卒業研究報告書

題目

Javaを用いた京都将棋AIの開発

指導教員

石水 隆 講師

報告者

15–1–037–0057

升田 夏美

近畿大学理工学部情報学科

平成31年1月31日提出

概要

京都将棋は 5×5 の盤面で「王」「香と」「銀角」「金桂」「飛歩」の 5 種類の駒を使い, 一手指すたびに駒を裏返すゲームである. 京都将棋の特徴は, 王以外は裏表にそれぞれ将棋の駒名が書かれており, 駒を打つ場合は裏表好きな方で打つことができることである.

本将棋では, 様々な将棋 AI が開発され, 今日ではプロ棋士に勝てるものまで出てきている. 一方, 京都将棋は知名度が低く, あまり研究されておらず, 有望な京都将棋 AI は存在しない. そこで本研究では, 強い京都将棋 AI の開発を目指す.

将棋の AI を作成する場合, 各駒に価値を割り当て, 敵味方の駒の価値の合計から局面の評価値を求める手法がよく用いられる. 本将棋では, プロ棋士により妥当と思われる駒の評価値はほぼ定まっているため, それを使用することができる. 一方, 京都将棋は本将棋とは駒の裏表が異なり, また盤面も 5×5 と小さいため, 本将棋の駒の評価値をそのまま使うことはできない. そこで本研究では, 京都将棋において最適となる駒の評価値を検証する. 本研究では, Java を用いて京都将棋 AI を作成し, 駒の評価値を変えながら京都将棋 AI を対局させ, 適切な評価値を求める.

目次

1 序論 1
  1.1 背景 1
  1.2 コンピュータ将棋の着手選択法 1
  1.3 京都将棋に関する既知の結果 1
  1.4 本報告書の構成 1

2 京都将棋とは 2
  2.1 京都将棋の概要 2
  2.2 京都将棋のルール 2

3 コンピュータ将棋の着手選択法 4
  3.1 局面の評価値 4
  3.2 Mini-Max 法 5

4 京都将棋プログラム 6

5 クラス分析 7
  5.1 Piece クラス 7
  5.2 Piece クラスのサブクラス 9
  5.3 Play クラス 9
  5.4 Position クラス 9

6 駒の評価値の検証 11
  6.1 評価方法 11
  6.2 検証 11

7 結論と今後の課題 14

付録 A ソースプログラム 17

1 序論

1.1 背景

京都将棋は, 香車の裏がと金, 銀将の裏が角行等, 裏と表に異なる駒が書かれた 5 種類の駒と 5 × 5 マスの盤面を使い, 駒を動かす度にその駒を裏返すという特別なルールがある将棋の一種である. 本将棋とは異なり駒を動かす度に役割が変わるため, 先を読むのが難しく, 見た目以上に奥の深いゲームであるが, 知名度は低い. 本将棋の AI はトップレベルに勝てるほど強い AI が存在しているが, 京都将棋は本将棋と比べマイナーであるため [2] 等いくつかの既存の AI はあるもののあまり強くはない. そこで本研究では, 京都将棋で強い AI を作成することを指す.

1.2 コンピュータ将棋の着手選択法

本節では, コンピュータ将棋の着手選択法について述べる. コンピュータ将棋が着手を決定するのに用いられている主な手法としては, 局面の評価値計算, 定跡データベースの利用, 一定手数の先読み, 終盤での詰み読み, モンテカルロ法, 機械学習等がある.

局面の評価値計算とはその局面が先手、後手にとってどれほど有利なのかを計算することである.

定跡データベースとは, プロ棋士などが指した棋譜を元に作成したデータベースである. 本将棋には, プロ棋士が長年構築してきた定跡があり, また, プロ棋士の棋譜は公開されているため, そこからデータベースを作成することができる. データベースに登録されている局面であれば, プロと同じ手を指すことができる. この手法の欠点は, 相手があえて定跡以外の手を指す, 新手を指すなどしてデータベースに無い局面に持ち込まれた場合に対応できないことである.

1.3 京都将棋に関する既知の結果

京都将棋の大会などは調べた限り存在しなかった.
株式会社ねこまどが 2015 年に制作したアプリが存在する.[2]

1.4 本報告書の構成

本報告書の構成は以下の通りである. まず第 2 章で, 本研究の対象である京都将棋について述べる.

2　京都将棋とは

本章では, 本研究の対象である京都将棋について説明する.

2.1　京都将棋の概要

京都将棋は 1976 年に田宮克哉が発案した.

京都将棋は 5 × 5 の盤面で「王」「香と」「銀角」「金桂」「飛歩」の 5 種類の駒を使い, 一手指すたびに駒を裏返すゲームである. 王以外は裏表にそれぞれ将棋の駒名が書かれており, 駒を打つ場合は裏表好きな方で打つことができる.

2.2　京都将棋のルール

本節では, 京都将棋のルールについて述べる.

図 1 に京都将棋の初期配置を示し, 以下に京都将棋のルールを示す.

5	4	3	2	1	
歩	金	王	銀	と	一
					二
					三
					四
と	銀	玉	金	歩	五

図 1　京都将棋の初期配置

- と銀王金歩の順に 5 種類の駒を並べる. 自陣, 敵陣はない.
- 駒の動かし方は通常の将棋と同じ.
- 「王」以外の全ての駒は一手指すごとにその駒を裏返す. (駒は一手ごとに動きが変わる)
- じゃんけんで順番を決め交互に駒を動かし, 先に相手の「王」を取ったほうが勝ち.
- 自分の駒のあるマスには進めない. 相手の駒のあるマスに進むとその駒を取ることができる. 取った駒は持ち駒という.
- 持ち駒は自分の手番に, 空いているマスならどこにでも打つことができる. どちらの面を上にしても良い.
- 打ち歩詰め, 二歩, 身動きの取れない駒などは禁止されていない.
- 千日手は引き分け.

1 手ごとに駒が変化するため, 京都将棋は本将棋の得意な人であっても上手に指すのは難しい. 図 2 に京都将棋の 3 手詰めの例を挙げる. この例の場合, ▲ 4 二香 (⇒と), △ 4 二同飛 (⇒歩), ▲ 3 二角 (⇒銀) までとなる. 2 手目に飛車を動かしたことにより, 歩になってしまい 3 二への利きが無くなるので詰む. 2 手目を△ 4 二同角 (⇒銀) としても同様に▲ 3 二角までである. このように駒が次々変化していくのが京都将棋の特徴である.

5	4	3	2	1	
角		王	香		一
飛				金	二
		金			三
角			飛		四
	香			玉	五

図 2　京都将棋の 3 手詰めの例

3　コンピュータ将棋の着手選択法

本章では, 本研究で作成した京都将棋 AI の着手選択法について述べる. 1.2 節で述べた通り, コンピュータ将棋は様々な手法を用いて着手選択している.

本将棋では, プロ棋士が構築してきた定跡がありそれを利用できるが, 京都将棋には競技人口も少なく, 定跡は確立していない. また, 各局面における駒同士の連携や玉の安全度などは, 本将棋では矢倉囲いや美濃囲いなど基本的な駒の配置や玉の守り方になっているかどうかで判断できるが, 京都将棋ではどのような配置が良いか明確ではない.

そこで本研究では, 各駒に価値を割り振り, その価値の合計から局面の優劣を評価する手法を取る.

3.1　局面の評価値

本節では, 本研究で作成する京都将棋 AI で用いている局面の評価値について述べる.

本研究で作成する京都将棋 AI は以下の項目から局面の評価値を求める.

- 駒の価値
- 駒の稼動範囲から値

本研究では, 各駒に価値を割り当て, 先手の駒なら価値を正の値, 後手の駒なら価値を負の値として駒の価値を合計したものを局面の評価値として用い, 評価値が正なら先手有利な局面, 負なら後手有利な局面として局面とみなす.

本将棋では, 表 3 に示す例のように, プロ棋士により各駒の評価値はほぼ定まっている. 一方, 京都将棋は駒の裏表が本将棋と異なり, また, 盤面も本将棋より狭いため, 本将棋の駒の価値をそのまま用いることはできない. そのため, 京都将棋に適切な駒の評価値を求める必要がある.

駒の稼動範囲は

5	4	3	2	1	
角		王	香		一
飛				金	二
		金			三
角			飛		四
	香			玉	五

図 3　駒の可動範囲

例として, 図 3 では動ける場所は 3 三金は 6 箇所, 1 二金は 4 箇所, 5 一の角 2 箇所, 5 四の角 3 箇所, というふうに評価している.

表 1　本将棋の駒の評価値の例 [1]

歩	香	桂	銀	金	角	飛	玉
100	600	700	1000	1200	1800	2000	∞
と	杏	圭	全		馬	竜	玉
1200	1200	1200	1200		2000	2200	

3.2　Mini-Max 法

現時点で評価値が高い局面でも, 数手先で不利になる場合もある. よって, 局面を評価するためには, 現在の局面だけではなく, 数手先の局面も考慮しなければならない.

本研究では, 最善手の選択アルゴリズムは min-max 法を用いた.

図 4 は, 3 手先まで読む例を示している. まず最初に今指せる手（1 手目）を全て展開する. 図では, それが 2 通りだった場合である. 次にその指し手に対して, 相手が指せる手（2 手目）を全て展開する. さらに, その後の指し手（3 手目）を展開する. 図では, それぞれ 2 通りの指し手がある場合を示している. その後, 3 手目を指した全ての局面の評価値を求める. この評価値は図では赤で示されている. この場合,10,3,7,5,2,8,3,6 がそれぞれ同じ局面の分岐（親）になっている. 3 手目は先手の手番であるため, それぞれの中の最大値（max）を選び, 親の評価値とする. 図では, この値は緑で示されている. ここで, 同様に 10,7,8,6 が同じ親を持っている. かつ、2 手目は後手ですから、この中の最小値（min）を選んで親の評価値とする. 同様に 1 手目は 7,6 で先手であるため, 最大の 7 の評価値の手が選ばれる. 一番高いあるいは低い評価値を持つ手が複数存在することがあり得るが, その時はその中からランダムに手を選択するものとする.

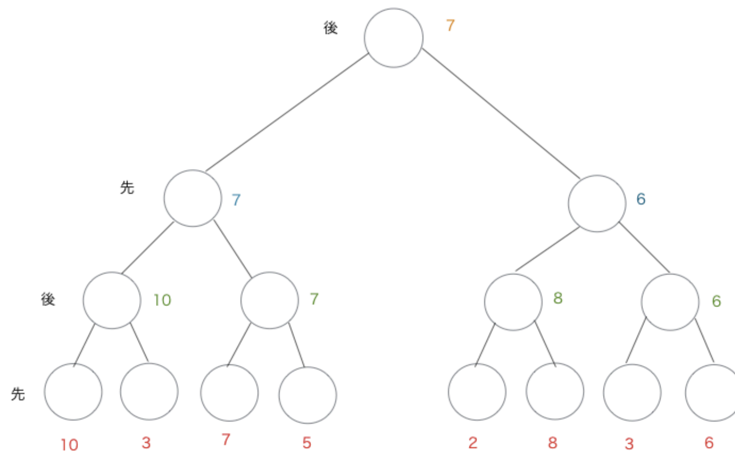

図 4　ミニマックス法

4 京都将棋プログラム

本章では本研究で作成した京都将棋のアプリケーションについて説明する.

本研究では [1] の将棋プログラムを参考にし, Java を用いて京都将棋プログラムを作成した. 付録に本研究で作成したプログラムのソースコードを示す.

本研究で作成した京都将棋プログラムは, 各駒に価値を割り当て, その価値の合計から局面の評価値を求める. そして Mini-Max 法により∗手先の局面まで先読みすることで着手選択している.

以下に本研究で作成した京都将棋プログラムの各クラスについて説明する.

5　クラス分析

5.1　Piece クラス

駒の抽象クラス

フィールド

isBlack boolean 型: true:先手の駒 false:後手の駒
name char 型:駒の名前
number int 型:駒の番号 (持ち駒の表示順序)
pieceValue int 型:駒の価値
pieceValue2 int 型:駒の価値 2
walk int 型:歩いて行ける場所
run int 型:走って行ける場所

メソッド

isBlack()boolean 型:先手の駒か
isWhite()boolean 型:後手の駒か
getName()char 型:駒の名前
getNumber()int 型:駒の番号
getPieceValue()int:駒の価値
getPieceValue2()int:駒の価値 2
notMove(int r, int c)boolean 型:動けない駒か (デフォルトで動ける)
checkPosition(int r, int c)boolean 型:r 行,c 列を与え場所が盤面内か調べる
addNextPositionByWalk
(Position position:現在の盤面,
int r:駒の行,
int c:駒の列,
int[][] walk:歩いて進める方向,
ArrayList<Position> positions:動いた後の盤面リスト)void 型:歩いて進む
addNextPositionByRun
(Position position:現在の盤面,
int r:駒の行,
int c:駒の列,
int[][] run:走って進める方向,
ArrayList<Position> positions:動いた後の盤面リスト)void 型:走って進む
moveCountByWalk
(Position position:現在の盤面,
int r:駒の行,
int c:駒の列,

int[][] walk:歩いて進める方向)int 型:歩いて進める場所の数

moveCountByRun

(Position position:現在の盤面,

int r:駒の行,

int c:駒の列,

int[][] run:走って進める場所の数)int 型:走って進める場所の数

canWalk

(Position position, 現在の盤面

int r, 駒の行

int c, 駒の列

int newR, 行き先の行

int newC, 行き先の列

int[][] walk)boolean 型:歩いて行ける場所かどうかのチェック

canRun

(Position position:現在の盤面,

int r:駒の行,

int c:駒の列,

int newR:行き先の行,

int newC:行き先の列,

int[][] run:走って進める方向)boolean 型:走って行ける場所かどうかのチェック

addNextPosition

(Position position:現在の盤面,

int r:駒の行,

int c:駒の列,

ArrayList<Position> positions)void 型:駒を動かした時の盤面リスト

moveCount

(Position position:現在の盤面,

int r:駒の行,

int c:駒の列)int 型:駒が動ける場所の数

canMove

(Position position:現在の盤面,

int r:動かす元の行（-1 なら持ち駒）,

int c:動かす元の列（持ち駒の場合は持ち駒の番号）,

int newR:動かす（打つ）行,

int newC:動かす（打つ）列)boolean 型:可能な着手かどうかのチェック

　コンストラクタ

Piece reverse()abstract 型:ひっくり返した時の駒

Piece originalPiece()abstract 型:表面の駒

5.2　Piece クラスのサブクラス

Fu クラス :「歩」を表す
Gin クラス:「銀」を表す
Gyok クラス:「玉」を表す
Hi クラス:「飛」を表す
Kak クラス:「角」を表す
Kei クラス:「桂」を表す
Kin クラス:「金」を表す
Kyo クラス:「香」を表す
To クラス:「と」を表す

5.3　Play クラス

メインメソッド
main(String[] args)void 型:対戦させる

5.4　Position クラス

盤面状態を表すクラス

フィールド
turn boolean 型: true:先手番、false:後手番
boad 盤
blackPocket 先手の持ち駒
whitePocket 後手の持ち駒
lastMoveR int 型:最後に動いた行
lastMoveC int 型:最後に動いた列
positionValue int 型:盤面の価値
turnNumber int 型:手数
depth int 型:読みの深さ
random Random 型:乱数
moveValue int 型:動ける場所の数に対する重み

コンストラクタ
Position(int depth:読みの深さ):初期盤面の設定をする
Position(Position oldPosition:前の盤面):前の盤面をコピーする
move

(int oldR:元の行,
int oldC:元の列,
int newR:動いた後の行,
int newC:動いた後の列):駒を動かす
drop
(int h:何番目の持ち駒を打つか,
boolean r:表で打つか裏で打つか,
int newR:打つ行,
int newC:打つ列):駒を打つ
bestMove():最善手を求める

　メソッド
getTurn()boolean 型:どちらの手番か
Piece getPiece
(int r:行,
int c:列)boolean 型:盤のマス目にある駒を得る (なければ null)
getLastMoveR()int 型:最後に動いた行を得る
getLastMoveC()int 型:最後に動いた列を得る
getBlackPocket()ArrayList<Piece>:先手の持ち駒を返す
getWhitePocket()ArrayList<Piece>:後手の持ち駒を返す
getPositionValue()int 型:盤面の価値を返す
canDrop
(int p:何番目の持ち駒か,
int newR:打つ行,
int newC:打つ列)boolean 型:駒が打てるかどうかのチェックをする
addHand
(ArrayList<Piece> hand:先手か後手の持ち駒リスト,
Piece piece:加える駒)void 型:持ち駒に加える
printBoad()void 型:盤面の表示
printLine()void 型:盤面の横線を引く
nextMoves()ArrayList<Position>:次に可能な盤面リストを求める
value()int 型:一番深く読んだ時の盤面の価値を求める
getPieceValue
(int r:駒の行,
int c:駒の列,
boolean t:手番)int 型:駒の価値を求める
value(int d:読みの深さ)int 型:盤面の価値を求める

6　駒の評価値の検証

6.1　評価方法

3.1 節で述べた通り, 本研究で実装京都将棋 AI は, 評価関数として次の項目を用いている.

- 駒の価値の和を増やす
- 駒の稼動範囲

通常の将棋ではそれぞれの駒に強弱の差があるが, 京都将棋では表と裏は基本的に強い駒と弱い駒の組み合わせになっている（飛車と歩兵等）ため, 適切な駒の評価値を定めるのが困難である．そこで本研究では, 駒の評価値を変えながら京都将棋 AI 同士を対戦させ, 適切な評価値を探す. ユーザインタフェースは今回は CUI で実装しており, 図 1 の実行例のように表示される．なお, 敵味方の駒, 及び最終着手は色で区別がつくようにしている.

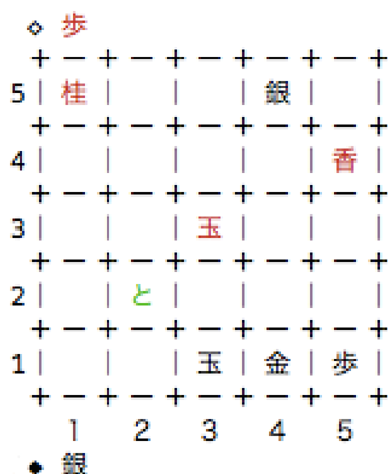

図 5　実行の様子

6.2　検証

先手, 後手のどれか一つの駒の値を 150, その他を 100 にし (-は全て 100), 各 AI 同士を 20 回対戦させたときの先手の勝数を表 3 に示す.

勝率 p の勝負を N 回行った場合, 標準偏差は以下の式で与えられる.

$$\sqrt{N \cdot p \cdot (1-p)}$$

$p = 0.5$ と仮定した場合, $N = 20$ なら標準偏差は

$$\sqrt{20 \cdot 0.5 \cdot 0.5} = 2.24$$

となる．したがって, 試行回数 20 回の場合. 危険率 95% の信頼区間に含まれる勝数は 5.6〜14.4 である．表 3

表 2　対戦結果（先手の勝数、試行回数 20 回）

先手＼後手	-	香と	銀角	金桂	飛歩	計
-	9	10	*4*	8	14	45
香と	7	6	*0*	9	*17*	39
銀角	9	12	*3*	6	6	36
金桂	11	*4*	14	9	*17*	55
飛歩	*0*	12	*0*	*2*	*15*	29
計	36	44	21	34	69	204

において有意差がある部分はイタリックになっている. また, 先手の勝数が 500 回中 204 回と有意に少ないこともわかった.

対角線上で折り返したら重なる場所, 例として先手香と, 後手香とを重要視した場合, 先手香と 454, 後手香と 476 と勝った数と負けた数がほぼ一致したため, 駒の差は見られないが, 先手の時は飛歩, 後手の時は金桂を重視する結果となった.

表 3　対戦結果（先手の勝数、試行回数 1000 回）

先手＼後手	-	香と	銀角	金桂	飛歩	計
-	407	476	440	388	458	2169
香と	454	488	450	452	520	2364
銀角	434	463	491	392	473	2254
金桂	354	434	388	327	438	1941
飛歩	478	526	472	422	525	2423
計	2127	2387	2241	1981	2412	2236

7　結論と今後の課題

本研究で京都将棋 AI の実装ができ, 実際に人間との対局が可能になった. 評価関数として, 各駒に付与した値を変化させながら AI 同士で対局させた.

結論としては, 先手後手に依存しそうだが, 先手後手を考えなければどの駒も差はなさそうである. しかし, 先手の時, 後手の時に強い駒が存在する可能性がある.

今後の課題としては, 他の AI と対局させること, もっと有効な数字を探すこと, またインターフェースを GUI に変更することも必要である.

謝辞

本研究にあたり, 近畿大学理工学部情報学科情報論理工学研究室の石水隆講師には, 大変お世話になりました. この場を借りて感謝を申し上げます.

参考文献

[1] 池 泰弘 :Java 将棋のアルゴリズム, 工学社 (2007)

[2] 京都将棋, Nekomado Co.Ltd, (2015), https://itunes.apple.com/jp/app/京都将棋/id1037596970?mt=8

[3] 将棋ゲームの時間, (2012) https://syouginojikan.web.fc2.com/kyouto.html

[4] 京都将棋, 株式会社幻冬舎エデュケーション, (2014)

付録 A　ソースプログラム

本研究で作成した京都将棋プログラムのソースを以下に示す.

- Fu クラス

```
/**
 * @author   NATSUMI MASUDA
 */

package kyotosyogi2;

/**
 * 歩
 *
 */

public class Fu extends Piece {
        private int[][] walkl = { { 1, 0 } }; // 前にだけ歩く

        public Fu(boolean isBlack) {
                this.isBlack = isBlack;
                this.name = '歩';
                this.number = 1;
                this.pieceValue = 100;
                this.pieceValue2 = 150;
                walk = walkl;
        }

        public Piece reverse() {
                return new Hi(isBlack);
        }

        public Piece originalPiece() {
                return new Fu(!isBlack);
        }

        public boolean notMove(int r, int c) {
                return (isBlack && r == 4) || (!isBlack && r == 0);
        }
```

- Gin クラス

```
/**
 * @author   NATSUMI MASUDA
 */

package kyotosyogi2;
```

```
/**
 * 銀
 *
 */

public class Gin extends Piece {
        private int[][] walkl = { { 1, -1 }, { 1, 0 }, { 1, 1 }, { -1, -1 }, { -1, 1 } }; // 斜
め４方向と前に歩く

        public Gin(boolean isBlack) {
                this.isBlack = isBlack;
                this.name = '銀';
                this.number = 7;
                this.pieceValue = 150;
                this.pieceValue2 = 100;
                walk = walkl;
        }

        public Piece reverse() {
                return new Kak(isBlack);
        }

        public Piece originalPiece() {
                return new Gin(!isBlack);
        }
```

- Gyok クラス

```
/**
 * @author   NATSUMI MASUDA
 */

package kyotosyogi2;

/**
 * 玉
 *
 */

public class Gyok extends Piece {
        private int[][] walkl = { { 1, -1 }, { 1, 0 }, { 1, 1 }, { 0, -1 }, { 0, 1 }, { -1, -1 }, { -1, 0 }, {
方向に歩く

        public Gyok(boolean isBlack) {
                this.isBlack = isBlack;
                this.name = '玉';
                this.number = 0;
                this.pieceValue = 10000;
```

```
		this.pieceValue2 = 10000;
		walk = walkl;
	}

	public Piece reverse() {
		return new Gyok(isBlack);
	}

	public Piece originalPiece() {
		return new Gyok(!isBlack);
	}
```

- Hi クラス

```
/**
 * @author   NATSUMI MASUDA
 */

package kyotosyogi2;

/**
 * 飛
 *
 */

public class Hi extends Piece {
	private int runl[][] = { { 1, 0 }, { 0, -1 }, { 0, 1 }, { -1, 0 } }; // 縦横に走る

	public Hi(boolean isBlack) {
		this.isBlack = isBlack;
		this.name = '飛';
		this.number = 2;
		this.pieceValue = 100;
		this.pieceValue2 = 150;
		run = runl;
	}

	public Piece reverse() {
		return new Fu(isBlack);
	}

	public Piece originalPiece() {
		return new Fu(!isBlack);
	}
```

- Kak クラス

```
/**
 * @author   NATSUMI MASUDA
 */
```

```
package kyotosyogi2;

/**
 * 角
 *
 */

public class Kak extends Piece {
        private int runl[][] = { { 1, -1 }, { 1, 1 }, { -1, -1 }, { -1, 1 } }; // 斜め４方に走る

        public Kak(boolean isBlack) {
                this.isBlack = isBlack;
                this.name = '角';
                this.number = 8;
                this.pieceValue = 150;
                this.pieceValue2 = 100;
                run = runl;
        }

        public Piece reverse() {
                return new Gin(isBlack);
        }

        public Piece originalPiece() {
                return new Gin(!isBlack);
        }
```

- Kei クラス

```
/**
 * @author   NATSUMI MASUDA
 */

package kyotosyogi2;

/**
 * 桂
 *
 */

public class Kei extends Piece {
        private int[][] walkl = { { 2, -1 }, { 2, 1 } }; // 前方からさらに斜めに歩く

        public Kei(boolean isBlack) {
                this.isBlack = isBlack;
                this.name = '桂';
                this.number = 6;
                this.pieceValue = 100;
```

```
                this.pieceValue2 = 100;
                walk = walkl;
        }

        public Piece reverse() {
                return new Kin(isBlack);
        }

        public Piece originalPiece() {
                return new Kin(!isBlack);
        }

        public boolean notMove(int r, int c) {
                return (isBlack && r >= 3) || (!isBlack && r >= 1);
        }
```

- Kin クラス

```
/**
 * @author   NATSUMI MASUDA
 */

package kyotosyogi2;

/**
 * 金
 *
 */

public class Kin extends Piece {
        private int[][] walkl = { { 1, -1 }, { 1, 0 }, { 1, 1 }, { 0, -1 }, { 0, 1 }, { -1, 0 } }; // 斜
め後ろ以外に歩く

        public Kin(boolean isBlack) {
                this.isBlack = isBlack;
                this.name = '金';
                this.number = 5;
                this.pieceValue = 100;
                this.pieceValue2 = 100;
                walk = walkl;
        }

        public Piece reverse() {
                return new Kei(isBlack);
        }

        public Piece originalPiece() {
                return new Kin(!isBlack);
        }
```

- Kyo クラス

```
/**
 * @author   NATSUMI MASUDA
 */

package kyotosyogi2;

/**
 * 香
 *
 */

public class Kyo extends Piece {
        private int runl[][] = { { 1, 0 } }; // 前に走る

        public Kyo(boolean isBlack) {
                this.isBlack = isBlack;
                this.name = '香';
                this.number = 3;
                this.pieceValue = 100;
                this.pieceValue2 = 100;
                run = runl;
        }

        public Piece reverse() {
                return new To(isBlack);
        }

        public Piece originalPiece() {
                return new Kyo(!isBlack);
        }

        public boolean notMove(int r, int c) {
                return (isBlack && r == 4) || (!isBlack && r == 0);
        }
```

- Piece クラス

```
/**
 * @author   NATSUMI MASUDA
 */
package kyotosyogi2;
import java.util.ArrayList;

/**
 * 駒の抽象クラス
 *
 */
```

```
public abstract class Piece {
        protected boolean isBlack; // true:先手の駒   false:後手の駒
        protected char name; // 駒の名前
        protected int number; // 駒の番号（持ち駒の表示順序）
        protected int pieceValue; // 駒の価値
        protected int pieceValue2; // 駒の価値２
        protected int[][] walk = {};
        protected int[][] run = {};

        /**
         * 先手の駒か？
         * @return
         */
        public boolean isBlack() {
                return isBlack;
        }

        /**
         * 後手の駒か？
         * @return
         */
        public boolean isWhite() {
                return !isBlack;
        }

        /**
         * 駒の名前
         * @return
         */
        public char getName() {
                return name;
        }

        /**
         * 駒の番号
         * @return
         */
        public int getNumber() {
                return number;
        }

        /**
         * 駒の価値
         * @return
         */
        public int getPieceValue() {
                return pieceValue;
        }
```

```
/**
 * 駒の価値 2
 * @return
 */
public int getPieceValue2() {
        return pieceValue2;
}

/**
 * 動けない駒か（デフォルトで動ける）
 * @return
 */
public boolean notMove(int r, int c) {
        return false;
}

/**
 * 場所が盤面内か？
 * @param r     行
 * @param c     列
 * @return
 */
public boolean checkPosition(int r, int c) {
        return (r >= 0 && r < 5 && c >= 0 && c < 5);
}

/**
 * 歩いて進む
 * @param position      現在の盤面
 * @param r     駒の行
 * @param c     駒の列
 * @param walk  歩いて進める方向
 * @param positions     動いた後の盤面リスト
 */
public void addNextPositionByWalk(Position position, int r, int c, int[][] walk, ArrayList<Position> po
        int newR;
        int newC;

        for (int i = 0; i < walk.length; i++) {
                if (isBlack) {
                        newR = r + walk[i][0];
                        newC = c + walk[i][1];
                } else {
                        newR = r - walk[i][0];
                        newC = c - walk[i][1];
                }
                if (checkPosition(newR, newC)) {
                        if (position.getPiece(newR, newC) == null) {
                                positions.add(position.move(r, c, newR, newC));
```

```
                        } else if (position.getPiece(newR, newC).isBlack() != isBlack) {
                                positions.add(position.move(r, c, newR, newC));
                        }
                }
        }
}

/**
 * 走って進む
 * @param position      現在の盤面
 * @param r     駒の行
 * @param c     駒の列
 * @param run   走って進める方向
 * @param positions     動いた後の盤面リスト
 */
public void addNextPositionByRun(Position position, int r, int c, int[][] run, ArrayList<Position> posi
        int newR;
        int newC;

        for (int i = 0; i < run.length; i++) {
                for (int j = 0; j < 4; j++) {
                        if (isBlack) {
                                newR = r + (run[i][0]) * (j + 1);
                                newC = c + run[i][1] * (j + 1);
                        } else {
                                newR = r - (run[i][0]) * (j + 1);
                                newC = c - run[i][1] * (j + 1);
                        }
                        if (checkPosition(newR, newC)) {
                                if (position.getPiece(newR, newC) == null) { // マスが空
いていたら進める

                                        positions.add(position.move(r, c, newR, newC));
                                } else if (position.getPiece(newR, newC).isBlack() != isBlack) { // 相
手の駒があれば

                                        positions.add(position.move(r, c, newR, newC));
                                        break; // 駒を取ったら終わり
                                } else {
                                        break; // 味方の駒があれば終わり
                                }
                        }
                }
        }
}

/**
 * 歩いて進める場所の数
 * @param position      現在の盤面
 * @param r     駒の行
 * @param c     駒の列
```

```
 * @param walk 歩いて進める方向
 * @return
 */
public int moveCountByWalk(Position position, int r, int c, int[][] walk) {
        int newR;
        int newC;
        int ret = 0;

        for (int i = 0; i < walk.length; i++) {
                if (isBlack) {
                        newR = r + walk[i][0];
                        newC = c + walk[i][1];
                } else {
                        newR = r - walk[i][0];
                        newC = c - walk[i][1];
                }
                if (checkPosition(newR, newC)) {
                        if (position.getPiece(newR, newC) == null) {
                                ret++;
                        } else if (position.getPiece(newR, newC).isBlack() != isBlack) {
                                ret++;
                        }
                }
        }
        return ret;
}

/**
 * 走って進める場所の数
 * @param position      現在の盤面
 * @param r     駒の行
 * @param c     駒の列
 * @param run   走って進める方向
 */
public int moveCountByRun(Position position, int r, int c, int[][] run) {
        int newR;
        int newC;
        int ret = 0;

        for (int i = 0; i < run.length; i++) {
                for (int j = 0; j < 4; j++) {
                        if (isBlack) {
                                newR = r + run[i][0] * (j + 1);
                                newC = c + (run[i][1]) * (j + 1);
                        } else {
                                newR = r - run[i][0] * (j + 1);
                                newC = c - (run[i][1]) * (j + 1);
                        }
                        if (checkPosition(newR, newC)) {
```

```
                                        if (position.getPiece(newR, newC) == null) {
                                                ret++;
                                        } else if (position.getPiece(newR, newC).isBlack() != isBlack) {
                                                ret++;
                                                break;
                                        } else {
                                                break;
                                        }
                                }
                        }
                }
                return ret;
        }

        /**
         * 歩いて行ける場所かどうかのチェック
         * @param position      現在の盤面
         * @param r     駒の行
         * @param c     駒の列
         * @param newR 行き先の行
         * @param newC 行き先の列
         * @param walk 歩いて進める方向
         * @return
         */
        public boolean canWalk(Position position, int r, int c, int newR, int newC, int[][] walk) {
                boolean ret = false;
                if (!checkPosition(newR, newC)) { // 番外に行こうとした
                        return ret;
                }
                if (position.getPiece(newR, newC) != null) {
                        if (position.getPiece(newR, newC).isBlack == isBlack) { // 自分の駒がある
場所に行こうとした
                                return ret;
                        }
                }
                for (int i = 0; i < walk.length; i++) {
                        if (isBlack) {
                                if (newR == r + walk[i][0] && newC == c + walk[i][1]) {
                                        ret = true;
                                        break;
                                }
                        } else {
                                if (newR == r - walk[i][0] && newC == c - walk[i][1]) {
                                        ret = true;
                                        break;
                                }
                        }
                }
                return ret;
```

```
	}

	/**
	 * 走って行ける場所かどうかのチェック
	 * @param position 現在の盤面
	 * @param r     駒の行
	 * @param c     駒の列
	 * @param newR 行き先の行
	 * @param newC 行き先の列
	 * @param walk 走って進める方向
	 * @return
	 */
	public boolean canRun(Position position, int r, int c, int newR, int newC, int[][] run) {
		boolean ret = false;
		if (!checkPosition(newR, newC)) { // 番外に行こうとした
			return ret;
		}
		if (position.getPiece(newR, newC) != null) {
			if (position.getPiece(newR, newC).isBlack == isBlack) { // 自分の駒がある
場所に行こうとした
				return ret;
			}
		}
		for (int i = 0; i < run.length; i++) {
			for (int j = 0; j < 4; j++) {
				int pr;
				int pc;
				if (isBlack) {
					pr = r + run[i][0] * (j + 1);
					pc = c + (run[i][1]) * (j + 1);
				} else {
					pr = r - run[i][0] * (j + 1);
					pc = c - (run[i][1]) * (j + 1);
				}
				if (!checkPosition(pr, pc)) {
					break;
				}
				if (position.getPiece(pr, pc) == null) {
					if (newR == pr && newC == pc) {
						return true;
					}
				} else if (position.getPiece(pr, pc).isBlack != isBlack) {
					if (newR == pr && newC == pc) {
						return true;
					} else {
						break;
					}
				} else {
					break;
```

```
                    }
                }
            }
            return ret;
        }

        /**
         * 駒を動かした時の盤面リスト
         * @param position      現在の盤面
         * @param r     駒の行
         * @param c     駒の列
         * @param positions
         */
        public void addNextPosition(Position position, int r, int c, ArrayList<Position> positions) {
                addNextPositionByWalk(position, r, c, walk, positions);
                addNextPositionByRun(position, r, c, run, positions);
        }

        /**
         * 駒が動ける場所の数
         * @param position      現在の盤面
         * @param r     駒の行
         * @param c     駒の列
         */
        public int moveCount(Position position, int r, int c) {
                return moveCountByWalk(position, r, c, walk) + moveCountByRun(position, r, c, run);
        }

        /**
         * 可能な着手かどうかのチェック
         * @param r 動かす元の行（-1 なら持ち駒）
         * @param c 動かす元の列（持ち駒の場合は持ち駒の番号）
         * @param newR 動かす（打つ）行
         * @param newC  動かす（打つ）列
         * @return
         */
        public boolean canMove(Position position, int r, int c, int newR, int newC) {
                return canWalk(position,r, c, newR, newC, walk) || canRun(position,r, c, newR, newC, run);
        }

        /**
         * ひっくり返した時の駒
         * @return
         */
        public abstract Piece reverse();

        /**
         * 表面の駒
         * @return
```

```
         */
        public abstract Piece originalPiece();
```

- Play クラス

```
/**
 * @author   NATSUMI MASUDA
 */
package kyotosyogi2;

import java.util.Scanner;

public class Play {

        public static void main(String[] args) {
                Position p = new Position(2);
                Position tmpP;
                Scanner kbd = new Scanner(System.in);
                int r, c, newR, newC;
                while (true) {

                        /* コンピューター */
                        p = p.bestMove();
                        p.printBoad();
                        int v = p.getPositionValue();
                        System.out.printf("### %d ###\n", v);
                        if (v > 15000 || v < -15000) {
                                break;
                        }
                        /* ここまで     */

                        /* 人間
                        while (true) {
                                System.out.print("Please intput next move:");
                                r = kbd.nextInt();
                                c = kbd.nextInt();
                                newR = kbd.nextInt();
                                newC = kbd.nextInt();
                                tmpP = p.humanMove(r, c, newR, newC);
                                if (tmpP != null) {
                                        p = tmpP;
                                        break;
                                }
                        } */
                        p.printBoad();
                        v = p.getPositionValue();
                        System.out.printf("### %d ###\n", v);
                        if (v > 15000 || v < -15000) {
                                break;
```

```
                }
                /* ここまで     */
            }
            kbd.close();
        }
```

- Position クラス

```
/**
 * @author   NATSUMI MASUDA
 */

package kyotosyogi2;
import java.util.ArrayList;
import java.util.Random;

/**
 * 盤面状態を表すクラス
 *
 */

public class Position {
        private boolean turn; // true:先手番、false:後手番
        private Piece[][] boad; // 盤（5^^e2^^9c^^96^^ef^^b8^^8f 5）
        private ArrayList<Piece> blackPocket; // 先手の持ち駒
        private ArrayList<Piece> whitePocket; // 後手の持ち駒
        private int lastMoveR; // 最後に動いた行
        private int lastMoveC; // 最後に動いた列
        private int positionValue; // 盤面の価値

        private static int turnNumber = 0; // 手数
        private static int depth; // 読みの深さ
        private static Random random = new Random();
        private final int moveValue = 1; // 動ける場所の数に対する重み

        /**
         * 初期盤面の設定
         */
        public Position(int depth) {
                boad = new Piece[5][5];
                for (int i = 0; i < 5; i++) {
                        for (int j = 0; j < 5; j++) {
                                boad[i][j] = null;
                        }
                }
                boad[0][0] = new To(true);
                boad[0][1] = new Gin(true);
                boad[0][2] = new Gyok(true);
                boad[0][3] = new Kin(true);
```

```
        boad[0][4] = new Fu(true);
        boad[4][4] = new To(false);
        boad[4][3] = new Gin(false);
        boad[4][2] = new Gyok(false);
        boad[4][1] = new Kin(false);
        boad[4][0] = new Fu(false);
        blackPocket = new ArrayList<Piece>();
        whitePocket = new ArrayList<Piece>();
        turn = true;
        Position.depth = depth;
}

/**
 * 前の盤面をコピーする
 * @param oldPosition 前の盤面
 */
public Position(Position oldPosition) {
        turn = !(oldPosition.turn);
        boad = new Piece[5][5];
        for (int i = 0; i < 5; i++) {
                for (int j = 0; j < 5; j++) {
                        boad[i][j] = oldPosition.boad[i][j];
                }
        }
        blackPocket = new ArrayList<Piece>(oldPosition.blackPocket);
        whitePocket = new ArrayList<Piece>(oldPosition.whitePocket);
}

/**
 * どちらの手番か？
 * @return
 */
public boolean getTurn() {
        return turn;
}

/**
 * 盤のマス目にある駒を得る（なければ null）
 * @param r 行
 * @param c 列
 * @return
 */
public Piece getPiece(int r, int c) {
        return boad[r][c];
}

/**
 * 最後に動いた行を得る
 * @return
```

```
 */
public int getLastMoveR() {
        return lastMoveR;
}

/**
 * 最後に動いた列を得る
 * @return
 */
public int getLastMoveC() {
        return lastMoveC;
}

/**
 * 先手の持ち駒を返す
 * @return
 */
public ArrayList<Piece> getBlackPocket() {
        return blackPocket;
}

/**
 * 後手の持ち駒を返す
 * @return
 */
public ArrayList<Piece> getWhitePocket() {
        return whitePocket;
}

/**
 * 盤面の価値を返す
 * @return
 */
public int getPositionValue() {
        return positionValue;
}

/**
 * 駒が打てるかどうかのチェック
 * @param position 現在の盤面
 * @param p 何番目の持ち駒か
 * @param newR 打つ行
 * @param newC 打つ列
 * @return
 */
public boolean canDrop(int p, int newR, int newC) {
        if (boad[newR][newC] != null) {
                return false;
        } else {
```

```
				if (turn) {
					return p < blackPocket.size();
				} else {
					return p < whitePocket.size();
				}
		}
	}

	/**
	 * 駒を動かす
	 * @param oldR 元の行
	 * @param oldC 元の列
	 * @param newR 動いた後の行
	 * @param newC 動いた後の列
	 * @return
	 */
	public Position move(int oldR, int oldC, int newR, int newC) {
		Position newPosition = new Position(this);
		newPosition.lastMoveR = newR;
		newPosition.lastMoveC = newC;
		if (boad[newR][newC] != null) {
			if (turn) {
				newPosition.blackPocket.add(boad[newR][newC].originalPiece());
			} else {
				newPosition.whitePocket.add(boad[newR][newC].originalPiece());
			}
		}
		newPosition.boad[newR][newC] = boad[oldR][oldC].reverse();
		newPosition.boad[oldR][oldC] = null;
		newPosition.turn = !turn;
		return newPosition;
	}

	/**
	 * 駒を打つ
	 * @param h 何番目の持ち駒を打つか
	 * @param r 表で打つか裏で打つか
	 * @param newR 打つ行
	 * @param newC 打つ列
	 * @return
	 */
	public Position drop(int h, boolean r, int newR, int newC) {
		Position newPosition = new Position(this);
		newPosition.lastMoveR = newR;
		newPosition.lastMoveC = newC;
		Piece piece;
		if (turn) {
			piece = newPosition.blackPocket.remove(h);
		} else {
```

```
                piece = newPosition.whitePocket.remove(h);
        }
        if (!r) {
                piece = piece.reverse();
        }
        newPosition.boad[newR][newC] = piece;
        newPosition.turn = !turn;
        return newPosition;
}

/**
 * 持ち駒に加える
 * @param hand 先手か後手の持ち駒リスト
 * @param piece 加える駒
 */
public void addHand(ArrayList<Piece> hand, Piece piece) {
        if (hand.size() == 0) {
                hand.add(piece);
        } else {
                for (int i = 0; i < hand.size(); i++) {
                        if (hand.get(i).getNumber() > piece.getNumber()) {
                                hand.add(i, piece);
                        }
                }
        }
}

/**
 * 盤面の表示
 */
public void printBoad() {
        if (turn) {
                System.out.printf("   ");
        } else {
                System.out.printf("\u001b[31m%3d\u001b[30m", turnNumber);
        }
        System.out.printf("◇ ");
        for (Piece p : whitePocket) {
                System.out.printf("\u001b[31m%c\u001b[30m ", p.getName());
        }
        System.out.println();
        printLine();
        for (int i = 0; i < 5; i++) {
                System.out.printf("  %d |", 5 - i);
                for (int j = 0; j < 5; j++) {
                        if (boad[4 - i][j] == null) {
                                System.out.printf("  |");
                        } else if (boad[4 - i][j].isBlack) {
                                if (4 - i == lastMoveR && j == lastMoveC) {
```

```
                                        System.out.printf("\u001b[33m%c\u001b[30m
|", boad[4 - i][j].getName());
                                } else {
                                        System.out.printf("%c |", boad[4 - i][j].getName());
                                }
                        } else {
                                if (4 - i == lastMoveR && j == lastMoveC) {
                                        System.out.printf("\u001b[32m%c\u001b[30m
|", boad[4 - i][j].getName());
                                } else {
                                        System.out.printf("\u001b[31m%c\u001b[30m
|", boad[4 - i][j].getName());
                                }
                        }
                }
                System.out.println();
                printLine();
        }
        System.out.println("     1  2  3  4  5");
        if (turn) {
                System.out.printf("%3d", turnNumber);
        } else {
                System.out.printf("   ");
        }
        System.out.printf("◆ ");
        for (Piece p : blackPocket) {
                System.out.printf("%c ", p.getName());
        }
        System.out.println();
        System.out.println();

}

/**
 * 盤面の横線を引く
 */
private void printLine() {
        System.out.println("   +—+—+—+—+—+");
}

/**
 * 最善手を求める
 * @return
 */
public Position bestMove() {
        turnNumber++;
        ArrayList<Position> newPositions = nextMoves();
        ArrayList<Position> bestPositions = new ArrayList<Position>();
        if (turn) {
```

```
            positionValue = -100000;
            for (Position p : newPositions) {
                int v = p.value(Position.depth);
                if (v > positionValue) {
                    positionValue = v;
                    p.positionValue = positionValue;
                    bestPositions.clear();
                    bestPositions.add(p);
                } else if (v == positionValue) {
                    p.positionValue = positionValue;
                    bestPositions.add(p);
                }
            }
        } else {
            positionValue = 100000;
            for (Position p : newPositions) {
                int v = p.value(Position.depth);
                if (v < positionValue) {
                    positionValue = v;
                    p.positionValue = positionValue;
                    bestPositions.clear();
                    bestPositions.add(p);
                } else if (v == positionValue) {
                    p.positionValue = positionValue;
                    bestPositions.add(p);
                }
            }
        }
        int r = random.nextInt(bestPositions.size());
        return bestPositions.get(r);
    }

    /*
     * 次に可能な盤面リストを求める
     */
    public ArrayList<Position> nextMoves() {
        ArrayList<Position> newPositions = new ArrayList<Position>();
        for (int i = 0; i < 5; i++) {
            for (int j = 0; j < 5; j++) {
                if (boad[i][j] == null) {
                } else if (boad[i][j].isBlack() == turn) {
                    boad[i][j].addNextPosition(this, i, j, newPositions);
                }
            }
        }
        for (int i = 0; i < 5; i++) {
            for (int j = 0; j < 5; j++) {
                if (boad[i][j] == null) {
                    if (turn) {
```

```
                                        for (int p = 0; p < blackPocket.size(); p++) {
                                                newPositions.add(drop(p, true, i, j));
                                                newPositions.add(drop(p, false, i, j));
                                        }
                                } else {
                                        for (int p = 0; p < whitePocket.size(); p++) {
                                                newPositions.add(drop(p, true, i, j));
                                                newPositions.add(drop(p, false, i, j));
                                        }
                                }
                        }
                }
        }
        return newPositions;
}

/**
 * 一番深く読んだ時の盤面の価値を求める
 * @return
 */
public int value() {
        return value(Position.depth);
}

/**
 * 駒の価値
 * @param r 駒の行
 * @param c 駒の列
 * @param d 手番
 * @return
 */
public int getPieceValue(int r, int c, boolean t) {
        if (t) {
                return boad[r][c].getPieceValue();
        } else {
                return boad[r][c].getPieceValue2();
        }
}

/**
 * 盤面の価値を求める
 * @param d 読みの深さ
 * @return
 */
public int value(int d) {
        int value = 0;
        boolean t = turnNumber % 2 == 1;
        if (d == 0) { // 最後の深さ
                for (int i = 0; i < 5; i++) {
```

```
                for (int j = 0; j < 5; j++) {
                    if (boad[i][j] == null) {
                    } else if (boad[i][j].isBlack()) {
                        if (boad[i][j].notMove(i, j)) { // 動けない駒は価
値がない
                            value += moveValue * boad[i][j].moveCount(this, i, j);
                        } else {
                            value += getPieceValue(i, j, t) + moveValue * boad[i][j
                        }
                    } else {
                        if (boad[i][j].notMove(i, j)) { // 動けない駒は価
値がない
                            value -= moveValue * boad[i][j].moveCount(this, i, j);
                        } else {
                            value -= getPieceValue(i, j, t) + moveValue * boad[i][j
                        }
                    }
                }
            }
            for (Piece p : blackPocket) {
                if (t) {
                    value += p.getPieceValue();
                } else {
                    value += p.getPieceValue2();
                }
            }
            for (Piece p : whitePocket) {
                if (t) {
                    value -= p.getPieceValue();
                } else {
                    value -= p.getPieceValue2();
                }
            }
        } else {
            int v;
            v = value(0);
            if (!turn) {
                if (v > 15000) { // 先手の勝ち
                    return v;
                }
            } else {
                if (v < -15000) { // 後手の勝ち
                    return v;
                }
            }
            ArrayList<Position> newPositions = nextMoves();
            if (turn) {
                value = -100000;
                for (Position p : newPositions) {
```

```
                v = p.value(d - 1);
                if (v > value) {
                    value = v;
                }
            }
        } else {
            value = 100000;
            for (Position p : newPositions) {
                v = p.value(d - 1);
                if (v < value) {
                    value = v;
                }
            }
        }
    }
    return value;
}

public Position humanMove(int r, int c, int newR, int newC) {
    Position ret = null;
    int d;
    if (r == 0) { // 駒を打つ
        if (c < 0) {
            d = -c;
        } else {
            d = c;
        }
        if (canDrop(d - 1, newR - 1, newC - 1)) {
            if (c < 0) {
                ret = drop(d - 1, false, newR - 1, newC - 1);
            } else {
                ret = drop(d - 1, true, newR - 1, newC - 1);
            }
        }
    } else { // 駒を動かす
        if (boad[r - 1][c - 1].canMove(this, r - 1, c - 1, newR - 1, newC - 1)) {
            ret = move(r - 1, c - 1, newR - 1, newC - 1);
        }
    }
    return ret;
}
```

- To クラス

```
/**
 * @author   NATSUMI MASUDA
 */

package kyotosyogi2;
```

```
/**
 * と
 *
 */

public class To extends Piece {
        private int[][] walkl = { { 1, -1 }, { 1, 0 }, { 1, 1 }, { 0, -1 }, { 0, 1 }, { -1, 0 } }; // 斜
め後ろ以外に歩く

        public To(boolean isBlack) {
                this.isBlack = isBlack;
                this.name = 'と';
                this.number = 4;
                this.pieceValue = 100;
                this.pieceValue2 = 100;
                walk = walkl;
        }

        public Piece reverse() {
                return new Kyo(isBlack);
        }

        public Piece originalPiece() {
                return new Kyo(!isBlack);
        }
```