# 够级智能出牌算法开发环境

## 一、软硬件环境

智能出牌算法的运行环境由“够级AI开发测试平台（简称：够级平台）”提供，够级平台运行于JavaSE7或更高版本之上。

够级平台实现够级打牌过程，同时加载AI类并创建其实例对象，由该对象实现智能出牌决策。当需要出牌时，够级平台调用AI对象的相应方法，向其提供当前牌局信息。AI对象返回决策结果，够级平台再执行该决策。



够级规则采用最基本的打法，不打点，不憋三，2和王可以随便挂。每把只计分不进贡。发到的牌没有2和王时也不允许“革命”。烧牌只允许烧第一把够级。

## 二、AI类编程接口

AI类需实现pers.zzk.gouji.model.GoujiAIInterface接口，该接口定义了出牌决策中的新发牌、普通管牌、烧牌、烧牌后发牌四种情形。具体定义如下：

public interface GoujiAIInterface {

// 返回够级AI的名称，AI名称将显示在座位上

String getName();

// 做出上牌、过牌或让牌的决策，用于普通管牌（即非新一轮先发牌）的情况

GoujiAIDecision chuPai(Situation situation);

// 提供新一轮先发牌时要出的牌，不能过牌或让牌，牌列表不能为空

GoujiAIDecision xianFaPai(Situation situation);

// 提供烧牌时要出的牌。烧牌的这一把可以不用挂王，但后面除最后一把外需要每一把都挂王。

GoujiAIDecision shaoPai(Situation situation);

// 烧牌以后的出牌。除最后一把外，每一把出牌都需要挂王。

GoujiAIDecision yiBaYiWang(Situation situation);

}

后四个方法的参数和返回值都是同一类型的。参数situation给出了决策时的牌局信息，返回值GoujiAIDecision类型给出了决策结果。

Situation类定义了牌局信息，具体定义如下：

public class Situation {

public int seat; // 当前出牌者座次

public List<Card> handCards; // 目前手中的牌

public int type; // 当前出牌类型，1：新发牌；2：普通管牌；3：烧牌；

// 4：烧牌后新发牌（一把一王）

public List<Card> lastChuPai; // 要管的牌，即本轮中上一个出的牌

public int lastChuPaiPosition; // 本论中上一个出牌的座次

public int faPaiQuanNextTurn; // 如果都过牌，下一轮先发牌权属于哪个座次，

// 一般情况下等于lastChuPaiPosition，

// 但是当lastChuPaiPosition出完牌后走科时，

// faPaiQuanNextTurn为借风者。

public PositionSituation[] position; // 各座次的状态，包括局中状态、是否走科及走了几科、轮中状态、还剩几张（如果满足报牌规则）

public GameRecord gameRecord; // 本局从最初开始到目前的全部出牌记录

public int[] numberGone; // 记牌器，即统计出的每种牌已经出了多少张

}

GoujiAIDecision定义了AI决策结果，具体定义如下：

public class GoujiAIDecision {

public enum Option {ShangPai, GuoPai, RangPai; } // 决策选项：上牌、过牌、或让牌

public static String[] decisionString = {"上牌", "过牌", "让牌"}; //对应的中文字符串

public Option decision; // 决策结果是上牌、过牌或让牌

public List<Card> cards; // 如果决定上牌，cards中保存要出的牌

}

其他相关的类的定义参见doc下的类文档。

## 三、AI类本地开发环境

1、下载够级平台开发包，包括以下文件：

gouji.jar

images\

doc\

icon.png

startup.bat

situation文件

2、创建Java Project

添加JAR包gouji.jar（在Project上点右键，依次选择Build Path - Libraries - Add External JARs，再选择gouji.jar）。

编写一个类实现pers.zzk.gouji.model.GoujiAIInterface接口，实现出牌所需的四个方法。

3、situation测试

方便起见，AI类编写期间先用Situation文件测试。随开发包提供了一些situation文件，文件名为situation\*\*\_\*\*.xml，用于测试AI类的各个方法。可以用Situation类的loadFromFile方法将situation文件加载到situation对象，然后座位参数提供给AI类的各方法。具体代码可参考第四部分的示例。

situation文件的例子包括：

situation01\_新发牌.xml

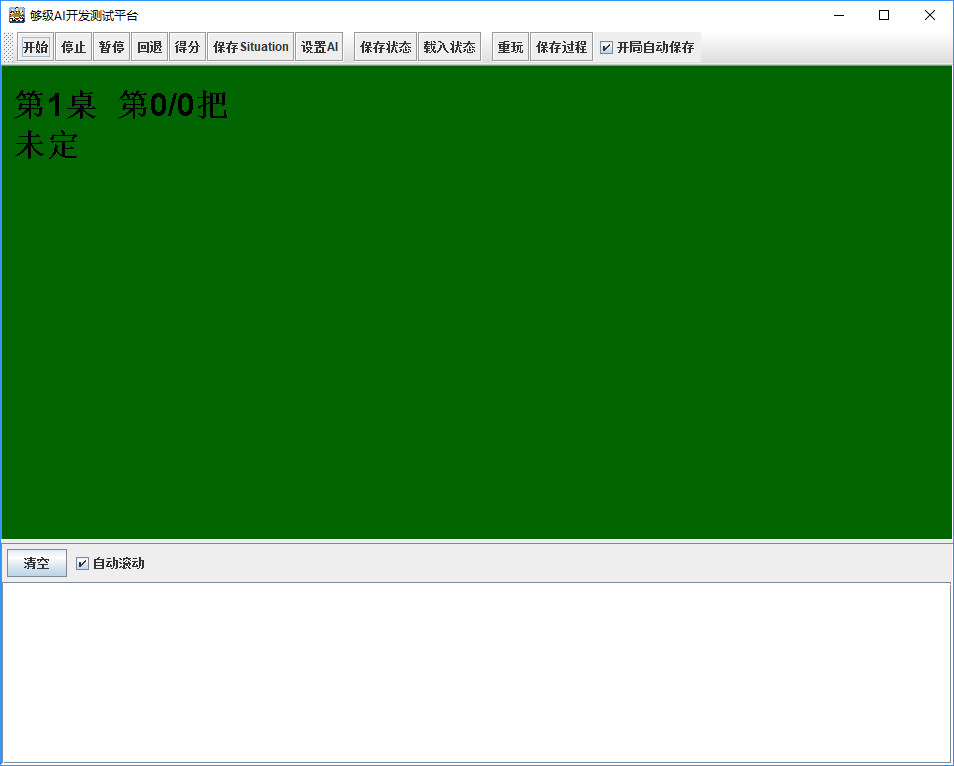
situation02\_管牌.xml

situation03\_烧牌.xml

situation04\_烧牌后发牌.xml

4、平台测试

运行startup.bat启动够级平台。

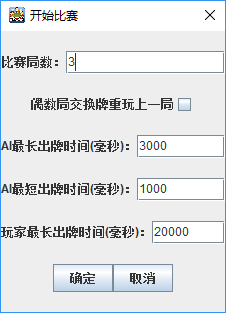


点击“设置AI”按钮，将开发的AI类设置为某个座位上的AI。为了让够级平台能够找到AI类的class文件，需要将其复制到够级平台目录下的与AI类的包路径对应的目录中。例如，goujitest.testAI是编写的AI类，那么需要将testAI.class文件复制到够级平台目录下的goujitest子目录下。



也可将类型设置为“本地玩家”，那么轮到该座次出牌时将由玩家决策。

然后点击“开始”即可让AI进行够级游戏，游戏中六家分别由对应的AI对象做出牌决策。可以设定比赛局数、AI出牌时间、玩家出牌时间。可以选择偶数局互换牌重玩上一局，在比较测试两个AI的强弱时，可以去除发牌随机性的影响。



点击“停止”按钮可以立即终止比赛。

点击“暂停”按钮可以暂停比赛，再次点击可以恢复比赛。

点击“回退”按钮可以回退一步，当前为新发牌时，回退到上一轮新发牌。

点击“得分”按钮可以查看每家的积分，积分方式：头科+2，二科+1，三科四科+0，二落-1，大落-2，圈三户每家+2，被圈三户每家-2。



点击“保存Situation”按钮，可以将当前状态保存到一个situation文件，以便记录当前决策条件，用于调试AI类。

点击“保存状态”按钮，可以将当前状态保存到一个xml文件，文件名为game\_record\_[序号].state。点击“载入状态”按钮，则可以从一个state文件中恢复状态并继续打牌。此功能可以用来复现特定的出牌情况，便于AI算法的编写调试。

点击“重玩”按钮，可以重玩当前牌局，各家发的牌会一样。

点击“保存过程”按钮，可以将游戏过程保存到一个xml文件，文件名为game\_record\_[序号].record。内容包括发牌、出牌过程、结果。一局结束后，也会自动将游戏过程保存到一个xml文件，文件名为record-desk1-[日期时间].xml。

比赛过程中，每结束一局会将结果保存在一个txt文件中，文件名为result-desk1-[日期时间].txt。比赛结束后，累计的结果也会保存在该文件中。

## 四、AI类示例

下面是一个基本的出牌算法示例，采用如下规则：

\* 1、先发牌时出最小的一套牌；

\* 2、不管联邦的10以上的牌（包括10）；

\* 3、其它的牌“有就上”，即能管住就上牌；

\* 4、上牌时选择牌力降低最少的牌，牌力降低=出牌前牌力-出牌后牌力；

\* 5、牌力计算采用 牌力计算表

|  |  |  |  |
| --- | --- | --- | --- |
|  | 构成一套够级 | 每多一张 | 不成套 |
| 大王 |  | 30 |  |
| 小王 |  | 20 |  |
| 2 |  | 10 |  |
| A | 8 | 5 | -1 |
| K~10 | 6 | 3 | -4 |
| 9~7 | 0(5张+) |  | -6 |
| 6~3 | 0(6张+) |  | -10 |

\* 6、在能烧对方的牌时烧牌；

\* 7、烧牌后从最小的牌开始挂王；

\* 8、没有考虑已出牌的统计、各家出牌情况、本局出牌记录。

代码如下：

public class TestAI implements GoujiAIInterface {

@Override

public String getName(){

return "测试AI";

}

@Override

public GoujiAIDecision chuPai(Situation situation) {

GoujiAIDecision decision = new GoujiAIDecision();

decision.clear();

CardCombo cardCombo = new CardCombo();

int position = situation.seat; // 当前出牌位置

List<Card> cards = situation.handCards;

int[] numberCount = generateCount(cards);

CardCombo lastChuPai = new CardCombo();

lastChuPai.fromCardsList(situation.lastChuPai);

CardCombo cc = null;

if(lastChuPai.getActualNumber()<7 || !isAlly(position, situation.lastChuPaiPosition)) { // 只管10以下的或者非联邦的牌

cc = findGuanPai(numberCount, lastChuPai);

}

if(cc == null) {

decision.decision = GoujiAIDecision.Option.GuoPai;

} else {

decision.decision = GoujiAIDecision.Option.ShangPai;

cardCombo.set(cc);

}

if(decision.decision==GoujiAIDecision.Option.ShangPai) {

selectCards(situation.handCards, cardCombo, decision.cards);

}

return decision;

}

@Override

public GoujiAIDecision xianFaPai(Situation situation) {

GoujiAIDecision decision = new GoujiAIDecision();

decision.clear();

CardCombo cardCombo = new CardCombo();

List<Card> cards = situation.handCards;

int[] numberCount = generateCount(cards);

for(int n=0; n<=11; n++) { // 0~11对应于'3' ~ 'A'

if(numberCount[n]>0) { // 如果手中有'3'~'A'，则出该套牌

decision.decision = GoujiAIDecision.Option.ShangPai;

cardCombo.set(n, numberCount[n]);

break;

}

}

if(decision.decision != GoujiAIDecision.Option.ShangPai) { // 如果手中没有'3'~'A'，则把'2'和'王'一起出掉

decision.decision = GoujiAIDecision.Option.ShangPai;

cardCombo.set(0, 0, numberCount[12], numberCount[13], numberCount[14]);

}

selectCards(situation.handCards, cardCombo, decision.cards);

return decision;

}

@Override

public GoujiAIDecision shaoPai(Situation situation) {

// 不烧联邦的牌

if(isAlly(situation.seat, situation.lastChuPaiPosition) ) return null;

// 根据手牌列表生成各种牌值的牌的张数

int[] numberCount = generateCount(situation.handCards);

// 根据要管的牌的列表生成组合表示

CardCombo lastChupai = new CardCombo();

lastChupai.fromCardsList(situation.lastChuPai);

// 找出手牌中能管住上一把牌的所有组合

List<CardCombo> guanPaiSet = findGuanpaiSet(numberCount, lastChupai);

int[] temp = new int[15];

Iterator<CardCombo> iter = guanPaiSet.iterator();

while(iter.hasNext()) { // 先找能直接发言走科的

CardCombo cc = iter.next();

// 把烧牌的牌出掉后，各牌张数保存到temp数组

System.arraycopy(numberCount, 0, temp, 0, 15);

if(cc.countOfNumber>0) temp[cc.number] -= cc.countOfNumber;

temp[12] -= cc.countOf2;

temp[13] -= cc.countOfViceKing;

temp[14] -= cc.countOfKing;

// 剩余的牌满足烧牌条件则烧牌，即除最后一把外可以一把挂一王

if(numberSetCount(temp)-kingCount(temp)<=1) {

GoujiAIDecision dec = new GoujiAIDecision();

dec.decision = GoujiAIDecision.Option.ShangPai;

selectCards(situation.handCards, cc, dec.cards);

return dec;

}

}

// 没有找到满足烧牌条件的出牌，则不烧牌

return null;

}

/\*\*

\* 统计numberCount记录的牌型中有几套牌，只统计3~A，不统计2和王。结果用来计算是否能够烧牌。

\* @param numberCount 3~A,2,小王,大王各有几张

\* @return 牌的套数

\*/

int numberSetCount(int[] numberCount){

int count = 0;

for(int i=0; i<12; i++) { // 3 ~ A

if(numberCount[i]>0) count ++;

}

return count;

}

/\*\*

\* 统计numberCount记录的牌型中有几张王。结果用来计算是否能够烧牌。

\* @param numberCount 3~A,2,小王,大王各有几张

\* @return 王的张数

\*/

int kingCount(int[] numberCount) {

return numberCount[13] + numberCount[14];

}

@Override

public GoujiAIDecision yiBaYiWang(Situation situation) {

GoujiAIDecision decision = new GoujiAIDecision();

decision.clear();

decision.decision = GoujiAIDecision.Option.ShangPai;

// 根据手牌列表生成各种牌值的牌的张数

int[] numberCount = generateCount(situation.handCards);

CardCombo cardCombo = new CardCombo();

// 选出最小的一套牌

for(int n=0; n<=11; n++) { // 0~11对应于'3' ~ 'A'

if(numberCount[n]>0) { // 如果手中有'3'~'A'，则出该套牌

cardCombo.set(n, numberCount[n]);

break;

}

}

if(kingCount(numberCount)==0) { // 手牌中没有王了，一般情况下是最后一把了

cardCombo.countOf2 = numberCount[12]; // 把所有2都挂上

} else { // 手牌中还有王

if(numberSetCount(numberCount)<=2) { // 在倒数第2把将所有王和2挂上

cardCombo.countOf2 = numberCount[12];

cardCombo.countOfViceKing = numberCount[13];

cardCombo.countOfKing = numberCount[14];

} else { // 不是最后一把或倒数第2把，只挂一个王，从小王开始挂

if(numberCount[13]>0) cardCombo.countOfViceKing = 1;

else if(numberCount[14]>0) cardCombo.countOfKing = 1;

// 其他情况属于诈烧

}

}

selectCards(situation.handCards, cardCombo, decision.cards);

return decision;

}

/\*\*

\* 判断两个座次是否为联邦

\* @param pos1 座次1，范围0~5

\* @param pos2 座次2，范围0~5

\* @return

\*/

public boolean isAlly(int pos1, int pos2){

return (pos1-pos2+6)%2==0;

}

/\*\*

\* 统计cards中各种大小的牌的数量，忽略花色

\* @param cards

\* @return 各种牌的数量保存在一个int数组中，0~14下标分别存放'3'~'A',2,小王，大王的张数

\*/

public static int[] generateCount(List<Card> cards) {

int[] count = new int[15];

Arrays.fill(count, 0);

Iterator<Card> iter = cards.iterator();

while(iter.hasNext()) {

Card c = iter.next();

count[c.getNumber()] ++;

}

return count;

}

/\*\*

\* 从counts中生成能管住obj的所有可能组合牌

\* @param counts

\* @param obj

\* @return

\*/

public static List<CardCombo> findGuanpaiSet(int[] counts, CardCombo obj) {

List<CardCombo> l = new LinkedList<CardCombo>();

CardCombo cc = new CardCombo(); // 如果cc.getActualNumber()==0, 表示过牌

cc.set(0, 0);

// 复制一份counts数组

int[] t = new int[counts.length];

for(int i=0; i<counts.length; i++) t[i] = counts[i];

// 如果有大王，返回空集

if(obj.countOfKing>0) return l;

//计算如何管小王

if(t[14] < obj.countOfViceKing) {

return l;

} else {

cc.countOfKing = obj.countOfViceKing;

}

if(obj.getActualNumber()==13) { // 要管的牌只有小王

l.add(cc);

return l;

}

t[14] -= cc.countOfKing;

//当要管的牌值为2时，计算如何管2

if(obj.getActualNumber()==12) {

if(t[13]+t[14]<obj.countOf2) { // 管不住

return l;

} else {

if(t[13]>=obj.countOf2) { // 只用小王就能管住

cc.countOfViceKing += obj.countOf2;

t[13] -= obj.countOf2;

} else if(t[13]+t[14]>=obj.countOf2) { // 小王+大王才能管住

cc.countOfViceKing += t[13];

cc.countOfKing += obj.countOf2 - t[13];

t[14] -= obj.countOf2 - t[13];

t[13] = 0;

}

l.add(cc);

return l;

}

}

// 要管的牌值为 3 ~ A时，如何管

int aCount = obj.getCountExceptKing(); // 除去王以外有几张牌

int aNumber = obj.getActualNumber(); // 实际牌值

int totalMoney = t[12] + t[13] + t[14]; // 手牌中所有能挂的牌总张数

for(int n = aNumber+1; n<12; n++) { // 尝试用 4 ~ A去管

if(t[n]==0 || t[n]+totalMoney < aCount) continue; // 没有这种牌值的牌，或者即使挂上所有2和王张数也不够

int maxN = t[n]>aCount ? aCount : t[n];

for(int countN=maxN; countN>0; countN--) { // 包含了拆牌和挂牌的各种可能

if(countN+totalMoney < aCount) break;

int max2 = t[12]>(aCount-countN) ? aCount-countN : t[12];

for(int count2=max2; count2>=0; count2--) {

if(countN+count2+t[13]+t[14] < aCount) break;

int maxVice = t[13]>(aCount-countN-count2) ? (aCount-countN-count2) : t[13];

for(int countVice=maxVice; countVice>=0; countVice--) {

if(countN+count2+countVice+t[14] < aCount) break;

int countKing = aCount-countN-count2-countVice;

CardCombo instance = new CardCombo();

instance.set(cc);

instance.number = n;

instance.countOfNumber = countN;

instance.countOf2 = count2;

instance.countOfViceKing += countVice;

instance.countOfKing += countKing;

l.add(instance);

}

}

}

}

// 只用2和王去管

int max2 = t[12]>aCount ? aCount : t[12];

for(int count2=max2; count2>=0; count2--) {

if(count2+t[13]+t[14] < aCount) break;

int maxVice = t[13]>(aCount-count2) ? aCount-count2 : t[13];

for(int countVice=maxVice; countVice>=0; countVice--) {

if(count2+countVice+t[14] < aCount) break;

int countKing = aCount-count2-countVice;

CardCombo instance = new CardCombo();

instance.set(cc);

instance.number = 0;

instance.countOfNumber = 0;

instance.countOf2 = count2;

instance.countOfViceKing += countVice;

instance.countOfKing += countKing;

l.add(instance);

}

}

return l;

}

/\*\*

\* 从手牌中选出能够管住一套牌的牌，包括挂2和王，选出的牌是手牌牌力损失最小的牌

\* @param numberCount 手牌中每种牌的张数

\* @param obj 要管的牌

\* @return 能管住obj的一套牌，在没有能管住的牌的情况下返回null

\*/

CardCombo findGuanPai(int[] numberCount, CardCombo obj){

if(obj.countOfKing>0) return null; // 没有牌能管住挂大王的牌

CardCombo cc = new CardCombo();

cc.set(0, 0);

if(obj.countOfViceKing>0) { // 管挂小王的牌需要用同样数量的大王

if(numberCount[14]>=obj.countOfViceKing) cc.countOfKing = obj.countOfViceKing;

else return null; // 如果大王数量不够，则管不住

}

if(obj.getActualNumber()==13) return cc; // 如果要管的牌中只有小王，没有其他牌

int[] t = Arrays.copyOf(numberCount, numberCount.length); // 使用numberCount的副本，防止后续操作修改原始数据

t[14] -= cc.countOfKing; // 从手牌中除去前面用掉的大王

if(obj.getActualNumber()==12) { // 如果要管的牌是一套2

if(t[13]>=obj.countOf2) { // 小王够用，则用小王管

cc.countOfViceKing += obj.countOf2;

t[13] -= obj.countOf2;

} else if(t[13]+t[14]>=obj.countOf2) { // 小王不够用，但加上大王够用

cc.countOfViceKing += t[13];

cc.countOfKing += obj.countOf2 - t[13];

t[14] -= obj.countOf2 - t[13];

t[13] = 0;

} else return null; // 大王加上小王都不够用，则管不住

return cc;

}

// 要管的牌是 3 ~ A

int aCount = obj.getCountExceptKing(); // 除去王以后，要管的牌的张数

int aNumber = obj.getActualNumber(); // 要管的牌的大小

/\*

\* 尝试手中每套能管住的牌，计算使用该套牌后的牌力，从中选出一套牌力损失最小的牌

\*/

int afterValue = -10000; // 管住牌以后剩余的最大牌力

int selectedNumber = -1; // 剩余最大牌力所对应的用掉的牌

int totalMoney = t[12] + t[13] + t[14]; // 能挂的所有张数：2、小王、大王

for(int n=aNumber+1; n<12; n++) { // 遍历比要管的牌大的牌（除去2，即n==12，因为2需要特殊处理）

if(t[n]==0) continue; // 手中没有大小为n的牌

if(t[n]+totalMoney < aCount) continue; // 手中大小为n的牌挂上2和王也不够数量

int tN = t[n]; // 暂存需要改动的张数

int t12 = t[12];

int t13 = t[13];

int t14 = t[14];

// 减去用掉的张数

if(t[n]>=aCount) { // 不用挂钱

t[n] -= aCount;

} else if(t[n]+t[12]>=aCount) { // 需要挂钱

t[12] -= (aCount - t[n]);

t[n] = 0;

} else if(t[n]+t[12]+t[13]>=aCount) { //需要小王

t[13] -= aCount - t[n] - t[12];

t[12] = 0;

t[n] = 0;

} else { // 需要大王

t[14] -= aCount - t[n] - t[12] - t[13];

t[13] = 0;

t[12] = 0;

t[n] = 0;

}

int v = calculatePaili1(t); // 计算用掉后的牌力

t[n] = tN; // 恢复到用掉之前的张数，以便计算用其他牌管的情况

t[12] = t12;

t[13] = t13;

t[14] = t14;

if(v>afterValue) { // 与之前的最大值比较，如果使用本套牌后剩余牌力较大，则修改剩余的最大牌力

selectedNumber = n;

afterValue = v;

}

}

if(selectedNumber<0) { // 没能选出 2以下的牌来管住，则尝试只用2和王来管

if(totalMoney>=aCount) {

selectedNumber = 12;

}

}

if(selectedNumber>=0) { // 选出了一套牌，则生成具体的各种牌的数量

int countMoney = aCount; // 保存需要挂的张数

if(selectedNumber<12) { // 用2以下的牌3~A来管

cc.number = selectedNumber;

if(t[selectedNumber]>=aCount) { // 该牌张数足够，不用挂

cc.countOfNumber = aCount;

countMoney = 0;

} else { // 该牌张数不够，需要挂

cc.countOfNumber = t[selectedNumber];

countMoney = aCount - t[selectedNumber];

}

}

if(countMoney>0) { // 需要挂的张数

if(countMoney<=t[12]) { // 2足够挂

cc.countOf2 = countMoney;

countMoney = 0;

} else { // 2不够时，先把2全挂上

cc.countOf2 = t[12];

countMoney -= t[12];

}

}

if(countMoney>0) { // 2不够时，再挂小王

if(countMoney<=t[13]) {

cc.countOfViceKing += countMoney;

countMoney = 0;

} else {

cc.countOfViceKing += t[13];

countMoney -= t[13];

}

}

if(countMoney>0) { // 小王不够时，再挂大王

cc.countOfKing += countMoney;

countMoney = 0;

}

return cc;

} else return null; // 没有能够管住的牌，返回null

}

/\*\* 按表计算牌力，没考虑实际出牌情况

\* 牌力计算表

\* 一套够级 每多一张 不成套

\* 大王 30

\* 小王 20

\* 2 10

\* A 8 5 -1

\* K~10 6 3 -4

\* 9~7 0(5+) -6

\* 6~3 0(6+) -10

\*/

public static int calculatePaili1(int[] counts) {

int p = 0;

for(int i=0; i<=11; i++) { // 3 ~ A

if(counts[i]==0) continue;

if(i<=3) { // 3 - 6 不成套的牌每套-10

if(counts[i]<6) { // 6张成套

p -= 10;

}

} else if(i<=6) { // 7 - 9 不成套的牌每套-6

if(counts[i]<5) { // 5张成套

p -= 6;

}

} else if(i<=10){ // 10 - K

int n = CardCombo.getGoujiCount(i); // 一种牌成为够级牌所需张数

if(counts[i]>=n) { // 够一套够级，+6，另外每多1张+3

p += 6;

p += (counts[i]-n) \* 3;

} else { // 不够够级，每套-4

p -= 4;

}

} else { // A

int n = CardCombo.getGoujiCount(i); // 一种牌成为够级牌所需张数

if(counts[i]>=n) { // 够一套够级，+8，另外每多1张+5

p += 8;

p += (counts[i]-n) \* 5;

} else { // 不够够级，每套-1

p -= 1;

}

}

}

p += counts[12] \* 10 + counts[13] \* 20 + counts[14] \* 30; // 2、小王、大王直接按张数算分

return p;

}

/\*\*

\* 从srcCards中，选出cardCombo描述的套牌所需的具体的牌，放入destCards中。

\* 并不从srcCards中移除选出的牌。

\* @param srcCards

\* @param cardCombo

\* @param destCards

\*/

void selectCards(List<Card> srcCards, CardCombo cardCombo, List<Card> destCards){

destCards.clear();

boolean[] used = new boolean[srcCards.size()];

Arrays.fill(used, false);

select(destCards, srcCards, 14, cardCombo.countOfKing, used);

select(destCards, srcCards, 13, cardCombo.countOfViceKing, used);

select(destCards, srcCards, 12, cardCombo.countOf2, used);

select(destCards, srcCards, cardCombo.number, cardCombo.countOfNumber, used);

}

void select(List<Card> destCards, List<Card> srcCards, int number, int count, boolean[] used) {

if(count>0) {

Iterator<Card> iter = srcCards.iterator();

int index = 0;

while(iter.hasNext()) {

Card c = iter.next();

if(c.getNumber()==number && !used[index]) {

used[index] = true;

destCards.add(c);

count --;

if(count==0) break;

}

index++;

}

}

}

/\*\*

\* 根据当前出牌状态（新发牌、普通管牌、烧牌、烧牌后新发牌），调用ai对象的相应方法

\* @return 决策结果，可能为上牌、过牌、让牌

\*/

GoujiAIDecision makeDecision(Situation situation) {

GoujiAIDecision decision = null;

switch(situation.type) {

case 1: // 新发牌

decision = xianFaPai(situation);

break;

case 2: // 普通管牌

decision = chuPai(situation);

break;

case 3: // 烧牌

decision = shaoPai(situation);

break;

case 4: // 烧牌以后一把一王

decision = yiBaYiWang(situation);

break;

}

return decision;

}

public static void main(String[] args)throws Exception{

Situation situation = new Situation();

situation.loadFromFile(new File("situation01\_新发牌.xml"));

// situation01\_新发牌.xml situation02\_管牌.xml situation03\_烧牌.xml

// situation04\_烧牌后发牌.xml

TestAI ai = new TestAI();

GoujiAIDecision decision = ai.makeDecision(situation);

if(decision!=null) {

System.out.println(decision.toString());

} else {

System.out.println("decision=null");

}

}

}