卒業研究報告書

題目

より強い京都将棋 AI の開発

指導教員

石水 隆 講師

報告者

16–1–037–0189

森田 和樹

近畿大学理工学部情報学科

令和 2 年 2 月 4 日提出

概要

近年，様々なゲームに対してディープラーニングを用いた AI が開発され，将棋や囲碁などはトッププロを凌ぐ AI も現れ始めている．これらのゲームはプロ棋士達による膨大な棋譜データがあるため，それを AI の学習に用いることで強いゲーム AI を作成することができる．

将棋に類似したゲームの一つに京都将棋がある．京都将棋は 5x5 格子盤面，双方 5 枚ずつの駒で行われる小規模将棋の一種である．京都将棋は玉以外の駒は一度その駒を動かすごとに駒をひっくり返し，駒の性能が変わるという大きな特徴がある．京都将棋は独特なルールと小さな盤面により通常より短い時間で内容豊富な将棋を楽しむことができるが知名度が低くあまり研究もされていない．本将棋ならばプロ棋士同士の対局の棋譜や，コンピュータ将棋の大会の棋譜などが大量に公開されているためそれらを学習用データとして用いることができるが，京都将棋の場合，公開されている棋譜データがないため機械学習やディープラーニングによる京都将棋の AI が作成が難しい．そこで本研究では今後、機械学習やディープラーニングでの学習データに用いるための棋譜データを作ることができる京都将棋の AI を Java を用いて作成する．

目次

1　序論

1.1　本研究の背景

京都将棋は田宮克哉氏によって考案された 5x5 格子盤面を用い，双方 5 枚ずつの駒で行われる小規模将棋の一種である．京都将棋の初期配置を図 1 に示す．京都将棋で用いる駒は本将棋とは異なり，『香』の裏が『と』，『銀』の裏が『角』，『金』の裏が『桂』，『飛』の裏が『歩』になっており，『玉』以外の駒はその駒を動かすたびに裏返し，一度駒を動かすごとに駒の性能が変わるという大きな特徴がある [1]．京都将棋は，上述のとおり特殊な駒を用いるため本将棋の駒や盤は使えず，商品として幻冬舎等が販売 [2] しているほか，アプリが販売されている [3] が，大きな大会などは調査した限り行われておらず，棋譜もデータとして公開されているものは無かった．

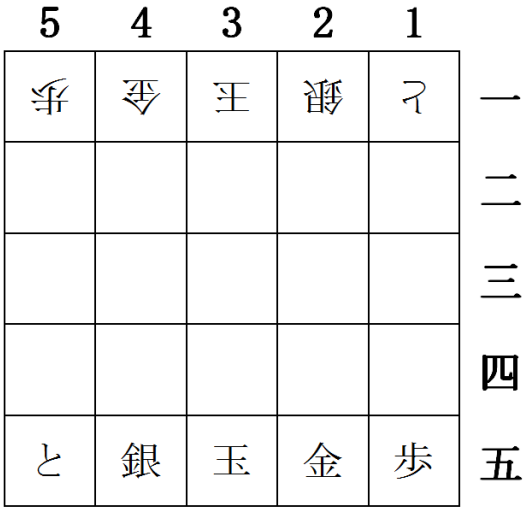

図 1　京都将棋の初期配置

1.2　本研究の目的

本研究の最終的な目的は，機械学習やディープラーニングを用いて強い京都将棋プログラムを開発することである．しかし，京都将棋があまり研究されておらず，学習データとして利用できるデータもあまりにも少なく実装が難しい．そこで，この問題を解決するべく，まず学習データにできる棋譜を多く生成できるプログラムを作る．またできるだけ良質な学習データにするために学習データ生成プログラムで用いる京都将棋 AI 自身の強化もはかる．

1.3　本報告書の構成

本報告書では，２章で研究対象について，３章では開発したプログラムについて，４章では検証内容の説明とその結果，最後に５章で結論と今後の課題を述べる．

2　研究対象について

本章では研究対象である京都将棋についてや棋譜データについて調査したことを述べる．

2.1　本将棋と京都将棋の違い

本将棋と京都将棋との違いを表 1 に示す．京都将棋で用いる駒は前章で述べた通り『香・と』『銀・角』『金・桂』『飛・歩』『玉』の 5 種類である．また，本将棋では禁止されている行為が京都将棋では禁止されていないものがある．移動できない駒とはその駒を動かそうとした際に盤面内に移動可能な場所がない状態や打ち駒した際に動けない駒になることで，京都将棋ではルール上問題ない．そのため京都将棋では，打つことで移動できない駒になる場合でも合い駒ができる．打ち歩詰めは手持ち駒の歩を盤面に置くことで敵の玉を詰ませる行為 [6] のことでこれも京都将棋では禁止されていない．この理由は京都将棋の場合，歩を打って詰ませることができる状況では飛車を表で打てば同じように詰ませることができるため禁則にしていないと考える．二歩に関しては本将棋の場合歩を打つ際にしか起こらないが京都将棋の場合は飛車の横移動の結果起こってしまう場合があるため，飛車の動きに制限をかけないために禁則にしていないのではないかと考える．また，千日手や王手放置のルールに関しては本将棋と同じである．

表 1　本将棋と京都将棋の違い

	本将棋	京都将棋
盤面	9x9	5x5
初期自陣駒数	20 枚	5 枚
駒を返す (成る) 条件	その駒が敵陣に入る (成らなくてもいい)	その駒を指す (裏返さなくてはならない)
手持ち駒を打つ際	表面のみ	表裏は任意
移動できない駒 (桂香歩に関する禁則)	禁止	禁止ではない
打ち歩詰め	禁止	禁止ではない
二歩	禁止	禁止ではない

2.2　棋譜データの形式

将棋の棋譜には様々な形式があるが今回は CSA 標準棋譜ファイル [4] を用いる.CSA 形式は駒の動きを先手か後手か，移動前の段と筋，移動後の段と筋を表記した後に移動後の駒の状態を表記する．先手の表記方法は『+』後手は『-』で表す．駒を打つ際は移動前の段と筋を『00』として表記する．各駒の csa 形式での表記方法を表 2 に示す．

例として京都将棋で先手が歩を初期配置 1 五から 1 四に差す『▲ 1 四歩成』の手の表記は『+1514HI』となり，後手が持ち駒を金を表にして 3 三に打つ『△ 3 三金打』の手の表記は『-0033KI』となる．

表 2　csa 形式での駒の表記方法

と	銀	金	歩	玉	香	角	桂	飛
TO	GI	KI	HU	OU	KY	KA	KE	HI

2.3 評価の判定方法

今回の京都将棋 AI で着手を決定する評価値には，盤面に置かれている駒の価値，盤面の利きの多さ，手持ちの駒の価値を用いる．この評価値を用い最善手を選択するアルゴリズムには相手は常に一番評価値が高くなる手，つまり自分にとって最悪の手を打ってくると仮定し探索するミニマックス法を用いた．各駒の価値の決め方は後ほど検証する．

3 開発したプログラムについて

本章では開発したプログラムについて述べる．本研究では，本将棋のプログラム [5] を参考に京都将棋に必要な機能を加えることで開発を進めた．12 個のクラスからなるプログラムのソースコードは付録にてまとめる．ここでは各クラスのメソッドの機能について説明する．

3.1 Game クラス

Game クラスはメインクラスである．ここで何試合行うのかを指定し連続で試合を行い棋譜データを作成することができる．

3.2 Board クラス

盤面の状況などを管理する Board クラスのメソッドとその機能を表 3 に示す．また，Board クラスのクラス図を図 2 に示す．

表 3　Board クラスのメソッドとその機能

メソッド	機能
getTurn:boolean 型	先手か後手かを得る
getKoma:Koma 型	指定座標にある駒を得る
getSHaveKoma:ArrayList 型	先手の持ち駒を得る
getGHaveKoma:ArrayList 型	後手の持ち駒を得る
getBanValue:int 型	盤面の価値を得る
getTurnNumber:int 型	手数を得る
getFinalTurnNumber:int 型	試合終了時の手数を得る
move:Board 型	駒を動かす
drop:Board 型	持ち駒を打つ
bestMove:Board 型	最善手を探索する
value:int 型	盤面の価値を求める
getKomaValue:int 型	駒の価値を得る
nextMoves:ArrayList 型	次の盤面を出す
CSAmake:void 型	csa 標準棋譜ファイルを書き出す

Board	# 盤面を定義するクラス
- turn : boolean	# true:先手 false:後手
- ban : Koma[][]	# 盤面
- sHaveKoma : ArrayList<Koma>	# 先手持ち駒
- gHaveKoma : ArrayList<Koma>	# 後手持ち駒
- banValue : int	# 盤面の評価値
- turnNumber : int	# 手数
- depth : int	# 読みの深さ
- random : Random	# 乱数生成用
- moveValue : int	# 利き一つに対する重み
+ Board (depth:int)	# コンストラクタ
+ Board (oldBoard:Board)	# コンストラクタ
+ getTurn() : boolean	# 先手か後手かを得る
+ getKoma (d:int, s:int)	# 指定座標にある駒を得る
+ getSHaveKoma() : ArrayList<Koma>	# 先手の持ち駒を得る
+ getGHaveKoma() : ArrayList<Koma>	# 後手の持ち駒を得る
+ getBanValue() : int	# 盤面の評価値を得る
+ getTurnNumber() : int	# 手数を得る
+ getFinalTurnNumber() : int	# 試合終了時の手数を得る
+ move (oldD:int, oldS:int, newD:int, newS:int) : Board	# 駒を動かす
+ drop (r:int, f:boolean, newD:int, newS:int) Board	# 持ち駒を打つ
+ bestMove() : Board	# 最善手を探索する
+ value() : int	# 盤面の価値を求める
+ value (dep:int) : int	# 盤面の価値を求める
+ getKomaValue (d:int, s:int, t:boolean) : int	# 駒の価値を得る
+ nextMoves() : ArrayList<Board>	# 次の盤面を出す
+ CSAmake (csa:ArrayList<Board>) : void	# csa 標準棋譜ファイルを書き出す

図 2　Board クラスのクラス図

3.3　Koma クラス

各駒についてを示す抽象クラスの Koma クラスのメソッドとその機能を表 4 に示す．また，Koma クラスのクラス図を図 3 に示す．

表 4　Koma クラスのメソッドとその機能

メソッド	機能
turn:boolean 型	先手かどうかを得る
getCSAname:String 型	棋譜データ時の名前を得る
getNumber:int 型	駒の種類を表す番号を得る
getKomaValue:int 型	駒の価値を得る
getKomaValue2:int 型	駒の価値 2 を得る
getHaveKomaValue:int 型	駒の手持ち時の価値を得る
getHaveKomaValue2:int 型	駒の手持ち時の価値 2 を得る
notMove:boolean 型	その駒に合法手があるかどうかを得る
checkPosition:boolean 型	盤面内かどうかを得る
addNextBoardByWalk:void 型	歩いて動いた際の盤面を作成
addNextBoardByRun:void 型	走って動いた際の盤面を作成
kikiByWalk:int 型	歩く駒の利きの数を得る
kikiByRun:int 型	走る駒の利きの数を得る
addNextBoard:void 型	駒を動かした盤面のリストを作成
moveCount:int 型	盤面全体の利きの数を得る
reverse:Koma 型	駒を返すことを示す抽象クラス
betrayal:Koma 型	敵の駒を味方にすることを示す抽象クラス

Koma	# 駒の種類を定義するクラス
# CSAname : String	# 棋譜データ時の名前
# sTurn : boolean	# true:先手 false:後手
# komaValue : int	# 駒の価値
# komaValue2 : int	# 駒の価値
# haveKomaValue : int	# 駒の手持ち時の価値
# haveKomaValue2 : int	# 駒の手持ち時の価値
# komaNumber : int	# 駒の種類を表す番号
# walk : int[][]	# 駒が 1 マス動く
# run : int[][]	# 駒が複数マス動ける
+ turn() : boolean	# 先手かどうかを得る
+ getCSAname() : String	# 棋譜データ時の名前を得る
+ getNumber() :String	# 駒の種類を表す番号を得る
+ getKomaValue() : int	# 駒の価値を得る
+ getKomaValue2() : int	# 駒の価値２を得る
+ getHaveKomaValue() : int	# 手持ち駒駒の価値を得る
+ getHaveKomaValue2() : int	# 手持ち駒の価値２を得る
+ notMove(r:int, c:int) : boolean	# 動けない駒かどうかを得る
+ addNextBoardByWalk(board:Board,d:int,s:int,walk:int[][],boards:ArrayList<Board>) : void	# 歩いて動くときの盤面リストを作成
+ addNextBoardByRun(board:Board,d:int,s:int,run:int[][],boards:ArrayList<Board>) : void	# 走って動くときの盤面リストを作成
+kikiByWalk(board:Board,d:int,s:int,walk:int[][]) : int	# 歩く駒の利きの数を求める
+kikiByRun(board:Board,d:int,s:int,run:int[][]) : int	# 走る駒の利きを求める
+addNextBoard(board:Board,d:int,s:int,boards:ArrayList<Board>) : void	# 次の盤面リストを作成
+moveCount(board:Board,d:int,s:int) : int	# 利きの数を得る
+ *reverse*() : Koma	# 駒を返すことを示す抽象メソッド
+ *betrayal*() : Koma	# 敵の駒を味方にすることを示す抽象メソッド

図 3　Koma クラスのクラス図

3.4　Koma クラスのサブクラス

Koma のサブクラスとして各駒を表す Hi クラス，Hu クラス，Gi クラス，Ka クラス，Ki クラス，Ke クラス，Ky クラス，To クラス，Ou クラスがある．それぞれの駒の価値をそれぞれのクラスのフィールドに記述する．抽象メソッドになっていた reverse メソッドでは歩の裏が飛，銀の裏が角であることなどその駒を裏返した際にどの駒になるのかを設定している．betrayal メソッドでは敵の駒を自分の駒として手持ちに加える際の処理を設定している．

3.5　実行の様子

本プログラムを実行し作成した棋譜ファイルの一つを例として図 4 に示す．図 4 により，csa 形式 [4] の通りに出力できたことが示される．また将棋の内容が正しいかどうか最終局面が正しく積んでいるかを [7] を用いて数戦確認したところ正しい合法手を示しており，図 4 の場合，図 5 のような最終局面になり先手が後手の玉を詰めていることが確認できる．

```
-HU-KI-OU-GI-TO
 *  *  *  *  *
 *  *  *  *  *
 *  *  *  *  *
+TO+GI+OU+KI+HU
_________________
+1514HI
-4142KE
+4544KA
-2122KA
+4422GI
-1122KY
+0043GI
-2223TO
+1424HU
-2322KY
+4334KA
-0033GI
+3443GI
-2224TO
+2524KE
-0023HI
+2432KI
%TORYO
```

図４　作成した棋譜データ

図 5　図 4 の最終局面

4　検証内容とその結果

本章ではこのプログラムを用いて検証する内容とその結果を示す.

4.1　検証内容

本研究で作成した京都将棋 AI は，駒の価値と敵味方の駒の各マスへの利きの数を用いて盤面を評価する．本将棋では，プロ棋士により各駒の価値はだいたい定まっている．しかし京都将棋ではそのような指標は無いため各駒の価値の適切な決め方を検証する．本検証で各駒に割りあてる駒の評価値を表 5 に示す．通常の将棋の場合は表より裏の方が強い動きができる駒になり，一度裏面になった駒は表に戻ることはない．京都将棋の場合は一番動きの弱い『歩』の裏に強い動きのできる『飛』があることや，一手ごとに表裏が変わること，さらに手持ちの駒は表裏どちらでも打てることから手持ち時の方がより駒の価値が高くなるのではないか等を考え，駒の価値を設定した．S は表裏で評価値を，A は駒ごとに違った評価値を設定し，その駒が手持ち時と盤面時で評価値が違う B，C をそれぞれ先手，基準とする S を後手として 100 回ずつ対戦させる．

表 5　駒に割り当てられた評価値

	香	と	銀	角	金	桂	飛	歩
S	60	120	100	180	120	70	200	10
A	90	90	140	140	95	95	105	105
B(盤面時)	60	120	100	180	120	70	200	10
B(手持ち時)	180	180	270	270	180	180	300	300
C(盤面時)	90	90	140	140	95	95	105	105
C(手持ち時)	135	135	210	210	142	142	157	157

4.2　結果と考察

対戦の結果を表 6 に示す.

勝率 p の勝負を N 回行った場合の標準偏差は以下の式で与えられる.

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

$p = 0.5$ とした場合，$N = 100$ なら標準偏差は

$$\sqrt{100 \cdot 0.5 \cdot 0.5} = 5.0$$

となり，試行回数 100 回の場合，危険率 95% の信頼区間は

$$50 \pm \frac{1.96 \cdot 5.0}{100} \cdot 100 = 50 \pm 9.8\%$$

となる．従って，勝率 60% を超えると強いとみなせる.

表 6 の S と A の比較により駒の価値は表裏で違う評価が，S と B の比較で盤面時と手持ち時の評価を変えることが有用であることがわかった. 強い駒が裏面の弱い駒の影響で正しく評価されなかったこと，手持ちの駒は好きな時に好きな面で好きな場所に打てることから盤面にある駒より良い動きにつながることが影響したのではないかと考える.

表 6　S との対戦結果 (試行回数 100 回)

	勝	敗	勝率
S	49	51	49%
A	36	64	36%
B	63	37	63%
C	43	57	43 %

5　結論・今後の課題

本研究によって京都将棋の棋譜が CSA 形式のデータとして収集できるようになった．また駒の評価値を用いる際の駒の価値の決め方がわかった．今後の課題として一つ目はこの学習データを生かしてディープラーニ

ングを用いたより強い京都将棋 AI の開発を行っていくこと，二つ目は今回わかった決め方に基づき，盤面時の裏表別の価値 8 種類と手持ち時の価値 4 種類の計 12 種類の価値についてより適切な値を探索しながら良質な棋譜データを収集することである．

謝辞

本研究を進めるに当たり，ご指導いただきました石水隆講師に感謝いたします.

参考文献

[1] 特開 2001-314544, 特許情報プラットフォーム,
URL: https://www.j-platpat.inpit.go.jp/web/PU/JPA _H13314544/B7A438836C5E182BF22D6131F2520D55
, 最終アクセス日：2019-10-10

[2] 京都将棋，幻冬舎,
URL: https://www.gentosha-edu.co.jp/book/b351544.html，最終アクセス日：2020 － 1 － 28

[3] iOS 版【京都将棋】アプリ　配信スタート，株式会社ねこまど,
URL: http://nekomado.com/entries/5407，最終アクセス日：2020-1-29

[4] 棋譜ファイル形式，コンピュータ将棋協会,
URL: http://www2.computer-shogi.org/protocol/recordv21.html 最終アクセス日：2020-1-29

[5] 池 泰弘:Java 将棋のアルゴリズム，工学社 (2007)

[6] 将棋の基礎知識，公益社団法人 日本将棋連盟,
URL: https://www.shogi.or.jp/knowledge/shogi/01.html 最終アクセス日：2020-1-29

[7] 京都将棋 将棋ゲームの時間，んとか将棋，URL:https://syouginojikan.web.fc2.com/kyouto.html 最終アクセス日：2020-1-30

付録 A　ソースファイル

本研究で作成したプログラムのソースファイルを以下に示す.

Game クラス

```
import java.util.ArrayList;

public class Game {

        public static void main(String[] args){
                int win = 0;//先手の勝利数
                int lose = 0;//後手の勝利数
                int draw = 0;//引き分けの数
                int count = 1;//行う試合回数
                ArrayList<Board> csa;//試合の流れのリスト
                for(int i = 0;i<count;i++){
                        System.out.println(i);
                        Board b = new Board(2);
                        csa = new ArrayList<Board>();
                        while(true){
                                csa.add(b);//現在の盤面をリストに追加
                                b = b.bestMove();//最善手を打つ
                                int v = b.getBanValue();//盤面の価値を計算
                                System.out.printf("### %d ###\n", v);
                                if (v > 15000) {
                                        System.out.println("先手の勝ち");
                                        System.out.println("ターン数:"+b.getFinalTurnNumber());
                                        win++;
                                        break;
                                }
                                if (v < -15000) {
                                        System.out.println("後手の勝ち");
                                        System.out.println("ターン数:"+b.getFinalTurnNumber());
                                        lose++;
                                        break;
                                }
                                if(b.getTurnNumber() >500){
                                        System.out.println("引き分け");
                                        System.out.println("ターン数:"+b.getFinalTurnNumber());
                                        draw++;
                                        break;
                                }
                        }
                        csa.add(b);//最後の盤面をリストに追加
                        b.CSAmake(csa);//csa 標準形式でファイルを作成
                }
                //平均をだす際は以下のコメントアウトを外す
```

```
                /*
                System.out.println(count+"戦で先手の"+win+"勝"+lose+"敗"+draw+"引き分けでした。");
                System.out.println("B 先手勝率:"+((double)win/(double)(count-draw))*100);
                System.out.println("S 後手勝率:"+((double)lose/(double)(count-draw))*100);
                */
        }
}
```

Board クラス

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.Random;


public class Board {
        private boolean turn; //true:先手 false:後手
        private Koma[][] ban; //盤面 (5x5)
        private ArrayList<Koma> sHaveKoma; //先手持ち駒
        private ArrayList<Koma> gHaveKoma; //後手持ち駒
        private int banValue; //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;//利き一つに対する重み


        public Board(int depth){
                ban = new Koma[5][5];
                for(int d = 0;d<5;d++){
                        for(int s = 0;s<5;s++){
                                ban[d][s] = null;
                        }
                }
                ban[0][0] = new To(true);
                ban[0][1] = new Gi(true);
                ban[0][2] = new Ou(true);
                ban[0][3] = new Ki(true);
                ban[0][4] = new Hu(true);

                ban[4][4] = new To(false);
                ban[4][3] = new Gi(false);
                ban[4][2] = new Ou(false);
                ban[4][1] = new Ki(false);
```

```
        ban[4][0] = new Hu(false);
        sHaveKoma = new ArrayList<Koma>();//先手の持ち駒
        gHaveKoma = new ArrayList<Koma>();//後手の持ち駒
        turn = true;//先手から
        Board.depth = depth;//読みの深さ
}

/**
 * 前の手番までの盤面をコピー
 * @param oldBoard 前の盤面
 */
public Board(Board oldBoard){
        turn = !(oldBoard.turn);//ターンの入れ替え
        ban = new Koma[5][5];
        for(int d = 0;d<5;d++){
                for(int s = 0;s<5;s++){
                        ban[d][s] = oldBoard.ban[d][s];
                }
        }
        sHaveKoma = new ArrayList<Koma>(oldBoard.sHaveKoma);//先手の持ち駒
        gHaveKoma = new ArrayList<Koma>(oldBoard.gHaveKoma);//後手の持ち駒
}

/**
 * 先手か後手か
 * @return true:先手 false:後手
 */
public boolean getTurn(){
        return turn;
}

/**
 * 盤面にある駒を得る
 * @param d 知りたい駒の段
 * @param s 知りたい駒の筋
 * @return その座標の駒（駒がなければ null）
 */
public Koma getKoma(int d, int s) {
        return ban[d][s];
}

/**
 * 先手の持ち駒を得る
 * @return 先手の持ち駒
 */
public ArrayList<Koma> getSHaveKoma(){
        return sHaveKoma;
}
```

```
/**
 * 後手の持ち駒を得る
 * @return 後手の持ち駒
 */
public ArrayList<Koma> getGHaveKoma(){
        return gHaveKoma;
}

/**
 * 盤面の価値を得る
 * @return 盤面の価値
 */
public int getBanValue(){
        return banValue;
}

/**
 * 手数を得る
 * @return  手数
 */
public int getTurnNumber(){
        return turnNumber;
}

/**
 * 試合終了時の手数を得る
 * @return 試合終了時の手数
 */
public int getFinalTurnNumber(){
        int finNumber = turnNumber;
        turnNumber = 0;
        return finNumber;
}


/**
 * 駒の移動
 * @param oldD  移動前の段
 * @param oldS  移動前の筋
 * @param newD  移動後の段
 * @param newS  移動後の筋
 * @return      移動後の盤面
 */

public Board move(int oldD, int oldS, int newD, int newS) {
        Board newBoard = new Board(this);
        if(ban[newD][newS] != null){    //行き先に駒があるなら
                if(turn){       //先手なら
                        newBoard.sHaveKoma.add(ban[newD][newS].betrayal());
```

```
                } else {        //後手なら
                        newBoard.gHaveKoma.add(ban[newD][newS].betrayal());
                }
        }
        newBoard.ban[newD][newS] = ban[oldD][oldS].reverse();   //移動後に駒を返す
        newBoard.ban[oldD][oldS] = null;
        newBoard.turn = !turn;  //ターン入れ替え
        return newBoard;
}

/**
 * 持ち駒を打つ
 * @param r     持ち駒リストの何番めの駒か
 * @param f  表で打つか裏で打つか
 * @param newD  打つ段
 * @param newS  打つ筋
 * @return      打ち終わった盤面
 */
public Board drop(int r,boolean f,int newD,int newS){
        Board newBoard = new Board(this);
        Koma koma;
        if(turn){       //先手なら先手の持ち駒リストから削除
                koma = newBoard.sHaveKoma.remove(r);
        } else{         //後手なら後手の持ち駒リストから削除
                koma = newBoard.gHaveKoma.remove(r);
        }
        if(!f){
                koma = koma.reverse();
        }
        newBoard.ban[newD][newS] = koma;
        newBoard.turn = !turn;  //ターン入れ替え
        return newBoard;
}

public Board bestMove(){
        turnNumber++;
        ArrayList<Board> newBoards = nextMoves();
        ArrayList<Board> bestMoves = new ArrayList<Board>();
        if(turn){       //先手なら
                banValue = -9999999;
                for(Board b:newBoards){
                        int v = b.value(Board.depth);
                        if(v > banValue){       //今までよりいい盤面なら bestMove リストを更
新
                                banValue = v;
                                b.banValue = banValue;
                                bestMoves.clear();
                                bestMoves.add(b);
                        } else if (v == banValue){      //今までの最高と同じなら bestMove リ
```

ストに追加

```
                                        b.banValue = banValue;
                                        bestMoves.add(b);
                                }
                        }
                }
                else {  //後手なら
            banValue = 9999999;
            for (Board b : newBoards) {
                    int v = b.value(Board.depth);
                    if (v < banValue) { //今までよりいい盤面なら bestMove リストを更新
                            banValue = v;
                            b.banValue = banValue;
                            bestMoves.clear();
                            bestMoves.add(b);
                    } else if (v == banValue) { //今までの最高と同じなら bestMove リストに追加
                            b.banValue = banValue;
                            bestMoves.add(b);
                    }
            }
    }
                int r = random.nextInt(bestMoves.size());
        return bestMoves.get(r);
        }

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

    public int value(int dep) {
        int value = 0;
        boolean t = turnNumber % 2 == 1;
        if (dep == 0) { // 最後の深さ
                for (int d = 0; d < 5; d++) {
                        for (int s = 0; s < 5; s++) {
                                if (ban[d][s] == null) ;
                                else if (ban[d][s].sTurn) {
                                        if (ban[d][s].notMove(d, s)) { // 動けない駒は価値がない
                                                value += moveValue * ban[d][s].moveCount(this, d, s);//moveCount
```

は0になる

```
                                        } else {
                                                value += getKomaValue(d, s, t) + moveValue * ban[d][s].moveCount(th
                                        }
                                } else {
                                        if (ban[d][s].notMove(d, s)) { // 動けない駒は価値がない
```

```
                                        value -= moveValue * ban[d][s].moveCount(this, d, s);//moveCount
は0になる
                                } else {
                                        value -= getKomaValue(d, s, t) + moveValue * ban[d][s].moveCount(th
                                }
                        }
                }
        }
        //持ち駒を価値に追加
        for (Koma k : sHaveKoma) {
                if (t) {
                        value += k.getHaveKomaValue();
                } else {
                        value += k.getHaveKomaValue2();
                }
        }
        for (Koma k : gHaveKoma) {
                if (t) {
                        value -= k.getHaveKomaValue();
                } else {
                        value -= k.getHaveKomaValue2();
                }
        }
} 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<Board> newBoards = nextMoves();
        if (turn) {
                value = -9999999;
                for (Board b : newBoards) {
                        v = b.value(dep - 1);
                        if (v > value) {
                                value = v;
                        }
                }
        } else {
                value = 9999999;
                for (Board b : newBoards) {
                        v = b.value(dep - 1);
                        if (v < value) {
```

```
                                        value = v;
                                }
                        }
                }
        }
        return value;
}


    /**
     * 駒の価値を得る
     * @param d 駒の段
     * @param s  駒の筋
     * @param t  手番
     * @return  駒の価値
     */
    public int getKomaValue(int d, int s, boolean t) {
        if (t) {
                return ban[d][s].getKomaValue();
        } else {
                return ban[d][s].getKomaValue２();
        }
}


    /**
     * 次の盤面リストを出す
     * @return  次の盤面リスト
     */
        private ArrayList<Board> nextMoves() {
                ArrayList<Board> newBoards = new ArrayList<Board>();
        for (int d = 0; d < 5; d++) {
                for (int s = 0; s < 5; s++) {
                        if (ban[d][s] == null) {
                        } else if (ban[d][s].sTurn == turn) {
                                ban[d][s].addNextBoard(this, d, s, newBoards);
                        }
                }
        }
        //持ち駒を打った際の盤面をリストに
        for (int d = 0; d < 5; d++) {
                for (int s = 0; s < 5; s++) {
                        if (ban[d][s] == null) {
                                if (turn) {
                                        for (int p = 0; p < sHaveKoma.size(); p++) {
                                                newBoards.add(drop(p, true, d, s));
                                                newBoards.add(drop(p, false, d, s));
                                        }
                                } else {
```

```
                                for (int p = 0; p < gHaveKoma.size(); p++) {
                                        newBoards.add(drop(p, true, d, s));
                                        newBoards.add(drop(p, false, d, s));
                                }
                        }
                }
        }
}

        return newBoards;
}
/**
 * csa 標準棋譜ファイルを書き出す
 * @param csa 試合の盤面のリスト
 */
public void CSAmake(ArrayList<Board> csa) {
try {
        //ファイルの作成
        File file = new File("CSA"+ LocalDateTime.now() +".txt");

          FileWriter filewriter = new FileWriter(file);
          //作成時間
          filewriter.write(LocalDateTime.now()+"\r\n");
          //初期配置を書き込み
          filewriter.write("-HU-KI-OU-GI-TO\r\n");
          filewriter.write(" *  *  *  *  *\r\n");
          filewriter.write(" *  *  *  *  *\r\n");
          filewriter.write(" *  *  *  *  *\r\n");
          filewriter.write("+TO+GI+OU+KI+HU\r\n");

          filewriter.write("_____________\r\n");

          for(int t = 1;t<csa.size();t++){
                  int new1 = 0;
                  int new2 = 0;
                  int old1 = 6;
                  int old2 = 6;
                  String turn = null;
                  for(int d = 0;d<5;d++){
                                for(int s = 0;s<5;s++){
                                        if(csa.get(t).ban[d][s]!=csa.get(t-1).ban[d][s]){
                                                if(csa.get(t-1).ban[d][s] == null){     //
動かした際の移動先の座標
                                                        new1 = d+1;
                                                        new2 = s+1;
                                                }
                                                else if(csa.get(t).ban[d][s] == null){ //
動かした際の移動前の座標
                                                        old1 = d+1;
```

```
                                                        old2 = s+1;
                                                }
                                                else { //打った際の駒の座標
                                                        new1 = d+1;
                                                        new2 = s+1;
                                                }
                                        }
                                }
                        }
                        if(csa.get(t-1).getTurn()) turn = "+";//先攻の動きなら＋をつける
                        else turn = "-";//後攻の動きなら−をつける
                        //＋ or −、移動前の段、移動前の筋、移動後の段、移動後の筋、移動後の駒の状態の順
に書き出し
                        filewriter.write(turn+(6-old2)+""+(6-old1)+""+(6-new2) +""+ (6-new1)+csa.get(t).ban[new1-
                }
                //引き分けの表示
                if(csa.size()>499) filewriter.write("%SENNICHITE");
                //投了の表示
                else filewriter.write("%TORYO");

                filewriter.close();
              } catch (IOException e) {
                      e.printStackTrace();
              }
        }

}
```

Koma クラス

```
import java.util.ArrayList;

public abstract class Koma {
        protected String CSAname; //棋譜データ時の名前
        protected boolean sTurn; //true:先手 false:後手
        protected int komaValue; //駒の価値
        protected int komaValue2; //駒の価値二つめ
        protected int haveKomaValue;//駒の手持ち時の価値
        protected int haveKomaValue2;//駒の手持ち時の価値２
        protected int komaNumber; //駒の種類を表す番号
        protected int [][] walk = {}; //駒が１マス動く
        protected int [][] run = {}; //駒が複数マス動ける

        /**
         * 先手駒かどうか
         * @return      true:先手 false:後手
         */
```

```
public boolean turn(){
        return sTurn;
}

/**
 * 棋譜データ時の名前を得る
 * @return   棋譜データ時の名前
 */
public String getCSAname(){
        return CSAname;
}

/**
 * 駒の種類を表す番号を得る
 * @return 駒の番号
 */
public int getNumber(){
        return komaNumber;
}

/**
 * 駒の価値を得る
 * @return 駒の価値
 */
public int getKomaValue(){
        return komaValue;
}

/**
 * 駒の価値 2 を得る
 * @return 駒の価値 2
 */
public int getKomaValue 2 (){
        return komaValue2;
}

/**
 * 駒の手持ち時の価値を得る
 * @return       駒の手持ち時の価値
 */
public int getHaveKomaValue(){
        return haveKomaValue;
}

/**
 * 駒の手持ち時の価値 2 を得る
 * @return       駒の手持ち時の価値 2
 */
public int getHaveKomaValue2(){
```

```
            return haveKomaValue2;
    }

    /**
 * 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;
}

    /**
     * その場所が盤面内か確認
     * @param d 駒の段
     * @param s 駒の筋
     * @return true 盤面内
     */
    public boolean checkPosition(int d, int s){
            return (d>0 && d<5 && s>0 && s<5);
    }

    /**
     * 歩いて動くときの盤面
     * @param board 現在の盤面
     * @param d 駒の段
     * @param s     駒の筋
     * @param walk  歩いて動ける方向
     * @param boards        移動後の盤面のリスト
     */
    public void addNextBoardByWalk(Board board,int d,int s,int[][] walk,ArrayList<Board> boards){
            int newD;
            int newS;

            for(int i = 0;i<walk.length;i++){
                    if(sTurn){
                            //先手なら
                                    newD = d + walk[i][0];
                                    newS = s + walk[i][1];
                    }       else{
                            //後手なら
                                    newD = d - walk[i][0];
                                    newS = s - walk[i][1];
                    }
                    if(checkPosition(newD,newS)){
                            //盤面ないか確認
                            if(board.getKoma(newD,newS) == null){
                                    //移動先に駒がないなら進める
                                    boards.add(board.move(d,s,newD,newS));
                            } else if (board.getKoma(newD, newS).turn() != sTurn){
```

```
                                        //移動先に駒があり，それが相手の駒ならその駒を取り進める
                                        boards.add(board.move(d,s,newD,newS));
                                }
                        }
                }
        }

        /**
         *
         * @param board 現在の盤面
         * @param d     駒の段
         * @param s     駒の筋
         * @param run   駒の走れる方向
         * @param boards 移動後の盤面のリスト
         */
        public void addNextBoardByRun(Board board,int d,int s,int[][] run,ArrayList<Board> boards){
                        int newD;
                        int newS;

                        for(int i = 0;i < run.length;i++){
                                for (int j = 0;j<4;j++){
                                        if(sTurn){
                                                //先手なら
                                                newD = d + (run[i][0]) * (j+1);
                                                newS = s + (run[i][1]) * (j+1);
                                        } else {
                                                //後手なら
                                                newD = d - (run[i][0]) * (j+1);
                                                newS = s - (run[i][1]) * (j+1);
                                        }
                                        if(checkPosition(newD,newS)){
                                                //盤面内か確認
                                                if(board.getKoma(newD,newS) == null){
                                                        //移動先に駒がないなら進める
                                                        boards.add(board.move(d,s,newD,newS));
                                                } else if (board.getKoma(newD, newS).turn() != sTurn){
                                                        //移動先に駒があり，それが相手の駒ならその駒
を取り進める

                                                        boards.add(board.move(d,s,newD,newS));
                                                        break; //相手駒の位置で終わり
                                                } else{
                                                        break; //味方の駒があるならそれ以上進めない
ので終わり
                                                }
                                        }
                                }
                        }
        }
```

```
/**
 * 歩く駒の利きの数
 * @param board 現在の盤面
 * @param d 駒の段
 * @param s 駒の筋
 * @param walk 歩いて動ける方向
 * @return 利きの数
 */

public int kikiByWalk(Board board, int d, int s, int[][] walk){
        int newD;
        int newS;
        int kiki = 0;

        for(int i = 0;i<walk.length;i++){
                if(sTurn){
                        //先手なら
                        newD = d + walk[i][0];
                        newS = s + walk[i][1];
                } else {
                        //後手なら
                        newD = d - walk[i][0];
                        newS = s - walk[i][1];
                }
                if(checkPosition(newD,newS)){
                        //盤面ないか確認
                        if(board.getKoma(newD, newS)==null){
                                //移動先に駒がないなら
                                kiki++;
                        } else if(board.getKoma(newD, newS).turn() != sTurn){
                                //移動先に駒があり，それが相手の駒ならその駒を取り進める
                                kiki++;
                        }
                }
        }
        return kiki;
}




/**
 *
 * @param board 現在の盤面
 * @param d     駒の段
 * @param s     駒の筋
 * @param run   駒の走れる方向
 * @return 利きの数
```

```
 */
public int kikiByRun(Board board, int d, int s, int[][] run){
        int newD;
        int newS;
        int kiki = 0;
        for(int i = 0;i<run.length;i++){
                for (int j = 0;j<4;j++){
                        if(sTurn){
                                //先手なら
                                newD = d + run[i][0] * (j+1);
                                newS = s + run[i][1] * (j+1);
                        }else {
                                //後手なら
                                newD = d - (run[i][0]) * (j+1);
                                newS = s - (run[i][1]) * (j+1);
                        }
                        if(checkPosition(newD,newS)){
                                //盤面内か確認
                                if(board.getKoma(newD, newS)==null){
                                        //移動先に駒がないなら
                                        kiki++;
                                } else if(board.getKoma(newD, newS).turn() != sTurn){
                                        //移動先に駒があり，それが相手の駒ならその駒を取り進め
る
                                        kiki++;
                                        break; //相手駒の位置で終わり
                                } else {
                                        break; //味方の駒があるならそれ以上進めないので終わり
                                }
                        }
                }
        }
        return kiki;
}

/**
 * 駒を動かした際の盤面のリスト
 * @param board   現在の盤面
 * @param d   駒の段
 * @param s   駒の筋
 * @param boards   移動後の盤面のリスト
 */
public void addNextBoard(Board board,int d,int s,ArrayList<Board> boards){
        addNextBoardByWalk(board,d,s,walk,boards);
        addNextBoardByRun(board,d,s,run,boards);
}

/**
 * 盤面の利きの数を得る
```

```
         * @param board   現在の盤面
         * @param d   駒の段
         * @param s   駒の筋
         * @param newD   移動後の駒の段
         * @param newS   移動後の駒の筋
         * @return   利きの数
         */
        public int moveCount(Board board,int d,int s){
                return kikiByWalk(board,d,s,walk) + kikiByRun(board,d,s,run);
        }

        /**
         * 駒を返す
         * @return 返った駒
         */
        public abstract Koma reverse();

        /**
         * 敵の駒を味方にする
         * @return 表の駒
         */
        public abstract Koma betrayal();
}
```

Ky クラス

```
public class Ky extends Koma{
        private int ky[][] = { { 1, 0 } }; //先手香は前に走る
        public Ky(boolean b){
                this.sTurn = b;
                this.CSAname = "KY";
                this.komaNumber = 3;

                this.komaValue = 60;//先攻
                this.komaValue2 = 60;//後攻

                this.haveKomaValue = 180;//先攻の持ち駒の価値
                this.haveKomaValue2 = komaValue2;//後攻の持ち駒の価値


                run = ky;
        }

        @Override
        public Koma reverse() {
                return new To(sTurn);
        }
```

```
        @Override
        public Koma betrayal() {
                return new To(!sTurn);
        }

}
```

To クラス

```
public class To extends Koma {
        private int to[][] = { { 1, -1 }, { 1, 0 }, { 1, 1 }, { 0, -1 }, { 0, 1 }, { -1, 0 } }; // と
は斜め後ろ以外に歩く
        public To(boolean b) {
                this.sTurn = b;
                this.CSAname = "TO";
                this.komaNumber = 4;

                this.komaValue = 120;//先攻の盤面時の価値
                this.komaValue2 = 120;//後攻の盤面時の価値

                this.haveKomaValue = 180;//先攻の持ち駒の価値
                this.haveKomaValue2 = komaValue2;//後攻の持ち駒の価値

                 walk = to;
                }


        @Override
        public Koma reverse() {
                return new Ky(sTurn);
        }

        @Override
        public Koma betrayal() {
                return new To(!sTurn);
        }

}
```

Gi クラス

```
public class Gi extends Koma {
        private int gi[][] = { { 1, -1 }, { 1, 0 }, { 1, 1 }, { -1, -1 }, { -1, 1 } }; // 先手の
銀は斜め４方向と前に歩く
        public Gi(boolean b) {
                this.sTurn = b;
```

```
                this.CSAname = "GI";
        this.komaNumber = 7;

        this.komaValue = 100;//先攻
        this.komaValue2 = 100;//後攻

        this.haveKomaValue = 270;//先攻の持ち駒の価値
        this.haveKomaValue2 = komaValue2;//後攻の持ち駒の価値

        walk = gi;
        }

        @Override
        public Koma reverse() {
                return new Ka(sTurn);
        }

        @Override
        public Koma betrayal() {
                return new Gi(!sTurn);
        }

}
```

Ka クラス

```
public class Ka extends Koma{
        private int ka[][] = { { 1, -1 }, { 1, 1 }, { -1, -1 }, { -1, 1 } }; // 角は斜め４方に走る

        public Ka(boolean b){
                this.sTurn = b;
                this.CSAname = "KA";
        this.komaNumber = 8;

        this.komaValue = 180;//先攻
        this.komaValue2 = 180;//後攻

        this.haveKomaValue = 270;//先攻の持ち駒の価値
        this.haveKomaValue2 = komaValue2;//後攻の持ち駒の価値

        run = ka;
        }

        @Override
        public Koma reverse() {
                return new Gi(sTurn);
        }
```

```
        @Override
        public Koma betrayal() {
                return new Gi(!sTurn);
        }

}
```

Ki クラス

```
public class Ki extends Koma {
        private int[][] ki = { { 1, -1 }, { 1, 0 }, { 1, 1 }, { 0, -1 }, { 0, 1 }, { -1, 0 } }; // 先
手金は斜め後ろ以外に歩く

        public Ki(boolean b) {
                this.sTurn = b;
                this.CSAname = "KI";
                this.komaNumber = 5;

                this.komaValue = 120;//先攻
                this.komaValue2 = 120;//後攻

                this.haveKomaValue = 180;//先攻の持ち駒の価値
                this.haveKomaValue2 = komaValue2;//後攻の持ち駒の価値



                walk = ki;
        }

        @Override
        public Koma reverse() {
                return new Ke(sTurn);
        }

        @Override
        public Koma betrayal() {
                return new Ki(!sTurn);
        }

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


}
```

Ke クラス

```
public class Ke extends Koma{
        private int[][] sKe = { { 2, -1 }, { 2, 1 } }; // 先手桂は前方からさらに斜めに歩く
        private int[][] gKe = { { -2, -1 }, { -2, 1 } };// 後手桂は後手から見て前方からさらに斜めに歩
く


        public Ke(boolean b){
                this.sTurn = b;
                this.CSAname = "KE";
                this.komaNumber = 6;

                this.komaValue = 70;//先攻
                this.komaValue2 = 70;//後攻

                this.haveKomaValue = 180;//先攻の持ち駒の価値
                this.haveKomaValue2 = komaValue2;//後攻の持ち駒の価値



                if(sTurn)walk = sKe;
                else walk = gKe;
        }
        @Override
        public Koma reverse() {
                return new Ki(sTurn);
        }

        @Override
        public Koma betrayal() {
                return new Ki(!sTurn);
        }

}
```

Hi クラス

```
public class Hi extends Koma {
        private int hi[][] =  { { 1, 0 }, { 0, -1 }, { 0, 1 }, { -1, 0 } }; // 飛は縦横に走る
        public Hi(boolean b){
                this.sTurn = b;
                this.CSAname = "HI";
        this.komaNumber = 2;
        this.komaValue = 200;//先攻
        this.komaValue2 = 200;//後攻

        this.haveKomaValue = 300;//先攻の持ち駒の価値
        this.haveKomaValue2 = komaValue2;//後攻の持ち駒の価値
```

```
        run=hi;
        }
        @Override
        public Koma reverse() {
                return new Hu(sTurn);
        }

        @Override
        public Koma betrayal() {
                return new Hu(!sTurn);
        }


}
```

Hu クラス

```
public class Hu extends Koma {
        private int hu[][] = { { 1, 0 } }; // 先手歩は前にだけ歩く



        public Hu(boolean b) {
                this.sTurn = b;
                this.CSAname = "HU";
                this.komaNumber=1;

                this.komaValue = 10;//先攻
                this.komaValue2 = 10;//後攻

                this.haveKomaValue = 300;//先攻の持ち駒の価値
                this.haveKomaValue2 = komaValue2;//後攻の持ち駒の価値

                if(sTurn) walk = hu;

        }

        @Override
        public Koma reverse() {
                return new Hi(sTurn);
        }


        @Override
        public Koma betrayal() {
                return new Hu(!sTurn);
```

```
        }

        public boolean notMove(int r, int c) {
                return (sTurn && r == 5) || (!sTurn && r == 1);
}


}
```

Ou クラス

```
public class Ou extends Koma {
        private int[][] ou =
        { { 1, -1 }, { 1, 0 }, { 1, 1 }, { 0, -1 }, { 0, 1 }, { -1, -1 }, { -1, 0 }, { -1, 1 } }; //
王は全方向に歩く
        public Ou(boolean b) {
                this.sTurn = b;
                this.CSAname = "OU";
                this.komaNumber = 0;
                this.komaValue = 99999;
                this.komaValue2 = 99999;
                this.haveKomaValue = 99999;
                this.haveKomaValue2 = 99999;
                walk = ou;
        }

        @Override
        public Koma reverse() {
                return new Ou(sTurn);
        }

        @Override
        public Koma betrayal() {
                return new Ou(!sTurn);
        }

}
```