人工智能 实验二

151220129 **计科 吴政亿** 18805156360@163.com

Task 1

`MiniMaxDecider.java` 的代码相比于书本中的样例，将min与max进行了合并，用一个 `boolean maximize` 代替，以及通过+1,-1将两个函数抽象成一个，下面对三个函数进行简要介绍。

变量定义

类型	变量名	意义
boolean	maximize	为true时最大化val,为false时最小化val
int	depth	限制了递归的深度
Map	computedStates	用Hash存储不同局面及得分，避免重复运算

MiniMaxDecider(boolean maximize, int depth)

对上面的变量进行初始化

public Action decide(State state)

1. 根据maximize将value初始化为负无穷或正无穷
2. 定义 `bestAction` 用于存储最高得分的动作序列
3. 初始化因子 `flag` 为-1或+1.
4. 遍历所有当前局势下所有可能的动作
    - 定义变量 `newState` 与 `newValue` 为此动作的预期局势与得分
    - 如果 `maximize == true` 则取其中的得分最高项，反之取最低得分
    - 两个if是为了保证 `bestAction` 中的actions得分相同且均为最优解
5. 从 `bestAction` 中随机选取一个action执行

public float miniMaxRecursor(State state, int depth, boolean maximize)

此函数与上一个函数 `decide` 相似，是抽象出来的递归部分

1. 如果 `computedStates` 中存在，即已经计算过得分，则直接返回
2. 如果游戏结束或者到达递归最底层，则直接返回当前局面由启发式函数计算的评估分数
3. 与函数 `decide` 大体相同
4. 最后返回评估分数

private float finalize(State state, float value)

未使用，返回 `value` 。

Task 2

对下列函数加入 `alpha` 与 `beta` 参数，进行 $\alpha - \beta$ 剪枝

```
public float miniMaxRecursor(State state, int depth, boolean maximize, float alpha, float beta) {
    // Has this state already been computed?
    if (computedStates.containsKey(state))
                // Return the stored result
                return computedStates.get(state);
    // Is this state done?
```

```
    if (state.getStatus() != Status.Ongoing)
                // Store and return
                return finalize(state, state.heuristic());
    // Have we reached the end of the line?
    if (depth == this.depth)
                //Return the heuristic value
                return state.heuristic();

    // If not, recurse further. Identify the best actions to take.
    float value = maximize ? Float.NEGATIVE_INFINITY : Float.POSITIVE_INFINITY;
    int flag = maximize ? 1 : -1;
    List<Action> test = state.getActions();
    for (Action action : test) {
        // Check it. Is it better? If so, keep it.
        try {
            State childState = action.applyTo(state);
            float newValue = this.miniMaxRecursor(childState, depth + 1, !maximize, alpha, beta);
            //Record the best value
            if (flag * newValue > flag * value) {
                value = newValue;
                //alpha-beta cut
                if (maximize && value > alpha) {
                    if (value > beta)
                        return finalize(state, value);
                    alpha = value;
                } else if (!maximize && value < beta) {
                    if (value < alpha)
                        return finalize(state, value);
                    beta = value;
                }
            }
        } catch (InvalidActionException e) {
            //Should not go here
            throw new RuntimeException("Invalid action!");
        }
    }
    // Store so we don't have to compute it again.
    return finalize(state, value);
}
```

在 `Othello` 中递归深度为2时，加入 $\alpha - \beta$ 剪枝后速度几乎毫无变化。 逐渐加深深度，由于深度不大，所以效果依旧不太明显。

最后将 `Othello` 中递归深度由2改为8

- 在原有未剪枝的版本，前两步还在1s左右，到第三步就默默的不动了。
- 在加入 $\alpha - \beta$ 剪枝后，每一步反应时间大约在1s左右。

可见加入了 $\alpha - \beta$ 剪枝后速度大幅度提升。

Task 3

先简单介绍一下框架基本内容

变量名	解释
dimension	棋局大小8*8
move	轮到哪个角色下棋
Board	分为h/v/d1/d2，分别是水平，竖直与两个对角线，在代码中经常分别对应于状态0,1,2,3.
getLines	得到四个方向的棋子,按照上面的顺序存在lines[]中

原有 `heuristic` 函数

```
public float heuristic() {
    Status s = this.getStatus();
    int winconstant = 0;
    switch (s) {
    case PlayerOneWon:
        winconstant = 5000;
        break;
    case PlayerTwoWon:
        winconstant = -5000;
        break;
    default:
        winconstant = 0;
        break;
    }
    return this.pieceDifferential() +
        8 * this.moveDifferential() +
        300 * this.cornerDifferential() +
        1 * this.stabilityDifferential() +
        winconstant;
}
```

其中 `winconstant` 是得分，在这里以玩家A举例，B同理。

- 如果A能赢则给5000分的分值
- 已有多少个棋子，分值1分
- 有多少个可动点，分值8分
- 有多少个顶角点，分值300分（顶角点绝对不会被翻转，而且变相相当于两边与一个对角线钦定为你的颜色）
- 有多少个可翻转棋子，分值1分

其中，他将棋盘分为水平，竖直与两个对角线，一个四个方向计分并累加。

优化后的 `heuristic` 函数

```
public float heuristic() {
    Status s = this.getStatus();
    int winconstant = 0;
    switch (s) {
    case PlayerOneWon:
        winconstant = 5000;
        break;
    case PlayerTwoWon:
        winconstant = -5000;
        break;
    default:
        winconstant = 0;
        break;
    }
    return this.pieceDifferential() +
        8 * this.moveDifferential() +
        300 * this.cornerDifferential() +
        150 * this.fixDifferential() +
        1 * this.stabilityDifferential() -
        50 * this.nextCornerDifferential() +
        20 * this.edgeDifferential() +
        20 * this.centerDifferential() +
        winconstant;
}
```

主要的改动在于我添加了新的考虑因素，简单介绍一下

`nextCornerDifferential()`

这个函数主要考虑了22点，22点的定义是指

.	.	.	.	.	.	.	.
.	x	.	.	.	.	x	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	x	.	.	.	.	x	.
.	.	.	.	.	.	.	.

图中的四个一定要慎重考虑，尽量逼迫对方棋子下在这里， 这样的话就可以将棋下在4个corner处，从而相当于霸占了两个边加一个对角线。 所以我将这四个点给予负分50的权重。

edgeDifferential()

这个函数用来计算四条边的棋子数量，因为中间的棋子贴边，相比之下他的比分要高于其他位置，

x	**x**	**x**	**x**	**x**	**x**	**x**	**x**
x	.	.	.	.	.	.	x
x	.	.	.	.	.	.	x
x	.	.	.	.	.	.	x
x	.	.	.	.	.	.	x
x	.	.	.	.	.	.	x
x	.	.	.	.	.	.	x
x	x	x	x	x	x	x	x

我将边上的棋子，赋予20的权重

centerDifferential

对中间的棋子赋予20的权重，因为他们可以分割战场

.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	x	x	.	.	.
.	.	.	x	x	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.

当占据了中间的位置时，可以更方便的打乱对面密集的棋局。

fixDifferential

从四个顶角开始，递归的计算哪些点同顶角一样不可能被翻转了。

x	→	.	.	.	.	←	**x**

x	→	**.**	**.**	**.**	**.**	←	**x**
↓	.	.	.	.	.	.	↓
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
↑	.	.	.	.	.	.	↑
x	→	.	.	.	.	←	x

由于这些棋子已经确定了他们的颜色不会被改变，所以给与150的权重，因为他们相比与corner战略性小一点

Task 4

MTD（f）通过仅执行零窗口 α - β 搜索来获得其效率，并具有"良好"的界限（变量 β ）。 在NegaScout中，使用宽搜索窗口调用搜索，就像 `AlphaBeta（root，-INFINITY，+ INFINITY，depth）` 一样，所以返回值在一次调用中位于 α 和 β 的值之间。 在MTD（f）中，AlphaBeta失败的高或低，分别返回minimax值的下界或上界。 零窗口调用会导致更多的截断，但返回的信息更少 - 仅限于最小值。 为了找到极小极大值，MTD（f）多次调用 `AlphaBeta` ，收敛它并最终找到确切的值。 转置表存储和检索存储器中树的先前搜索的部分以减少重新探索搜索树的部分的开销。

其中， `Map<State，SearchNode> transpositionTable` 作为一个转置表，减少了多次搜索对于重复情况的开销。

下面给出 `MTDF` 的伪代码：

```
function MTDF(root, f, d)
    g := f
    upperBound := +∞
    lowerBound := -∞
    while lowerBound < upperBound
        β := max(g, lowerBound+1)
        g := AlphaBetaWithMemory(root, β-1, β, d)
        if g < β then
            upperBound := g
        else
            lowerBound := g
    return g
```

其中f为猜测的值，为动作action a的分值，最快的算法收敛，第一次通话可能为0。

d为深度，迭代加深深度优先搜索可以通过多次调用 `MTDF()` 并增加d来完成，并提供f中最好的先前结果