**基于博弈树算法的爱因斯坦棋V1.0说明书**

1需求分析

1.1开发背景

棋盘游戏是一种有着悠久历史和广泛群众基础的智力活动项目，棋盘类游戏道具相对简单、所需空间较小、游戏乐趣无穷，因而在其诞生之时就吸引了不少群众参与到这类游戏中，棋盘类游戏丰富了人们的业余生活，锻炼了人们的智力，促进了朋友之间的交流，是一类健康有益的游戏活动。

计算机自产生以来促进了棋盘类游戏的发展，计算机作为高科技代表也作为选手参与了进来。由此衍生出了一个新的领域--机器博弈，机器博弈作为人工智能的一部分，从最初代替真实玩家满足用户的娱乐需求，到挑战人类智慧的极限，再到将成功运用到其他领域促进人工智能领域的发展，一步步证明了机器博弈存在的重要意义。

机器博弈领域最引人注目的事件莫过于2016年3月9日-15日，AlphaGo以总比分4:1战胜李世石，至此棋类项目中最难以克服的围棋也被智能机器人攻破了。机器博弈领域带来的突破让人们认识到了人工智能方向的无限可能性，因此越来越多的计算机编程爱好者参与到机器博弈项目中。

棋牌类游戏种类繁多、玩法多样，每一种棋盘类项目的攻克都是对新知识的探索，由此带来的成果不仅对计算机科学领域，甚至对于人类科学的其他领域都有重要的研究价值。

1.2开发目的

1.设计一款规则合理、操作简便，供人闲暇之余陶冶情操的益智小游戏。

2.学习机器博弈的基本原理、算法，掌握机器博弈项目的基本建模方案。

3.深入学习C++编程与图形化界面的结合，将抽象的数据结构转化为可视化的图形界面，实现抽象与现实的结合，数字到图形的转化，展示C++编程语言的魅力。

4.利用计算机强大的计算能力和准确的数据处理能力，将机器博弈的原理和思想充分展现出来，充分发挥计算机的优势。

5.通过开发棋类游戏学会分析博弈过程中的各类影响因素，包含主观支配因素和客观决定因素，并且掌握如何进行大规模测试、进行算法改进和测试分析。

1.3开发环境

•操作系统：Windows 10

•开发语言：C++

•开发平台：Qt

2软件总体设计

2.1软件总体框架

软件主要分为以下6种模块

1. 数据结构模块
2. 走法规则模块
3. 估值量化模块
4. 博弈树算法模块
5. 胜负判定模块
6. 图形界面模块

2.2软件总体框架说明

1. 数据结构模块

数据结构模块包含了该程序中用到的棋子，棋子步法保留器等数据集合，数据结构模块是整个程序的基础，它决定了其他模块的构建原理。比如，棋子的颜色属性是判断棋子玩家的重要依据，棋子的横纵坐标确定了棋子的位置，棋子的ID标识着每个棋子的独一无二。数据结构模块的确立为其他模块的构建奠定了坚实的基础。

1. 走法规则模块

走法规则模块确定了棋子的移动规则，是主要体现游戏规则的模块。棋子每次可以移动的位置由走法规则决定，比如，红色棋子只能向正右、正下或者右下方移动，蓝色棋子只能向正左、正上或者左上方移动。走法规则模块为游戏玩家制定了合理公平的原则，是游戏得以存在的条件，也为后面的人工智能算法奠定了基础。

1. 估值量化模块

估值量化模块是判定棋盘局面局势好坏的算法，是专门为计算机设定的走法判断标准，计算机走棋的目的是为了获得更有利的棋盘局面，以获得最终的胜利。判断棋盘局面的算法就是估值算法，在本模块中，我们经过多次计算和反复的测试，选择了相对合理的局势判断标准，这个标准的确定正是实现机器博弈的重要一环。

1. 博弈树算法模块

博弈树算法模块是爱因斯坦棋的核心模块，博弈树本质就是极大极小的搜索过程，利用递归的设计方法，通过对局面的逐层分析，将极大极小搜索过程结合起来最终确定对计算机收益最大的一步，这是机器博弈的关键算法。在本程序中，我们在极大极小搜索算法的基础上利用了剪枝算法，减去没有必要继续搜索的枝，节省了内存空间，提高了代码的运行效率，从而节省了计算机作出反应的时间。

1. 胜负判定模块

胜负判定模块是爱因斯坦棋的重要模块，游戏一般都是有胜有负，爱因斯坦棋也不例外，胜负判定模块根据爱因斯坦棋的规则确定判定的标准，然后结合棋子的数据结构，用代码的形式判定当前棋局是否已经结束，以及胜利的是哪一方。

1. 图形界面模块

图形界面模块利用Qt图形界面用户程序开发框架，将C++编程与图形化界面相结合，将抽象的数据结构转化为可视化的图形界面，实现抽象与现实的结合，数字到图形的转化，展示了C++编程语言的魅力。图形界面模块是软件用户最直观感受到的模块。

3软件设计方案

3.1爱因斯坦棋规则

爱因斯坦棋是2004年由德国人Ingo Althofer所推出的两人骰棋类游戏，为奥林匹亚电脑游戏诚实竞赛的指定棋类之一。游戏道具包括一个5\*5的棋盘，以及1到6数字的棋子（红方一套，棋子颜色为红色，蓝方一套，棋子颜色为蓝色）。

游戏初始时红方将六枚棋子任意摆放在棋盘左上方的3\*3的直角三角形区域，蓝方将六枚棋子任意摆放在棋盘右下方的3\*3的直角三角形区域。游戏开始时，由红方掷骰子，根据骰子最上面的数字决定红方可以移动的棋子，红方只能将指定旗子朝下、右或右下方移动一格至任何棋位，如果骰子提示的棋子已经死亡，那么改移动最接近该数字的棋子之一，如果棋子移动的目的位置有其他棋子，那么将这枚棋子移出棋盘不再使用。红方棋子移动结束后，由蓝方掷骰子，根据骰子最上面的数字决定蓝方可以移动的棋子，蓝方只能将指定旗子朝上、左或左上方移动一格至任何棋位，如果骰子提示的棋子已经死亡，那么改移动最接近该数字的棋子之一，如果棋子移动的目的位置有其他棋子，那么将这枚棋子移出棋盘不再使用。如此交替掷骰子走棋，以先抵达敌方角落位，或消灭所有敌棋为胜。

3.2软件各模块原理说明

1.数据结构表示（棋子和棋子步法保留器）

根据爱因斯坦棋的规则，首先棋子有颜色之分，利用颜色区别棋子属于哪一方，所以我们给了棋子一个bool类型的属性red（red为1，表示棋子属于红方，red为0表示棋子属于蓝方）；棋子在棋盘上是有位置的，所以我们给了棋子两个整数型的属性，x和y，分别表示棋子所在的横纵坐标；棋子被杀死之后是不能留在棋盘上的，为了判断棋子是否被杀死，我们赋予bool类型的棋子属性dead；每方的棋子都有1-6棋子，为了区别各个棋子，我们赋予棋子整数型属性id，其中红方的六枚棋子分配id为0-5，蓝方的六枚棋子分配id为6-11。

棋子步法保留器是为了保留棋子的上一步状态的，主要用于机器博弈过程中的博弈树算法，因此与棋子类似，我们要分配给它坐标属性和id属性，不过棋子步法保留器并非棋子，所以它不需要颜色和dead属性。经过不断修改，我们确定了四个属性， moveId（移动的棋子），killId（杀死的棋子），rowFrom（初始横坐标），colFrom（初始纵坐标），rowTo（目的横坐标），colTo（目的纵坐标）。

具体结构如下：

class Einsteinchess

{

public:

Einsteinchess();

~Einsteinchess();

void begin(int id);

int x; //横坐标

int y; //纵坐标

bool dead; //棋子存活状态

bool red; //棋子颜色

int id; //棋子编号

QString name();

};

class Footwork: public QObject

{

Q\_OBJECT

public:

explicit Footwork(QObject \*parent = 0);

~Footwork();

int moveId; //移动的棋子编号

int killId; //被杀的棋子编号

int xFrom; //初始横坐标

int yFrom; //初始纵坐标

int xTo; //目的横坐标

int yTo; //目的纵坐标

};

1. 走法规则算法

根据爱因斯坦棋的规则，玩家根据骰子最上面的数字决定可以移动的棋子，红方只能将指定旗子朝下、右或右下方移动一格至任何棋位，蓝方只能将指定旗子朝上、左或左上方移动一格至任何棋位，如果骰子提示的棋子已经死亡，那么改移动最接近该数字的棋子之一，如果棋子移动的目的位置有其他棋子，那么将这枚棋子移出棋盘不再使用。在这一模块，我们分为三部分，第一部分，掷骰子；第二部分，选择将要移动的棋子；第三部分，判断棋子走法是否符合规则。

掷骰子部分主要由qrand()函数实现，因为真实的骰子只有1-6这6个数，而我们在设计棋子的id时为了使用数组的方便，我们设置为0-11，考虑到这点，我们利用qrand()%6生成0-5这五个数（轮到蓝方走棋的时候根据qrand()%6+5选择棋子），而在图形化时，骰子示数的显示默认为加1后的值，这样解决骰子示数与id设置不符的矛盾。除此之外，为了游戏公平，我们只允许每个玩家走棋前掷一次骰子，如果掷骰子多次，我们只认为第一次有效。函数如下：

void Appearance::getorder()

{

If(对弈模式已经选择好&&游戏已经开始&&骰子未被掷出过)

{

获得骰子示数;

骰子被掷出次数设置为1;

重新绘制游戏界面;

}

}

选择棋子部分主要分为四种情况进行考虑，第一种情况是红方掷得骰子对应的棋子存在，此时可以直接选择对应棋子；第二种情况是红方掷得骰子对应的棋子不存在，此时要对棋盘上剩余的红方棋子进行选择，可以选择的棋子应该是id最接近该数字的棋子之一；第三种情况是蓝方掷得骰子对应的棋子存在，此时需要将骰子掷得得数字进行相应转换，然后可以直接选择对应棋子；第四种情况是蓝方掷得骰子对应的棋子不存在，此时要对棋盘上剩余的蓝方棋子进行选择，需要先将骰子掷得的数字进行相应转换，然后选择id最接近该数字的蓝色棋子之一。选择棋子之前可以进行一些基础的判断以节省一些不必要的时间，先判断当前是否已经有棋子被选中、当前要选择的棋子的编号是否符合规范、可供选择的棋是否恰为鼠标点击的棋。

这部分的具体代码如下：

trySelectEinsteinchess(int id)

1. If id == -1 Then return

2 //如果当前已有棋子被选中

3 If selected!=-1 Then return

4 //如果当前是红方回合

5 If redTurn Then

6 //对应棋子已经被杀死

7 If this->einsteinchess[order].dead Then

8 //寻找与骰子指示的棋子编号最近的棋子

9 For j=0 to 5

10 If this->einsteinchess[j].dead Then continue

11 If qAbs(j-order)<min Then

12 min=qAbs(j-order)

13 For k=0 to 5

14 If qAbs(k-order)==min&&!this->einsteinchess[k].dead

15 &&k==id Then

16 selected=id

17 //对应棋子未被杀死

18 If !this->einsteinchess[order].dead Then

19 //如果鼠标点击的棋子就是骰子指示的棋子

20 If id==order Then

21 selected=id

22 //重新绘制游戏棋盘

23 update()

24 //如果当前是蓝方回合

25 If !redTurn Then

26 //对应棋子已经被杀死

27 If this->einsteinchess[order+6].dead Then

28 //寻找与骰子指示的棋子编号最近的棋子

29 For j=6 to 11

30 If this->einsteinchess[j].dead Then continue

31 If qAbs(j-order-6)<min Then

32 min=qAbs(j-order-6)

33 For k=6 to 11

34 If qAbs(k-order-6)==min&&

35 !this->einsteinchess[k].dead&&k==id Then

36 selected=id;

37 //对应棋子未被杀死

38 If !this->einsteinchess[order+6].dead Then

39 If id==order+6

40 selected=id

41 //重新绘制游戏棋盘

42 update()

判断棋子走法是否符合规则部分主要根据前面介绍的爱因斯坦棋走法规则。

具体代码如下：

ruleOfChess(int moveid, int x, int y)

1 If 要移动的棋子是红方棋子 Then

2 If 目的位置是当前棋子的正右方相邻位置&&目的位置在棋盘上 Then

3 return true

4 If 目的位置是当前棋子的正下方相邻位置&&目的位置在棋盘上 Then 5 return true

6 If 目的位置是当前棋子的正右下方相邻位置&&目的位置在棋盘 Then 7 return true

8 If 要移动的棋子是蓝方棋子 Then

9 If 目的位置是当前棋子的正左方相邻位置&&目的位置在棋盘上 Then 10 return true

11 If 目的位置是当前棋子的正上方相邻位置&&目的位置在棋盘上 Then 12 return true

13 If 目的位置是当前棋子的正左上方相邻位置&&目的位置在棋盘 Then 14 return true

15 return false

3.估值算法

估值算法是指用量化的方式表现当前局面的优劣，也就是用具体的数字描述当前局面的好坏，这套算法主要根据爱因斯坦棋的规则而来，我们在估值算法中主要考量的因素是每方棋子的在场数量和棋子离目的位置的距离，因为在游戏中我们默认红方是真实玩家，黑方是电脑，所以估值算法是以电脑的角度进行设计的，具体代码如下：

value()

1 For i=0 to 5

2 If einsteinchess[i].dead Then continue

3 //红方棋子场上数量计算

4 number1++

5 //红方棋子距离目的位置形势量化

6 valueRed+= einsteinchess[i].y+einsteinchess[i].x)\*

7 (einsteinchess[i].y+einsteinchess[i].x)/

8 (8-einsteinchess[i].y-einsteinchess[i].x)

9 valueRed=valueRed \*number1 //红方场面局势得分

10 For i=6 to 11

11 If einsteinchess[i].dead Then continue;

12 number2++ //蓝方棋子场上数量计算

13 valueBlack += (8-einsteinchess[i].y-einsteinchess[i].x)\*

14 (8-einsteinchess[i].y-einsteinchess[i].x)

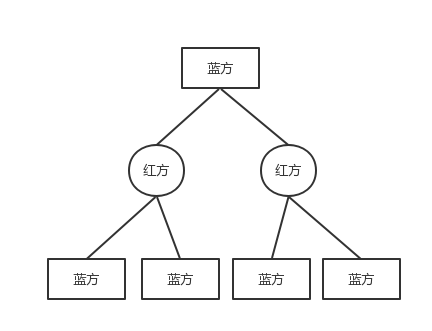
15 /(einsteinchess[i].y+einsteinchess[i].x)

16 valueBlack=valueBlack \*number2 //蓝方场面局势得分

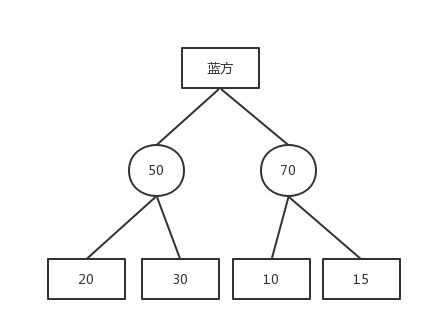
17 return valueBlack - valueRed //当前局面对电脑而言的分数

1. 博弈树算法

博弈树是指由于动态博弈参与者的行动有先后次序，因此可以依次将参与者的行动展开成一个树状图形。博弈的初始格局是初始节点。在博弈树中，“或”节点和“与”节点是逐层交替出现的。自己一方扩展的节点之间是“或”关系，对方扩展的节点之间是"与"关系。双方轮流地扩展节点。所有自己一方获胜的终局都是本原问题，相应的节点是可解节点；所有使对方获胜的终局都认为是不可解节点。如下图所示：

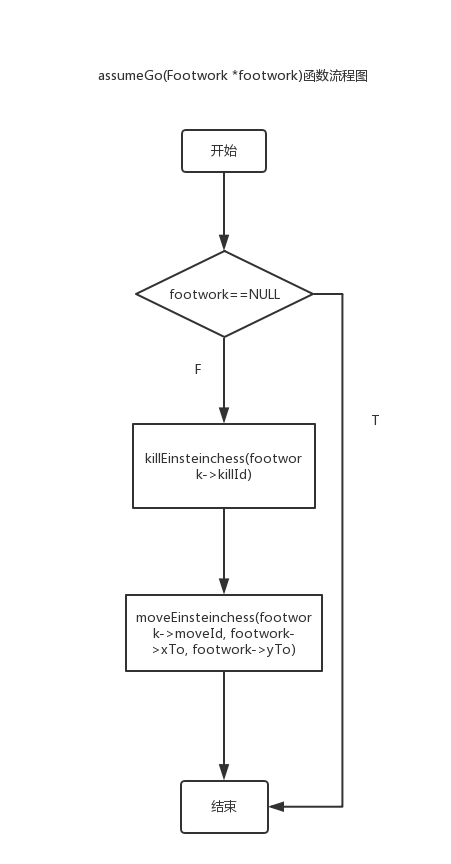


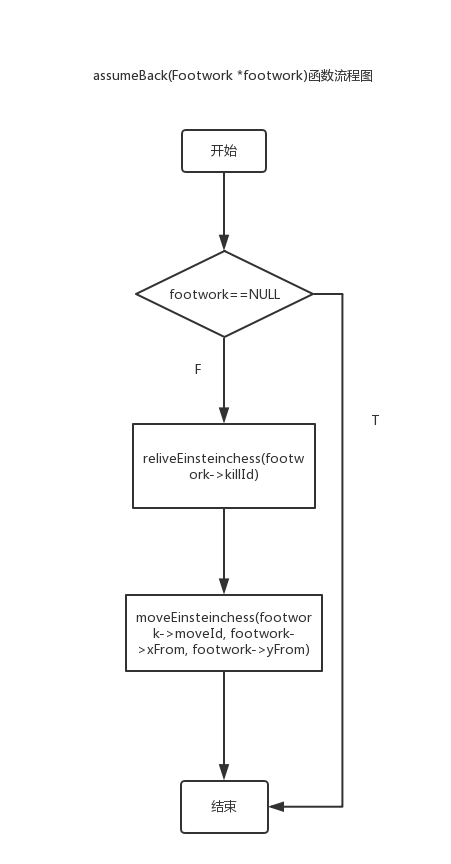
对于爱因斯坦棋游戏而言，蓝方为电脑，它在走棋前要进行思考，假如走出某一步之后，红方会选择哪一步棋，红方走出这步棋之后，自己的局面形势会如何，假如最初走的是另一步，会不会有更好的结果……



对于上图而言，如果不仔细考虑，蓝方会选择70分的那步，而红方的想法是尽量让蓝方的优势减小，所以他会选择让蓝方的局面只有10分的路，而如果蓝方最开始选择的是50分的路，红方依旧会尽量让蓝方的优势减小，而此时红方最好的选择是让蓝方的局面分为20分，20>10，所以蓝方初始时选择50分的路对于以后的局面形势更有利。由此引出了博弈树算法，本质就是极大极小的搜索过程。具体搜索过程是这样的：红方（真实玩家）和蓝方（电脑）对弈，轮到电脑走棋了，那么电脑会遍历蓝方的每一个可能走棋方法，然后对于前面电脑的每一个走棋方法，遍历红方的每一个走棋方法，然后接着遍历蓝方的每一个走棋方法，如此下去，直到得到确定的结果或者达到了搜索深度的限制。当达到了搜索深度限制，此时无法判断结局如何，一般都是根据当前局面的形式，给出一个得分，这个得分就是由前面我们所介绍的估值算法确定的。在搜索树中，表示电脑走棋的节点即为极大节点，表示红方走棋的节点为极小节点。在上面的例图中，搜索深度为2，如果要想得到更合理的走棋，搜索深度要尽量的增加，这儿就用到了递归处理方法。

对于上述所说的博弈树搜索算法（极大极小搜索策略），有一个问题需要解决，就是假设走出的棋子并不是真实走出去的，它只是作为电脑思考的一部分，在真正决定好要走出哪步棋子前，前面每一步假设走出的棋子都要走回来，为此，我们设计了“假设走出去函数”assumeGo(Footwork \*footwork)和“假设走回来函数”assumeBack(Footwork \*footwork)。





解决了上述问题之后在设计算法前还要考虑另外一个问题：怎样遍历己方当前所有可能的走法。在设计这一部分时，我们分成了两种情况考虑，第一种情况，对于搜索算法的第一层，当前可以移动的棋子只能是由骰子示数指定的棋子，所以此时我们不必考虑其他棋子，只搜索棋盘上可供目标棋子移动的位置；第二种情况，对于搜索算法的第n层（n>1），在这一层中没有掷骰子的过程，每枚棋子都可能被移动，所以我们要考虑所有棋子，搜索棋盘上所有可供我方棋子移动的位置。第一种搜索棋子的伪代码如下：

getPossibleMove(QVector<Footwork \*> &footworks)

1 //如果骰子指定的棋子已经死亡

2 If this->einsteinchess[order].dead then

3 //寻找场上存在的与骰子数最接近的棋子编号

4 For j=6 to 11

5 If this->einsteinchess[j].dead then continue

6 If qAbs(j-order)<min then

7 min=qAbs(j-order);

8 //将场上存在的与骰子数最接近的棋子确定为目标棋子

9 For k=6 to 11

10 If qAbs(k-order)==min&&!this->einsteinchess[k].dead then

11 //搜索场上所有位置

12 For x=0 to 4

13 For y=0 to 5

14 killId = this->getEinsteinchessId(x, y)

15 //如果当前位置可供目标棋子移动

16 If ruleOfChess(k, x, y) then

17 //将棋子、目的位置棋子、目的位置保存

18 saveFootwork(k, killId, x, y, footworks)

19 //如果骰子指定的棋子没有死亡

20 If !this->einsteinchess[order].dead then

21 //搜索场上所有位置

22 For x=0 to 4

23 For y=0 to 4

24 killId = this->getEinsteinchessId(x, y)

25 //如果当前位置可供目标棋子移动

26 If ruleOfChess(order, x, y) then

27 //将棋子、目的位置棋子、目的位置保存

28 saveFootwork(order, killId, x, y, footworks)

在上述算法中，我们分为了两种情况进行考虑，第一种情况是骰子指定的棋子已经死亡，第二种情况是骰子指定的棋子没有死亡。

对于第一种情况，我们可以遍历场上存在的棋子，寻找与骰子示数最接近的棋子，一般这样的棋子存在两个。然后我们把找到的棋子设为目标棋子。然后对棋盘上的所有位置进行遍历，寻找可供棋子移动的位置，然后我们将棋子、目的位置棋子、目的位置保存在前面介绍的棋子步法保留器中。

对于第二种情况，目标棋子只有一个，就是骰子指示的那个棋子。我们同样要遍历场上所有位置，寻找可供棋子移动的位置，然后我们将棋子、目的位置棋子、目的位置保存在前面介绍的棋子步法保留器中。

第二种搜索方法与第一种有所区别，在这种搜索算法中要将所有的己方棋子设为目标棋子（在这种搜索算法中，没有骰子指示将要移动的棋子），然后在棋盘上寻找可供棋子移动的位置，然后将棋保留在步法保留器中，具体伪代码如下：

getAllPossibleMove(QVector<Footwork \*> &footworks)

1 //如果当前是红方走棋

2 If redTurn then

3 minId = 0, maxId = 6

4 //如果当前是蓝方走棋

5 Else then

6 minId = 6, maxId = 12

7 //遍历所有己方棋子

8 For i=minId to maxId-1

9 If this->einsteinchess[i].dead then continue

10 //遍历场上所有位置

11 For x=0 to 4

12 For y=0 to 4 {

13 killId = this->getEinsteinchessId(x, y)

14 //如果当前位置可供目标棋子移动

15 If ruleOfChess(i, x, y) then

16 //将棋子、目的位置棋子、目的位置保存

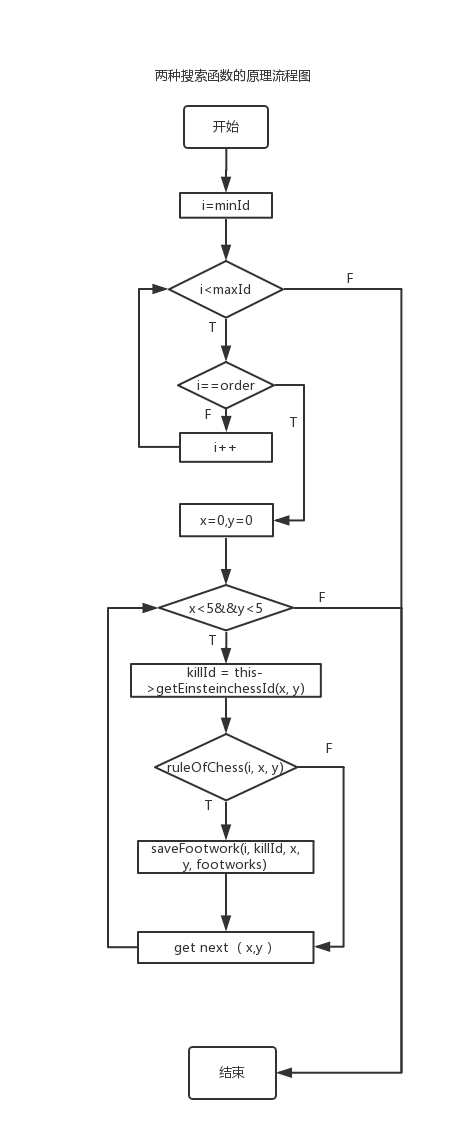
17 saveFootwork(i, killId, x, y, footworks)

第二种搜索函数主要用在最大最小搜索策略的递归过程（详见下文）中，因为在递归过程中，我们无法确定骰子掷出后会显示什么数，因此为了全面考虑，防止随意选择目标棋子，造成与骰子示数不合而影响计算机的计算，我们假设递归过程中每枚棋子被选为目标棋子的概率相同，均为1/6，，即每枚棋子都可能成为目标棋子。

基于上述考虑，我们在递归过程中将棋盘上所有编号范围内的棋子的所有可以移动的步法均视为可能步法，都可以看做棋子将来的选择之一，因此，要遍历棋盘上所有的位置，利用走法规则为每枚棋子寻找所有的可能走法，然后将其保存。

第二种搜索算法中，还有一个细节，就是最前面的minId和maxId的确定，它们确定了棋子编号的两个范围，0-5和6-11，也就是红方棋子和蓝方棋子。在递归过程中，计算机既要考虑它自己的所有可能走法，也要站在对方的角度上考虑走法，综合所有可能情形选择最佳走法。递归过程中如果在机器方考虑，那么棋子的编号范围为6-11，如果在人的角度考虑，那么棋子的编号范围为0-5。

对于上述两种搜索算法，原理是一样的，都是在棋盘上搜索所有的可能走法，然后将走法保存在棋子步法保留器中，两种搜索算法的原理可以用如下流程图表示：



上述准备工作全部做完之后就是最重要的搜索算法了，在搜索算法中我们设计了三个函数:

getBestMove(),

getMinvalue(int depth, intcurrentMinvalue),

getMaxvalue(depth-1, minInAllMaxvalue)。

第一个函数是搜索算法的入口和出口，第二个函数是对方玩家用来寻找对我方玩家最坏局面的算法，第三个函数是我方玩家用来寻找对我方玩家最好局面的算法。三种搜索算法运用了极大极小搜索策略配合剪枝算法达到准确高效率的求解效果。第一个函数的伪代码如下：

getBestMove()

1 //获得当前所有走法集合

2 getPossibleMove(footworks)

3 //遍历所有走法

4 while footworks.count() do

5 //取其中一步

6 footwork=footworks.last()

7 //将这一步移除步法保留器

8 footworks.removeLast()

9 //按照步法保留器走出这一步

10 assumeGo(footwork)

11 //如果走出这一步恰好胜利

12 If win()==2 then

13 //将这一步保留

14 a=footwork

15 //走回之前状态

16 assumeBack(footwork)

17 //将步法保留器清空，释放空间

18 While footworks.count() do

19 footwork=footworks.last()

20 footworks.removeLast()

21 delete footwork

22 //返回最优步法

23 return a;

24 //获得走出这一步后下层返回的局面分

25 minvalue=getMinvalue(this->depth-1, maxInAllMinvalue);

26 //走回之前状态

27 assumeBack(footwork)

28 //对比之前所有步法可能获得的最高分

29 If minvalue>maxInAllMinvalue then

30 If a then delete a

31 //将这一步确定为最优步

32 a=footwork

33 //更新n层搜索后能获得的最大局面分

34 maxInAllMinvalue = minvalue

36 Else then

37 //将这一步淘汰

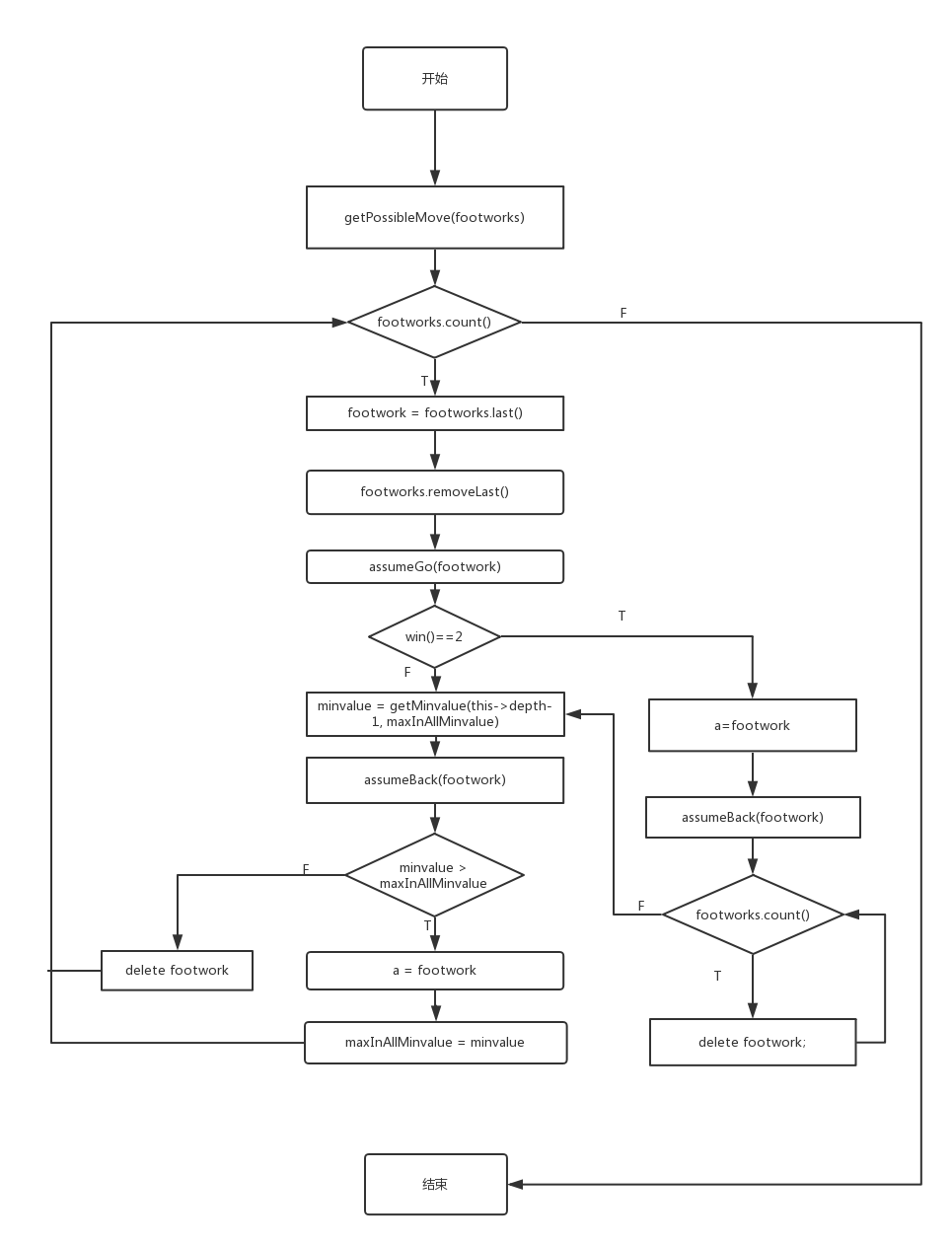
38 delete footwork

39 //返回最优步

40 return a

上述算法是在极大极小搜索算法和剪枝算法基础上改进后得到的，在搜索过程中，如果找到一种可以直接获得胜利的步法，那么就会舍弃其他所有步法，直接选择这一步，有了这个判断后，我们的搜索算法可以高效处理极限问题，从而节省了时间，使得程序更加高效。

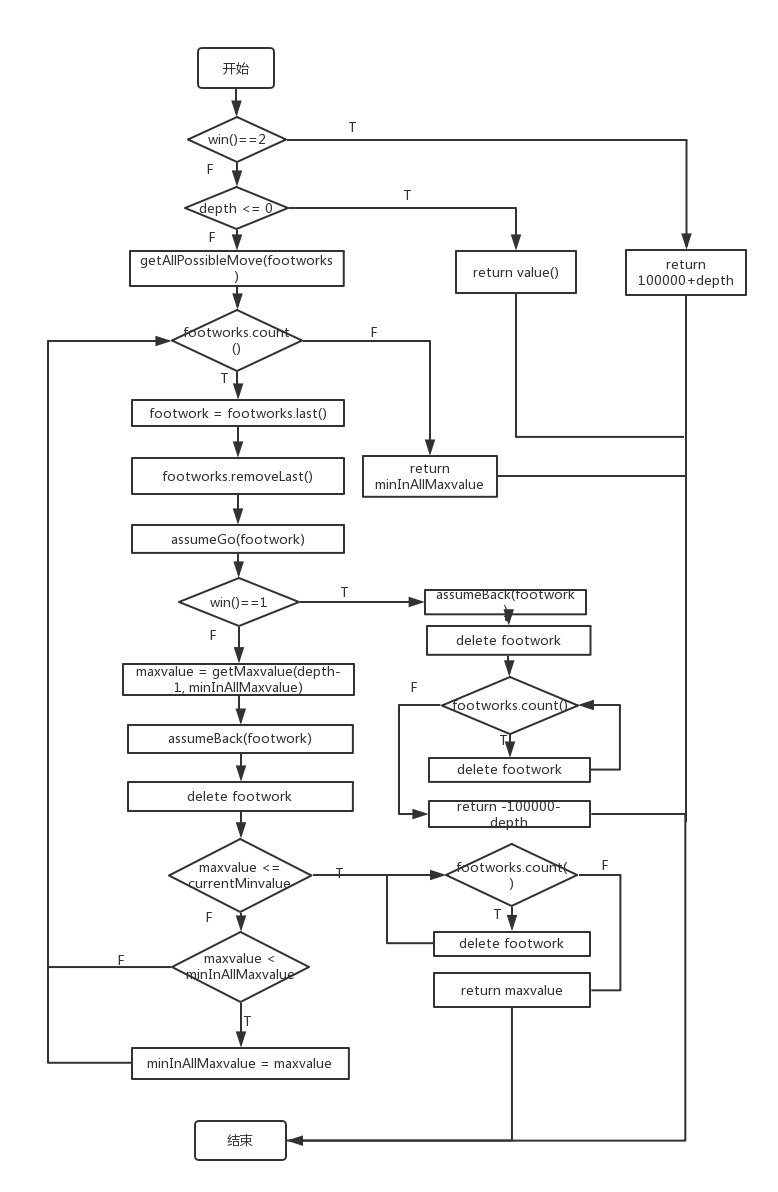
getBestMove()的流程图如下：



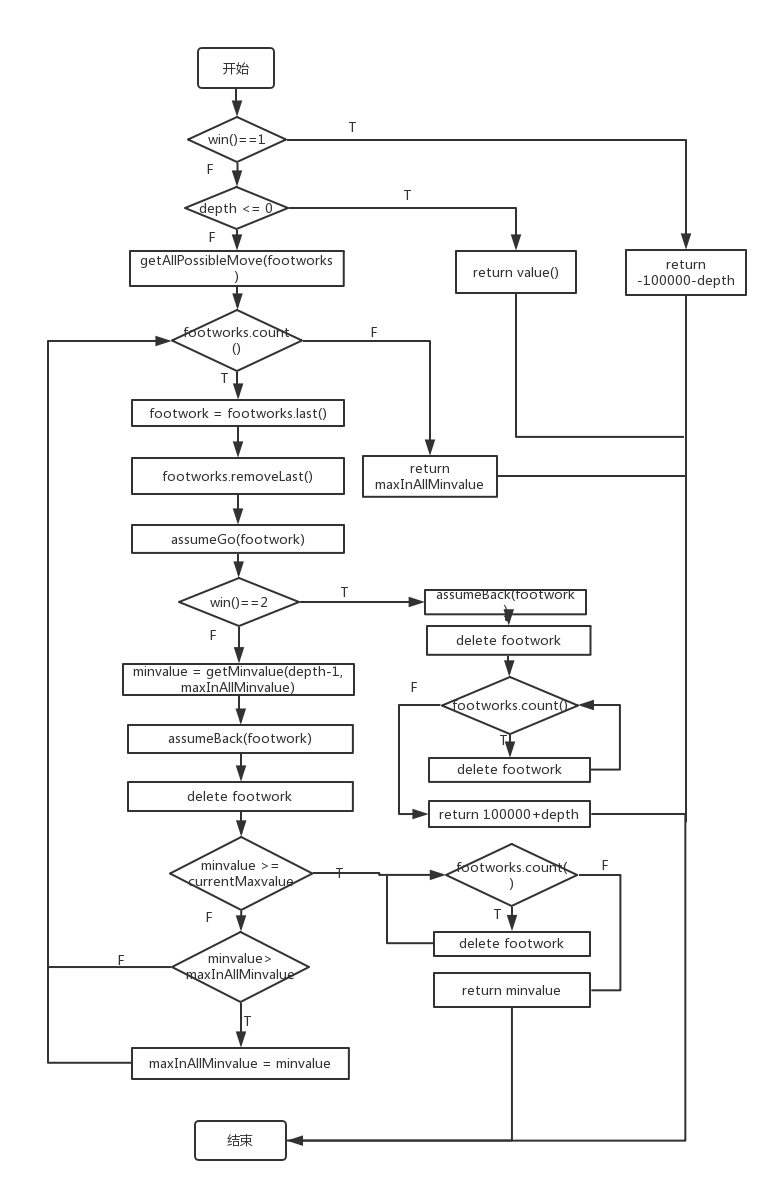
getMinvalue(int depth, int currentMinvalue)和getMaxvalue(depth-1, minInAllMaxvalue)函数的原理与getBestMove()函数相似，都是通过比较下一层返回来的局面分数获得最优解，不过这两个函数用到的搜索算法是前面讲的第二个，因为在递归过程中无法确定骰子的示数，所以只能把左右的棋子都看做是可以移动的棋子。

getMinvalue(int depth, int currentMinvalue)函数的功能是在红方的角度搜索令电脑获得的最小局面分，利用递归的方法，获得下一层函数返回的最大局面分，然后在最大局面分中挑选最小的局面分，对于极限情况，即红方可以走出直接获胜的棋，电脑要竭力避免这种情况，只要发现红方有直接获胜的方法可以选择，那么电脑直接淘汰这一路，选择其他路。

getMinvalue(int depth, int currentMinvalue)函数的流程图如下：



getMaxvalue(depth-1, minInAllMaxvalue)的流程图如下：



getMaxvalue(int depth, int currentMaxvalue)算法与极小值搜索算法的思想是相同的，不同的是它获得下一层函数返回的最小局面分，然后在最小局面分中挑选最大的局面分。对于极限情况，即电脑可以走出直接获胜的棋，我们设想的是电脑要尽可能获得这种情况，虽然骰子的示数具有随机性，但是只要存在这种可能性，我们要尽量为电脑争取。因此，只要发现蓝方有直接获胜的方法可以选择，那么电脑直接选择这一路。

5.胜负判定算法

胜负判定算法的设计原则就是爱因斯坦棋的规则，以先到达敌方角落位或者杀死敌方所有棋子者为胜，为了便于判别哪方取得胜利，我们为算法设定了返回值，如果棋局未结束，返回0；如果红方胜，返回1；如果蓝方胜，返回0。具体代码如下：

win()

1 //如果红方角落位有蓝方棋子

2 If getEinsteinchessId(0,0)>5 then

3 return 2

4 //如果蓝方角落位有红方棋子

5 If getEinsteinchessId(4,4)>=0&&getEinsteinchessId(4,4)<6 then

6 return 1

7 //查找当前棋盘上存在的红色棋子数和蓝色棋子数

8 For i=0 to 4

9 For j=0 to 4

10 If getEinsteinchessId(i,j)>5 then

11 num2++

12 If getEinsteinchessId(i,j)>=0&&getEinsteinchessId(i,j)<6 then 13 num1++

14 //如果蓝方棋子数为0

15 If num2==0 then

16 return 1

17 //如果红方棋子数为0

18 If num1==0 then

19 return 2

20 return 0

6.图形界面

图形界面是人机交互的重要环节，qt软件提供的接口和函数给爱因斯坦棋提供了很大帮助。图形界面部分分为游戏界面绘制和鼠标点击事件两部分组成。绘制游戏界面时，我们用drawEinsteinchess(QPainter &p, int id)绘制爱因斯坦棋子，其中用到了qt提供的setPen、setBrush、drawEllipse、setFont、drawText等功能函数，绘制棋盘时我们主要用到drawLine函数绘制棋盘线，除此之外我们还用到了qt提供的窗口部件比如QPushButton、QLineEdit、QMenu等，具体效果参见后面的软件测试说明。在设计鼠标点击事件时，我们设计了getorder()实现掷骰子功能，用start\_clicked()实现开始游戏功能，还设计restart\_clicked()实现重玩功能，用mode1\_clicked()实现选择人人对战模式，用mode2\_clicked()实现选择人机对战模式，用about\_clicked()实现查看游戏规则功能，具体效果参见后面的软件测试说明。图形界面部分还要解决用数字坐标表示棋盘位置的问题，为此我们设计了center(int row, int col)、center(int id)、topLeft(int row, int col)、topLeft(int id)、cell(int row, int col)、cell(int id)等函数将棋盘上的位置转化为用数字表示的坐标。其中，为了获得相对而言更加美观的棋子视觉效果，我们经过多次修整后确定了棋子的半径为20。

4系统测试

4.1系统运行说明

程序经过构建之后，点击运行，弹出爱因斯坦的游戏界面：

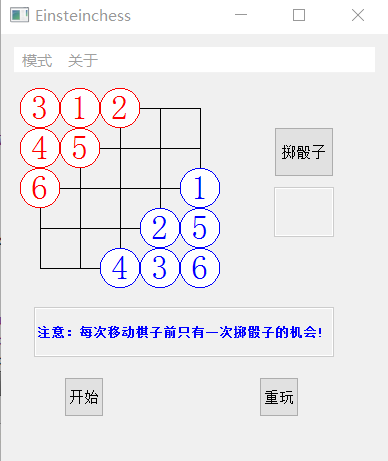


图1.游戏初始界面

最上面的菜单栏“模式”用来选择人人对战模式或者人机模式，点击按钮“关于”会弹出消息框告诉玩家游戏规则，如下图：

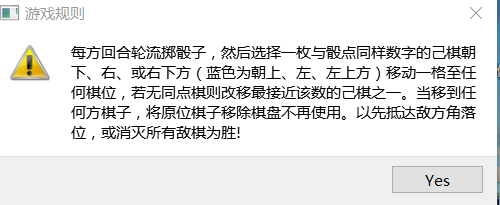


图2.点击“关于”按钮

“开始”按钮令游戏开始，但是点击“开始”按钮前需要首先选择游戏模式，如果没有选择游戏就点击“开始”按钮会弹出消息框，如下图：

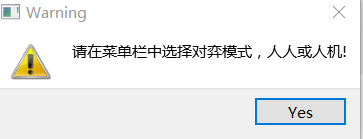


图3.点击“开始”按钮

“重玩”按钮用来将当前棋局终止，使游戏重新回到游戏初始界面。“掷骰子”按钮用来生成1-6的随机数，指示当前玩家可以走哪一个棋子。游戏初始界面骰子示数没有示数，即使点击“掷骰子”按钮也没有反应。只有游戏开始后，玩家才可以掷骰子，然后骰子的示数会显示在按钮下面的文本框中。文本框通过setEnabled(false)设置为不可编辑的，只能靠“掷骰子”获得示数。棋盘下面还有一个文本框，告诉玩家为了公平，每次移动棋子前只能掷一次骰子。

为了实现上述效果，我们为棋盘设置了两个bool型变量end和start来表示当前游戏进行的状态；两个int型变量ordercount和modechoose分别记录当前掷骰子的次数和选择的模式。对于“掷骰子”事件，必须满足游戏已经开始，模式已经选择，骰子第一次被掷出这三个条件才可以掷出骰子；对于“开始”按钮必须满足游戏是结束状态，游戏模式已经选择才可以启动游戏。

如图1所示，游戏初始界面中，红方蓝方的6枚棋子是以乱序的排列方式分别排在左上角和右下角的。在这一部分，我们利用了LuanXu(int\*a,int min,int max)实现，具体代码如下：

LuanXu(int\*a,int min,int max)

1. //遍历数组指定范围

2 For i=min to max-1

3 //获得指定范围内某个棋子的编号

4 r = min+qrand()%6

5 //赋值当前棋子

6 temp=a[i]

7 //交换两个棋子

8 a[i]=a[r]

9 a[r]=temp

如果游戏模式选择人人对战，游戏开始后默认红方先走，此时由红方掷骰子：

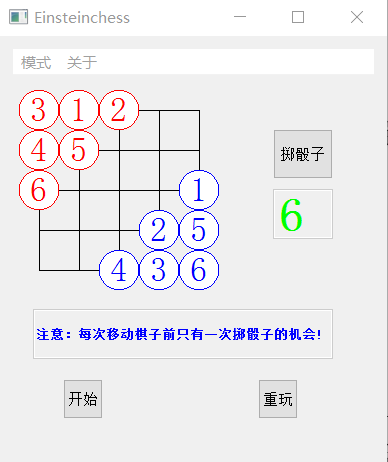


图4.红方掷出骰子

骰子投出数字6，此时红方应该移动6号棋子：

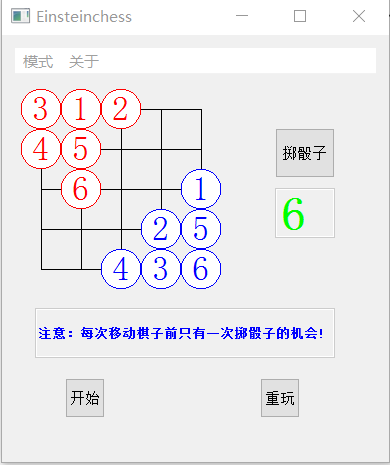


图5.走6号棋子

红方棋子走后由蓝方棋子掷骰子：

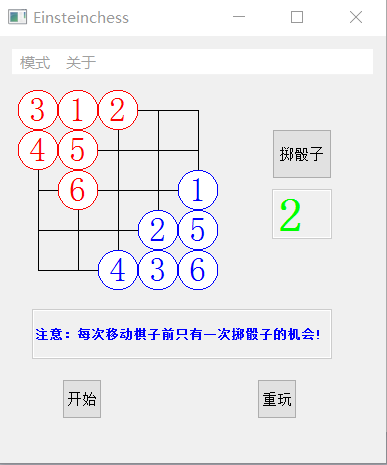


图6.蓝方掷骰子

此时蓝方掷骰子得到的数为2，所以蓝方应走2号棋：

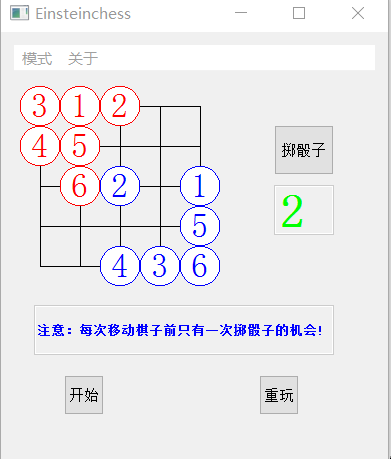


图7.蓝方走棋

如此循环往复直到决出胜负。

如果骰子掷出的数对应的棋子已经死亡。如图8所示，蓝方掷出骰子数为2，此时应当走2号棋，但是2号棋已经死亡，根据爱因斯坦棋的规则，只能选择与2最接近的棋子，蓝方此时共有4枚棋子，最接近2的是图中的1号棋，所以此时蓝方只能选择移动1号棋，其他棋不能移动。

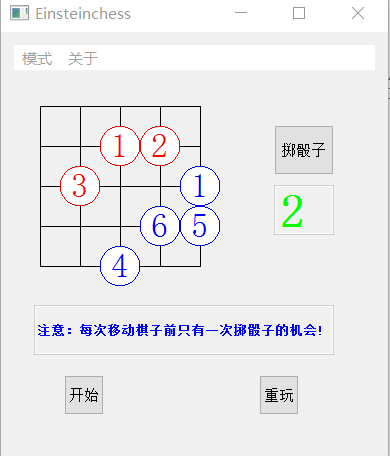


图8.骰子数指示的棋子已经死亡

游戏结束后会弹出消息框告诉玩家哪方取得了胜利，如下图：

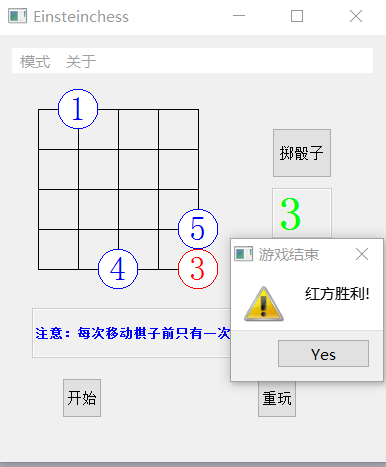


图9.红方胜利

从图中可以看出红方先于对方到达了敌方角落位，所以红方取得了胜利。此时点击“yes”后，游戏重新布局如图1所示。

假如我们选择了人机对战模式，游戏默认红方为真实玩家，蓝方为电脑，红方先走棋，骰子示数为2，2号棋向右方向走一格，杀死己方5号棋。由红方走出第一步棋后：

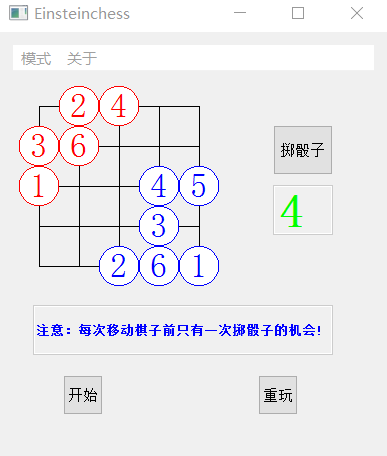


图10.人机模式

电脑自动掷骰子数为4，然后电脑利用前面所说的搜索算法进行缜密计算后，确定左上方为最优位置，然后将4号棋移动到目的位置。由于电脑的计算速度太快，我们几乎观察不到电脑的思考过程，只观察到电脑迅速走出了4号棋，这正是电脑的优势所在。

按照上述玩法循环往复，直到游戏结束。

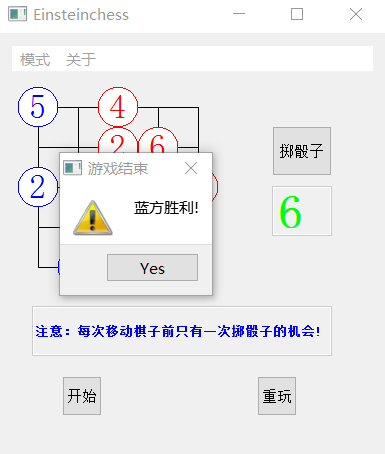


图11.游戏结束

按照爱因斯坦棋规则，由蓝方首先到达敌方角落位，所以蓝方最终获得了胜利。点击“yes”按钮，重新选择模式开始新的一局。

4.2系统测试总结

软件经过多次调试修改后，爱因斯坦棋的使用过程如4.1所示。

总体而言，爱因斯坦棋的运行界面符合我们的预期效果，软件运行过程与我们的设计思路一致。经过检测，软件各个功能完美实现，软件运行原理合理，没有出现异常情况。