五子棋游戏设计报告

面向对象程序设计

柳昊天 3150105318
任意 3150103692
赖杰文 3150105192
吴越 3150101992

浙江大学

2017 年 6 月 23 日

目录

插图

1 总概述

1. 总功能：通过图形界面，用户可以通过本游戏程序进行五子棋游戏的双人游戏或者人机对战，游戏程序可以正确判断出游戏的胜负
2. 开发工具为 C++ （语言标准为 C++11）语言，使用了 QT 库以实现图形和跨平台
3. 总体的设计思路：

   总体的架构采用了 MVC（Model-View—Controller）模式，Model 为 ChessBoard 类，View 为 MainWindow 类，Controller 为 ChessController 类。
4. 命名规则统一使用驼峰命名法，缩进为 4 空格，中文部分编码为 UTF-8

2 模块功能

2.1 MainWindow

- `MainWindow` 类是 View 中负责用户交互的部分。
- 设置游戏模式
- 监听点击事件，并将事件转换为棋盘中的逻辑位置
- 和 ChessController 交互
- 提供菜单和开始游戏的选项

2.2 ChessBoard

- 记录当前棋盘的状态
- 可以增添或者删除棋子
- 判断一个即将添加进棋盘的棋子是否越界或者重复
- 对于 `VirtualBoard` 子类，还需提供整个棋盘的估值函数，保存 AI 预测的下法
- 对于 `RealBoard` 子类，还需绘制整个棋盘内的每个棋子

2.3 ChessController

- 绑定 RealBoard 和 VirtualBoard 实例
- 管理游戏的状态，开始和结束游戏等
- 判断游戏的输赢
- 接受转换过的点击时间，添加棋子进棋盘内
- 提供 AI 的下一步走法

2.4 AI

- 根据当前棋盘的形势给出下一步的最优解

3 类之间的关系和模块间接口

3.1 MainWindows

```
// 绑定 ChessConroller 实例
public:
void chessInstance(ChessController *cc);
```

3.2 ChessController

```
// ChessContrller 的构造器，通过绑定一个 RealBoard
    实例来控制棋子的显示，并且默认将 virtualBoard 实例用 WHITE
    初始化（也就是说AI是白子后手）。
ChessController(RealBoard &cb) : cb(cb),
    curChessBoard(cb.getChessBoard()),virtualBoard(WHITE) { }

// 初始化游戏化的状态，重置virtualBoard，使程序进入游戏状态。
// 参数决定游戏是以人机模式还是双人对战模式运行
void startGame(bool isAIPlayer);

// 在AI模式下，提供AI算出来的下一步
ChessItem AIgo();
```

3.3 AI

```
// 返回AI算出的最优解
ChessItem AI::getBestItem(const VirtualBoard &board)

// AI算法的核心部分，通过negaMax算法算出一方的分数
int negaMax(const VirtualBoard &thisBoard, int deep, int alpha, int beta,
    ItemType itemType, vector<ChessItem> &thisBestItems,vector<ChessItem>&
    steps);
```

3.4 ChessBoard

3.4.1 Public 成员

```
// 默认构造器，生成一个空棋盘
ChessBoard();

// 复制构造器
ChessBoard(const ChessBoard &board);

// 重载+=操作符，用于在棋盘上添加一个棋子
ChessBoard &operator+=(const ChessItem &next);

// 重载-=操作符，用于在棋盘上移除一个棋子
ChessBoard &operator-=(const ChessItem &next);

// 用于判断棋子是否越出棋盘的界或者出现在已被占用的地方
bool isBlock(const ChessItem point) const;

// 判断棋盘的这个位置是否是空的
bool isEmpty(const ChessItem point) const;

bool isItem(const ChessItem point) const;

// 虚函数，清空整个棋盘
virtual void clear();
```

3.4.2 Protected 成员

```
// 在棋盘的对应位置放置棋子（不做任何检查）
void updateChess(int x, int y, ItemType val);

// +=操作符的具体实现（检查棋盘的位置是否为空）
virtual void set(const ChessItem &next);

// -=操作符的具体实现（不做任何检查）
void remove(const ChessItem &next);
```

3.5 VirtualBoard: ChessBoard

```
// 继承自ChessBoard的虚方法，不仅调用父方法，还清空了AI分数的估值数据
void VirtualBoard::clear()

// 继承自ChessBoard的虚方法，调用父方法并设置分数信息以供估值函数使用
void set(const ChessItem &next);
```

```
6
7  // 估值函数，估算这个棋子下之后双方的得分（形势）的值
8  int evalGlobalScore(ItemType itemType) const;
```

3.5.1 Private 成员

```
1  // 在一个方向（上下左右对角线）上进行估值
2  int evalScoreInOneDir(ChessItem item, int updateDir);
3
4  // 按照棋子的连续情况进行分类估值，例如是0000是连续四个白棋，估值为
       FOUR（1000000），若00E0（E代表空），则估值为 THREE （1000）
5  int calcTypeScore(int totalItems, int firstEmpty, int blockType,bool
       isEmpty);
```

3.6 RealBoard : public QWidget, public ChessBoard

因为这是个控制棋盘显示的控件，所以还继承自 QWidget

```
1  // 开始游戏状态
2  void play(bool isAIPlayer);
3
4  // 供 ChessController 调用，结束游戏
5  void win(int winner);
6
7  // 在画布上绘制棋子
8  void paintEvent(QPaintEvent *e);
```

3.7 数据结构

1. ItemType 枚举类型，用于表示棋盘上每个位置的状态是黑子还是白子或者是空。
2. TypeScore 枚举类型，用于定义每种棋型对应的分数。
3. vector< vector<T> > 类似于一个二维数组，用于存储 15 * 15 棋盘中的信息，例如棋子还有各个位置的估值分数。

4 设计思路

4.1 UI

棋盘数据使用了 vector<vector<ItemType> > 进行存储，而在 VirtualBoard 内，用 vector<vector<Score> > 来估值。

棋盘实际上是一个背景贴图，路径在 /resources/chessBoard.png 下，棋盘格与棋子通过 QPainter 进行实时绘制。

同时，MainWindow 还会监听鼠标的点击释放事件，并且经过换算后转换为棋盘上的具体位置。但是棋盘中的棋子并不在 MainWindow 中绘制。

4.2 分模块

1. ChessBoard 是 Model 中负责记录当前棋盘数据的类。在内部使用了 vector<vector<ItemType> >（类似于一个二维数组）来表示棋盘里的黑白棋子和空位。

   ChessBoard 有两个子类，分别为 RealBoard 和 VirtualBorad, RealBoard 是 ChessController 中持有的实际棋盘，负责绘制棋盘中的棋子以及显示当前的游戏状态。VirtualBoard 则是提供给 AI 模块用来预先演算 5 - 6 步的棋盘以及供 AI 估算当前局面的双方分数。

2. ChessController 扮演了 Controller 角色，它持有了两个棋盘的私有成员变量：RealBoard 和 VirtualBoard。ChessController 负责控制游戏的状态，包括判断游戏是否到达终止条件（即有一方获得胜利）以及确定当前下的棋子的颜色等。

3. AI 模块是本游戏中最复杂的模块，算法上使用了 negaMax 和估值函数（位于 VirtualBoard 中）以及 alpha-beta 剪枝。

   negaMax 是一种 minMax 算法在零和游戏中的变种。由于零和游戏的性质，对一方有利的一步会对另一方不利。negaMax 相对 minMax 的改进在于，剪枝和搜寻对于玩家和 AI 双方都是可用的，所以一个函数可以既用来预测玩家的下一步，也可以运算 AI 的下一步。估值函数由于和当前棋盘的局面有关，所以定义为 VirtualBoard 的成员函数。

   alpha-beta 剪枝可以通过剪去比已经搜索过的节点更坏的节点来减少 DFS 搜索树时节点的个数，这个剪枝在 AI 预测玩家的策略时也会用上。在当前的设置中，negaMax 的搜索深度定义在 config.h 中（DEEP = 4），即 AI 会预测 DEEP 步（玩家和 AI 各 $DEEP/2$ 步）。

4.3 性能

1. AI 模块在运算时较为耗时，为了不阻塞 UI 的绘制工作，在每次执行 nagaMax 的末尾，都会调用一次 QCoreApplication::processEvents() 来执行当前正在等待的事件。

2. 为了保证 AI 的性能，我们将搜索的深度定为 4 。

4.4 算法（AI 伪代码）

```
1 negaMax(board, depth, alpha, beta) {
2     score ← board.evalation();
3     if (depth <= 0 || score > threshold) {
4         //达到了搜索深度或者阈值（胜率很高）
5         return score;
6     }
```

```
7       best ← MIN;
8
9       for (candidate: candidates) {
10          board += candidates;
11          // this is why it is called negaMax
12          // dfs procedure
13          nextScore ← negaMax(board, depth − 1, − beta, −max(alpha, best));
14          board ← board − cadidates;
15          if (nextScore >= best) {
16              best ← nextScore;
17              clear resultList ;
18              add candidate to resultList ;
19          }
20          ...
21          some special cases
22          ...
23          // alpha−beta pruning
24          if (best >= beta) {
25              break;
26          }
27      }
28  }
```

5 整体架构

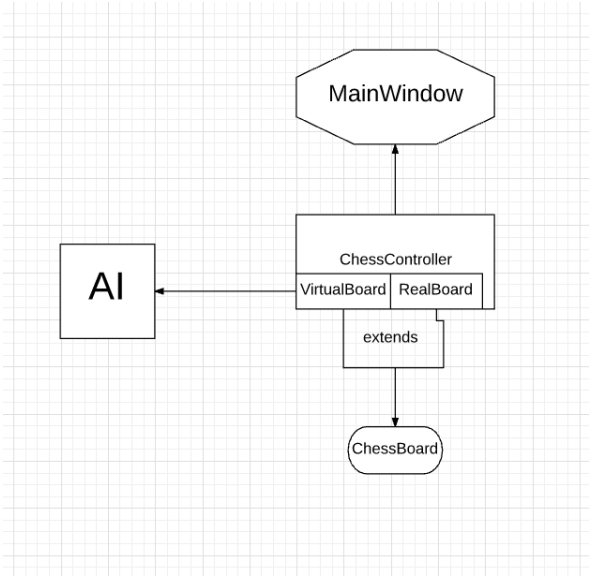

图 1: 模块间联系图

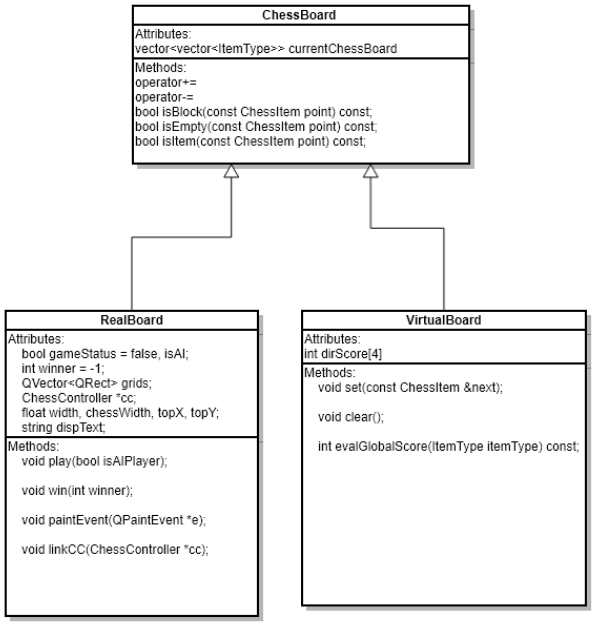

图 2: 类间继承关系

1. 整体架构采用 MVC 模式，Model 类为 ChessBoard，VirtualBoard；View 类为 MainWindow 和 RealBoard；Controller 类为 ChessController 和 RealBoard。此处的 RealBoard 较为特殊，身兼 Model 和 View 类两职，因为它存着实际显示的棋子布局，所以棋子的绘制也交由它来处理

2. MVC 是一种较为常见的软件架构，但是由于 project 的规模较小和功能并不复杂，所以各个部件的还是有相当的耦合。

6 测试结果

6.1 双人对战（玩家一先手执黑）

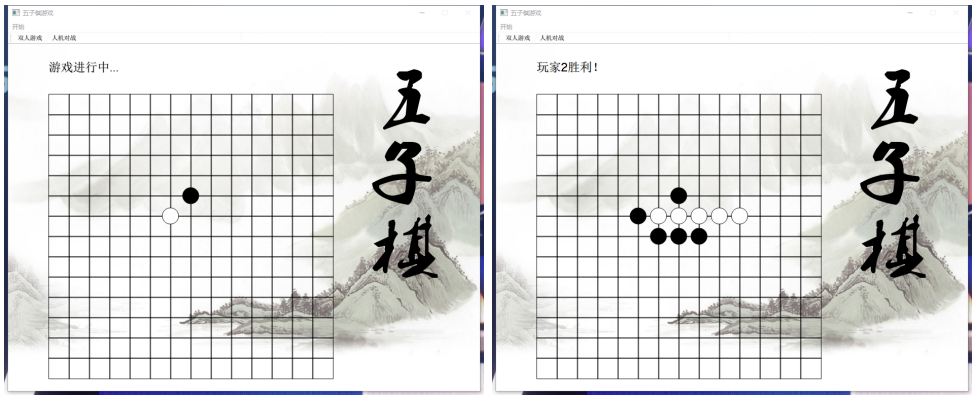

图 3: 玩家 2 获胜，游戏正确判断正负并终止游戏

6.2 人机对战（玩家先手执黑）

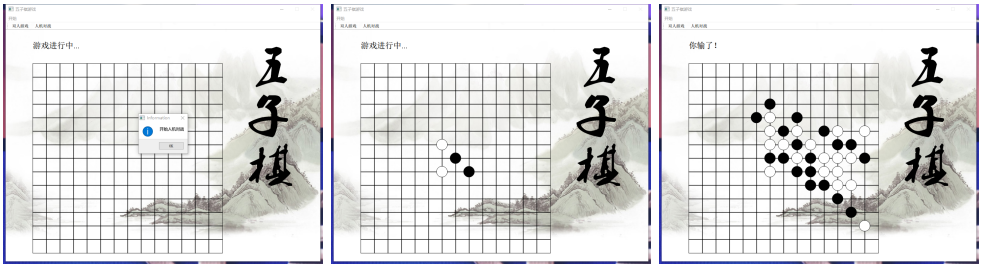

图 4: AI 获胜，游戏正常结束，AI 的运算速度很快

7 组员分工

- 柳昊天：OOP 接口与整体架构设计、编写，QT 界面与交互；
- 任意：五子棋 AI 部分，架构设计、调整；
- 赖杰文：五子棋 Controller，架构设计，测试，报告撰写；
- 吴越：五子棋 Model，报告撰写。