卒業研究報告書

題目

京都将棋における適切な評価関数について

指導教員

石水 隆　講師

報告者

14-1-037-0142

油井 千雅

近畿大学理工学部情報学科

令和２年２月４日提出

概要

京都将棋は，『玉』『香と』『金桂』『飛歩』の５種類の駒と縦横５マスの盤を用いる将棋である．それぞれの駒の動きは本将棋と同様であるが，『玉』以外の駒は一手ごとにその駒を必ず裏返すという特別なルールがあり，玉以外は一手ごとに性能が変わる．このため，本将棋に精通した者であっても京都将棋で上手に指せるとは限らず，また，盤面が小さいこともあり短時間で勝負が決まるため，京都将棋は本将棋とはまた違った趣を持つゲームとなっている．

本将棋では，以前よりコンピュータ将棋は広く研究されており，最近ではディープラーニングを用いた将棋 AI がプロ棋士を上回る棋力を持つようになっている．一方，京都将棋はマイナーなゲームであるため本将棋と比べ対戦 AI の開発は進んでいない．

本将棋では，プロ棋士による膨大な棋譜を学習データとして用いる事ができ，またプロ棋士の長年の経験から局面の評価値をどのように評価するかが大方定まっている．しかし，京都将棋のようなミニ将棋では本将棋ほどデータとなる棋譜もなく評価値の算出方法も定まっていない．本研究では，学習により評価関数を調整しながら AI 同士を対戦させ，京都将棋の適切な評価値の求め方を検証する．

目次

1 序論

1.1. 京都将棋とは

京都将棋は，５×５マスのミニ将棋で玉以外の駒は『香』の裏は『と』というように裏表に異なる駒が書かれており，一手ごとに駒を裏返すという特別なルールがある．このため，本将棋に精通した者であっても京都将棋で上手に指せるとは限らず，また，盤面が小さいこともあり短時間で勝負が決まるため，京都将棋は本将棋とはまた違った趣を持つゲームとなっている．

1.2. 京都将棋の既知の結果

京都将棋について調べたところ定跡や評価値についての研究結果は見当たらなかった．現在，京都将棋のソフトは CPU 相手にプレイできるアプリとして株式会社ねこまどが開発した [4]や[5]が存在する．[4]では CPU の強さを 4 段階で調整できる．また，[5]を実際にプレイしたところ[5]のアプリケーションの方が勝つことが多かったが将棋に精通しているプレイヤと対戦した場合はどの程度の強さであるかは不明である．CUP 戦ではないが Web 上での対戦が行える[6]も存在する．

1.3. 将棋 AI の手法

将棋のような可能な局面数の多いゲームでは完全解析を行うことは困難である．そのようなゲームに対しては，駒の価値や玉の危険度から現在の局面を数値化し有利か不利かを算出する局面の評価値計算や，数手先の手を読みその手の局面で評価値計算を行い評価値が高い手を選択する一定手数の先読みなどの方法で有利だと思われる手を選択している．また，ゲームの終盤では残りの手数が少ないため勝敗が着くまで手を読み切ることも可能である．

本将棋にはプロ棋士の対戦により膨大な棋譜データが存在するため，定跡データベースから有利な局面を指す手を選択することも可能である．

将棋 AI として有名な Bonanza は棋譜データを元に機械学習を用いて有利な手を選択しており，プロ棋士に勝つことも可能なほど強い将棋 AI となっている．

将棋では，局面の評価値計算の要素として各駒に評価値を割り当てる手法がよく用いられる．各駒の価値は，本将棋では長年のプロ棋士の研究によりおおかた評価値が定まっているが，京都将棋では一手ごとに駒の性能が変化することから本将棋の評価値をそのまま利用することはできない．

1.4. 本研究の目的

前節で述べた通り，京都将棋では盤面の評価値としてどのような要素を用い，各駒に割り当てる価値をどのようか値にするべきかは定まっていない．そこで，より強い京都将棋 AI を作成するため，本研究では AI 同士を対戦させながら評価関数を調整し，京都将棋において適切な評価値を検証する．

1.5. 本報告書の構成

本報告書の構成は以下の通りである．
２章では京都将棋について解説し，３・４章では作成したプログラムについて，５章には検証結果を示す．

2. 京都将棋について

本章では，京都将棋について説明する．

2.1. 京都将棋の概要

京都将棋とは 1976 年に田宮克哉が発表した将棋の一種である．
基本的なルールは本将棋と同様であるが，玉以外の駒には『香』と『と』，『銀』と『角』，『金』と『桂』，『飛』と『歩』はそれぞれ一つの駒の裏表になっており一手ごとに駒を裏返さなければならないという特別なルールがある．

2.2. 京都将棋のルール

京都将棋の駒は，『玉』『香と』『銀角』『金桂』『飛歩』の５駒からなり，一手ごとに駒を裏返すということ以外それぞれの駒の動きは本将棋と同じである．本将棋の不成のように裏返さないことは許されず，また，持ち駒を打つ際は裏表どちらで打っても良い．
本将棋では禁止だが，京都将棋では禁止されていないルールとして二歩，行き所のない駒，打ち歩詰めがある．京都将棋において持ち駒は表裏どちらで打ってもいいため，本将棋では動くことのできない駒は成らなければならないことから禁止されている行き所のない駒や，歩を打って詰む場合，飛を打っても詰むため打ち歩詰めは禁止する意味が無い．また，京都将棋で歩は本将棋に比べて枚数が少なく打つたびに裏表が変わり飛になることから二歩も禁止されていない．
千日手，連続王手の千日手で引き分けである．
京都将棋の初期盤面を図 1 に示す．

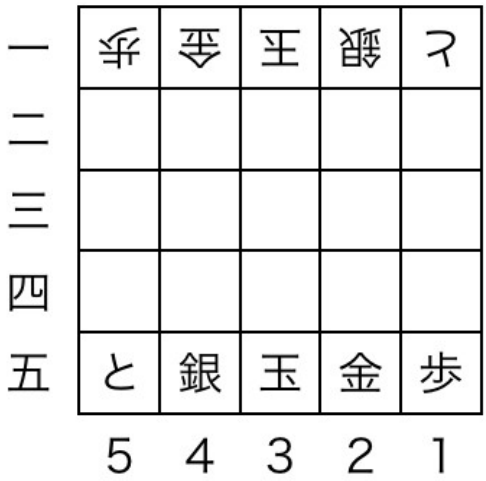

図 1　京都将棋の初期盤面

3. 京都将棋の評価方法

本章では，作成する京都将棋 AI の着手方法について述べる．

本将棋ではプロ棋士による膨大な棋譜から定跡があり，各駒に割り当てる評価値も表 1 に示す通りおおかた定まっている．

京都将棋では棋譜データも少なく適切な値が定まっていない．そこで，本研究では以下のような方法で局面の評価値を求める．

表 1 本将棋における駒の評価値

歩	香	桂	銀	金	角	飛	と	玉
100	600	700	1000	1200	1800	2000	1200	∞

3.1. 局面の評価方法について

本研究では，局面の評価値を算出する要素として，

- 各駒に割り当てた評価値　V_p
- 玉の危険度　D
- 着手可能数　M

を用いる．

本研究で用いる局面の評価値はパラメタを用いて以下の式で与えられる．

$$v*\sum_{p}\left(V_p*\left(N_{1,p}-N_{2,p}\right)\right)+d*\left(D_1-D_2\right)+m*\left(M_1-M_2\right)$$

V_p ：駒({ 香と，銀角，金桂，飛歩 }) の価値
$N_{1,p}$ ：先手の駒の枚数,　　$N_{2,p}$ ：後手の駒の枚数
D_1 ：先手玉の危険度,　　D_2 ：後手玉の危険度
M_1 ：先手の着手可能数,　　M_2 ：後手の着手可能数

先に述べた通り，駒の価値 V_pは京都将棋では適切な値は不明である．自玉の周囲に自駒の利きが多ければ玉を守ることができるので安全，敵駒の利きが多ければ玉が攻められるので危険と考えられる．そこで，自玉の周囲８マスに対して，各マスに効いている自駒の数の和から敵駒の数の和を引いたものを玉の危険度 D とする．着手可能数 M は各手番において着手可能な合法手の数である．一般に，着手可能手が多ければ選択の幅が広がるので有利となり，少なければ不利な手でも指さなければならなくなるため不利と考えられる．

3.2. MiniMax 法

MiniMax 法では数手先の局面の評価値を求め，敵が常に自分にとって最も不利な手を指してくることを仮定して自分の指す手を選択する．

図２に MiniMax 法の例を示す．先手番の時，３手先まで読んだ場合の局面の評価値は１～８にあたる．１～８は先手番のため評価値の大きい方を選択する．次に９～１２は後手番にあたるので先手にとって不利である局面の評価値すなわち小さい方の評価値を選択する．１３～１４は先手番のため評価値の大きい物が選ばれ，評価値２５の手が選ばれる．

それぞれの手番において全ての局面を読むため，深く読むにつれて計算量が増えるため探索に時間がかかる．MiniMax 法を改良した手法として，評価値の探索を行う際に α 以下 β 以上を切り捨てることにより計算量の短縮された $\alpha\beta$ 法もある．

本研究では，最善手の選択にまず MiniMax 法を用いた β 版のプログラムを作成し，β 版上で正しく着手選択しているかの動作確認を行う．動作確認後，計算量の改善を測るために $\alpha\beta$ 法を用いた改良版のプログラムを作成する．

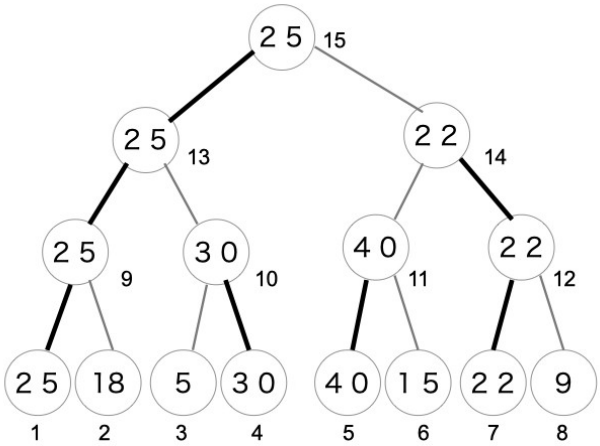

図２　MiniMax 法について

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プログラムについて説明する．本研究で作成した β 版京都将棋プログラムの実行時の様子を図３に示す．

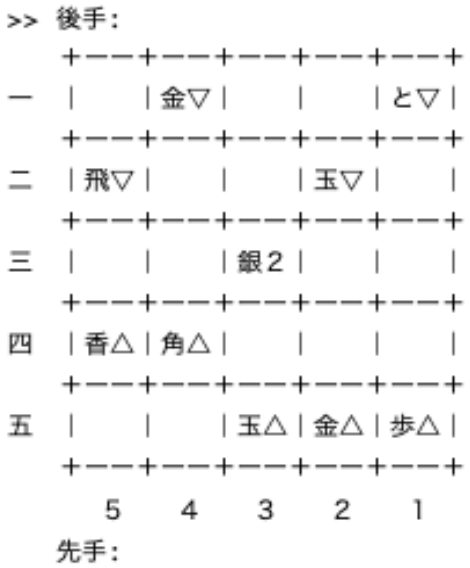

図３　実行の様子

以下では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．

4.1. Koma クラス

Koma クラスは，駒の抽象クラスである．図４に Koma クラスのクラス図を示す．Koma クラスは，駒を打つ先が盤内であるかどうかチェックする checkPosition メソッドや，canWalk メソッド，canRun メソッドなど駒が進める場所を求めるメソッドを持つ．着手可能なマスの数 moveCount メソッドもこのクラスで求める．図３において着手可能数は，先手の４四角の場合３，後手の５二飛の場合５のように計算する．
各駒を表すサブクラスとして，『歩』を表す Fu クラス，『銀』を表す Gin クラスなどがある．図６～１４に示す．

	Koma			#駒を表す抽象クラス
#	isSente	:	boolean	#先手の駒か
#	name	:	char	#駒の番号
#	komaValue1	:	int	#駒の価値 1
#	komaValue2	:	int	#駒の価値 2
#	walk	:	int[][]	#駒が歩いて移動できる方向
#	run	:	int[][]	#駒が走って移動できる方向
+	isSente ()	:	boolean	#先手の駒か
+	isGote ()	:	boolean	#後手の駒か
+	getName ()	:	char	#name フィールドの getter
+	getNumber ()	:	int	#number フィールドの getter
+	getKomaValue1 ()	:	int	#komaValue1 フィールドの getter
+	getKomaValue2 ()	:	int	#komaValue2 フィールドの getter
+	notMove (r:int, c:int)	:	boolean	#動けない駒か
+	checkPosition (r:int, c:int)	:	boolean	#盤面内か
+	addNextPositionByWalk (position:Position, r:int, c:int, walk:int[][], positions:ArrayList<Position>)	:	void	#駒を歩いて進める
+	addNextPositionByRun (position:Position, r:int, c:int, run:int[][], positions: ArrayList<Position>)	:	void	#駒を走って進める
+	moveCountByWalk (position:Position, r:int, c:int, walk:int[][])	:	int	#歩いて進めるマスの数
+	moveCountByRun (position:Position, r:int, c:int, run:int[][])	:	int	#走って進めるマスの数
+	canWalk (position:Position, r:int, c:int, newR:int, newC:int, walk:int[][])	:	booleanm	#歩いて行ける場所か
+	canRun (position:Position, r:int, c:int, newR:int, newC:int, run:int[][])	:	booleanm	#走って行ける場所か
+	addNextPosition (position:Position, r:int, c:int, positions:ArrayList<Position>)	:	void	#駒を動かした時の盤面リスト
+	moveCount (position:Position, r:int, c:int)	:	int	#駒が動ける場所の数
+	canMove (position:Position, r:int, c:int, newR:int, newC:int)	:	boolean	#着手可能な手か
+	*reverse ()*	*:*	*Koma*	#ひっくり返した時の駒
+	*originalKoma ()*	*:*	*Koma*	#表面の駒

図 4 Koma クラスのクラス図

4.2 Position クラス

Position クラスは，盤面を表すクラスである．図 5 に Position クラスのクラス図を示す．Position クラスは，局面の評価値計算を行う value メソッドや，最善手を選択する bestMove メソッドを持つ．

riskGyoku メソッドにより評価値の玉の危険度を求める．玉の危険度は，玉の周り 8 マスの自駒の利きと敵駒の利き数の差によって定められる．戻り値が正の数であれは安全度が高く，負の数であれば危険度が高い．図 3 の場合，３五にある先手の玉の周囲のマスについて自駒の利きが２五の金が３四と２四の２箇所に利いており，敵駒の利きが３三の銀が２四と３四の２箇所に利いている．よってこの場合 riskGyoku メソッドが返す値は 0 である．

Position				# 盤面状態を表すクラス
-	turn	:	boolean	# 先手の駒か
-	boad	:	Koma[][]	# 盤上の駒
-	sentePocket	:	ArrayList<Position>	# 先手の持ち駒
-	gotePocket	:	ArrayList<Position>	# 後手の持ち駒
-	lastMoveR	:	int	# 最後に動いた行
-	lastMoveC	:	int	# 最後に動いた列
-	positionValue	:	int	# 盤面の価値
-	kingR1	:	int	# 先手玉の行
-	kingC1	:	int	# 先手玉の列
-	kingR2	:	int	# 後手玉の行
-	kingC2	:	int	# 後手玉の列
-	turnNumber	:	int	# 手数の番号
-	depth	:	int	# 読みの深さ
-	v1	:	int	# 駒の価値のパラメタ
-	d1	:	int	# 玉の危険度のパラメタ
-	m1	:	int	# 着手可能数のパラメタ
-	v2	:	int	# 駒の価値のパラメタ
-	d2	:	int	# 玉の危険度のパラメタ
-	m2	:	int	# 着手可能数のパラメタ
+	Position (depth: int)	:		# 初期盤面の設定
+	Position (oldPosition: Position)	:		# 盤面のコピーする
+	getTurn ()	:	boolean	# どちらの手番か
+	getKoma (r: int, c: int)	:	Koma	# 盤のマス目にある駒を得る
+	getLastMoveR ()	:	int	# 最後に動いた行を得る
+	getLastMoveC ()	:	int	# 最後に動いた列を得る
+	getSentePocket ()	:	ArrayList<Koma>	# 先手の持ち駒を返す
+	getGotePocket ()	:	ArrayList<Koma>	# 後手の持ち駒を返す
+	getPositionValue ()	:	int	# 駒の価値を返す
+	canDrop (p: int, newR: int, newC: int)	:	boolean	# 駒が打てるかどうか
+	move (oldR: int, oldC: int, newR: int, newC: int)	:	Position	# 駒を動かす
+	drop (h: int, r: boolean, newR: int, newC: int)	:	Position	# 表で打つか
+	addHand (hand: ArrayList<Koma>, koma: Koma)	:	void	# 持ち駒に加える
+	printBoad ()	:	void	# 盤面の表示
+	printLine ()	:	void	# 盤面の横線を引く
+	bestMove ()	:	Position	# 最善手を求める
+	nextMoves ()	:	ArrayList<Position>	# 次に可能な盤面リストを求める
+	getKomaValue (r: int, c: int, t: boolean)	:	int	# 駒の価値
+	value (d: int)	:	int	# 盤面の価値を決める
+	searchKing ()	:	void	# 玉の位置を返す
+	riskGyoku (p: Position)	:	int	# 玉の危険度

図 5 Position クラスのクラス図

Fu			# 『歩』を表すサブクラス
- walk1	:	int[][]	# 駒が歩く方向
+ Fu (isSente: boolean, k1: int, k2: int)	:		
+ reverse ()	:	Koma	# 裏返した駒
+ originalKoma ()	:	Koma	# 表の駒
+ notMove (r: int, c: int)	:	boolean	# 動けない駒か

図6　Fu クラスのクラス図

Gin			# 『銀』を表すサブクラス
- walk1	:	int[][]	# 駒が歩く方向
+ Gin (isSente: boolean, k1: int, k2: int)	:		
+ reverse ()	:	Koma	# 裏返した駒
+ originalKoma ()	:	Koma	# 表の駒

図7　Gin クラスのクラス図

Gyoku			# 『玉』を表すサブクラス
- walk1	:	int[][]	# 駒が歩く方向
+ Gyoku (isSente: boolean, k1: int, k2: int)	:		
+ reverse ()	:	Koma	# 裏返した駒
+ originalKoma ()	:	Koma	# 表の駒

図8　Gyoku クラスのクラス図

Hi			# 『飛』を表すサブクラス
- walk1	:	int[][]	# 駒が歩く方向
+ Hi (isSente: boolean, k1: int, k2: int)	:		
+ reverse ()	:	Koma	# 裏返した駒
+ originalKoma ()	:	Koma	# 表の駒

図9　Hi クラスのクラス図

Kaku			# 『角』を表すサブクラス
- walk1	:	int[][]	# 駒が歩く方向
+ Kaku (isSente: boolean, k1: int, k2: int)	:		
+ reverse ()	:	Koma	# 裏返した駒
+ originalKoma ()	:	Koma	# 表の駒

図１０　Kaku クラスのクラス図

Kei			# 『桂』を表すサブクラス
- walk1	:	int[][]	# 駒が歩く方向
+ Kei (isSente: boolean, k1: int, k2: int)	:		
+ reverse ()	:	Koma	# 裏返した駒
+ originalKoma ()	:	Koma	# 表の駒
+ notMove (r: int, c: int)	:	boolean	# 動けない駒か

図１１　Kei クラスのクラス図

Kin			# 『金』を表すサブクラス
- walk1	:	int[][]	# 駒が歩く方向
+ Kin (isSente: boolean, k1: int, k2: int)	:		
+ reverse ()	:	Koma	# 裏返した駒
+ originalKoma ()	:	Koma	# 表の駒

図１２　Kin クラスのクラス図

Kyo			# 『香』を表すサブクラス
- walk1	:	int[][]	# 駒が歩く方向
+ Kyo (isSente: boolean, k1: int, k2: int)	:		
+ reverse ()	:	Koma	# 裏返した駒
+ originalKoma ()	:	Koma	# 表の駒
+ notMove (r: int, c: int)	:	boolean	# 動けない駒か

図１３　Kyo クラスのクラス図

To			# 『と』を表すサブクラス
- walk1	:	int[][]	# 駒が歩く方向
+ To (isSente: boolean, k1: int, k2: int)	:		
+ reverse ()	:	Koma	# 裏返した駒
+ originalKoma ()	:	Koma	# 表の駒

図１４　To クラスのクラス図

5. 京都将棋における局面の評価値の検証

本章では，３章で述べた評価関数の値を変えながら AI 同士を対戦させ評価値を検証する．

評価方法として表２のような５つの評価値を用意した．特定の条件によって駒が成る本将棋とは異なり京都将棋では駒の裏表の変化が頻繁であるため裏表で一つの駒として考えた．京都将棋で利用される駒のうち本将棋で利用されている評価値は一番小さいもので歩が 100，一番大きいもので飛 2000 であるため 100 から 2000 を目安に評価関数の初期値を割り当てた．何度か実行した結果パラメタｖについては値の大きい方が圧倒的な強さになってしまうため本研究では一定とした．

それぞれの評価値で対戦させ結果から負けた方の評価値の一部を勝った方に近づけ，得られた評価値で再び対戦を繰り返す．具体的には，負けた方の４つの駒に割り当てる評価値のうち２～３つの駒の評価値をそれぞれ 100～300，パラメタｄ，ｍのうちどちらかを１～２勝った方に近づけた．

評価値 A の対戦結果を表３に，対戦結果により値を変えた評価値を表４に示した．

表 2　評価値 A

	香と	銀角	金桂	飛歩	v	d	m
A1	1000	1000	1000	1000	1	1	5
A2	500	2000	100	1500	1	2	4
A3	1500	100	2000	500	1	3	3
A4	2000	1500	500	100	1	4	2
A5	100	500	1500	2000	1	5	1

表 3　対戦結果 A (試行回数各 100 回)

評価値	勝:負:引		評価値	勝:負:引
A1:A2	100:0:0		A2:A4	46:54:0
A1:A3	37:0:63		A2:A5	36:60:2
A1:A4	91:7:2		A3:A4	65:0:35
A1:A5	96:4:0		A3:A5	26:74:0
A2:A3	0:100:0		A4:A5	100:0:0

表 4　評価値 B

	香と	銀角	金桂	飛歩	v	d	m
B1	600	1800	200	1500	1	1	4
B2	1500	300	1800	600	1	2	3
B3	1900	1500	600	200	1	2	2
B4	200	600	1500	1800	1	5	3
B5	600	1900	300	1500	1	3	4
B6	700	1900	100	1300	1	2	3
B7	200	2000	300	1600	1	3	4
B8	2000	1300	800	200	1	4	3
B9	1300	100	1900	700	1	4	3
B10	300	600	1300	2000	1	4	1

表 5　評価値 W

	香と	銀角	金桂	飛歩	v	d	m
W1	1100	700	1100	800	1	4	3
W2	900	800	1200	900	1	4	2
W3	1100	900	1000	900	1	4	2

この実験を繰り返したところ表 5 の結果が得られた.

４つの駒の評価値に大きな差は見られなかったが，香と，金桂を重視する結果となった．本将棋における評価値が香 600 と 1200 や，銀 1000 角 1800 などからわかるように弱い駒と強う駒が裏表になっていることが原因として考えられる．また，パラメタ d，m は d の値が大きい方が強い AI となり玉の危険度を重視することでより強い AI になることがわかった．

6. 結論・今後の課題

本研究では京都将棋 AI を作成し，京都将棋において適切な評価関数の検証を行った．評価値を調整しながら対戦を繰り返すことで収束した評価値の結果から，強い AI にするためには，銀角，飛歩よりも香と，金桂の価値を高めるべきであること，玉の危険度を重視すべきであることがわかった．

今後の課題として，得られた評価値がどの程度の強さであるかを検証しさらに強い AI の作成を目指すことや，最善手選択手法を MiniMax 法から $\alpha\beta$ 法に改良しより深く探索を行うことがあげられる．本研究では裏表の駒を一つの駒として検証したが裏表で評価値を変えた場合の強さの変化も検証する必要がある．また，検証方法についても手動で行うのではなく自動化することでより網羅的な検証が期待できる．

他にもより多くの棋譜データを集め定跡データベースの選定や機械学習など他の手法も試すことでさらに強い AI になることが考えられる．

謝辞

本研究を行うにあたり，石水隆講師には大変お世話になりました．至らない点もたくさんありご迷惑をおかけしましたが最後までご指導いただき本当にありがとうございました．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] 山岡忠夫, 将棋 AI で学ぶディープラーニング, マイナビ出版 (2018)

[3] Android 版【京都将棋】アプリ配信スタート！, 将棋をもっと楽しく親しみやすく、世界へ 2016 年 9 月 26 日, 株式会社ねこまど, http://nekomadoblog.jugem.jp/?eid=1385

[4] 京都将棋, んとか将棋, 将棋ゲームの時間, https://syouginojikan.web.fc2.com/kyouto.html

[5] 京都銀閣将棋, shogitter, http://shogitter.com/rule/19

付録

本研究で作成したソースコードを以下に示す.

Position クラス

```
package kyotoshogi;

import java.util.ArrayList;
import java.util.Random;

/**
 *
 * @author abui
 *
 * 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 Koma[][] boad; //
        private ArrayList<Koma> sentePocket; // 先手の持ち駒
        private ArrayList<Koma> gotePocket; // 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 int kingR1; // 行
        private int kingC1; // 列
        private int kingR2;
        private int kingC2;

        private static int turnNumber = 0; // 手数
        private static int depth; // 読みの深さ
        private static Random random = new Random();

        /* パラメタ */
        private final int v1 = 1; // 駒の価値
        private final int d1 = 4; // 玉の危険度
        private final int m1 = 2; // 着手可能数

        private final int v2 = 1;
        private final int d2 = 4;
        private final int m2 = 2;

        /**
         * 初期盤面の設定
         * @param depth
         */
        public Position(int depth){
           boad = new Koma[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, 900, 1100);
        boad[0][1] = new Gin(true, 800, 900);
        boad[0][2] = new Gyoku(true);
        boad[0][3] = new Kin(true, 1200, 1000);
        boad[0][4] = new Fu(true, 900, 900);

        boad[4][4] = new To(false, 900, 1100);
        boad[4][3] = new Gin(false, 800, 900);
        boad[4][2] = new Gyoku(false);
        boad[4][1] = new Kin(false, 1200, 1000);
        boad[4][0] = new Fu(false, 900, 900);

        sentePocket = new ArrayList<Koma>();
        gotePocket = new ArrayList<Koma>();
        turn = true;
        Position.depth = depth;
        turnNumber = 0;
    }

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

    /**
     * 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
     * @param r  行
     * @param c  列
     * @return
     */
    public Koma getKoma(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<Koma> getSentePocket(){
    return sentePocket;
}

/**
 * 後手の持ち駒を返す
 * @return
 */
public ArrayList<Koma> getGotePocket(){
    return gotePocket;
}

/**
 * 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 p
 * @param newR
 * @param newC
 * @return
 */
public boolean canDrop(int p, int newR, int newC){
    if(boad[newR][newC] != null){
        return false;
    } else {
        if(turn){
            return p < sentePocket.size();
        } else {
            return p < gotePocket.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.sentePocket.add(boad[newR][newC].originalKoma());
     } else {
         newPosition.gotePocket.add(boad[newR][newC].originalKoma());
     }
 }
 newPosition.boad[newR][newC] = boad[oldR][oldC].reverse();
 newPosition.boad[oldR][oldC] = null;
 newPosition.turn = !turn;
 return newPosition;
}

/**
 * 駒をうつ
 * @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;
    Koma koma;
    if(turn){
        koma = newPosition.sentePocket.remove(h);
    } else {
        koma = newPosition.gotePocket.remove(h);
    }
    if(!r){
        koma = koma.reverse();
    }
    newPosition.boad[newR][newC] = koma;
    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 koma  加える駒
 */
public void addHand(ArrayList<Koma> hand, Koma koma){
    if(hand.size() == 0){
        hand.add(koma);
    } else {
        for(int i = 0; i < hand.size(); i++){
            if(hand.get(i).getNumber() > koma.getNumber()){
                hand.add(i, koma);
            }
        }
    }
}
```

```
/**
 * 盤面の表示
 */
public void printBoad(){
    System.out.printf("  === %2d ====================\n", turnNumber);
       if(turn){
          System.out.printf(">> ");
       } else {
          System.out.print("   ");
       }
    System.out.printf("後手: ");
    for(Koma k : gotePocket){
       System.out.printf("%c ", k.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].isSente){
             if(4 - i == lastMoveR && j == lastMoveC){
                System.out.printf("%c１|", 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("%c２|", boad[4 - i][j].getName());
             } else {
                System.out.printf("%c▽|", boad[4 - i][j].getName());
             }
          }
       }
    System.out.println();
    printLine();
    }
    System.out.println("     １   ２   ３   ４   ５");
    if(turn){
       System.out.print("   ");
    } else {
       System.out.printf(">> ");
    }
    System.out.printf("先手: ");
    for(Koma k : sentePocket){
       System.out.printf("%c ", k.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
 * @return
 */
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].isSente() == 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 < sentePocket.size(); p++){
                        newPositions.add(drop(p, true, i, j));
                        newPositions.add(drop(p, false, i, j));
                    }
                } else {
                    for(int p = 0;p < gotePocket.size(); p++){
                        newPositions.add(drop(p, true, i, j));
                        newPositions.add(drop(p, false, i, j));
                    }
                }
            }
        }
    }
    return newPositions;
}

/**
 * 駒の価値
 * @param r  駒の行
 * @param c  駒の価値
 * @param t  手番
 * @return
 */
public int getKomaValue(int r, int c, boolean t){
    if(t){
        return boad[r][c].getKomaValue1();
    } else {
        return boad[r][c].getKomaValue2();
    }
}

/**
 * 盤面の価値を決める
 * @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].isSente()){
                    if(boad[i][j].notMove(i, j)){
                        value += m1*boad[i][j].moveCount(this, i, j);
                    } else {
                        value += v1*getKomaValue(i, j, t)
                                  + m1*boad[i][j].moveCount(this, i, j);
                    }
                } else {
                    if(boad[i][j].notMove(i, j)){
```

```
                    value -= m2*boad[i][j].moveCount(this, i,j);
                } else {
                    value -= v2*getKomaValue(i, j, t)
                              + m2*boad[i][j].moveCount(this, i, j);
                }
            }
        }
    }
    for(Koma k : sentePocket){
        if(t){
            value += v1*k.getKomaValue1();
        } else {
            value += v2*k.getKomaValue2();
        }
    }
    for(Koma k : gotePocket){
        if(t){
            value -= v1*k.getKomaValue1();
        } else {
            value -= v2*k.getKomaValue2();
        }
    }
} else {
    int  v;
    v = value(0);
    if(!turn){
        if(v > 15000){
            return v;
        }
    } else {
        if(v < -15000){
            return v;
        }
    }
    ArrayList<Position> newPositions = nextMoves();
    if(turn){
        value = -100000;
        for(Position p : newPositions){
            v = p.value(d-1) + d1*riskGyoku(p);
            if(v > value){
                value = v;
                }
            }
    } else {
        value = 100000;
        for(Position p : newPositions){
            v = p.value(d-1) - d2*riskGyoku(p);
            if(v < value){
                value = v;
            }
        }
    }
}
return value;
}

public Position humaMove(int r, int c, int newR, int newC){
```

```
    Position ret = null;
    int d;
    if(r == 0){
       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 {
       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;
}

/**
 * 玉の位置を探す
 */
public void searchKing(){
    for(int i = 0; i < 5; i++){
       for(int j = 0; j < 5; j++){
          if(boad[i][j]==null){
          } else if(boad[i][j].isSente){
             if(boad[i][j].getNumber()==0){
                kingR1 = i;
                kingC1 = j;
             }
          } else if(!boad[i][j].isSente){
             if(boad[i][j].getNumber()==0){
                kingR2 = i;
                kingC2 = j;
          }
       }
    }
  }
}

/**
 * 玉の危険度
 * @param p
 * @param t
 * @return
 */
public int riskGyoku(Position p){
    int[][] k1 = new int[5][5];
    int[][] k2 = new int[5][5];
    p.searchKing();
    int count1 = 0; // 玉の周り8マスの自駒の利き数
    int count2 = 0; // 玉の周り8マスの敵駒の利き数
```

```
if(turn){
    for(int i = 0; i < 5; i++){
        for(int j = 0; j < 5; j++){
            if(p.getKoma(i, j) != null){
              if(p.getKoma(i, j).isSente
                                        && p.getKoma(i, j).getNumber()!=0){
                  for(int s = -1; s < 2; s++){
                      for(int t = -1; t < 2; t++){
                          int r = kingR1 + s;
                          int c = kingC1 + t;
                          if(p.getKoma(i, j).canMove(p, i, j, r, c)
                                                    && k1[r][c]==0){
                              k1[r][c] = 1;
                          }
                      }
                  }
              } else if(!p.getKoma(i, j).isSente){
                  for(int s = -1; s < 2; s++){
                      for(int t = -1; t < 2; t++){
                          int r = kingR1 + s;
                          int c = kingC1 + t;
                          if(p.getKoma(i, j).canMove(p, i, j, r, c)
                                                    && k2[r][c]==0){
                              k2[r][c] = 1;
                          }
                      }
                  }
              }
            }
        }
    }
} else {
    for(int i = 0; i < 5; i++){
        for(int j = 0; j < 5; j++){
            if(p.getKoma(i, j) != null){
                if(!p.getKoma(i, j).isSente
                                        && p.getKoma(i, j).getNumber()!=0){
                  for(int s = -1; s < 2; s++){
                      for(int t = -1; t < 2; t++){
                          int r = kingR2 + s;
                          int c = kingC2 + t;
                          if(p.getKoma(i, j).canMove(p, i, j, r, c)
                                                    && k1[r][c]==0){
                              k1[r][c] = 1;
                          }
                      }
                  }
              } else if(p.getKoma(i, j).isSente){
                  for(int s = -1; s < 2; s++){
                      for(int t = -1; t < 2; t++){
                          int r = kingR2 + s;
                          int c = kingC2 + t;
                          if(p.getKoma(i, j).canMove(p, i, j, r, c)
                                                    && k2[r][c]==0){
                              k2[r][c] = 1;
                          }
```

```
                            }
                        }
                    }
                }
            }
        }
        for(int i = 0; i < 5; i++){
            for(int j = 0; j < 5; j++){
                if(k1[i][j]!=0){
                    count1++;
                }
            }
        }
        for(int i = 0; i < 5; i++){
            for(int j = 0; j < 5; j++){
                if(k2[i][j]!=0){
                    count2++;
                }
            }
        }
        return count1-count2;
    }

    /**
     * 局面の着手可能数
     * @param p
     * @param t
     * @return
     */
    public int canPut(Position p){
        int count = 0;
        if(turn){
            for(int i = 0; i < 5; i++){
                for(int j = 0; j < 5; j++){
                    if(p.getKoma(i, j) != null && p.getKoma(i, j).isSente){
                        count += p.getKoma(i, j).moveCount(p, i, j);
                    }
                }
            }
        } else {
            for(int i = 0; i < 5; i++){
                for(int j = 0; j < 5; j++){
                    if(p.getKoma(i, j) != null && !p.getKoma(i, j).isSente){
                        count += p.getKoma(i, j).moveCount(p, i, j);
                    }
                }
            }
        }
        return count;
    }

    public int getTurnNumber(){
        return turnNumber;
    }

}
```

Koma クラス

```
package kyotoshogi;

import java.util.ArrayList;

public abstract class Koma {

        protected boolean isSente; // true:先手の駒、false:後手の駒
        protected char name; // 駒の名前
        protected int number; // 駒の番号
        protected int komaValue1; // 駒の価値1
        protected int komaValue2; // 駒の番号2
        protected int[][] walk = {};
        protected int[][] run = {};

        /**
         * 先手の駒かどうか
         * @return
         */
        public boolean isSente(){
            return isSente;
        }

        /**
         * 後手の駒かどうか
         * @return
         */
        public boolean isGote(){
            return !isSente;
        }

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

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

        /**
         * 駒の価値1
         * @return
         */
        public int getKomaValue1(){
            return komaValue1;
        }

        /**
         * 駒の価値2
         * @return
         */
        public int getKomaValue2(){
```

```
    return komaValue2;
}

/**
 * 動けない駒かどうか
 * @param r  行
 * @param c  列
 * @return
 */
public boolean notMove(int r, int c){
    return false;
}

/**
 * 盤面内かどうか
 * @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  行
 * @param c  列
 * @param walk  歩いて進める方向
 * @param positions  動いた後の盤面
 */
public void addNextPositionByWalk(Position position, int r, int c
                            , int[][] walk, ArrayList<Position> positions){
    int newR;
    int newC;
    for(int i = 0; i < walk.length; i++){
        if(isSente){
            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.getKoma(newR, newC)==null){
                positions.add(position.move(r, c, newR, newC));
            } else if(position.getKoma(newR, newC).isSente()!=isSente){
                positions.add(position.move(r, c, newR, newC));
            }
        }
    }
}

/**
 * 走って進む
 * @param position  現在の盤面
 * @param r  行
 * @param c  列
```

```
 * @param run  走って進める方向
 * @param positions  動いた後の盤面
 */
public void addNextPositionByRun(Position position, int r, int c
                            , int[][] run, ArrayList<Position> positions){
    int newR;
    int newC;
    for(int i = 0; i < run.length; i++){
        for(int j = 0; j < 4; j++){
            if(isSente){
                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.getKoma(newR, newC)==null){
                positions.add(position.move(r, c, newR, newC));
            } else if (position.getKoma(newR, newC).isSente!=isSente){
                positions.add(position.move(r, c, newR, newC));
                    break; // 駒をとったら終わり
                } else {
                    break; // 味方の駒があれば
                }
            }
        }
    }
}

/**
 * 歩いて進める場所の数
 * @param position  現在の盤面
 * @param r  行
 * @param c  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(isSente){
            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.getKoma(newR, newC)==null){
                ret++;
            } else if(position.getKoma(newR, newC).isSente()!=isSente){
                ret++;
            }
```

```
        }
    }
    return ret;
}

/**
 * 走って進める場所の数
 * @param positon  現在の盤面
 * @param r  行
 * @param c  列
 * @param run  走って進める方向
 * @return
 */
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(isSente){
                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.getKoma(newR, newC)==null){
                    ret++;
                } else if(position.getKoma(newR, newC).isSente()!=isSente){
                    ret++;
                    break;
                } else {
                    break;
                }
            }
        }
    }
    return ret;
}

/**
 * 歩いていける場所かどうかのチェック
 * @param position  現在の盤面
 * @param r  行
 * @param c  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)){
        return ret;
    }
```

```
            if(position.getKoma(newR, newC)!=null){
                if(position.getKoma(newR, newC).isSente==isSente){
                    return ret;
                }
            }
            for(int i = 0; i < walk.length; i++){
                if(isSente){
                    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;
        }

/**
* 走っていける場所かどうかのチェック
* @param position  現在の盤面
* @param r  行
* @param c  列
* @param newR  行き先の行
* @param newC  行き先の列
* @param run  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)){
                return ret;
            }
            if (position.getKoma(newR, newC) != null){
                if(position.getKoma(newR, newC).isSente == isSente){
                    return ret;
                }
            }
            for(int i = 0; i < run.length; i++){
                for(int j = 0; j < 4; j++){
                    int pr;
                    int pc;
                    if(isSente){
                        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.getKoma(pr, pc) == null){
            if(newR == pr && newC == pc){
               return true;
            }
         } else if(position.getKoma(pr, pc).isSente != isSente){
            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  行
 * @param c  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  行
 * @param c  列
 * @return
 */
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 position  現在の盤面
 * @param r  行(持ち駒は-1)
 * @param c  列
 * @param newR  動かす行
 * @param newC  動かす行
 * @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 Koma reverse();

        /**
         * 表面の駒
         * @return
         */
        public abstract Koma originalKoma();

}
```

Fu クラス

```
package kyotoshogi;

/**
* 歩
* @author abui
*
* 前に 1 マスだけ動ける
*
*/
public class Fu extends Koma{

        private int[][] walk1 = {{1,0}};

        public Fu(boolean isSente, int k1, int k2){
            this.isSente = isSente;
            this.name = '歩';
            this.number = 1;
            this.komaValue1 = k1;
            this.komaValue2 = k2;
            walk = walk1;
        }

        public Koma reverse(){
            return new Hi(isSente, komaValue1, komaValue2);
        }

        public Koma originalKoma(){
            return new Fu(!isSente, komaValue1, komaValue2);
        }

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

}
```

Gin クラス

```
package kyotoshogi;

/**
* 銀
```

```
 * @author abui
 *
 * 斜めと前に 1 マスだけ動ける
 *
 */
public class Gin extends Koma{

        private int[][] walk1 = {{1,-1},{1,0},{1,1},{-1,-1},{-1,1}};

        public Gin(boolean isSente, int k1, int k2){
            this.isSente = isSente;
            this.name = '銀';
            this.number = 7;
            this.komaValue1 = k1;
            this.komaValue2 = k2;
            walk = walk1;
        }

        public Koma reverse(){
            return new Kaku(isSente, komaValue1, komaValue2);
        }

        public Koma originalKoma(){
            return new Gin(!isSente, komaValue1, komaValue2);
        }

}
```

Gyoku クラス

```
package kyotoshogi;

/**
 * 玉
 * @author abui
 *
 * 全方向に 1 マスだけ動ける
 *
 */
public class Gyoku extends Koma{

        private int[][] walk1 = {{1,-1},{1,0},{1,1},{0,-1}
                                                    ,{0,1},{-1,-1},{-1,0},{-1,1}};

        public Gyoku(boolean isSente){
            this.isSente = isSente;
            this.name = '玉';
            this.number = 0;
            this.komaValue1 = 10000;
            this.komaValue2 = 10000;
            walk = walk1;
        }

        public Koma reverse(){
            return new Gyoku(isSente);
        }

        public Koma originalKoma(){
```

```
            return new Gyoku(!isSente);
        }

}
```

Hi クラス

```
package kyotoshogi;

/**
 * 飛
 * @author abui
 *
 * 縦横に何マスでも動ける
 *
 */
public class Hi extends Koma{

        private int run1[][] = {{1,0},{0,-1},{0,1},{-1,0}};

        public Hi(boolean isSente, int k1, int k2){
            this.isSente = isSente;
            this.name = '飛';
            this.number = 2;
            this.komaValue1 = k1;
            this.komaValue2 = k2;
            run = run1;
        }

        public Koma reverse(){
            return new Fu(isSente, komaValue1, komaValue2);
        }

        public Koma originalKoma(){
            return new Fu(!isSente, komaValue1, komaValue2);
        }

}
```

Kaku クラス

```
package kyotoshogi;

/**
 * 角
 * @author abui
 *
 *  斜めに何マスでも動ける
 *
 */
public class Kaku extends Koma{

        private int run1[][] = {{1,-1},{1,1},{-1,-1},{-1,1}};

        public Kaku(boolean isSente, int k1, int k2){
            this.isSente = isSente;
```

```
            this.name = '角';
            this.number = 8;
            this.komaValue1 = k1;
            this.komaValue2 = k2;
            run = run1;
        }

        public Koma reverse(){
            return new Gin(isSente, komaValue1, komaValue2);
        }

        public Koma originalKoma(){
            return new Gin(!isSente, komaValue1, komaValue2);
        }

}
```

Kei クラス

```
package kyotoshogi;

/**
 * 桂
 * @author abui
 *
 * 前1,横2の位置に動ける
 *
 */
public class Kei extends Koma{

        private int[][] walk1 = {{2,-1},{2,1}};

        public Kei(boolean isSente, int k1, int k2){
            this.isSente = isSente;
            this.name = '桂';
            this.number = 6;
            this.komaValue1 = k1;
            this.komaValue2 = k2;
            walk = walk1;
        }

        public Koma reverse(){
            return new Kin(isSente, komaValue1, komaValue2);
        }

        public Koma originalKoma(){
            return new Kin(!isSente, komaValue1, komaValue2);
        }

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

}
```

Kin クラス

```
package kyotoshogi;

/**
 * 金
 * @author abui
 *
 * 縦横と斜め前に 1 マスだけ動ける
 *
 */
public class Kin extends Koma{

        private int[][] walk1 = {{1,-1},{1,0},{1,1},{0,-1},{0,1},{-1,0}};

        public Kin(boolean isSente, int k1, int k2){
            this.isSente = isSente;
            this.name = '金';
            this.number = 5;
            this.komaValue1 = k1;
            this.komaValue2 = k2;
            walk = walk1;
        }

        public Koma reverse(){
            return new Kin(isSente, komaValue1, komaValue2);
        }

        public Koma originalKoma(){
            return new Kin(!isSente, komaValue1, komaValue2);
        }

}
```

Kyo クラス

```
package kyotoshogi;

/**
 * 香
 * @author abui
 *
 * 前に何マスでも動ける
 *
 */
public class Kyo extends Koma{

        private int run1[][] = {{1,0}};

        public Kyo(boolean isSente, int k1, int k2){
            this.isSente = isSente;
            this.name = '香';
            this.number = 3;
            this.komaValue1 = k1;
            this.komaValue2 = k2;
            run = run1;
        }

        public Koma reverse(){
                return new To(isSente, komaValue1, komaValue2);
        }
```

```
        public Koma originalKoma(){
            return new Kyo(!isSente, komaValue1, komaValue2);
        }

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

}
```

To クラス

```
package kyotoshogi;

/**
 * と
 * @author abui
 *
 * 縦横と斜め前に動ける
 *
 */
public class To extends Koma{

        private int[][] walk1 = {{1,-1},{1,0},{1,1},{0,-1},{0,1},{-1,0}};

        public To(boolean isSente, int k1, int k2){
            this.isSente = isSente;
            this.name = 'と';
            this.number = 4;
            this.komaValue1 = k1;
            this.komaValue2 = k2;
            walk = walk1;
        }

        public Koma reverse(){
            return new Kyo(isSente, komaValue1, komaValue2);
        }

        public Koma originalKoma(){
            return new Kyo(!isSente, komaValue1, komaValue2);
        }

}
```

Play クラス

```
package kyotoshogi;

import java.util.Scanner;

public class Play {

        public static void main(String[] args){
            int s = 0;
            int g = 0;
            int h = 0;
         for(int i = 0; i < 100; i++){
```

```
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 */
    p = p.bestMove();
    p.printBoad();
    int v = p.getPositionValue();
    System.out.printf("(%d)### %d ###\n", i+1, v);

    int tn = p.getTurnNumber();

    if(v > 15000  && tn < 50){
        System.out.println("先手の勝ち");
        s++;
        break;
    } else if(v < -15000 && tn < 50){
        System.out.println("後手の勝ち");
        g++;
        break;
    } else if(tn >= 50){
        h++;
        //i--;
        break;
    }
    /* ここまで */


    /*人間
    while (true){
        System.out.print("Please input next move:");
        r = kbd.nextInt();
        c = kbd.nextInt();
        newR = kbd.nextInt();
        newC = kbd.nextInt();
        tmpP = p.humaMove(r, c, newR, newC);
        if(tmpP != null){
            p = tmpP;
            break;
        }
    }

    p.printBoad();
    v = p.getPositionValue();
    System.out.printf("### %d ###\n", v);

    if(v > 15000  && tn < 50){
        System.out.println("先手の勝ち");
        s++;
        break;
    } else if(v < -15000 && tn < 50){
```

```
                    System.out.println("後手の勝ち");
                    g++;
                    break;
                }

                //if(v > 15000 || v < -15000){
                //if(t){
                //    System.out.println("後手の勝ち");
                //} else {
                //System.out.println("先手の勝ち");
                //}
                   break;
             //}
                        /* ここまで */

           }

           kbd.close();
       }
       System.out.println("先手：" + s + "," + "後手：" + g + ", 引分：" + h);
   }

}
```