作业2: 黑白棋游戏 实验报告

吴政亿 151220129 18805156360@163.com
(南京大学 计算机科学与技术系, 南京 210093)

摘要：介绍MiniMax搜索的实现；应用AlphaBeta剪枝并比较速度；理解heuristic函数并改进；阅读并理解MTDDecider类并与MiniMaxDecider类比较异同
关键词：MiniMax搜索，AlphaBeta剪枝，heuristic函数，MTD(f)算法

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2s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
棋盘表示	8*8棋盘每个地方用2bits存储,共65536种情况,00-空,01-墙,10-P1,11-P2

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
- 已有多少个棋子的差值，分值1分
- 有多少个可动点的差值，分值8分
- 有多少个顶角点的差值，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的差值(在水平竖直与两个对角线四个方向，能翻转的棋子数的差值)，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

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)算法介绍

该算法通过使用零大小的搜索窗口多次调用 `AlphaBetaWithMemory` 来工作。通过放大 `minimax` 值来搜索。每个 `AlphaBeta` 调用返回 `minimax` 值的界限。边界存储在上边界和下边界中，形成围绕该搜索深度的真正最小值的区间。`Plus` 和 `minus INFINITY` 是叶值范围之外的值的简写。当上限和下限碰撞时，找到最小值。

`MTD(f)` 仅从零窗口 `alpha-beta` 搜索获得效率，并使用“良好”边界(变量beta)进行零窗口搜索。通常情况下，`AlphaBeta` 会在 `AlphaBeta(root，-INFINITY，+ INFINITY，depth)` 中使用广泛的搜索窗口进行调用，确保返回值位于 `alpha` 和 `beta` 的值之间。在 `MTD(f)` 中，使用了一个零大小的窗口，这样在每次调用时 `AlphaBeta` 将会失败或失败，分别返回最小值或最小值的上限或下限。零窗口调用会导致更多的截断，但返回的信息更少 - 仅限于最小最大值。然而，要找到它， `MTD(f)` 必须多次调用 `AlphaBeta` ，才能收敛它。在重复调用 `AlphaBeta` 时重新搜索部分搜索树的开销会在使用存储和检索其在内存中看到的节点的 `AlphaBeta` 版本时消失。

MTD(f)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

MTDDecider类

Name	introduction
Map<State, SearchNode> transpositionTable	转置表，减少了多次搜索对于重复情况的开销。
class ActionValuePaipublic	存储上一次迭代信息
Action iterative_deepening(State root)	限制时间内进行零窗口搜索，根据 `USE_MTDF` 选择搜索算法
int MTDF(State root, int firstGuess, int depth)	MTDF算法，迭代搜索返回当前局面分值
int AlphaBetaWithMemory(State state, int alpha, int beta, int depth, int color)	应用了置换表的 $\alpha \quad - \quad \beta$ 剪枝算法，将已搜索的节点存储在置换表中，如果深度>4,则分为depth与depth-2进行搜索

MTD与MiniMax对比

共同点

MTD是基于MiniMax的改进算法，保留了极大极小值与深度限制的部分

不同点

MTD在MiniMax的基础上，引入了置换表技术，减少了重复计算带来的开销，并且增加了AlphaBeta剪枝