# *...学院*

课程

大作业报告

课 程：

姓 名：

学 号：

同组姓名：

专业班级：

指导教师：

设计时间： 2021/5/10

评阅意见

1. **需求分析**

五子棋是我国古代传统的黑白棋种之一，又称作连珠棋。五子棋游戏首先需要棋盘，并绘制棋子，人机对战还要为计算机设置一定的算法，以使其能够自动判断落棋的位置，此外，还需要有一定的判断系统来判定胜负，还有悔棋功能。综上，五子棋人机对战游戏需要提供以下功能：

1)使用图形界面，绘制棋盘，并能够提供虚拟棋盘来作为计算机运算的依据。

2)判断玩家的落子位置，并相应的画出对应颜色的棋子，判断落子位置时误差要很小。另外，

需要记录玩家的落子情况。

3)通过运算判定电脑的落子位置，如防范玩家连成五子，或进攻使自己连成五子取得胜利，

并相应的显示对应颜色的棋子。另外，需要记录电脑的落子情况。

4)根据规则判断出胜负，先连成五子者获得胜利，并显示出胜利的一方。

1. **APP项目设计**

2.1 系统功能设计

五子棋人机对战游戏包括四个方面的功能，分别是绘制棋盘和棋子等图形化显示功能，获取玩家系统功能设计落子功能，计算并判断得到电脑落子位置的功能以及判断胜负的功能。

具体设计

1）棋盘：

棋盘采用14\*14的矩形棋盘，即应用14\*14的二位数组fivemap[14][14]，用来保存当前棋盘的落子资料组，其中对每个成员来说，0表示无子，1表示落1号玩家棋子，2表示落2号玩家棋子。

2）简单说明：

胜利显示你赢了，是否重新开局，输了显示你输了是否重新开局。

3）棋盘清空：

重新开局以及开局的时候棋盘上无任何棋子。

4）判断棋子下的位置及胜利情况

既要识别屏幕被触屏的x,y位置，并计算出属于那个格子内并绘制棋子，并判断是否五子连成。

5）电脑对应的算法

电脑要通过对手下的棋子计算出此时下载的最好的位置。

2.2 系统功能简介

五子棋的人机对战，通过人下棋，电脑可自动通过算法运算合适的位置进行人机对战。

1. **APP项目实现**

3.1 APP界面设计

简单的棋盘设计，利用比较偏国风的图片作为背景。

3.2 APP功能实现关键代码

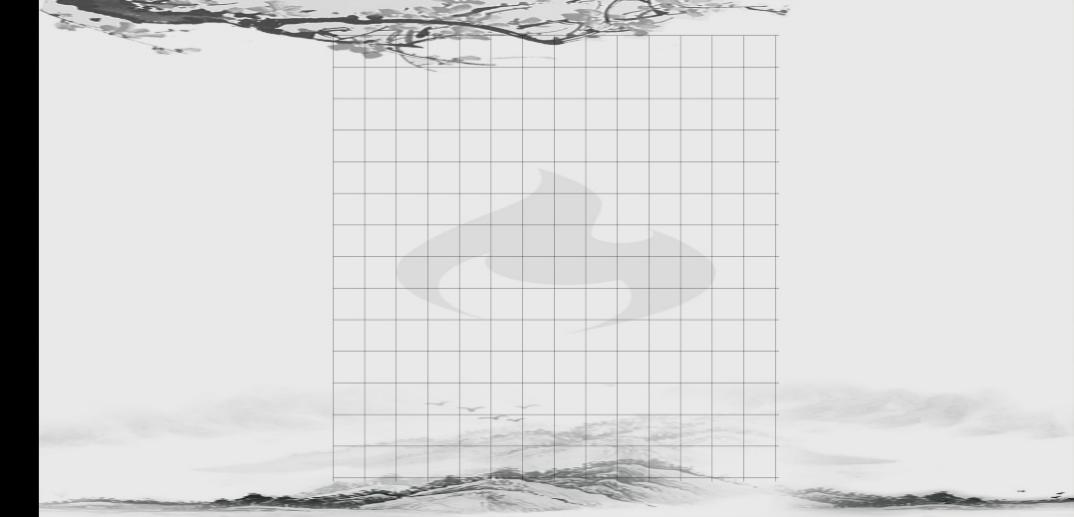
电脑下棋的核心算法：

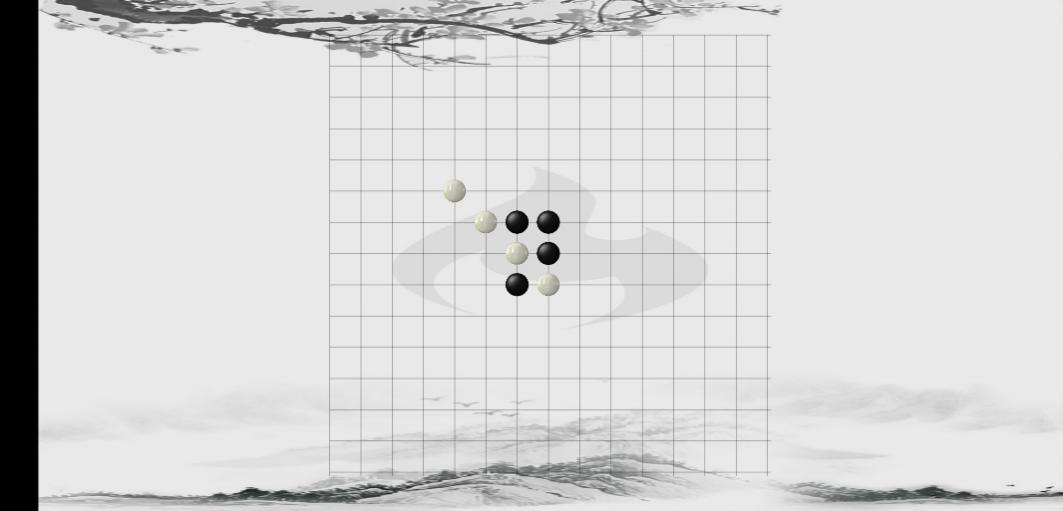
|  |
| --- |
| public class BaseComputerAi extends BasePlayer {  // 四个方向，横- 、纵| 、正斜/ 、反斜\  private static final int HENG = 0;  private static final int ZHONG = 1;  private static final int ZHENG\_XIE = 2;  private static final int FAN\_XIE = 3;  //往前往后  private static final boolean FORWARD = true;  private static final boolean BACKWARD = false;  //标示分析结果当前点位是两头通（ALIVE）还是只有一头通（HALF\_ALIVE），封死的棋子分析过程自动屏蔽，不作为待选棋子  private static final int ALIVE = 1;  private static final int HALF\_ALIVE = 0;  //private static final int DEAD = -1;  //计算范围，太大的范围会有性能问题  private class CalcuteRange {  int xStart, yStart, xStop, yStop;  private CalcuteRange(int xStart, int yStart, int xStop, int yStop) {  this.xStart = xStart;  this.yStart = yStart;  this.xStop = xStop;  this.yStop = yStop;  }  }  //限定电脑计算范围，如果整个棋盘计算，性能太差，目前是根据所有已下的棋子的边界值加RANGE\_STEP值形成，目前为1  private static final int RANGE\_STEP = 1;  CalcuteRange currentRange = new CalcuteRange(0, 0, 0, 0);  private void initRange(List<Point> comuters, List<Point> humans) {  currentRange.xStart = humans.get(0).getX() - RANGE\_STEP;  currentRange.yStart = humans.get(0).getY() - RANGE\_STEP;  currentRange.xStop = humans.get(0).getX() + RANGE\_STEP;  currentRange.yStop = humans.get(0).getY() + RANGE\_STEP;  for (Point point : humans) {  if (point.getX() - RANGE\_STEP < currentRange.xStart) {  currentRange.xStart = point.getX() - RANGE\_STEP;  } else if (point.getX() + RANGE\_STEP > currentRange.xStop) {  currentRange.xStop = point.getX() + RANGE\_STEP;  }  if (point.getY() - RANGE\_STEP < currentRange.yStart) {  currentRange.yStart = point.getY() - RANGE\_STEP;  } else if (point.getY() + RANGE\_STEP > currentRange.yStop) {  currentRange.yStop = point.getY() + RANGE\_STEP;  }  }  for (Point point : comuters) {  if (point.getX() - RANGE\_STEP < currentRange.xStart) {  currentRange.xStart = point.getX() - RANGE\_STEP;  } else if (point.getX() + RANGE\_STEP > currentRange.xStop) {  currentRange.xStop = point.getX() + RANGE\_STEP;  }  if (point.getY() - RANGE\_STEP < currentRange.yStart) {  currentRange.yStart = point.getY() - RANGE\_STEP;  } else if (point.getY() + RANGE\_STEP > currentRange.yStop) {  currentRange.yStop = point.getY() + RANGE\_STEP;  }  }  //如果范围扩大后超过了棋盘，则等于棋盘  currentRange.xStart = currentRange.xStart < 0 ? 0 : currentRange.xStart;  currentRange.yStart = currentRange.yStart < 0 ? 0 : currentRange.yStart;  currentRange.xStop = currentRange.xStop >= maxX ? maxX - 1 : currentRange.xStop;  currentRange.yStop = currentRange.yStop >= maxY ? maxY - 1 : currentRange.yStop;  }  // 分析当前形式的入口方法，分析总共分三个步骤，第三步骤可由子类干预以作难度控制  private Point doAnalysis(List<Point> comuters, List<Point> humans) {  if (humans.size() == 1) {//第一步  return getFirstPoint(humans);  }  //初始化计算范围  initRange(comuters, humans);  //清除以前的结果  initAnalysisResults();  // 开始分析，扫描所有空白点，形成第一次分析结果  Point bestPoint = doFirstAnalysis(comuters, humans);  if (bestPoint != null) {  //System.out.println("这个棋子最重要，只能下这个棋子");  return bestPoint;  }  // 分析第一次结果，找到自己的最佳点位  bestPoint = doComputerSencondAnalysis(computerFirstResults, computerSencodResults);  if (bestPoint != null) {  //System.out.println("快要赢了，就下这个棋子");  return bestPoint;  }  computerFirstResults.clear();  System.gc();  // 分析第一次结果，找到敌人的最佳点位  bestPoint = doHumanSencondAnalysis(humanFirstResults, humanSencodResults);  if (bestPoint != null) {  //System.out.println("再不下这个棋子就输了");  return bestPoint;  }  humanFirstResults.clear();  System.gc();  //没找到绝杀点，第三次结果分析  return doThirdAnalysis();  }  private static final HashMap<Integer, Integer> fMap = new HashMap<>();  //BUG修复：当电脑是黑棋的时候，白棋下在黑棋右边一颗（即开局黑(7,7)白(7,8)）会出现bug.  //下第一步棋子，不需要复杂的计算，根据人类第一步棋子X值减1完成  private Point getFirstPoint(List<Point> humans) {  Point point = humans.get(0);  if (myPoints.isEmpty()) {  //人类已经下了一颗，我还没有下，所以人类是先手  return xm1(point);  } else if (point.x == 6 && point.y == 6) {  return new Point(6, 8);  } else if (point.x == 6 && point.y == 7) {  return new Point(6, 6);  } else if (point.x == 6 && point.y == 8) {  return new Point(6, 6);  } else if (point.x == 7 && point.y == 6) {  return new Point(8, 8);  } else if (point.x == 7 && point.y == 8) {  return new Point(6, 8);  } else if (point.x == 8 && point.y == 6) {  return new Point(6, 6);  } else if (point.x == 8 && point.y == 7) {  return new Point(6, 6);  } else if (point.x == 8 && point.y == 8) {  return new Point(6, 8);  } else if (point.x == 5 && point.y == 5) {  return new Point(6, 8);  } else {  return new Point(6, 6);  }  }  private Point xm1(Point point) {  if (point.getX() == 0 || point.getY() == 0 || point.getX() == maxX && point.getY() == maxY)  return new Point(maxX / 2, maxY / 2);  else {  return new Point(point.getX() - 1, point.getY());  }  }  // private int debugx,debugy;//用于DEBUG  // 开始分析，扫描所有空白点，形成第一次分析结果  private Point doFirstAnalysis(List<Point> comuters, List<Point> humans) {  int size = allFreePoints.size();  Point computerPoint = null;  Point humanPoint = null;  int x, y;  FirstAnalysisResult firstAnalysisResult;  for (int i = 0; i < size; i++) {  computerPoint = allFreePoints.get(i);  //先把X、Y坐标记下来，因为在分析过程中会改变原来的对象  x = computerPoint.getX();  y = computerPoint.getY();  if (x < currentRange.xStart || x > currentRange.xStop || y < currentRange.yStart || y > currentRange.yStop) {  continue;  }  // if(x==debugx && y==debugy){  // System.out.println("sssssssssssss");  // }  //尝试在此位置上下一个棋子，并分析在“横向”这个方向上我方可形成的状态，如活4，活3，半活4，活2等所有状态  firstAnalysisResult = tryAndCountResult(comuters, humans, computerPoint, HENG);  computerPoint.setX(x).setY(y);//回复点位的原值，以供下次分析  if (firstAnalysisResult != null) {//无返回结果此方向上不可能达到五个棋子，  if (firstAnalysisResult.count == 5)//等于5表示在此点上下棋子即可连成5个，胜利了，不再往下进行分析  return computerPoint;  //记录第一次分析结果  addToFirstAnalysisResult(firstAnalysisResult, computerFirstResults);  }  //在“纵向”这个方向上重复上面的步骤  firstAnalysisResult = tryAndCountResult(comuters, humans, computerPoint, ZHONG);  computerPoint.setX(x).setY(y);  if (firstAnalysisResult != null) {//死棋，不下  if (firstAnalysisResult.count == 5)  return computerPoint;  addToFirstAnalysisResult(firstAnalysisResult, computerFirstResults);  }  //正斜向  firstAnalysisResult = tryAndCountResult(comuters, humans, computerPoint, ZHENG\_XIE);  computerPoint.setX(x).setY(y);  if (firstAnalysisResult != null) {//死棋，不下  if (firstAnalysisResult.count == 5)  return computerPoint;  addToFirstAnalysisResult(firstAnalysisResult, computerFirstResults);  }  //反斜向  firstAnalysisResult = tryAndCountResult(comuters, humans, computerPoint, FAN\_XIE);  computerPoint.setX(x).setY(y);  if (firstAnalysisResult != null) {//死棋，不下  if (firstAnalysisResult.count == 5)  return computerPoint;  addToFirstAnalysisResult(firstAnalysisResult, computerFirstResults);  }  //在“横向”上分析此棋子可在敌方形成如何状态，如敌方的活3、半活4等  firstAnalysisResult = tryAndCountResult(humans, comuters, computerPoint, HENG);  computerPoint.setX(x).setY(y);  if (firstAnalysisResult != null) {//死棋，不下  if (firstAnalysisResult.count == 5)  humanPoint = computerPoint;  addToFirstAnalysisResult(firstAnalysisResult, humanFirstResults);  }  //“纵向”  firstAnalysisResult = tryAndCountResult(humans, comuters, computerPoint, ZHONG);  computerPoint.setX(x).setY(y);  if (firstAnalysisResult != null) {//死棋，不下  if (firstAnalysisResult.count == 5)  humanPoint = computerPoint;  addToFirstAnalysisResult(firstAnalysisResult, humanFirstResults);  }  //“正斜”  firstAnalysisResult = tryAndCountResult(humans, comuters, computerPoint, ZHENG\_XIE);  computerPoint.setX(x).setY(y);  if (firstAnalysisResult != null) {//死棋，不下  if (firstAnalysisResult.count == 5)  humanPoint = computerPoint;  addToFirstAnalysisResult(firstAnalysisResult, humanFirstResults);  }  //“反斜”  firstAnalysisResult = tryAndCountResult(humans, comuters, computerPoint, FAN\_XIE);  computerPoint.setX(x).setY(y);  if (firstAnalysisResult != null) {//死棋，不下  if (firstAnalysisResult.count == 5)  humanPoint = computerPoint;  addToFirstAnalysisResult(firstAnalysisResult, humanFirstResults);  }  }  //如果没有绝杀棋子，第一次分析不需要返回结果  return humanPoint;  }  //第二次分析，分析第一次形成的结果，第一次分析结果会把一步棋在四个方向上可形成的结果生成最多四个FirstAnalysisResult对象（敌我各四）  //这里要把这四个对象组合成一个SencondAnalysisResult对象，  private Point doComputerSencondAnalysis(Map<Point, List<FirstAnalysisResult>> firstResults, List<SencondAnalysisResult> sencodResults) {  List<FirstAnalysisResult> list = null;  SencondAnalysisResult sr = null;  for (Point p : firstResults.keySet()) {  sr = new SencondAnalysisResult(p);  list = firstResults.get(p);  for (FirstAnalysisResult result : list) {  if (result.count == 4) {  if (result.aliveState == ALIVE) {//经过前面的过滤，双方都排除了绝杀棋，有活4就下这一步了，再下一步就赢了  return result.point;//如果有绝杀，第一轮已返回，在此轮活4已经是好的棋子，直接返回，不再往下分析  } else {  sr.halfAlive4++;  computer4HalfAlives.add(sr);  }  } else if (result.count == 3) {  if (result.aliveState == ALIVE) {  sr.alive3++;  if (sr.alive3 == 1) {  computer3Alives.add(sr);  } else {  computerDouble3Alives.add(sr);  }  } else {  sr.halfAlive3++;  computer3HalfAlives.add(sr);  }  } else {//半活2在第一阶段已被排除，不再处理  sr.alive2++;  if (sr.alive2 == 1) {  computer2Alives.add(sr);  } else {  computerDouble2Alives.add(sr);  }  }  }  sencodResults.add(sr);  }  //没有找到活4  return null;  }  //这个方法和上面的基本一样，但为了性能，少作几次判断，将人类和电脑的分开了  private Point doHumanSencondAnalysis(Map<Point, List<FirstAnalysisResult>> firstResults, List<SencondAnalysisResult> sencodResults) {  List<FirstAnalysisResult> list = null;  SencondAnalysisResult sr = null;  for (Point p : firstResults.keySet()) {  sr = new SencondAnalysisResult(p);  list = firstResults.get(p);  for (FirstAnalysisResult result : list) {  if (result.count == 4) {  if (result.aliveState == ALIVE) {  human4Alives.add(sr);  } else {  sr.halfAlive4++;  human4HalfAlives.add(sr);  }  } else if (result.count == 3) {  if (result.aliveState == ALIVE) {  sr.alive3++;  if (sr.alive3 == 1) {  human3Alives.add(sr);  } else {  humanDouble3Alives.add(sr);  }  } else {  sr.halfAlive3++;  human3HalfAlives.add(sr);  }  } else {  sr.alive2++;  if (sr.alive2 == 1) {  human2Alives.add(sr);  } else {  humanDouble2Alives.add(sr);  }  }  }  sencodResults.add(sr);  }  //没有找到活4  return null;  }  private void sleep(int miniSecond) {  try {  Thread.sleep(miniSecond);  } catch (InterruptedException e) {  }  }  //第三次分析，双方都不可以制造活4，找双活3棋子，不行就找半活4，再不行就找单活3，双活2  private Point doThirdAnalysis() {  if (!computer4HalfAlives.isEmpty()) {  return computer4HalfAlives.get(0).point;  }  System.gc();  sleep(300);  Collections.sort(computerSencodResults);  System.gc();  //即将单活4，且我没有半活4以上的，只能堵  Point mostBest = getBestPoint(human4Alives, computerSencodResults);  if (mostBest != null)  return mostBest;  Collections.sort(humanSencodResults);  System.gc();  mostBest = getBestPoint();  if (mostBest != null)  return mostBest;  //拿出各自排第一的，谁好就下谁  return computerSencodResults.get(0).point;  }  //子类实现这个方法，并改变其顺序可以实现防守为主还是猛攻  protected Point getBestPoint() {  //即将单活4，且我没有半活4以上的，只能堵  Point mostBest = getBestPoint(computerDouble3Alives, humanSencodResults);  if (mostBest != null)  return mostBest;  mostBest = getBestPoint(computer3Alives, humanSencodResults);  if (mostBest != null)  return mostBest;  mostBest = getBestPoint(humanDouble3Alives, computerSencodResults);  if (mostBest != null)  return mostBest;  mostBest = getBestPoint(human3Alives, computerSencodResults);  if (mostBest != null)  return mostBest;  mostBest = getBestPoint(computerDouble2Alives, humanSencodResults);  if (mostBest != null)  return mostBest;  mostBest = getBestPoint(computer2Alives, humanSencodResults);  if (mostBest != null)  return mostBest;  mostBest = getBestPoint(computer3HalfAlives, humanSencodResults);  if (mostBest != null)  return mostBest;  mostBest = getBestPoint(human4HalfAlives, computerSencodResults);  if (mostBest != null)  return mostBest;  mostBest = getBestPoint(humanDouble2Alives, computerSencodResults);  if (mostBest != null)  return mostBest;  mostBest = getBestPoint(human2Alives, computerSencodResults);  if (mostBest != null)  return mostBest;  mostBest = getBestPoint(human3HalfAlives, computerSencodResults);  return mostBest;  }  //第三次分析的最后一步，第二次结果已经过排序，在此可以从前面选出最好的棋子  protected Point getBestPoint(List<SencondAnalysisResult> myBest, List<SencondAnalysisResult> yourSencodResults) {  if (!myBest.isEmpty()) {  if (myBest.size() > 1) {  for (SencondAnalysisResult your : yourSencodResults) {  if (myBest.contains(your)) {  return your.point;  }  }  return myBest.get(0).point;  } else {  return myBest.get(0).point;  }  }  return null;  }  //第一次分析结果  private final Map<Point, List<FirstAnalysisResult>> computerFirstResults = new HashMap<Point, List<FirstAnalysisResult>>();  private final Map<Point, List<FirstAnalysisResult>> humanFirstResults = new HashMap<Point, List<FirstAnalysisResult>>();  //第二次总结果  protected final List<SencondAnalysisResult> computerSencodResults = new ArrayList<SencondAnalysisResult>();  protected final List<SencondAnalysisResult> humanSencodResults = new ArrayList<SencondAnalysisResult>();  //第二次分结果，电脑  protected final List<SencondAnalysisResult> computer4HalfAlives = new ArrayList<SencondAnalysisResult>(2);  protected final List<SencondAnalysisResult> computerDouble3Alives = new ArrayList<SencondAnalysisResult>(4);  protected final List<SencondAnalysisResult> computer3Alives = new ArrayList<SencondAnalysisResult>(5);  protected final List<SencondAnalysisResult> computerDouble2Alives = new ArrayList<SencondAnalysisResult>();  protected final List<SencondAnalysisResult> computer2Alives = new ArrayList<SencondAnalysisResult>();  protected final List<SencondAnalysisResult> computer3HalfAlives = new ArrayList<SencondAnalysisResult>();  //第二次分结果，人类  protected final List<SencondAnalysisResult> human4Alives = new ArrayList<SencondAnalysisResult>(2);  protected final List<SencondAnalysisResult> human4HalfAlives = new ArrayList<SencondAnalysisResult>(5);  protected final List<SencondAnalysisResult> humanDouble3Alives = new ArrayList<SencondAnalysisResult>(2);  protected final List<SencondAnalysisResult> human3Alives = new ArrayList<SencondAnalysisResult>(10);  protected final List<SencondAnalysisResult> humanDouble2Alives = new ArrayList<SencondAnalysisResult>(3);  protected final List<SencondAnalysisResult> human2Alives = new ArrayList<SencondAnalysisResult>();  protected final List<SencondAnalysisResult> human3HalfAlives = new ArrayList<SencondAnalysisResult>();  //第一次分析前清空上一步棋子的分析结果  private void initAnalysisResults() {  computerFirstResults.clear();  humanFirstResults.clear();  //第二次总结果  computerSencodResults.clear();  humanSencodResults.clear();  //第二次分结果  computer4HalfAlives.clear();  computerDouble3Alives.clear();  computer3Alives.clear();  computerDouble2Alives.clear();  computer2Alives.clear();  computer3HalfAlives.clear();  //第二次分结果，人类  human4Alives.clear();  human4HalfAlives.clear();  humanDouble3Alives.clear();  human3Alives.clear();  humanDouble2Alives.clear();  human2Alives.clear();  human3HalfAlives.clear();  System.gc();  }  //加入到第一次分析结果中  private void addToFirstAnalysisResult(FirstAnalysisResult result, Map<Point, List<FirstAnalysisResult>> dest) {  if (dest.containsKey(result.point)) {  dest.get(result.point).add(result);  } else {  List<FirstAnalysisResult> list = new ArrayList<FirstAnalysisResult>(1);  list.add(result);  dest.put(result.point, list);  }  }  //第一次分析结果类  private class FirstAnalysisResult {  //连续数  int count;  //点位  Point point;  //方向  int direction;  //状态  int aliveState;  private FirstAnalysisResult(int count, Point point, int direction) {  this(count, point, direction, ALIVE);  }  private FirstAnalysisResult(int count, Point point, int direction, int aliveState) {  this.count = count;  this.point = point;  this.direction = direction;  this.aliveState = aliveState;  }  private FirstAnalysisResult init(Point point, int direction, int aliveState) {  this.count = 1;  this.point = point;  this.direction = direction;  this.aliveState = aliveState;  return this;  }  private FirstAnalysisResult cloneMe() {  return new FirstAnalysisResult(count, point, direction, aliveState);  }  }  //第二次分析结果类  class SencondAnalysisResult implements Comparable<SencondAnalysisResult> {  int alive4 = 0;  //活3数量  int alive3 = 0;  //半活4，一头封的  int halfAlive4 = 0;  //半活3，一头封的  int halfAlive3 = 0;  //活2数量  int alive2 = 0;  //点位  Point point;  @Override  public int hashCode() {  final int prime = 31;  int result = 1;  result = prime \* result + ((point == null) ? 0 : point.hashCode());  return result;  }  @Override  public boolean equals(Object obj) {  SencondAnalysisResult other = (SencondAnalysisResult) obj;  if (point == null) {  if (other.point != null)  return false;  } else if (!point.equals(other.point))  return false;  return true;  }  private SencondAnalysisResult(Point point) {  this.point = point;  }  //第三次分析时，对第二次分析结果进行排序，此为排序回调函数  @Override  public int compareTo(SencondAnalysisResult another) {  return compareTowResult(this, another);  }  }  //返加-1则第一个参数优先，1则第二个参数优先，0则按原来顺序  private int compareTowResult(SencondAnalysisResult oneResult, SencondAnalysisResult another) {  if (oneResult.alive4 > another.alive4) {  return -1;  }  if (oneResult.alive4 < another.alive4) {  return 1;  }  if (oneResult.halfAlive4 > another.halfAlive4) {  return -1;  }  if (oneResult.halfAlive4 < another.halfAlive4) {  return 1;  }  if (oneResult.alive3 > another.alive3) {  return -1;  }  if (oneResult.alive3 < another.alive3) {  return 1;  }  if (oneResult.alive2 > another.alive2) {  return -1;  }  if (oneResult.alive2 < another.alive2) {  return 1;  }  if (oneResult.halfAlive3 > another.halfAlive3) {  return -1;  }  if (oneResult.halfAlive3 > another.halfAlive3) {  return 1;  }  return 0;  }  //一个临时对象，供第一次分析时临时存放分析结果使用，如果分析出有活1以上（不含）的结果，则调用其cloneMe方法获得结果，否则抛弃此结果  private final FirstAnalysisResult far = new FirstAnalysisResult(1, null, HENG);  // 分析如果在当前位下一子，会形成某个方向上多少个子，参数：当前己方已下的所有点，当前要假设的点，需要判断的方向  private FirstAnalysisResult tryAndCountResult(List<Point> myPoints, List<Point> enemyPoints, Point point, int direction) {  int x = point.getX();  int y = point.getY();  FirstAnalysisResult fr = null;  int maxCountOnThisDirection = maxCountOnThisDirection(point, enemyPoints, direction, 1);  if (maxCountOnThisDirection < 5) {  //无意义的棋子  return null;//此方向不足五个空位，已排除己方已下的棋子  } else if (maxCountOnThisDirection == 5) {  //半死状态，当是一头通  fr = far.init(point, direction, HALF\_ALIVE);  } else {  //两头皆通  fr = far.init(point, direction, ALIVE);  }  //在前和后的方向上计算一次  countPoint(myPoints, enemyPoints, point.setX(x).setY(y), fr, direction, FORWARD);  countPoint(myPoints, enemyPoints, point.setX(x).setY(y), fr, direction, BACKWARD);  if (fr.count <= 1 || (fr.count == 2 && fr.aliveState == HALF\_ALIVE)) {//活1，半活2及其以下结果，抛弃  return null;  }  //返回复制的结果  return fr.cloneMe();  }  //棋子出了墙  private boolean isOutSideOfWall(Point point, int direction) {  if (direction == HENG) {  return point.getX() < 0 || point.getX() >= maxX;//最大的X和Y值均在墙外所以用等号  } else if (direction == ZHONG) {  return point.getY() < 0 || point.getY() >= maxY;  } else {//这里可能有问题  return point.getX() < 0 || point.getY() < 0 || point.getX() >= maxX || point.getY() >= maxY;  }  }  private Point pointToNext(Point point, int direction, boolean forward) {  switch (direction) {  case HENG:  if (forward)  point.x++;  else  point.x--;  break;  case ZHONG:  if (forward)  point.y++;  else  point.y--;  break;  case ZHENG\_XIE:  if (forward) {  point.x++;  point.y--;  } else {  point.x--;  point.y++;  }  break;  case FAN\_XIE:  if (forward) {  point.x++;  point.y++;  } else {  point.x--;  point.y--;  }  break;  }  return point;  }  //在某个方向（八个中的一个）可下多少棋子，这个方法是第一分析中的核心方法  private void countPoint(List<Point> myPoints, List<Point> enemyPoints, Point point, FirstAnalysisResult fr, int direction, boolean forward) {  if (myPoints.contains(pointToNext(point, direction, forward))) {  fr.count++;  if (myPoints.contains(pointToNext(point, direction, forward))) {  fr.count++;  if (myPoints.contains(pointToNext(point, direction, forward))) {  fr.count++;  if (myPoints.contains(pointToNext(point, direction, forward))) {  fr.count++;  } else if (enemyPoints.contains(point) || isOutSideOfWall(point, direction)) {  fr.aliveState = HALF\_ALIVE;  }  } else if (enemyPoints.contains(point) || isOutSideOfWall(point, direction)) {  fr.aliveState = HALF\_ALIVE;  }  } else if (enemyPoints.contains(point) || isOutSideOfWall(point, direction)) {  fr.aliveState = HALF\_ALIVE;  }  } else if (enemyPoints.contains(point) || isOutSideOfWall(point, direction)) {  fr.aliveState = HALF\_ALIVE;  }  }  //在某个方向上是否还能下到满五个棋子  private int maxCountOnThisDirection(Point point, List<Point> enemyPoints, int direction, int count) {  int x = point.getX(), y = point.getY();  switch (direction) {  //横向  case HENG:  while (!enemyPoints.contains(point.setX(point.getX() - 1)) && point.getX() >= 0 && count < 6) {  count++;  }  point.setX(x);  while (!enemyPoints.contains(point.setX(point.getX() + 1)) && point.getX() < maxX && count < 6) {  count++;  }  break;  //纵向  case ZHONG:  while (!enemyPoints.contains(point.setY(point.getY() - 1)) && point.getY() >= 0) {  count++;  }  point.setY(y);  while (!enemyPoints.contains(point.setY(point.getY() + 1)) && point.getY() < maxY && count < 6) {  count++;  }  break;  //正斜向 /  case ZHENG\_XIE:  while (!enemyPoints.contains(point.setX(point.getX() - 1).setY(point.getY() + 1)) && point.getX() >= 0 && point.getY() < maxY) {  count++;  }  point.setX(x).setY(y);  while (!enemyPoints.contains(point.setX(point.getX() + 1).setY(point.getY() - 1)) && point.getX() < maxX && point.getY() >= 0 && count < 6) {  count++;  }  break;  //反斜 /  case FAN\_XIE:  while (!enemyPoints.contains(point.setX(point.getX() - 1).setY(point.getY() - 1)) && point.getX() >= 0 && point.getY() >= 0) {  count++;  }  point.setX(x).setY(y);  while (!enemyPoints.contains(point.setX(point.getX() + 1).setY(point.getY() + 1)) && point.getX() < maxX && point.getY() < maxY && count < 6) {  count++;  }  break;  }  return count;  }  //下棋子，对外接口  @Override  public void run(List<Point> humans, Point p) {  //把人类下的最后一步棋子去除  allFreePoints.remove(humans.get(humans.size() - 1));  //电脑可以下的一步棋子  Point result = null;  try {  result = doAnalysis(myPoints, humans);  } catch (NullPointerException | IndexOutOfBoundsException e) {  //修复可能出现的罕见bug：当快下满的时候（和棋时）可能出现AI崩溃的bug，暂定的解决方式是：AI随机下子。  Random random = new Random();  int i = random.nextInt(allFreePoints.size());  result = allFreePoints.get(i);  }  //去除电脑下的棋子  allFreePoints.remove(result);  //加入到电脑棋子中，下棋了  myPoints.add(result);  } |

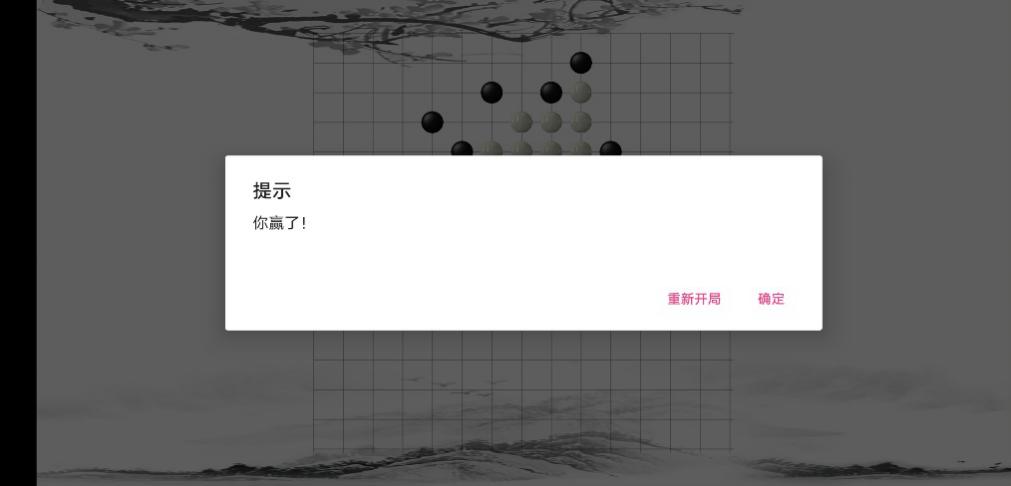
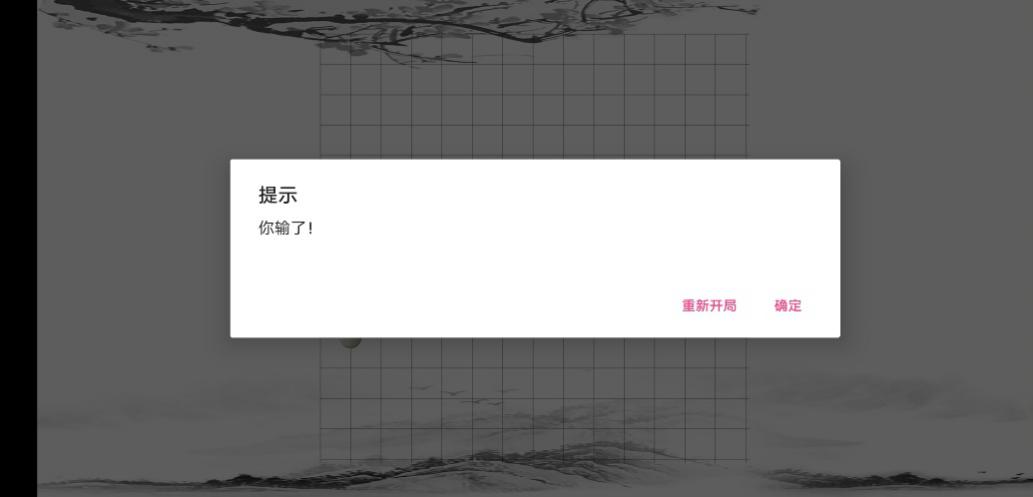
棋子及棋盘绘制的核心算法：

|  |
| --- |
| public class ChessView extends View {  private int mPanelWidth;  private static int MAX\_LINE = 15;  private float mLineHeight;  private Paint paint = new Paint();  //棋子图片  private Bitmap whiteChess;  private Bitmap blackChess;  private float pieceLineHeight = 0.75f;  private boolean isBlack = true; //玩家是否为黑棋  private boolean isWin = false;//是否已获胜  private IPlayer humanPlayer; //玩家  private IPlayer aiPlayer; //电脑  private IChessboard chessboard = new ChessBoard(MAX\_LINE);//棋盘  public ChessView(Context context, AttributeSet attrs) {  super(context, attrs);  setKeepScreenOn(true);//设置屏幕常亮  //setBackgroundColor(0x44ff0000);  Resources resources = getResources();  blackChess = BitmapFactory.decodeResource(resources, R.drawable.stone\_b1);  whiteChess = BitmapFactory.decodeResource(resources, R.drawable.stone\_w2);  initPaint();  initGame();  }  //音效  private SoundPool soundWin;  private SoundPool soundDefeat;  private SoundPool soundChess;  private void initGame() {  humanPlayer = new HumanPlayer();  humanPlayer.setChessboard(chessboard);  aiPlayer = new BaseComputerAi();  aiPlayer.setChessboard(chessboard);  humanPlayer.clear();  aiPlayer.clear();  soundWin = new SoundPool(1, AudioManager.STREAM\_SYSTEM,0);  soundDefeat = new SoundPool(1, AudioManager.STREAM\_SYSTEM,0);  soundChess = new SoundPool(1, AudioManager.STREAM\_SYSTEM,0);  soundWin.load(getContext(),R.raw.win,1);  soundDefeat.load(getContext(),R.raw.defeat,1);  soundChess.load(getContext(),R.raw.chess,1);  }  private void initPaint() {  paint.setColor(0x88000000);  paint.setAntiAlias(true);  paint.setDither(true);  paint.setStyle(Paint.Style.STROKE);  }  @Override  protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {  super.onMeasure(widthMeasureSpec, heightMeasureSpec);  int widthSize = MeasureSpec.getSize(widthMeasureSpec);  int widthModel = MeasureSpec.getMode(widthMeasureSpec);  int heightSize = MeasureSpec.getSize(heightMeasureSpec);  int heightModel = MeasureSpec.getMode(heightMeasureSpec);  int size = 0;  if (widthModel == MeasureSpec.UNSPECIFIED) {  size = heightSize;  } else if (heightModel == MeasureSpec.UNSPECIFIED) {  size = widthSize;  } else {  size = Math.min(widthSize, heightSize);  }  setMeasuredDimension(size, size);  }  @Override  protected void onSizeChanged(int w, int h, int oldw, int oldh) {  super.onSizeChanged(w, h, oldw, oldh);  mPanelWidth = w;  mLineHeight = mPanelWidth / MAX\_LINE;  int pieceWidth = (int) (mLineHeight \* pieceLineHeight);  whiteChess = Bitmap.createScaledBitmap(whiteChess, pieceWidth, pieceWidth, false);  blackChess = Bitmap.createScaledBitmap(blackChess, pieceWidth, pieceWidth, false);  }  @Override  protected void onDraw(Canvas canvas) {  super.onDraw(canvas);  drawChessBoard(canvas);  drawPieces(canvas);  }  private void drawPieces(Canvas canvas) {  for (Point point : humanPlayer.getMyPoints()) {  canvas.drawBitmap(isBlack ? blackChess : whiteChess,  (point.x + (1 - pieceLineHeight) / 2) \* mLineHeight,  (point.y + (1 - pieceLineHeight) / 2) \* mLineHeight,  null);  }  for (Point point : aiPlayer.getMyPoints()) {  canvas.drawBitmap(!isBlack ? blackChess : whiteChess,  (point.x + (1 - pieceLineHeight) / 2) \* mLineHeight,  (point.y + (1 - pieceLineHeight) / 2) \* mLineHeight,  null);  }  }  private void drawChessBoard(Canvas canvas) {  int w = mPanelWidth;  float lineHeight = mLineHeight;  for (int i = 0; i < MAX\_LINE; i++) {  float startX = lineHeight / 2;  float endX = w - lineHeight / 2;  float y = (float) ((0.5 + i) \* lineHeight);  canvas.drawLine(startX, y, endX, y, paint);  canvas.drawLine(y, startX, y, endX, paint);  }  }  @Override  public boolean onTouchEvent(MotionEvent event) {  int action = event.getAction();  if (action == MotionEvent.ACTION\_UP) {  int x = (int) (event.getX() / mLineHeight);  int y = (int) (event.getY() / mLineHeight);  if (x >= 0 && x < MAX\_LINE && y >= 0 && y < MAX\_LINE) {  onPoint(x,y);  }  return true;  } else if (action == MotionEvent.ACTION\_DOWN) {  return true;  }  return super.onTouchEvent(event);  }  private void onPoint(int x,int y){  if(isWin){  return;  }  Point point = new Point(x,y);  if(chessboard.getFreePoints().contains(point)){  humanPlayer.run(aiPlayer.getMyPoints(),point);  invalidate();  soundChess.play(1,1, 1, 0, 0, 1);  checkWin(true);  if(!isWin){  aiPlayer.run(humanPlayer.getMyPoints(),null);  invalidate();  checkWin(false);  }  }  }  private void checkWin(boolean player){  if (player && humanPlayer.hasWin()) {  isWin = true;  soundWin.play(1,1, 1, 0, 0, 1);  alert("你赢了！");  }  if(!player && aiPlayer.hasWin()){  isWin = true;  soundDefeat.play(1,1, 1, 0, 0, 1);  alert("你输了！");  }  if (chessboard.getFreePoints().isEmpty()) {  isWin = true;  alert("和棋！");  }  }  private void alert(String msg){  AlertDialog.Builder builder = new AlertDialog.Builder(getContext());  builder.setTitle("提示");  builder.setMessage(msg);  builder.setPositiveButton("确定", new DialogInterface.OnClickListener() {  @Override  public void onClick(DialogInterface dialogInterface, int i) {  dialogInterface.dismiss();  }  });  builder.setNegativeButton("重新开局", new DialogInterface.OnClickListener() {  @Override  public void onClick(DialogInterface dialogInterface, int i) {  isWin = false;  dialogInterface.dismiss();  clear();  }  });  AlertDialog dialog = builder.create();  dialog.show();  }  public synchronized void startBlack(){  isBlack = true;  clear();  }  public synchronized void startWhite(){  isBlack = false;  clear();  }  private void clear(){  isWin = false;  chessboard.clear();  aiPlayer.clear();  humanPlayer.clear();  //我方是白棋，电脑第一个棋子一定下在正中心  if(!isBlack){  Point point = new Point(MAX\_LINE/2, MAX\_LINE/2);  aiPlayer.getMyPoints().add(point);  chessboard.getFreePoints().remove(point);  }  invalidate();  }  public synchronized void back(){  //悔棋  if(!humanPlayer.getMyPoints().isEmpty()&&!aiPlayer.getMyPoints().isEmpty()){  LinkedList<Point> list1 = (LinkedList<Point>) humanPlayer.getMyPoints();  LinkedList<Point> list2 = (LinkedList<Point>) aiPlayer.getMyPoints();  Point p1 = list1.removeLast();  Point p2 = list2.removeLast();  chessboard.getFreePoints().add(p1);  chessboard.getFreePoints().add(p2);  isWin = false;  invalidate();  }  }  private static final String instance = "INSTANCE";  private static final String win = "ISWIN";  private static final String black = "ISBLACK";  private static final String human = "HUMAN";  private static final String ai = "AI";  //view的存储与恢复  @Override  protected Parcelable onSaveInstanceState() {  Bundle bundle = new Bundle();  Log.d("ChessView","保存棋局");  bundle.putParcelable(instance,super.onSaveInstanceState());  bundle.putBoolean(win,isWin);  bundle.putBoolean(black,isBlack);  bundle.putSerializable(human, (Serializable) humanPlayer.getMyPoints());  bundle.putSerializable(ai, (Serializable) aiPlayer.getMyPoints());  return bundle;  }  @Override  protected void onRestoreInstanceState(Parcelable state) {  if(state instanceof Bundle){  Log.d("ChessView","加载棋局");  Bundle bundle = (Bundle)state;  isWin = bundle.getBoolean(win);  isBlack = bundle.getBoolean(black);  chessboard = new ChessBoard(MAX\_LINE);  humanPlayer = new HumanPlayer();  humanPlayer.setChessboard(chessboard);  aiPlayer = new BaseComputerAi();  aiPlayer.setChessboard(chessboard);  List<Point> humanPoints = (List<Point>) bundle.getSerializable(human);  List<Point> aiPoints = (List<Point>) bundle.getSerializable(ai);  humanPlayer.getMyPoints().addAll(humanPoints);  aiPlayer.getMyPoints().addAll(aiPoints);  for(Point point:humanPoints){  chessboard.getFreePoints().remove(point);  }  for(Point point:aiPoints){  chessboard.getFreePoints().remove(point);  }  super.onRestoreInstanceState(bundle.getParcelable(instance));  return;  }  super.onRestoreInstanceState(state);  }  } |

1. 实例运行效果图





、

1. **总结**

本项目做的比较简单，由于时间问题界面没有太过于优化，导致界面看上去很单调，功能只是实现了人机下棋，悔棋等操作也没有实现。