α-β剪枝算法实验报告

1. 导言

在本次实验中，我们要使用α-β剪枝算法，编写一个中国象棋博弈程序，实现人机对弈。

α-β剪枝算法是一种特殊的极小极大搜索算法。

极小极大搜索策略是考虑双方对弈若干步后，从可能的步中选一步相对好的走法来走。假设 MAX 代表程序方，MIN 代表对手方，P 代表一个状态，f 是一个关于可能的状态的静态估计函数。

当轮到 MIN 走步的节点时，MAX 应考虑最坏的情况（ $f(P)$ 取极小值）；当轮到 MAX 走步的节点时，MAX 应考虑最好的情况（ $f(P)$ 取极大值）。

以上的一般性的 MAX / MIN 过程，是先生成全部的搜索树，然后再进行端点静态估计和倒推值计算，这样效率很低。而α-β剪枝算法把生成后继和倒推值估计结合起来，及时剪掉一些无用分枝，以此提高算法效率。

2. 实验过程

α-β剪枝算法流程

1. α 剪枝：若任一 MIN 层结点的 β 值不大于它任一前驱 MAX 层结点的 α 值，即 α(前驱层)>=β(后继层)，则可终止该 MIN 层中这个 MIN 结点以下的搜索过程。该 MIN 结点的最终倒推值就确定为这个 β 值。
2. β 剪枝：若任一 MAX 层结点的 α 值不小于它任一前驱 MIN 层结点的 β 值，即 α(后继层)>=β(前驱层)，则可终止该 MAX 层中这个 MAX 结点以下的搜索过程。该 MAX 结点的最终倒推值就确定为这个 α 值。
3. 对 MAX 层结点，它的 α 值随着它后继的 MIN 层结点的倒推值的增大而增大；对 MIN 层结点，它的 β 值随着它后继的 MAX 层结点的倒推值的增大而增大。
4. 对每个结点，在遍历其后继结点时，记录下与其最优的倒推值对应的后继结点。

代码架构

- package `main`
  - class `Main` ： 程序的入口类
- package `alphabeta` ： α-β剪枝算法的实现
  - interface `BaseNode` ： α-β搜索的基本结点
  - class `BaseSearch` ： α-β搜索的实现
- package `chess` ： 中国象棋博弈的实现
  - class `Utils` ： 辅助类，提供关于棋局的各种常量定义、对某颗棋子下一步可走的位置的计算
  - package `calculate` ： 中国象棋博弈的计算
    - class `SimplePiece` ： 基本的棋子数据
    - class `ChessCalculate` ： 评估棋局、检测棋局是否已决出胜负
    - class `ChessNode` ： 棋局的状态结点
  - package `ui` ： UI相关处理

- class `Controller` ：控制UI、控制人机对弈过程
- class `Piece` ：棋子的UI

棋局规则

- 囿于有限的时间和精力，本AI的棋局规则只包括基本的棋子走法，以及将帅不能对面，而不实现比较复杂的长将、长杀、长捉等规则。
- 玩家点击本方棋子后，程序会显示玩家可以着棋的地方，玩家只能在这些地方中的一个着棋。

α-β剪枝算法

α-β剪枝算法的实现由 `BaseSearch` 类完成，代码如下：

```
public class BaseSearch {
    private final static int DEPTH = 5;
    BaseNode best;

    public BaseNode search(BaseNode current) {
        best = null;
        alphabeta(current, DEPTH, Integer.MIN_VALUE, Integer.MAX_VALUE);
        //if(value == Integer.MAX_VALUE || value == Integer.MIN_VALUE) return null;
        return best;
    }

    private int alphabeta(BaseNode node, int depth, int alpha, int beta) {
        ArrayList<BaseNode> children;
        int value = node.getValue();
        if(depth == 0 || value == Integer.MAX_VALUE || value == Integer.MIN_VALUE) return
value;

        if(node.isMax()) {  // 极大层
            children = node.getChildren();
            if(depth == DEPTH) {
                best = children.get(0);
            }
            for (BaseNode child: children) {
                int v = alphabeta(child, depth - 1, alpha, beta);
                if(alpha < v) {
                    alpha = v;
                    if(depth == DEPTH) {
                        best = child;
                    }
                }
                if(alpha >= beta){  // beta剪枝
                    break;
                }
            }
            return alpha;
        }
        else {              // 极小层
            children = node.getChildren();
            for (BaseNode child: children) {
                int v = alphabeta(child, depth - 1, alpha, beta);
```

```
                if(beta > v) {
                    beta = v;
                    if(depth == DEPTH) {
                        best = child;
                    }
                }
                if(alpha >= beta){  // alpha剪枝
                    break;
                }
            }
            return beta;
        }
    }
}
```

中国象棋博弈

棋子

SimplePiece 表示最基本的棋子，包括棋子类型、颜色、列、行。

有虚棋和实棋，实棋即可以被操作的实际的棋子，虚棋即棋盘上看不见的棋子，占据位置。

部分代码如下：

```
public class SimplePiece {
    private int type;
    private int color;
    private int col;
    private int row;

    // 棋子被吃掉，成为虚棋
    public void eaten() {
        this.type = TYPE_NONE;
        this.color = COLOR_NONE;
    }

    // 棋子移动
    public void moveTo(int col, int row) {
        this.col = col;
        this.row = row;
    }
}
```

棋局

每一步棋都生成一个新的棋局，棋局也是搜索过程中的状态结点。

棋局状态结点的关键代码如下：

```
public class ChessNode implements BaseNode {    // 棋局的状态结点
    private SimplePiece[] redPieces;
    private SimplePiece[] blackPieces;
```

```
    private SimplePiece[][] piecesAt;
    private boolean isPlayer;               // 当前下棋方，true-玩家，false-程序
    private int value;                      // 棋局的评估值
    private SimplePiece lastFrom = null;
    private SimplePiece lastTo = null;      // 生成当前棋局的前一步棋子的移动

    // ...

    // 把from位置的棋子移动到move位置
    public void move(SimplePiece from, SimplePiece to) {
        int oldc = from.getCol();
        int oldr = from.getRow();
        int newc = to.getCol();
        int newr = to.getRow();

        SimplePiece eater = piecesAt[oldc - 1][oldr - 1];
        SimplePiece food = piecesAt[newc - 1][newr - 1];
        piecesAt[newc - 1][newr - 1] = eater;
        piecesAt[oldc - 1][oldr - 1] = food;
        eater.moveTo(newc, newr);
        food.moveTo(oldc, oldr);
        food.eaten();

        isPlayer = !isPlayer;
    }

    @Override
    public ArrayList<BaseNode> getChildren() {      // 获取根据棋子的基本下法生成的所有可能的后继棋局
        SimplePiece[] pieces = isPlayer ? redPieces : blackPieces;
        ArrayList<BaseNode> children = new ArrayList<>();
        for(SimplePiece from : pieces) {
            if(from.getType() == Utils.TYPE_NONE) continue;
            ArrayList<SimplePiece> arrivableList = Utils.getArrivable(piecesAt, from);
            for (SimplePiece to: arrivableList) {
                ChessNode child = new ChessNode(this, from, to);
                children.add(child);
            }
        }
        return children;
    }
}
```

棋局评估

不同的棋子有不同的棋力，采取简单的棋力相加的结果作为一方的评估值。

```
        int v = 0;
        for(SimplePiece p : redPieces) {
            switch (p.getType()) {
                case Utils.TYPE_GENERAL :
                    v += 10000;
                    break;
                case Utils.TYPE_ADVISOR :
```

```
                    v += 150;
                    break;
                case Utils.TYPE_ELEPHANT :
                    v += 150;
                    break;
                case Utils.TYPE_HORSE :
                    v += 320;
                    break;
                case Utils.TYPE_CHARIOT :
                    v += 700;
                    break;
                case Utils.TYPE_CANNON :
                    v += 300;
                    break;
                case Utils.TYPE_SOLDIER :
                    v += 100;
                    break;
            }
        }
        redValue = v;
```

黑方是程序方，所以棋局评估值 = 黑方的评估值 - 红方的评估值。

```
        value = blackValue - redValue;
```

对棋局的评估在类 `ChessCalculate` 中实现。

人机对弈

部分关键代码如下：

```
public class Controller {
    // ...

    public void onClick(Piece piece) {

        // ...
        else if(redArrivable.indexOf(piece) != -1) {    // 玩家移动棋子到可以着棋的地方
            humanTurn(piece);
            computerTurn();     // 人机对弈
        }
    }

    // ...

    private void computerTurn() {   // 轮到电脑下棋
        human.setVisible(false);

        Thread thread = new Thread(()->{

            // 判断是否结束
            int result = ChessCalculate.getResult(current);
            // ...
```

```
            // 开始计算下一步
            ChessNode best = (ChessNode) new BaseSearch().search(current);  // 计算结果
            // ...

                current.move(best.getFrom(), best.getTo());     // 电脑走棋

                // ...

                    // 判断是否结束
                    int result2 = ChessCalculate.getResult(current);
                    // ...

                    // 轮到玩家下棋
                    // ...
        });

        thread.start();
    }
}
```

3. 结果分析

运行

```
java -jar cnchess.jar
```

结果

1. α-β搜索深度设为4和设为5时，对弈刚开始时的电脑反应速度明显不一样。深度为5时，电脑要经过大量计算才会得出下一步，所以看起来很慢。但为了更好的计算，我还是将深度设为5。
2. 由于未实现长将、长捉等规则，所以到最后，可能会出现重复走步，而迟迟不能决胜负。
3. 总的来说，在有限的规则下，该中国象棋AI表现尚可。至少下棋水平很低的我是打不过它的。

4. 结论

1. 可以看出，α-β剪枝算法简化了极小极大搜索的过程。
2. 可以推想，只要时间和运行空间足够，搜索深度越大、计算机计算出的选择越优。但是在实践中，深度每增大一层，搜索空间的大小就可能暴涨几个数量级，搜索时间也相应地增长。所以选择恰当的深度很有必要。
3. 由于包含了对局面的评估，极小极大搜索算法的效果也在相当程度上依赖于评估函数。囿于时间和精力，我只采取了简单地将棋力相加的评估方法。根据文献，在更进一步的实践中，除了由棋子类型决定的棋力，还可以进一步定义棋子位置的被攻击值、被保护值、灵活性等等。象棋局面的模型和评估函数是有很大的研究空间的。

主要参考文献

[1]黎利辉.基于Alpha-Beta剪枝法的中国象棋博弈系统研究[J].福建电脑,2014,30(03):29-30+103.

[2]朱福喜.人工智能基础教程.