传递，到达终局节点的时候会根据结果对父节点的 α 或 β 进行更新，这样的更新会不断向上传递。在我们进行探索的过程中，当我们发现某个赛局节点中 $\beta \leq \alpha$ 时，我们就进行一次剪枝，不再探索这个赛局节点的其他未探索过的子节点，并将记录的行动和效益值向上传递，最终会回到初始赛局 s ，至此我们就可以在赛局 s 中作出最优选择。

值得注意的是Alpha-Beta剪枝只是剪去了没必要探索的路径，并不会影响MinMax算法的最终结果。

```
function alphabeta(node, depth, α, β, MinMax)
      if depth = 0 or node is left_node
          return benefit of node
      if MinMax = Max
          v := -∞
          foreach child of node
              v := max(v, alphabeta(child, depth - 1, α, β, FALSE))
              α := max(α, v)
              if β ≤ α
                  break
          return v
      else
          v := ∞
          foreach child of node
              v := min(v, alphabeta(child, depth - 1, α, β, TRUE))
              β := min(β, v)
              if β ≤ α
                  break
          return v
```

关键代码展示

1. 关键变量的定义和类型的重命名

```
const int row_size = 8; //row_size定义了棋盘的
const int board_size = row_size * row_size; //board_size定义了
typedef std::bitset<board_size> state_t; //已落子状态的类型
typedef std::pair<int, int> pos_t; //落子位置
state_t left_shift_base;
state_t right_shift_base;
enum turn{black_turn = 1, white_turn}; //持方
enum direction{Up, Down, Left, Right, UpLeft, UpRight, DownLeft,
DownRight};
```

2. 黑白棋逻辑设计

```
class State{
private:
    state_t black; //黑子落子的情况，1为占有
    state_t white; //白字落子的情况，1为占有
    turn cur_turn; //当前状态下的持方
    state_t valid_pos; //当前持方可以下的位置，1为合法位置

public:
    State();
    friend std::ostream& operator << (std::ostream& , State);
    friend std::ostream& operator << (std::ostream& , state_t);
    state_t shift(state_t , direction);
    state_t getValid(); //获取当前持方可以下的位置
    std::vector<pos_t> getValids(); //获取当前持方可以下的位置的二维坐标形式
    bool action(int x, int y); //当前持方执行一次行动
    int score(); //当前局面的效益值（评价函数）
    bool isFinish(); //判断当前赛局是否结束
    turn getPlayer(); //获取当前持方
    std::string getWinner(); //获取优胜者
};
```

3. Alpha-Beta剪枝

```
int minMax(State cur_state, int cur_height, int Alpha, int Beta, int max_height, pos_t& best_pos){
    if(cur_height == max_height || cur_state.isFinish()){//当棋局到达终局或探索深度达到预设额最大深度时退出递归
        return cur_state.score();
    }
    pos_t best = std::make_pair(0, 0);//表示为二维坐标形式
    pos_t t = best;//best是
    int cost = 0;
    if(cur_state.getPlayer() == black_turn){
        std::vector<pos_t> actions = cur_state.getValids();
        cost = INT_MIN + 1;
        for(auto iter = actions.begin(); iter != actions.end(); iter++){
            State sub_state = cur_state;
            sub_state.action(iter->first, iter->second);
            int tmp = minMax(sub_state, cur_height + 1, Alpha, Beta, max_height, t);
            best = tmp > cost? std::make_pair(iter->first, iter->second) : best;
```

```
            cost = cost > tmp? cost : tmp;
            Alpha = cost > Alpha? cost : Alpha;
            if(Beta <= Alpha)
                break;
        }
    }
    else if(cur_state.getPlayer() == white_turn){
        std::vector<pos_t> actions = cur_state.getValids();
        cost = INT_MAX;
        for(auto iter = actions.begin(); iter != actions.end(); iter++)
{
            State sub_state = cur_state;
            sub_state.action(iter->first, iter->second);
            int tmp = minMax(sub_state, cur_height + 1, Alpha, Beta,
max_height, t);
            best = tmp < cost? std::make_pair(iter->first, iter-
>second) : best;
            cost = cost < tmp? cost : tmp;
            Beta = cost < Beta? cost : Beta;
            if(Beta <= Alpha)
                break;
        }
    }
    best_pos = best;//best_pos是全局变量，保存最优位置
    return cost;
}
```

评价函数设计

1. 基于黑白子个数

   记当前状态 S_i 时， 黑色子个数为 B_i^1 ，白色子个数为 W_i^1 ，则估值函数 $H_1(S_i) = B_i^1 - W_i^1$ 。黑色方落子时AI选取可能达到最大的 H_1 的落子位置，白色方落子时AI 选取可能达到最小的 H_1 的落子位置。

2. 基于地势值

   为 8×8 的棋盘上的每一格都赋一个地势值（根据经验赋值），如果当前黑色方占据该格子，则黑色方具有其地势值，记当前状态为 S_i ，记当前黑棋地势值为 B_i^2 ， 白色子地势值为 W_i^2 ，则估值函数 $H_2(S_i) = B_i^2 - W_i^2$ 。黑色方落子时AI选取可能达到最大的 H_2 的落子位置，白色方落子时AI 选取可能达到最小的 H_2 的落子位置。

   本次实验使用中所使用的地势图

	1	2	3	4	5	6	7	8
1	90	-20	10	10	10	10	-20	90
2	-20	-20	1	1	1	1	-20	-20
3	10	1	1	1	1	1	1	10
4	10	1	1	1	1	1	1	10
5	10	1	1	1	1	1	1	10
6	10	1	1	1	1	1	1	10
7	-20	-20	1	1	1	1	-20	-20
8	90	-20	10	10	10	10	-20	90

3. 基于双方行动力

   记当前状态 S_i 时， 如果轮到黑色方下，黑色方可下位置的总数记为的为 B_i^3 ，如果是白色方下，白色方可下位置的总数为 W_i^3 ，则估值函数 $H_3(S_i) = B_i^3 - W_i^3$ 。黑色方落子时AI选取可能达到最大的 H_3 的落子位置，白色方落子时AI 选取可能达到最小的 H_3 的落子位置。

4. 基于稳定子

   记当前状态 S_i 时， 不可被翻转的黑色棋子总数记为的为 B_i^4 ，不可被翻转的白色棋子总数为 W_i^4 ，则估值函数 $H_4(S_i) = B_i^4 - W_i^4$ 。黑色方落子时AI选取可能达到最大的 H_4 的落子位置，白色方落子时AI 选取可能达到最小的 H_4 的落子位置。

所以有估价函数

$$H(S_i) = \sum_{i=1}^{4} \alpha_i H_i$$

其中，α_i 可以是变量，根据当前回合数决定。

根据目前个人对黑白棋的了解，在探索深度一定的情况下：开局时，行动力和地势值是值得优先考虑的，而到中局时，稳定子开始变得逐渐重要，而接近终局时，双方子数才是值得优先考虑的。

结果展示及分析

人机对弈

人先手，AI后手

1. 开局3回合

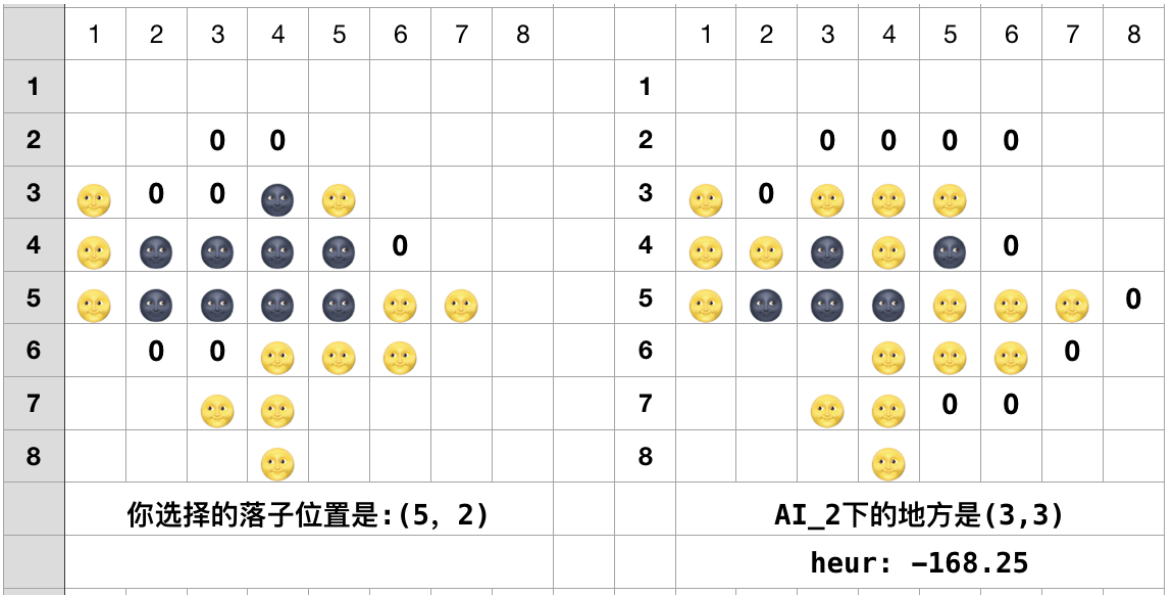
	1	2	3	4	5	6	7	8			1	2	3	4	5	6	7	8
1										1								
2										2								
3			0	●	0					3				●				
4				●	●					4				●	●			
5			0	●	○					5			○	○	○			
6										6		0	0	0	0	0		
7										7								
8										8								
	你选择的落子位置是:(3,4)										AI_2下的地方是(5,3)							
											heur: -79.2656							

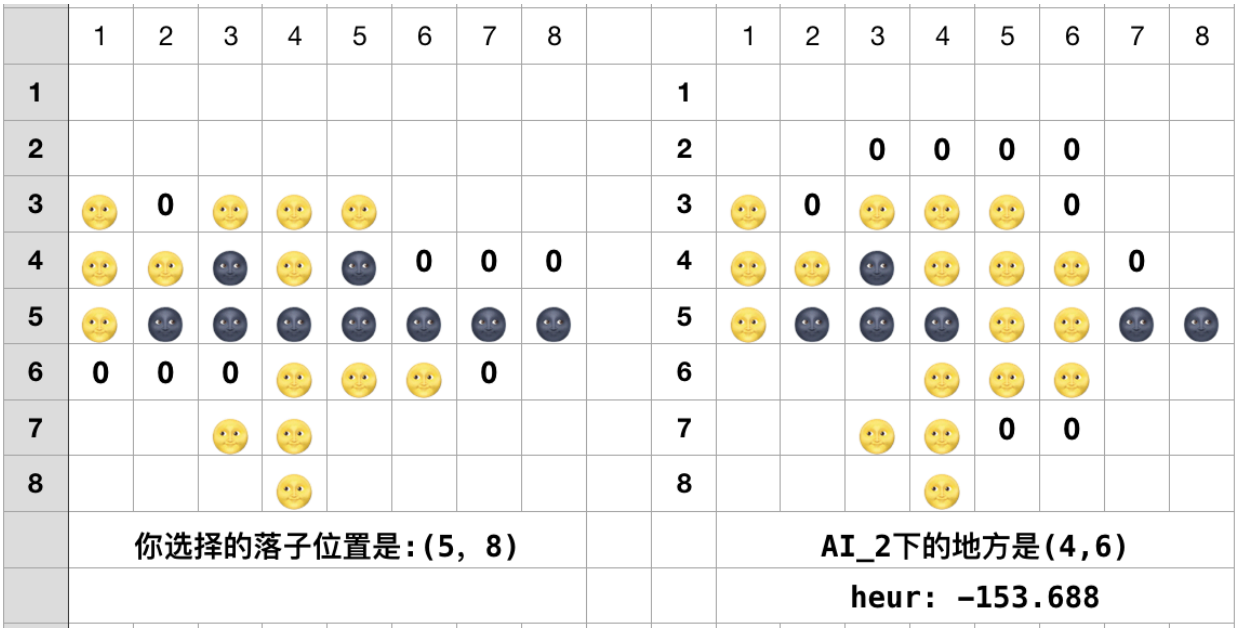
	1	2	3	4	5	6	7	8			1	2	3	4	5	6	7	8
1										1								
2										2								
3			0	●	0					3				●				
4				●	●					4				●	●			
5			○	●	○					5		0	○	○	○	0		
6				●						6		0		○	0	0		
7			0		0					7			○	0				
8										8								
	你选择的落子位置是:(3,4)										AI_2下的地方是(7,3)							
											heur: -76.4844							

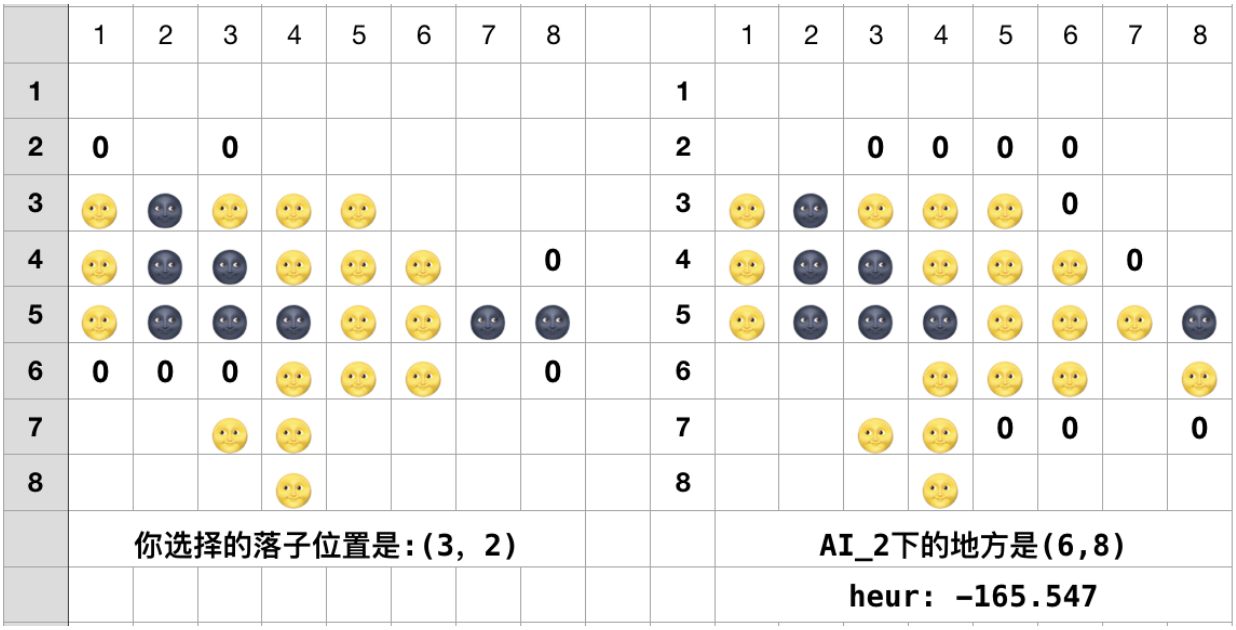
	1	2	3	4	5	6	7	8			1	2	3	4	5	6	7	8
1										1								
2										2								
3			0	●	0					3				●	○	0		
4				●	●					4		0		○	○	0		
5			○	●	○					5		0	○	●	○	0		
6				●						6				●				
7			○	●	0					7		0	○	●				
8										8		0						
	你选择的落子位置是:(7,4)										AI_2下的地方是(7,3)							
											heur: -76.4844							

很显然，开局几步对后续结果影响变化不大，AI作为后手预估的值甚至在增大，而AI要获取最小值，说面我作为先手方棋下得不错。

2. 中局阶段

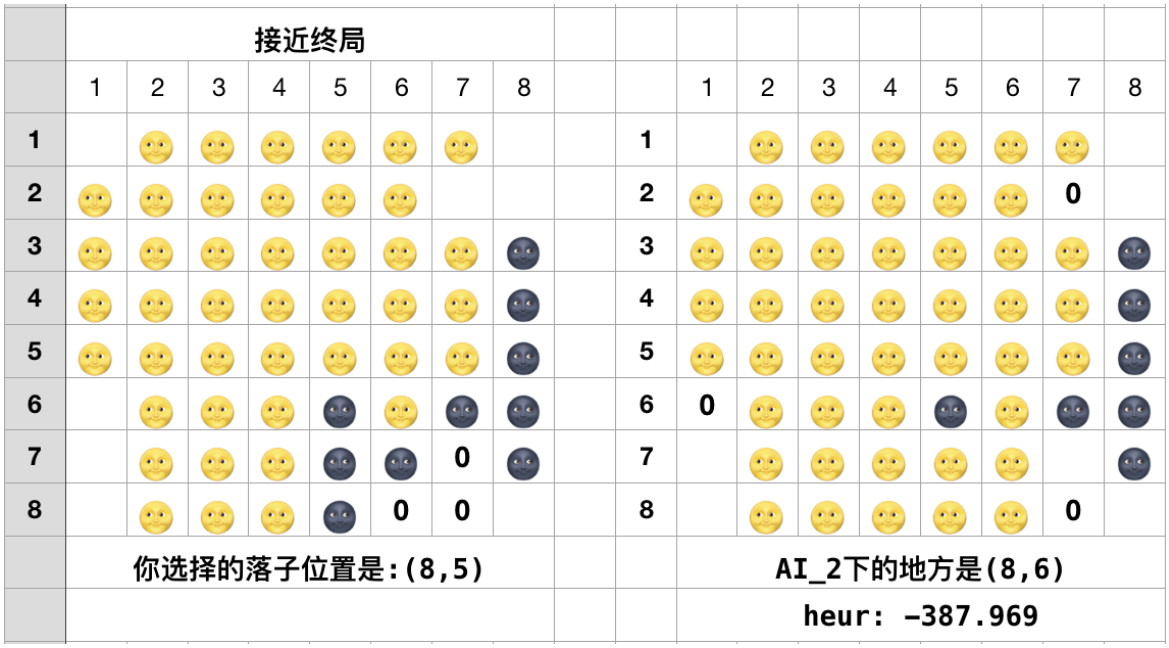

你选择的落子位置是:(5, 2)
AI_2下的地方是(3,3)
heur: -168.25

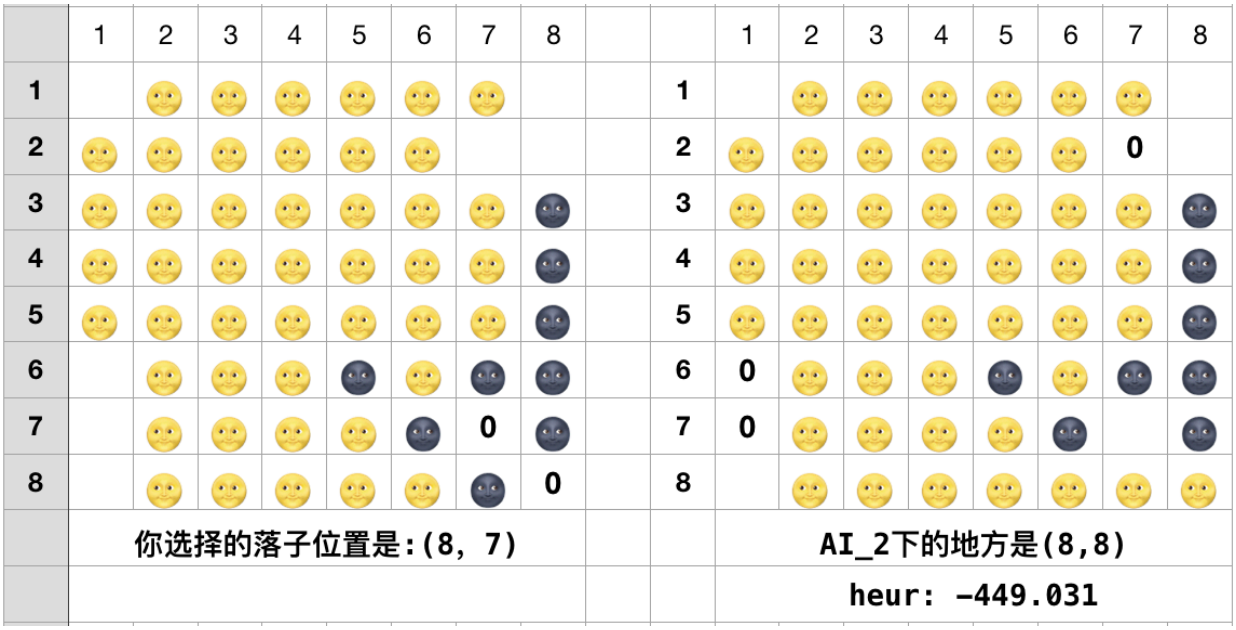

你选择的落子位置是:(5, 8)
AI_2下的地方是(4,6)
heur: -153.688


你选择的落子位置是:(3, 2)
AI_2下的地方是(6,8)
heur: -165.547

这时候开始不对劲了，我很想让自己被包起来，但是AI似乎总能找到方式化解，而且AI的估值越来越小，说明它越有把握赢了。

3. 接近终局





	1	2	3	4	5	6	7	8			1	2	3	4	5	6	7	8
1		⚪	⚪	⚪	⚪	⚪	⚪			1		⚪	⚪	⚪	⚪	⚪	⚪	
2	⚪	⚪	⚪	⚪	⚪	⚪		0		2	⚪	⚪	⚪	⚪	⚪	⚪		0
3	⚪	⚪	⚪	⚪	⚪	⚪	⚪	⚫		3	⚪	⚪	⚪	⚪	⚪	⚪	⚪	⚫
4	⚪	⚪	⚪	⚪	⚪	⚪	⚪	⚫		4	⚪	⚪	⚪	⚪	⚪	⚪	⚪	⚫
5	⚪	⚪	⚪	⚪	⚪	⚪	⚪	⚫		5	⚪	⚪	⚪	⚪	⚪	⚪	⚪	⚫
6	⚫	⚫	⚫	⚫	⚫	⚪	⚫	⚫		6	⚫	⚫	⚫	⚫	⚫	⚪	⚫	⚫
7	0	⚪	⚪	⚪	⚪	⚫	0	⚫		7	0	⚪	⚪	⚪	⚪	⚫	0	⚫
8		⚪	⚪	⚪	⚪	⚪	⚪	⚪		8		⚪	⚪	⚪	⚪	⚪	⚪	⚪
	你选择的落子位置是:(6, 1)										AI_2下的地方是(7,1)							
											heur: -544.688							

下到这里，我基本凉了，行动力不多，子也不多，也基本没有稳定子。AI的估值变化也越来越小，说明它赢的把握越来越大。

实验思考

我在本次实验中坐的使用的估价函数还是比较简单的，甚至可以说与人工智能基本没关系，像是在做算法作业一样，上网了解到了如下的一些方法可以用来运用到黑白棋的AI设计中：（因为个人能力和时间所限，我并没有尝试过）

1. 蒙特卡洛方法：

   就我所了解的蒙特卡洛方法来说（欢迎助教指正我，我真的是小白），蒙特卡洛方法就是利用随机采样的样本值来估计真实值。我们可以用模拟很多的简单随机比赛，通过跟踪某落子位置和最终胜负之间的关系，我们可以大致得到某一步棋的评价。这几乎不用考虑太多的下棋技巧，但是需要进行大量的随机对局。

2. 时序差分学习

   时序差分学习结合了动态规划和蒙特卡洛方法。

   "蒙特卡洛的方法是模拟（或者经历）一段序列，在序列结束后，根据序列上各个状态的价值，来估计状态价值。 时序差分学习是模拟（或者经历）一段序列，每行动一步（或者几步），根据新状态的价值，然后估计执行前的状态价"（——转自https://blog.csdn.net/qq_30159351）

   在黑白棋中，我们也是可以追踪一个落子位置的评价，可以每落一次子就动态更新这个位置的评价函数。

3. 导入海量数据库

   我发现黑白棋其实算是有一定受众的游戏，如果将所有对局情况记录下来，应该可以达到上千万的局的对局数据，而 8×8 黑白棋所有可能的状态有 3^{64} 种，决定AI的落子时，我们找到所有具有相同过程状态的对局，然后选取最后赢的概率最大的落子位置下。

但是这方法光是想想都很害怕，就像KNN一样巨慢。