人工智能导论作业 2 报告

李佳鑫（211220006、211220006@smail.nju.edu.cn）

(南京大学 计算机科学与技术系，南京 210093)

1 MinMaxDecider 理解介绍

```
// Are we maximizing or minimizing?
private boolean maximize;
// The depth to which we should analyze the search space
private int depth;
// HashMap to avoid recalculating States
private Map<State, Float> computedStates;
// Used to generate a graph of the search space for each turn in SVG format
private static final boolean DEBUG = true;
```

成员对象介绍

maximize：为 true 为 max，false 为 min

depth：搜索深度限制

computedStates：该状态是否已经出现过，如果已经出现并计算过启发得分那么就可以免于继续搜索直接返回

```
public MiniMaxDecider(boolean maximize, int depth) {
    this.maximize = maximize;
    this.depth = depth;
    computedStates = new HashMap<State, Float>();
}
```

构造函数

```java
public Action decide(State state) {
    // Choose randomly between equally good options
    float value = maximize ? Float.NEGATIVE_INFINITY : Float.POSITIVE_INFINITY;
    List<Action> bestActions = new ArrayList<Action>();
    // Iterate!
    int flag = maximize ? 1 : -1;
    for (Action action : state.getActions()) {
        try {
            // Algorithm!
            State newState = action.applyTo(state);
            float newValue = this.miniMaxRecursor(newState, depth:1, !this.maximize);
            // Better candidates?
            if (flag * newValue > flag * value) {
                value = newValue;
                bestActions.clear();
            }
            // Add it to the list of candidates?
            if (flag * newValue >= flag * value) bestActions.add(action);
        } catch (InvalidActionException e) {
            throw new RuntimeException(message:"Invalid action!");
        }
    }
    // If there are more than one best actions, pick one of the best randomly
    Collections.shuffle(bestActions);
    return bestActions.get(index:0);
}
```

decide 函数介绍：

1. 首先由 maximize 确定 value 的初始值便于后续的比较和更新。flag 变量同理
2. 初始化 bestActions 列表用于 bestActions 的选择。
3. 遍历可选择的行动，对于每个行动通过 miniMaxRecursor
4. 获取该行动的得分，并根据该节点是 max 还是 min 更新最大值和最小值，如果得到了更大（小）的值则更新 bestActions，如果得分相同则直接加入 bestActions
5. 随机选取一个 bestAction 返回

```java
public float miniMaxRecursor(State state, int depth, boolean maximize) {
    // 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);
            //Record the best value
                            if (flag * newValue > flag * value)
                                value = newValue;
        } catch (InvalidActionException e) {
                            //Should not go here
            throw new RuntimeException(message:"Invalid action!");
        }
    }
    // Store so we don't have to compute it again.
    return finalize(state, value);
}
```

miniMaxRecursor 函数介绍：

1. 如果当前状态已经出现过那么可直接返回该节点的得分
2. 如果该节点已经游戏结束或者达到深度限制，那么可以直接返回该节点的启发得分。
3. 剩余函数和 decide 类似：
    a) 首先由 maximize 确定 value 的初始值便于后续的比较和更新。flag 变量同理
    b) 初始化 bestActions 列表用于 bestActions 的选择。
    c) 遍历可选择的行动，对于每个行动通过 miniMaxRecursor 获取递归该行动的得分，并根据该节点是 max 还是 min 更新最大值和最小值。
    d) 返回该节点的得分

2 AlphaBeta 剪枝

1. 设计思路，剪枝需要将上层已经得到的值向下传递，那么在函数多加两个 alpha，beta 参数，用于搜索时比较，如果满足剪枝条件就可以停止搜索，break 出 for 循环，同时更新 alpha，beta 的值。

2. 函数修改：decide 函数无需修改，仅需对 minMaxRecursor 进行修改即可
函数体修改，新增加 alpha，beta,用于向下一层传递本层的

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

设置剪枝条件，当本层为 minimize 时，如果新得到的值小于上层值（上层为 maximize），那么一定不会选取该层的值，则可以 break 当前循环，若本层为 maximize，如果新得到的值大于上层值（上层为 minimize）则一定不会选择该层的值，直接 break 当前循环。由此逻辑完成剪枝。

```
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;
    if(maximize){
        if(value>=beta){
            break;
        }
        if(value>alpha){
            alpha=value;
        }
    }
    else{
        if(value<=alpha){
            break;
        }
        if(value>beta){
            beta=value;
        }
    }
```

3. 速度变化，将搜索深度设置为 10，剪枝前走完第一步，程序约 56s 走出第二步，剪枝后走完第一步，程序约 26s 走出第二步，经过剪枝用时明显减少。

3 启发式函数设计

1. 出发点：游戏输赢决定了黑白棋的本质，如果局面能判定游戏输赢则该节点的评分会大幅上升，之后权重高的为玩家可占据角落棋子的个数。个人认为四个角几乎决定了整局游戏的输赢，角落的棋子不会被翻转，同时其能覆盖的地点也很广泛，之后是两边玩家可选择的移动，之后是棋子的个数的区别，这虽然决定了游戏的最终输赢，但是在棋局过程中这一指标并没有那么重要。以及一些稳定子的区别，就是那些不会再被翻转的棋子，比如和角落相邻，以及和稳定子相邻的棋子为稳定子。
2. 修改方式：既然除游戏输赢外，最重要的指标是角落个数，那么避免对方下入角落同样重要，那么星位（角落的最近对角位）C 位（角落的相邻位就是要比避免下入的位置）。

```java
private float starOrCDifferential() {
    float diff = 0;
    short[] stars = new short[4];
    short[] Cs=new short[8];
    stars[0] = getSpotOnLine(hBoard[1], (byte)1);
    stars[1] = getSpotOnLine(hBoard[1], (byte)(dimension - 2));
    stars[2] = getSpotOnLine(hBoard[dimension - 2], (byte)1);
    stars[3] = getSpotOnLine(hBoard[dimension - 2], (byte)(dimension - 2));
    for (short star : stars) if (star != 0) diff += star == 2 ? 1 : -1;
    Cs[0]=getSpotOnLine(hBoard[1], (byte)0);
    Cs[1]=getSpotOnLine(hBoard[0], (byte)1);
    Cs[2]=getSpotOnLine(hBoard[1], (byte)(dimension-1));
    Cs[3]=getSpotOnLine(hBoard[0], (byte)(dimension-2));
    Cs[4]=getSpotOnLine(hBoard[dimension-1], (byte)1);
    Cs[5]=getSpotOnLine(hBoard[dimension-2], (byte)0);
    Cs[6]=getSpotOnLine(hBoard[dimension-1], (byte)(dimension - 2));
    Cs[7]=getSpotOnLine(hBoard[dimension-2], (byte)(dimension - 1));
    for (short C : Cs) if (C != 0) diff += C == 2 ? 1 : -1;
    return diff;
}
```

```java
return this.pieceDifferential() +
    8 * this.moveDifferential() +
  400 * this.cornerDifferential() +
   10 * this.stabilityDifferential() -
   30 *this.starOrCDifferential()+
   winconstant;
```

4 MTDDecider 类

1. MTD 算法：只使用零窗口进行搜索，搜索是否存在比下界值要大的值，如果是将新返回的值设置为新下界值，否则则设置为新上界值。最后设置为一个固定值。

   成员对象：EXACT_VALUE, LOWERBOUND, UPPERBOUND：确切值，下界，上界

   SearchNode:搜索节点

   SearchStatistics：辅助节点，记录深度，用时，以及已经搜索过的节点

   成员函数：decide：记录开始事件，创建搜索状态表，开始迭代搜索

   iterative_deepening:首先获取可执行的行动，根据 USE_MTDF 选择是用 MTDF 函数还是 AlphaBetaWithMemory 函数获取 maximize。

   MTDF：循环调用 AlphaBetaWithMemory 获取下一节点的最佳估计值。而 AlphaBetaWithMemory 函数则可以看作一个加入置换表和 alphabeta 剪枝的 miniMaxRecursor 函数，该函数同样进行了状态记录，避免重复遍历，同时将 depth 进行区分，对于深度大于 4 的节点，如果则超时则直接抛出异常，否则则进行 depth 和 depth-2 的两轮搜索。否则则进行 saveAndReturnState

   saveAndReturnState：函数主要功能是根据条件修改 EntryType 的值，更新状态表

异同：核心计算 value 的值相同，利用 minmax 以及 alphabeta 剪枝获取 value 值，有状态记录，相同的启发函数。MTDDecider 在 minimax 的基础上做出了很多改进，比如采用置换表储存历史状态，采用零窗口进行搜索。