卒業研究報告書

題目

チェスのポーンエンディングの解析

指導教員

石水 隆　講師

報告者

10-1-037-0054

吉田 駿

近畿大学理工学部情報学科

平成 26 年 1 月 31 日提出

概要

チェスは、囲碁・将棋と同じような盤上競技の遊びであり、二人の対戦者が交互に白と黒の駒を動かし、相手のキング（将棋でいう王将）を追い詰めることで勝利することができる。ルールでよく似ているとされている将棋との違いは主に駒で、将棋でいう成りがチェスではポーンしかなく、飛車と角を混ぜたようなクイーンという駒が存在する。また、桂馬の全方位番のようなナイトも存在する。他には、盤面の大きさ、一度取った相手の駒は使用することができないことなどが挙げられる。将棋においてコンピュータと対戦できるコンピュータ将棋が多数作られたのと同様に、チェスもコンピュータチェスが数多く作られている。コンピュータチェスは 1997 年に IBM のスーパーコンピュータ「ディープブルー」が当時チェスの世界チャンピオンだったゲイリー・カスパロフと対戦し、勝利したことがあり、その後も多くのチェスの名人がコンピュータチェスに敗れている。このことから、現在ではコンピュータチェスは人間より強いといわれている。また、チェスは、先ほど記述したように、将棋と異なり一度取った駒は使用できないため、終盤では少数の駒のみの局面がよく現れる。終盤の場面の中でも、双方のキング以外はポーンのみが残っている局面はポーンエンディング（以下 P.E.とする）と呼ばれる。

本研究では、チェスの P.E.の解析を行った。そのために、P.E.のプログラムを Java のプログラムを用いて作成した。具体的な内容としては、攻撃側がポーンとキング、防御側がキングのみという局面を作り、勝率、勝敗が決定するまでの手数を解析した。

目次

1 序論 1

2P.E.について 2

3 研究内容 6

4 結果・考察 10

5 結論・今後の課題 10

参考文献 13

付録 14

1　序論

1.1.本研究の背景

チェスは、囲碁・将棋と同じような盤上将棋の仲間で、二人の対戦者が交互に白と黒の駒を動かし、相手のキングを追い詰める遊びである。将棋とよく比較されるチェスは、類似点としてポーンと歩、ルークと飛車など似たような駒があることが挙げられる。相違点としては将棋は一度取った相手の駒を再使用することに対し、チェスでは一度取った相手の駒を使用することができない。そのためチェスは将棋に比べて局面数が少ないと言える。

チェスは二人零和有限確定完全情報ゲームに分類され、双方が最善手を指した場合、先手勝ち、後手勝ち、引き分けのいずれになるかはゲーム開始時点て決定している。理論上は、可能な全ての局面を解析できれば最善手を指すことができる。しかし、チェスの可能な局面数はおよそ10^{50}通りあるとされており、現在の計算機能力では完全解析は不可能である。

しかし、完全解析はできなくとも、強いとされる手を計算機によって求めることはできる。将棋においてコンピュータと対戦できるコンピュータ将棋が多数作られたのと同様に、チェスもコンピュータチェスが数多く作られている。代表的なチェスソフトはCHESS4.6、BELLE、DEEP THOUGHT、HITECH などが挙げられる[4]。現在、コンピュータチェスの強さは人間を上回るとされている。1997年にニューヨークで当時チェスのチャンピオンだったガリ・カスパロフがIBMのチェス専用マシン「ディープブルー」に敗れている[15]。

コンピュータチェスにおいて使われている戦略の代表的なものとしては、局面の評価値計算、定跡データベース、対戦データベースの利用、終盤での完全読みきりなどがある。評価値計算は先読みした盤面ごとに評価値を良い手を選択する戦略、定跡データベースは定跡を予め記憶させそれに従い手を指す戦略、対戦データベースの利用は様々なチェスの対戦のデータを参照し良い手を指す戦略、終盤での完全読みきりは残りの少ない局面数から勝てる局面を先読みし良い手を指す戦略である。

チェスは将棋と違い取った相手の駒を使用することができない。そのためにチェスの終盤局面では駒の数が少ない場面がよく現れる。特に双方のキング以外の駒がポーンのみの場面をポーンエンディングと呼ぶ（以下P.E.とする）。

終盤において双方の駒が少なくなると、可能な局面数が減るため、完全解析可能となる。1980年代には、残り駒数が5駒以下の場合の完全解析がされている。1984年、Nefkens は、盤上に双方のキング以外はクイーンが1個のみあるKQKエンディングにおける完全解析データベースを作成した[4][7]。Nefkensの結果によれば、KQKエンディングの可能な局面数は40,960通りであり、双方最善手を指せば最大 19 手でチェックメイトできる。また、ルークが 1 個のみある KRK エンディングでは、可能な 40,960 局面全てで 31 手以内にチェックメイトできる。P,E,のうち、ポーンが 1 個のみある KPK エンディングについては、1989 年に Zennler により完全解析データベースが作られている[4][8]。Zennler の結果によれば、KPK エンディングでは、可能な局面数は98,304 通りであり、そのうち 62,480 局面は白勝ちの局面であり、最大 37 手でクイーンに昇格でき、その後最大17手でチェックメイトできる。5駒以下の場合の解析については、1970年代から特定の駒組み合わせに対して解析が行われてきた。例えば、1978 年には、Arlazarov らにより KROKR エンディングが、1986 年には、Komissarchik らにより KQPKQ エンディングに解析がなされている[4][10][11]。1989年には、Stillerにより、残り駒数が5駒以下の場合についての完全解析がなされた[4][9]。現在では、6 駒以下のエンディングについては完全解析されており、フリーのデータベースとして利用できるようになっている[12][12]。このため、多くのチェスソフトでは、データベースを組み込むことにより残り駒数が少なくなると最善手を指せるようになっている。

チェスの終盤での完全解析は6駒以下が完了しているため、今後は7駒以下の局面の解析が求められる。そこで、本研究では終盤でのチェスの完全解析の第一段階として P.E.について解析する。

1.2.本研究の目的

本研究では、駒が少ないチェスの終盤における完全解析の第一段階として、P.E.の解析を行う。本研究では P.E.について解析することにより勝率、勝敗が決まる最短の手数、有利な局面などを導き出す。

1.3.本報告書の構成

本報告書では、2 章で P.E.の基本的なルールについて述べる。また、3 章では P.E.の局面、作成したプログラムの説明をする。そして、4 章では本研究の結果・考察を述べ、最後に 5 章で今後の課題を述べる。

2　P.E.について

2.1. P.E.についての概要

チェスは、将棋と違い取られた駒は使用することができないため終盤局面では少数の駒のみの局面がよく現れる。特に、双方のキング以外はポーンのみの場合をポーンエンディング（以下 P.E.とする）と呼ばれる。本研究では最も簡単な P.E.である、盤上に双方のキングとポーンが 1 個のみ存在する場合について解析する。図 1 に本研究で解析する P.E の初期配置、表 1 に P.E.で使用する駒を示す。以下では、ポーンとキングのみのプレイヤーを攻撃側、キングのみのプレイヤーを防御側とする。攻撃側は、如何にしてポーンを防御側のキングに取られずにチェスの最強の駒であるクイーンに昇格させるかが P.E.の解析における重要点である。そのため、本研究で行う解析はポーンが防御側に取られるか、ポーンが昇格するまでかの最短手順を求めることまでとし、ポーンが取られた後引き分けになるまで、あるいは、昇格後チェックメイトするまでの最短手順は求めない。

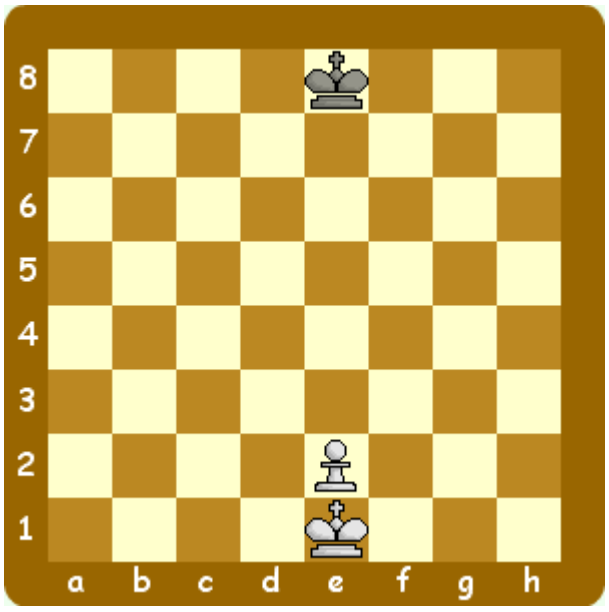

図 1　初期配置

表 1 P.E.で使用する駒

駒	略称	記号	個数
白のキング	K		1
白のポーン	P		1
黒のキング	K’		1

2.2.　各駒の動き

チェスの動き方を x 軸と y 軸を用いて以下で説明する.

・ポーン

ポーンは、その前方のマスが空いている場合、一つ前に進めることができる(y+1)。また、その左前もしくは右前に相手の駒がある場合、その駒を取って相手の駒がいたマスに進むことができる(x±1, y+1)。また、初期位置(白駒なら第 2 ランク、黒駒なら第 7 ランク)にいるポーンは、その前方 2 マスが空いていれば、1 手で 2 マス進むことができる。図 2 にポーンの動き方を示す。ポーンは最前列(白駒なら第 8 ランク、黒駒なら第 1 ランク)まで進むと昇格(プロモーション)できる。これは、将棋で言う成りのことでポーンの場合は、キング以外の任意の自駒に成ることができる。加えて、ポーンの白駒が第5ランク(黒駒の場合は第4ランク)に進んでいるとき、その隣の行の黒駒が初期位置から二歩進んですれ違おうとした場合、これを一歩進んできたと同様と見なして白駒は黒駒を取りながら斜め前方に進むことができる[5]。ポーンのこの動きはアンパサンと呼ばれる。 図 3 にポーンの昇格、図 4 にアンパサンでの動き方を示す。

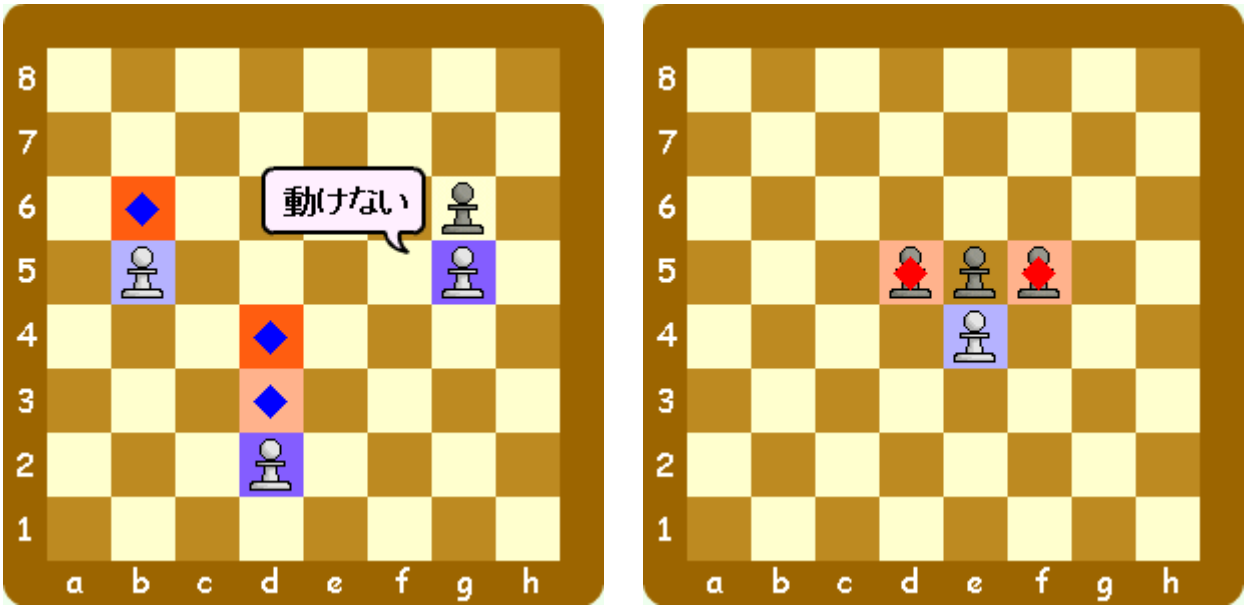

図 2 ポーンの動き方

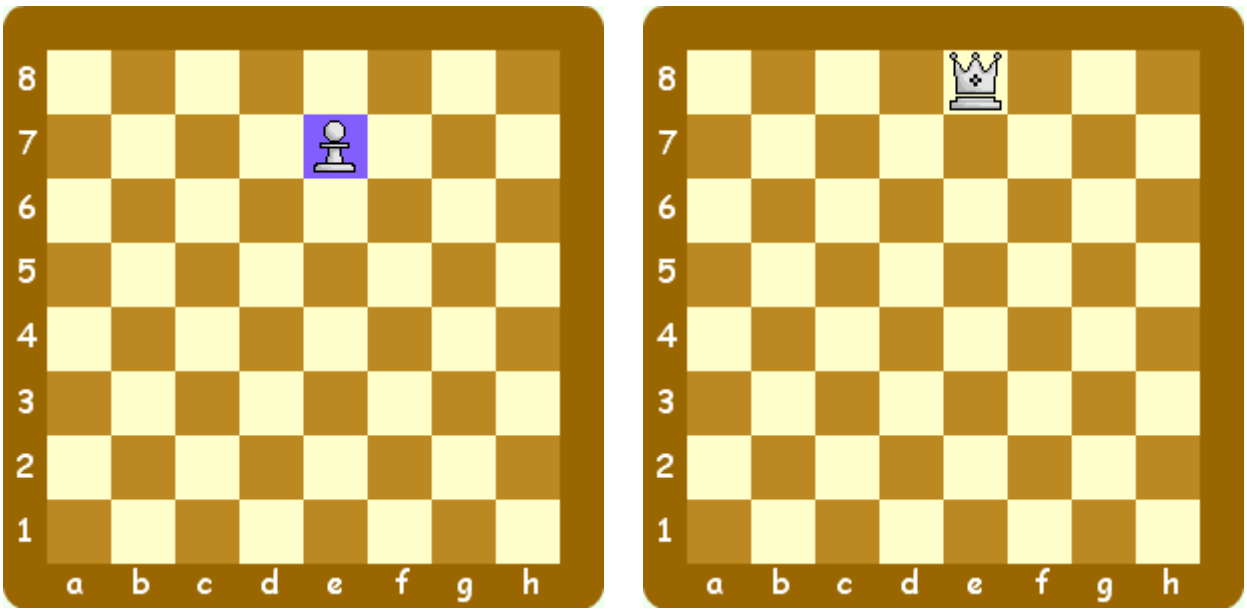

図 3 ポーンの昇格

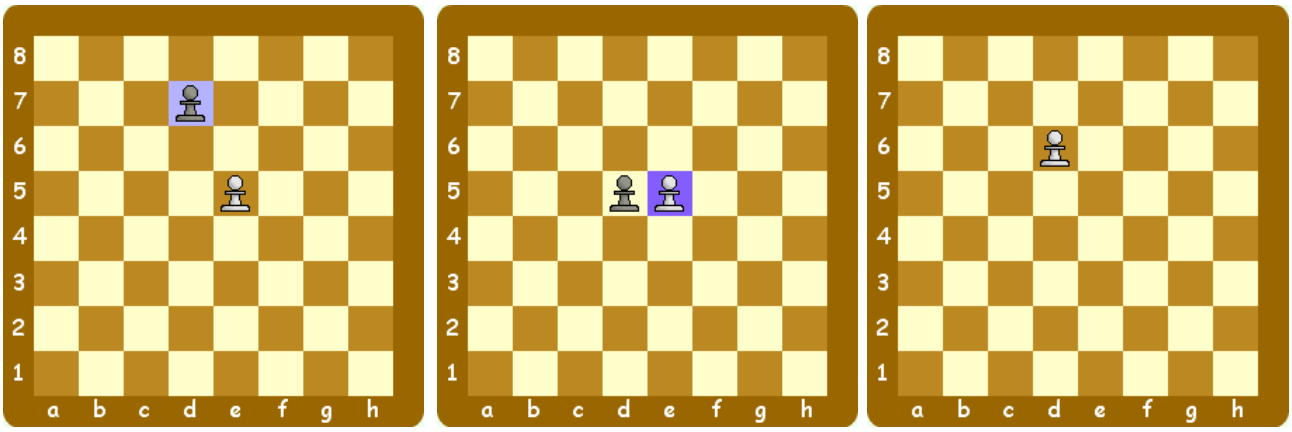

図 4 アンパサンでの動き方

・キング

キングは、その周囲 8 マスのうち任意のマスに進むことができる(x±1, y±1)。また、周囲 8 マスの敵駒がある場合は、その駒を取りながらそのマスに進むことができる。だし、キングは、相手の駒の効いているマスへは移動することができない。図 5 にキングの動き方を示す。

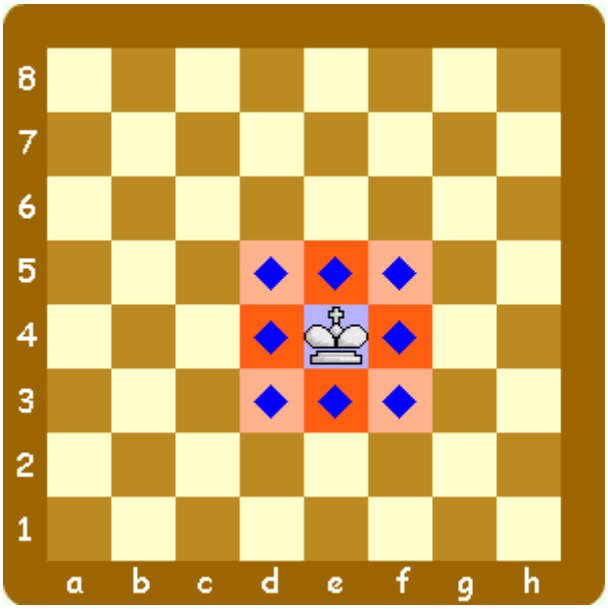

図 5 キングの動き方

2.2. その他のルール

・ステイルメイト

自分のキングにチェックがかかっていなくて、自分の番なのに動かせる駒がない場合、引き分けになる。図 ~~46~~ のような局面の場合防御側のキングはどの位置に進めても自殺手になるため動かせる駒がなくステイルメイトになる。

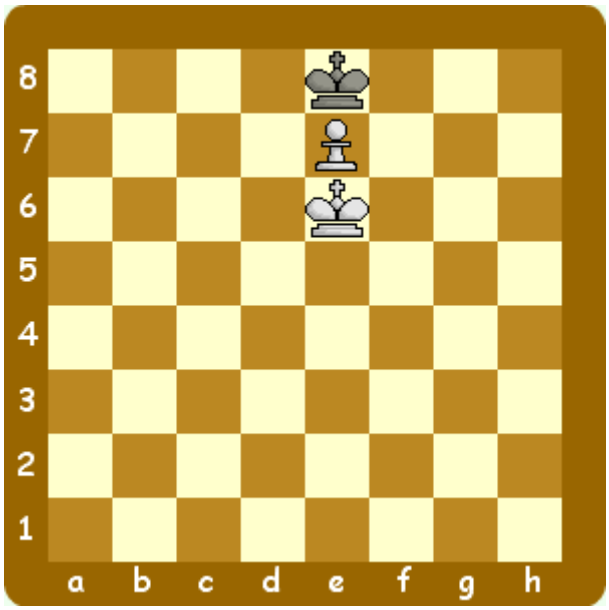

図 6 ステイルメイト

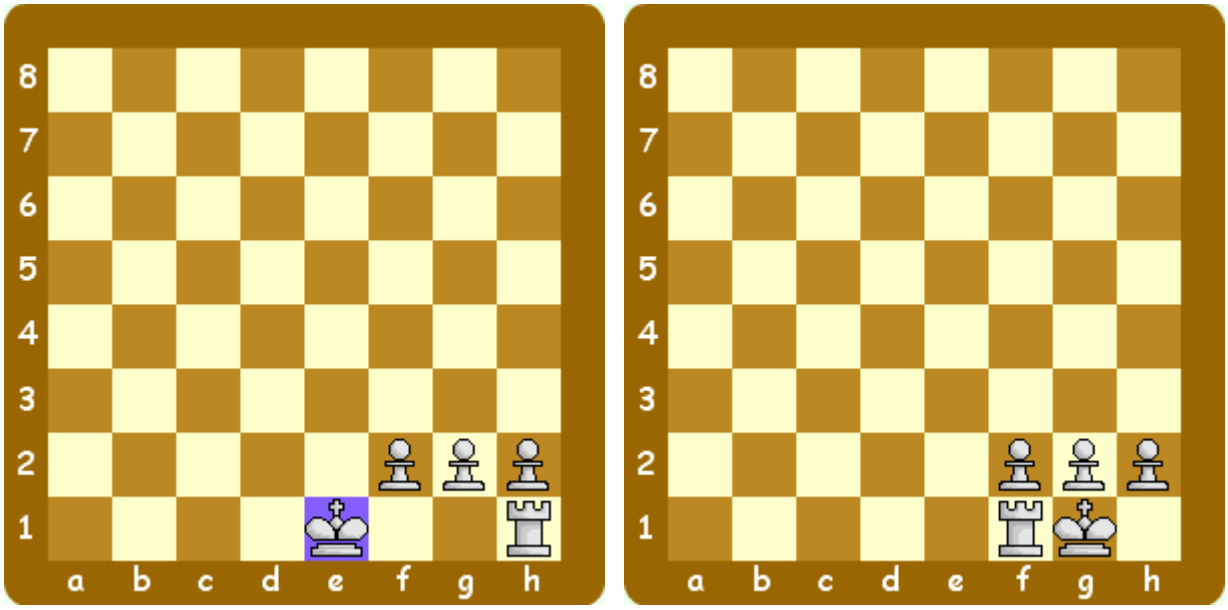

図 7　キャスリングでの動き方

・千日手(パペチュアル)

将棋の千日手と同じであるが、将棋が同一局面 4 回で成立するのに対してチェスでは 3 回で引き分けになる。将棋と一番違うのは王(チェック)の連続でも引き分けになることである[5]。

・キャスリング

キャスリングは、キングを安全地帯に移動すると同時に、隅のルークを中央に向かって繰り出すことを 1 手で行う特殊ルールである。キャスリングは次の場合に限って行うことができる。

- キングとキャスリングさせるルークが共に初期配置から移動していない
- キングとキャスリングさせるルークとの間に駒が無い

- 現在キングがチェックされておらず、キングが通過するマスおよびキングの移動後のマスに敵の駒が利いていない
- キングとキャスリングさせるルークが同ランク上にある

以上の条件を満たしているときにキングがそのルークに向かって、一度に 2 マス移動し、キングが移動した側のルークがキングを飛び越えて、そのすぐ隣マスに移動する[5]。図 7 にキャスリングでの動き方を示す。本研究では、ルークを使用していないためキャスリングが行われる場面はない。

2.3. P.E.での定跡

P.E.では、攻撃側はキングをポーンよりも前に出すこと、攻撃側防御側双方ともに自分の手番で相手キングと 1 マス開けた位置(オポジション)を取ることが定跡として知られている。P.E.プログラムは、この定跡に従い、キングを有利な位置へ移動させることを優先的に行う。そして、ポーンが最前列へ進むまでに防御側のキングに取られない局面になるとポーンを進めていく。

2.3.1. P.E.についての既知の結果

図 1 の P.E.の場合、双方が最善手を指した場合、以下に示す 14 手でポーンが昇格できる[14]。

1. Kf2, Kf8
2. Ke3, Ke7
3. Ke4, Ke6
4. e3, Kd6
5. Kf5, Ke7
6. Ke5, Kd7
7. Kf6, Ke8
8. Ke6, Kd8
9. e4, Ke8
10. e5 Kd8
11. Kf7, Kd7
12. e6+, Kd6
13. e7, Kd5
14. e8=Q

よって、本研究での目標は、図 1 から 14 手で昇格できる手を探索することとなる。

2.4. P.E.のプログラムで用いられる手法

本研究では、P.E.のコンピュータを Java を用いて作成する。本研究では、指定した局面数までを先読みし、局面による評価値を求めだし良い結果の出た手を打つプログラムを作成した。局面の評価値は、攻撃側に有利だと高く、防御側に有利だと低くなるように設定する、よって攻撃側は、評価値の高い手を指し、防御側は、評価値の低い手を指す。

3 研究内容

本研究では、Java で作成した局面の先読みによる評価値にしたがって動作する P.E.プログラム（以下 PEAI とする）とランダムに動作する P.E.プログラム（以下 PERM）を作成した。

3.1. 先読み評価値

PEAI は局面を先読みし評価値のよくなる手を選択する。攻撃側は評価値が高い手を選択し、

防御側は評価値が低くなる手を選択する。P.E.では、攻撃側はキングをポーンよりも前に出すこと、攻撃側防御側双方ともに自分の手番で相手キングと１マス開けた位置を取ることが定跡として知られている。P.E.プログラムは、この定跡に従い、キングを有利な位置へ移動させることを優先的に行うので、このような局面になる場合は評価値が高くなるようにしてある。つまり、攻撃側は自陣のキングがポーンよりも前に進む場合評価値が高くなる。P.E.の防御側は勝利することができないため、引き分けにするためポーンを取るように、またはスティールメイトとなるように動かす。そのため防御側はキングがポーンに近づくと評価値が低くなる

3.2. 勝てる局面

序盤は上記で説明したように攻撃側はポーンより前にキングを出すという定跡に従い評価値を高くつけて先読みし評価値の高い局面を選択し駒を進めている。しかし、キングをポーンより前に進めているだけではポーンを昇格させることができず勝敗を決めることができない。そこで、ポーンが昇格できる条件かつ次の相手番で昇格したポーンが取られない局面に成る場合は、評価値を最大にしポーンを進める。例えばポーンが昇格した局面で防御側のキングがポーンの周囲１マス以内に存在していなければ次の番にポーンが取られることはない。また、ポーンが昇格した場合にポーンの周囲１マス以内に防御側のキングが存在した場合でも攻撃側のキングがポーンの周囲にあれば防御側のキングはポーンを取るのは自殺手になるため取ることができない。さらにポーンから最前列までの距離よりポーンと防御側のキングの距離が短くなる場合もポーンが昇格するまでに防御側のキングがポーンに追いつけないため勝てる局面となる。これらの局面になる場合は攻撃側が有利になると考えられるので評価値を高くしてある。図 8 に攻撃側が勝利できる局面を示す。

また、ポーンが防御側のキングに取られる局面では評価値が低くなるようにしてある。ポーンが昇格した場合でも次の相手の番に昇格したポーンが取られる場合は引き分けとなるので評価値は低くしている。また、P.E.ではポーンを相手側のキングに取られないようにキングを配置することが重要で自分のキングとポーンの距離より相手のキングとポーンの距離の方が近い場合は相手キングにポーンを取られてしまうために引き分けとなる。そのため自分のキングとポーンの距離より相手のキングとポーンの距離が近くなる局面の評価値は低くなる。図 9 に引き分けとなる局面を示す。

PEAI は以上に記述したような評価値の付け方で局面を先読みし最も良い手となるものを選んでいるが本研究では P.E.の攻撃側、防御側の勝率を求めることによりどちらが有利なのかなどを調査したので少し評価値に乱数を加えている。一方、対照実験用の PERM は指せる手からランダムに選択して手を決めている。ただし、残り 1 手で勝敗が決まる場合はその手を選択する。

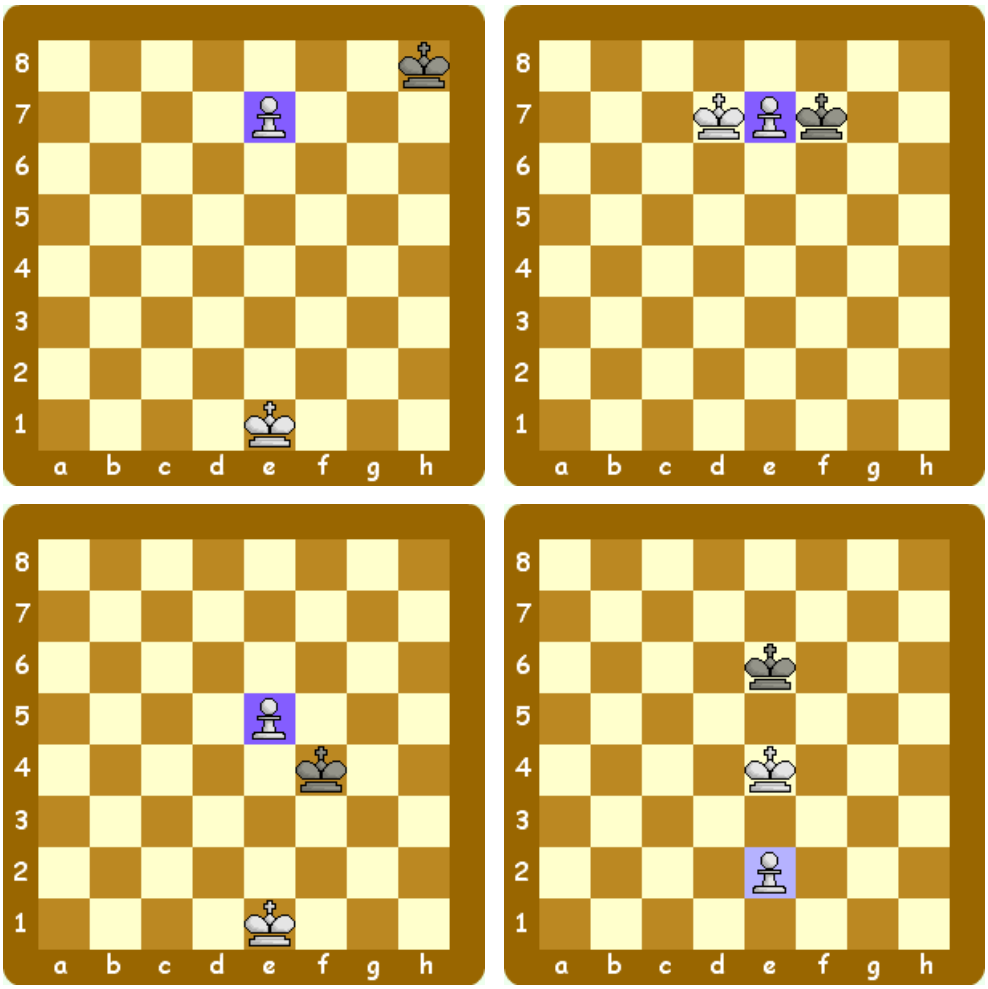

図 8 攻撃側が勝利できる局面

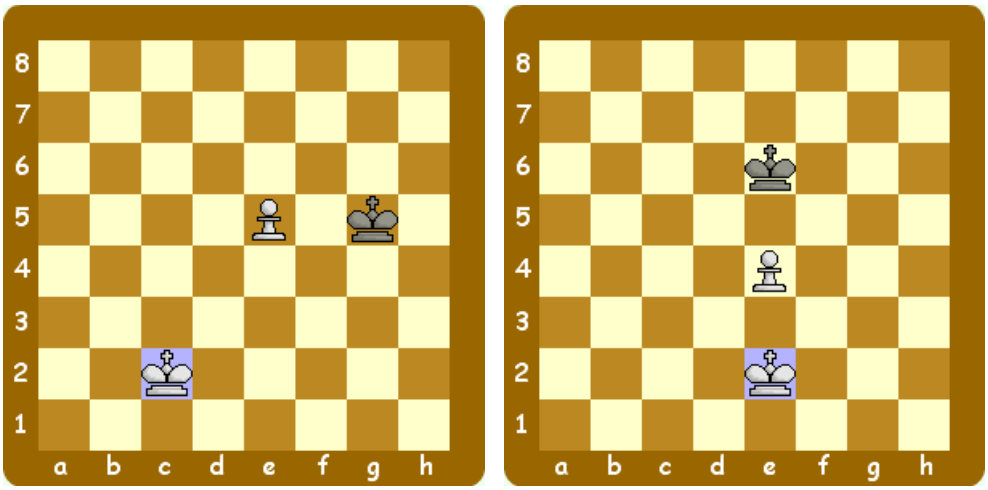

図 9 引き分けとなる局面

3.2. PEAI のプログラム

本節では、本研究で作成した PEAI のプログラムについて説明する。

・Chess クラス

Chess クラスは P.E.を実行するクラスである。プレイヤーを人間かコンピュータかを選択し、P.E.を実行させる。

・Board クラス

Board クラスはチェス盤を管理するクラスである。本研究でもっとも重要だと思われる局面の評価値や先読みはこのクラスで行われている。表 1~~2~~2 に Board クラスの各メソッドについてまとめる。

・Piece クラス

Piece クラスは P.E.で使用される駒を管理するクラスである。駒の動きや座標はこのクラスで設定されている。表 3 に Piece クラスのメソッドを示す。

表 2 Board クラスのメソッド

メソッド	処理内容
Board()	コンストラクタ
Board(int[][])	コンストラクタ
void showBoard()	盤面を表示する
String showPiece (int)	駒を表示
String player (int)	各プレイヤーの手番
String com (int)	com の手番
String ran (int)	com の手番(ランダム)
String movePiece (Piece, int, int, int)	指定した位置に駒を移動させ、その棋譜を表示する
void removePiece (int, int)	指定した位置にある駒を盤から取り除く
boolean checkWin (int)	勝負がついたか
boolean idChecked (int)	現在チェックがかかっているか
boolean isMate (int)	詰みの判定
void createMovaleList	候補手の判定
int value (int)	現在の盤の評価値を表示
Int value (int, int)	現在の盤の評価値を表示
Board nextBoard (NextMove, int)	指定した駒を指定した位置に動かした後の盤を得る
String name (int)	駒名を返す
void recordBoard()	現在の盤面を記憶
boolean checkPerpetual()	千日手になったか 判定
boolean precheckPerpetual()	先読み時に同じ局面になったか判定
void setMaxDepth (int)	先読みの上限数を指定
void resetMoves()	手数をリセット
int winner()	勝者の番号を返す

表 3 Piece クラスのメソッド

メソッド	処理内容
Piece(int)	コンストラクタ
Piece(int, int, int)	コンストラクタ
void setMovableList()	駒の移動可能方向をセット
Void setInitialPosition()	駒を盤上の初期位置にセット
Void setPosition(int, int)	駒を盤上の指定した座標にセット
Void setOnBoard()	駒を盤上に置く
Void removeFromBord()	駒を盤上から削除
Boolean isOnBoard()	盤上に駒が存在するか
Boolean isThere(int, int)	指定した座標に駒が存在するか
Int x()	X 座標を返す
Int y()	Y 座標を返す
Int type()	駒の種類を返す
String name()	駒名を返す
Void move(int, int)	駒を指定した座標に移動する
Boolean movable(int, int)	駒が指定した座標に移動できるか
Boolean movable(int[][], int, int)	駒が指定した座標に移動できるか
Boolean checkAndMove(int, int)	駒を指定した座標に移動する
Boolean checkAndMove(int[][], int,	駒を指定した座標に移動する

表 4　NextMove クラスのメソッド

メソッド	処理内容
NextMove(int, int, int)	コンストラクタ
Int type()	駒の種類を返す
Int nextX()	移動先の X 座標を返す
Int nextY()	移動先の Y 座標を返す
Int value()	移動した場合の盤面の評価値を返す
Void setValue(int)	評価値をセットする

・NextMove クラス

NexeMove クラスは P.E.クラスで使用される駒の次の局面での座標を管理するクラスである。主に先読みするときに使用される。表 4 に NextMove クラスのメソッドを示す。

表 5 終盤定跡に従うプログラム同士の対戦結果

結果	攻撃側勝利	引き分け
回数	80	20

4　結果・考察

本研究では、終盤定跡に従い着手を決定する P.E.のプログラムを Java を用いて作成した。表 5 にプログラム同士の対戦結果を示す。ただし試行回数は 100 回である。表 5 より、双方終盤定跡に従った場合、攻撃側が勝つことが多いことが分かる。また、攻撃側が勝つ場合ポーンが昇格できるまでの手数は 30 手以内で終わることが多い。PERM のコンピュータで実行した場合、100 手以内で勝敗が決まらない場合が多い。このことから P.E.のような駒数が少ない場合でもランダムで手を指すと勝負が決まらないことがわかる。また、PEAI と PERM が対戦した場合、PEAI の攻撃側の勝率は 100％である。本研究では PEAI に乱数を加えているため、PEAI 同士の勝負の場合、2.3.1 節で説明したような、今回の配置なら双方最善手を指せば 14 手で昇格できるような手はでず互いのキング同士が向かい合い拮抗し同じ局面に陥ることがあり最善手の 14 手で勝負が決まらない場合の確率が高かった。また、乱数を乱数を加えていない状態で対戦した場合、14 手で決まらなかったため本研究で作成した評価値によるプログラムは未完成だといえる。しかし、防御側のみに乱数を加えた場合は 14 以内で勝敗が決まることから勝ちに向かうようには手を指せていることがわかる。このことから最善手の手を指さなかった場合、手数が多くなることが分かる。

5　結論・今後の課題

本研究では、先手がキングとポーン、後手がキングのみと非常に駒の数が少ない場合の P.E.のプログラムの作成と解析を行った。攻撃側はポーンとキングの間が一マス以内だと防御側のキングにポーンを取られることがないため有利に局面を進めることができる。勝敗は PEAI 同士で戦った場合、攻撃側が勝つことが多い。また、PEAI のような盤面の評価値に従い先読みするプログラムでは本研究ような P.E.の最善手の最短 14 手で終わらせることが困難であると分かる。1 章で説明したように、現在では 6 駒以下の場合については完全解析がされているが今回の研究で評価値によるチェスプログラムでは最善手を指すのが困難であると分かる。このことからデータベースを組み込んだプログラムの方が最善手を指すことができ強いことが分かる。

今後の課題としては、P.E.の局面が与えられたときに、その最短手順を求めるプログラムを作ること、駒の種類や数がより多い場合のエンディングのプログラムを作成することが挙げられる。

謝辞

本研究において、最初から最後まで石水隆講師には多大なるご迷惑をかけた事をお詫びするとともに、ご指導いただきました感謝を申し上げます。また、研究室の皆様にもご協力いただき、感謝と御礼を申し上げたく、この場を謝辞として使わせていただきます。

参考文献

[1] 加藤暢他, オブジェクト指向 Java プログラミング入門, 近代科学者, 2008

[2] 田中哲郎「どうぶつしょうぎ」の完全解析, 情報処理学会研究報告, Vol.2009-GL-22 No.3, pp.1-8, 2009.

[3] 池泰弘, Java 将棋のアルゴリズム、工学社, 2007.

[4] D.Levy, M.Newborn 著, 飯田弘之, 吉村信弘 訳, コンピュータチェス 世界チャンピオンへの挑戦, サイエンス社, 1994

[5] 渡井美代子 著, 松本康司 監修, 図解早わかりチェス 初歩の定石と必勝のコツ, 日東書院, 2002

[6] 湯川博士 著, 若島正 監修, 将棋ファンでも楽しめる初めてのチェス１手・２手詰集, 山海堂, 2003

[7] Harry Nefkens, Constructing Data Bases to Fit a Microcomputer, ICCA Journal, Vol. 8, No. 4, pp. 219-224, 1985.

[8] Hans Zellner, The KPK Database Revisited. ICCA Journal, Vol. 12, No. 2, pp.78-82, 1989.

[9] Lewis Stiller, Parallel Analysis of Certain Endgames. ICCA Journal, Vol. 12, No. 2, pp. 55-64, 1989.

[10] Vladimir Arlazarov and Aron Futer, Computer analysis of a rook endgame, Machine Intelligence 9, University of Edinburgh Press, 1978.

[11] Edik Komisssarchink, Aron Futer and Vladimir Arlazarov, Computer analysis of a Queen endgame, ICCA Journal, vol.9, No.4, pp.189-200, 1986.

[12] Kirill Kryukov, Endgame Tablebases Online, 6-men endgame analysis free for everyone, 2013, http://kirill-kryukov.com/chess/tablebases-online/

[13] Aaron Tay, A guide to Endgames Tablebase, 2006, http://horizonchess.com/FAQ/Winboard/egtb.html

[14] Jacques Marie Pinoeu, ジャック・ピノーのダイナミックチェス入門, 山海堂, 1995.

[15] IBM 元開発者「チェス王者にスパコンが勝てたのは、バグのおかげ」WIRED
http://wired.jp/2012/10/03/deep-blue-computer-bug/

付録

以下に本研究で作成したプログラムのソースを示す。

・Chess クラス

```
package chess;

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;

import chess.Board;

public class Chess {
	final static int KING     = 1;                                          //  キング(先手)
	final static int PAWN     = 2;                                          // ポーン
	final static int KING2    = -1;                                         //  キング(後手) // 変更 3
	final static int EMPTY    = 0;                                          // 空白
	final static int BORDER = Integer.MAX_VALUE;                            // 盤外外
	static FileWriter pass, rPass;                                          // 棋譜出力用
	static PrintWriter fp, rfp;

	public static void main(String[] args) {
		Board board = new Board();
		boolean isCom[] = {false, false};
		Scanner keyBoardScanner = new Scanner(System.in);
		String input;
		// 入力用;
		String score;
		// 棋譜;
		FileWriter pass = null;                                             //  棋譜出力用ファイル
		PrintWriter fp = null;                                              //  棋譜出力用ファイルのポインタ

		try {
			pass = new FileWriter("ChessScore.txt");
			fp = new PrintWriter(pass);
		} catch (IOException e) {
			System.err.println (e);
		}
```

```
                System.out.print("先手チームは COM が持ちますか？ (Y/N) ");
                input = keyBoardScanner.next();
                if (input.equals("Y") || input.equals("y")) {
                        isCom[0] = true;
                }

                System.out.print("後手チームは COM が持ちますか？ (Y/N) ");
                input = keyBoardScanner.next();
                if (input.equals("Y") || input.equals("y")) {
                        isCom[1] = true;
                }

                /* コンピュータ同士の対戦 */
//              isCom[0] = true;
//              isCom[1] = true;

                while (true) {
                        board.showBoard();
                        board.createMovableList (0); // 先手チームが移動可能な手を求める
                        //System.out.println(board.value(1));
                        if (board.isChecked(0)) System.out.println("チェック");
                        if (board.checkWin(1)) break;
                        if (isCom[0]) {
                                score = board.com(0);
                        } else {
                                score = board.player (0);
                        }
                        fp.print(score); // 棋譜出力

                        board.showBoard();
                        board.createMovableList(1); // 後手チームが移動可能な手を求める
                        //System.out.println(board.value(0));
                        if (board.isChecked(1)) System.out.println("チェック");
                        if (board.checkWin(0)) break;
                        if (isCom[1]) {
                                score = board.com(1);
                        } else {
                                score = board.player(1);
                        }
                        fp.println(score); // 棋譜出力
                }
                fp.close();
        }
}
```

・Board クラス

```
package chess;

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

public class Board {
	final static int KING     = 1;
		// キング(先手)
	final static int PAWN     = 2;
	// ポーン
	final static int KING2   = -1;
		// キング(後手)
	final static int EMPTY   = 0;
	// 空白
	final static int BORDER = Integer.MAX_VALUE;					//  盤
外
	final static boolean isChessStyleScore = true;				// 棋譜表記チ
ェス式

	/* チェス盤 */
	public int[][] board
				= {{BORDER, BORDER, BORDER, BORDER, BORDER, BORDER,
BORDER, BORDER, BORDER, BORDER},
				{BORDER,  EMPTY,  EMPTY,  EMPTY,  EMPTY,  KING2,
EMPTY,  EMPTY,  EMPTY, BORDER},
				{BORDER,  EMPTY,  EMPTY,  EMPTY,  EMPTY,  EMPTY,
EMPTY,  EMPTY,  EMPTY, BORDER},
				{BORDER,  EMPTY,  EMPTY,  EMPTY,  EMPTY,  EMPTY,
EMPTY,  EMPTY,  EMPTY, BORDER},
				{BORDER,  EMPTY,  EMPTY,  EMPTY,  EMPTY,  EMPTY,
EMPTY,  EMPTY,  EMPTY, BORDER},
				{BORDER,  EMPTY,  EMPTY,  EMPTY,  EMPTY,  EMPTY,
EMPTY,  EMPTY,  EMPTY, BORDER},
				{BORDER,  EMPTY,  EMPTY,  EMPTY,  EMPTY,  EMPTY,
EMPTY,  EMPTY,  EMPTY, BORDER},
				{BORDER,  EMPTY,  EMPTY,  EMPTY,  EMPTY,  PAWN,
EMPTY,  EMPTY,  EMPTY, BORDER},
				{BORDER,  EMPTY,  EMPTY,  EMPTY,  EMPTY,  KING,
EMPTY,  EMPTY,  EMPTY, BORDER},
				{BORDER, BORDER, BORDER, BORDER, BORDER, BORDER,
BORDER, BORDER, BORDER, BORDER}};

	int nextX;
						// 移動先の X 座標
	int nextY;
						// 移動先の Y 座標
```

```
Piece king, king2, pawn;
        // 駒
Boolean resign = false;
        // 投了したか
Boolean checkmate = false;
        // チェックメイトか
Boolean stalemate = false;
        // スティールメイトか
Boolean promotion = false;
        // 昇格したか(プロモーション)
ArrayList<NextMove> movableList;
        // 候補手のリスト
int maxDepth = 1;
                // 先読みする手数の上限;
boolean doResign = false;
        // 負けが確定したときに COM が投了するか
static int moves = 0;
                // 手数
static int lookAheadMoves = 0;
        // 先読み手数
final static int maxMove = 100;
// 手数の上限
static int[][][] boardRecord = new int [maxMove][8][8];    // 盤面記録用
int winner = 0;
                // 勝者  1:先手勝ち  -1:後手勝ち  0:引き分け

/* コンストラクタ */
public Board() {
        pawn     = new Piece(PAWN);
        king     = new Piece(KING);
        king2    = new Piece(KING2);
}

/**
 * コンストラクタ
 * 盤上に駒を指定した位置に配置
 * @param int[][] board  現在の盤
 */
public Board(int[][] board) {
        for (int y = 1; y <= 8; ++y)
                for (int x = 1; x <= 8; ++x)
                        this.board[y][x] = board[y][x];
        for (int y = 1 ; y <= 8; ++y) {
                for (int x = 1; x <= 8; ++x) {
                        switch (board [y][x]) {
                        case PAWN:
                                pawn = new Piece(PAWN, x, y);
```

```
                                        break;
                                case KING:
                                        king = new Piece(KING, x, y);
                                        break;
                                case KING2:
                                        king2 = new Piece(KING2, x, y);
                                        break;
                                }
                        }
                }
}

/* 盤を表示 */
public void showBoard() {
        System.out.println("    abcdefgh");

        for (int y = 0; y < board.length; ++y) {
                switch (y) {
                case 0:
                        System.out.print("  ");
                        break;
                case 1:
                        System.out.print("8");
                        break;
                case 2:
                        System.out.print("7");
                        break;
                case 3:
                        System.out.print("6");
                        break;
                case 4:
                        System.out.print("5");
                        break;
                case 5:
                        System.out.print("4");
                        break;
                case 6:
                        System.out.print("3");
                        break;
                case 7:
                        System.out.print("2");
                        break;
                case 8:
                        System.out.print("1");
                        break;
                case 9:
                        System.out.print(" ");
```

```
                        break;
                }

                for (int x = 0; x < board[y].length; ++x) {
                        System.out.print(showPiece(board[y][x]));
                }

                System.out.println();
        }
}

/**
 * 駒を表示
 * @param 駒の種類
 * @return 駒の文字列表現
 */
public String showPiece(int type) {
        switch (type) {
        case PAWN:
                return "P";
        case KING:
                return "K";
        case KING2:
                return "k";
        case EMPTY:
                return "  ";
        case BORDER:
                return "■";
        default:
                return "？";
        }
}

/**
 * 各プレイヤーの手番
 * @paran int playerNum プレイヤー番号
 * @return 指した手の棋譜
 */
public String player(int playerNum) {
        Scanner keyBoardScanner = new Scanner(System.in);
        String inputPiece;
                // 駒の種類入力用
        Piece piece;
                // 動かす駒
        int type;
                // 動かす駒の種類
        int nextX, nextY;
```

```
                // 移動先
        ArrayList<NextMove> onesMovableList;
// ある駒が移動可能な手のリスト

        /* 適切な駒が選択されるまでループ */
        while (true) {
                if (playerNum == 0) {
                        System.out.println("先手の番です");
                        System.out.print("進める駒(k,p)を選んでください(R=投了):");
                } else {
                        System.out.println("後手の番です");
                        System.out.print("進める駒(k2)を選んでください(R=投了):");
                }

                inputPiece = keyBoardScanner.next();
                if (inputPiece.equals("K") || inputPiece.equals("k")) {
                        piece = king;
                        type = KING;
                } else if (inputPiece.equals("P") || inputPiece.equals("p")) {
                        piece = pawn;
                        type = PAWN;
                } else if (inputPiece.equals("K2") || inputPiece.equals("k2")) {
                        piece = king2;
                        type = KING2;
                } else if (inputPiece.equals("R") || inputPiece.equals("r")) {
                        System.out.println("投了します");
                        resign = true;
                        if (isChessStyleScore) {
                                if (playerNum == 0) {
                                        return "1-0";
                                } else {
                                        return "0-1";
                                }
                        } else {
                                return "投了";
                        }
                } else {
                        if (playerNum == 0) {
                                System.out.println("K,P から選んでください");
                        } else {
                                System.out.println("K2 から選んでください");
                        }
                        continue; // 選び直し
                }

                if (playerNum == 0 && !(type == KING || type == PAWN)) {
                        System.out.println("K,P から選んでください");
```

```
                    continue; //  選び直し
                } else if (playerNum == 1 && !(type == KING2)) {
                    System.out.println("K2  から選んでください");
                    continue; //  選び直し
                }

                if (piece == null) {
                    System.out.println(name(type) + "はすでに取られています");
                    continue; //  選び直し
                }

                onesMovableList = new ArrayList<NextMove>(); //  指定した駒が移動可能な手のリスト
                for (int i = 0;   i < movableList.size(); ++i) {
                    if (movableList.get(i).type() == type) {          //  指定した駒を動かす手か？
                        onesMovableList.add  (movableList.get(i));   //  指定した駒が移動可能な手の数を加える
                    }
                }

                if (onesMovableList.size() == 0) { //  指定した駒が移動可能な位置が無い場合
                    System.out.println(name(type) + "が進める位置はありません");
                    continue; //  選び直し
                }

                System.out.println(name(type) + "が進む場所を選んでください");
                System.out.print(name(type) + "は現在(" + piece.x() + "," + piece.y() + ")にいて");
                for(int i = 0; i < onesMovableList.size(); ++i) {
                    System.out.print("(" +   onesMovableList.get(i).nextX() + "," + onesMovableList.get(i).nextY() + ")");
                }
                System.out.println("へ移動できます");

                System.out.print("x 座標 (1～8):");
                nextX = keyBoardScanner.nextInt();
                System.out.print("y 座標 (1～8):");
                nextY = keyBoardScanner.nextInt();

                boolean movable = false; //  指定した位置に移動可能か
                for (int i = 0; i < onesMovableList.size(); ++i) {
                    if  ((nextX  ==  onesMovableList.get(i).nextX())  &&  (nextY  == onesMovableList.get(i).nextY())) {
                        movable = true;
                        break;
```

```
                        }
                    }

                    if (!movable) {
                        System.out.println(name(type) + "は(" + nextX + "," + nextY + ")
へは移動できません");
                        continue; //  選び直し
                    }
                    break; // while ループから脱出
                }

                String score = movePiece(piece, type, nextX, nextY); //  駒を移動させる

                System.out.println(moves + ":" + score + "[" + value(playerNum) + "]");
                ++moves;
                return score;
            }

            /**
             * COM の手番
             * maxDepth で指定した手数を先読みして最も良い手を指す
             * @paran int playerNum プレイヤー番号
             * @return 指した手の棋譜
             */

            public String com(int playerNum) {
                int type, nextX, nextY;         //  移動する駒の種類, 移動先の座標
                Piece piece = null;                     //  移動する駒
                int bestValue = 0;                       //  最も良い盤面の評価値
                NextMove bestMove = null;                //  最も良い手
                int nextPlayerNum;                       //  次の手番プレイヤー

                if (movableList.size() == 0) { //  候補手が存在しない＝詰み
                    System.out.println("投了します");
                    resign = true;
                    if (isChessStyleScore) {
                        return (playerNum == 0) ? "1-0" : "0-1";
                    } else {
                        return "投了";
                    }
                } else if (movableList.size() == 1) { //  候補手が 1 つしか無い場合は評価値を求めな
い
                    bestMove = movableList.get(0);
                } else {
                    if (playerNum == 0) { //  先手チームの場合  評価値が高いほど良い手と見
做す
                        nextPlayerNum = 1;              //  次の手番プレイヤー
```

```
                    bestValue = Integer.MIN_VALUE;
                    bestMove = movableList.get(0);

                    ++lookAheadMoves; // 先読みのために手数を 1 増やす
                    for (int i = 0; i < movableList.size(); ++i) {
                        NextMove nextMove = movableList.get(i);      // i 番目
の候補手
                        Board nextBoard = nextBoard(nextMove,
nextPlayerNum);      // 次の盤面を生成
                        int value = nextBoard.value(nextPlayerNum,
maxDepth);// 盤面の評価値を計算
                        if (value > bestValue) {                    // 高評価の
手を発見した
                            bestMove = nextMove;                    //
高評価の手を記憶
                            bestValue = value;                      //
評価値を記憶
                        }
                        if (bestValue == Integer.MAX_VALUE) {      // 評価無
限大の手＝必勝の手を発見
                            break;                                  //
ループ脱出
                        }
                    }
                    --lookAheadMoves; // 先読みが終了したので手数を 1 戻す
                    if (bestValue == Integer.MIN_VALUE && doResign) {   // 評価値
無限小の手しか無い＝負け確定
                        System.out.println("投了します");
                        resign = true;
                        if (isChessStyleScore) {
                            return "1-0";
                        } else {
                            return "投了";
                        }
                    }
                } else { // 後手チームの場合 評価値が低いほど良い手と見做す
                    nextPlayerNum = 0;              // 次の手番プレイヤー
                    bestValue = Integer.MAX_VALUE;
                    bestMove = movableList.get(0);
                    ++lookAheadMoves; // 先読みのために手数を 1 増やす
                    for (int i = 0; i < movableList.size(); ++i) {
                        NextMove nextMove = movableList.get(i);      // i 番目
の候補手
                        Board nextBoard = nextBoard(nextMove,
nextPlayerNum);      // 次の盤面を生成
                        int value = nextBoard.value(nextPlayerNum,
maxDepth);// 盤面の評価値を計算
```

```
                                        if (value < bestValue) {                    // 高評価の
手を発見した
                                                bestMove = nextMove;                //
高評価の手を記憶
                                                bestValue = value;                  //
評価値を記憶
                                        }
                                        if (bestValue == Integer.MIN_VALUE) {       // 評価無
限小の手＝必勝の手を発見
                                                --lookAheadMoves; // 先読みが終了したの
で手数を 1 戻す
                                                break;                              //
ループ脱出
                                        }
                                }
                                --lookAheadMoves; // 先読みが終了したので手数を 1 戻す
                                if (bestValue == Integer.MAX_VALUE && doResign) {  // 評価
値無限大の手しか無い＝負け確定
                                        System.out.println("投了します");
                                        resign = true;
                                        if (isChessStyleScore) {
                                                return "0-1";
                                        } else {
                                                return "投了";
                                        }
                                }
                        }
                }
                type = bestMove.type();             // 移動する駒の種類
                nextX = bestMove.nextX(); // 移動先の X 座標
                nextY = bestMove.nextY(); // 移動先の Y 座標

                switch (type) {
                case PAWN:      piece = pawn;      break;
                case KING:      piece = king;      break;
                case KING2:     piece = king2;     break;
                }
                String score = movePiece(piece, type, nextX, nextY); // 駒を移動させる
                System.out.println(moves + ":" + score + "[" + bestValue + "]");
                ++moves;
                return score;
        }

        /**
         * COM の手番
         * ランダムに手を指す
         * @paran int playerNum プレイヤー番号
```

```
     * @return 指した手の棋譜
     */
    public String ran(int playerNum) {
        int type, nextX, nextY;          // 移動する駒の種類，移動先の座標
        Piece piece = null;                    // 移動する駒

        NextMove ranMove = null;               // ランダム手
        boolean isMateToOne = false;           // 1 手詰めか
        int nextPlayerNum;                     // 次の手番プレイヤー

        if (movableList.size() == 0) { //  候補手が存在しない＝詰み
            System.out.println("投了します");
            resign = true;
            if (isChessStyleScore) {
                return (playerNum == 0) ? "1-0" : "0-1";
            } else {
                return "投了";
            }
        } else if (movableList.size() == 1) { //  候補手が 1 つしか無い場合は評価値を求めない
            ranMove = movableList.get(0);
        } else {
            if (playerNum == 0) { //  先手チームの場合  評価値が高いほど良い手と見做す
                nextPlayerNum = 1;             //  次の手番プレイヤー
                for (int i = 0; i < movableList.size(); ++i) {
                    NextMove nextMove = movableList.get(i);      // i 番目の候補手
                    Board nextBoard = nextBoard(nextMove, nextPlayerNum);      //  次の盤面を生成
                    //nextBoard.createMovableList(playerNum);      //  候補手を生成
                    int value = nextBoard.value(nextPlayerNum, 0);//  先読み無しで盤面の評価値を計算
                    if (value == Integer.MAX_VALUE) {            // 1 手詰めの手を発見した
                        ranMove = nextMove;                      // 1 手詰めの手を記憶
                        isMateToOne = true;
                        break;                         //  ループ脱出
                    }
                }
            } else { //  後手チームの場合  評価値が低いほど良い手と見做す
                nextPlayerNum = 0;             //  次の手番プレイヤー
                for (int i = 0; i < movableList.size(); ++i) {
                    NextMove nextMove = movableList.get(i);      // i 番目の候補手
```

```
                                        Board nextBoard = nextBoard(nextMove,
nextPlayerNum);     // 次の盤面を生成
                                        //nextBoard.createMovableList(playerNum);     // 候
補手を生成
                                        int value = nextBoard.value(nextPlayerNum, 0);// 先読
み無しで盤面の評価値を計算
                                        if (value == Integer.MIN_VALUE) {           // 1 手詰め
の手を発見した
                                                ranMove = nextMove;                 //
1 手詰めの手を記憶
                                                isMateToOne = true;
                                                break;                      // ループ脱出
                                        }
                                }
                        }
                        if (!isMateToOne) { // 1 手詰めの手は無かった
                                Random rnd = new Random();
                                int ran = rnd.nextInt(movableList.size()); // 候補手数までの乱数
を生成
                                ranMove = movableList.get(ran);
                        }
                }
                type = ranMove.type();              // 移動する駒の種類
                nextX = ranMove.nextX(); // 移動先の X 座標
                nextY = ranMove.nextY(); // 移動先の Y 座標

                switch (type) {
                case PAWN:    piece = pawn;    break;
                case KING:    piece = king;    break;
                case KING2:   piece = king2;   break;
                }
                String score = movePiece(piece, type, nextX, nextY); // 駒を移動させる
                System.out.println(moves + ":" + score + "[?]");
                ++moves;
                return score;
        }

        /**
         * 指定した位置に駒を移動させ、その棋譜表記を返す
         * @param Piece piece  移動させる駒
         * @param int type  移動させる駒の種類
         * @param int nextX  移動先の X 座標
         * @param int nextY  移動先の Y 座標
         */
        public String movePiece(Piece piece, int type, int nextX, int nextY) {
                String score; // 棋譜
```

```
        if (isChessStyleScore) {                                    // チェス風の棋譜を作成
                switch (type) {
                case PAWN:     score = "P"; break;
                case KING:     score = "K"; break;
                case KING2: score = "K2"; break;
                default:       score = "?"; break;
                }

                if (board[nextY][nextX] != EMPTY) score += "x";
                switch (nextX) {
                case 1:   score += "a"; break;
                case 2:   score += "b"; break;
                case 3:   score += "c"; break;
                case 4:   score += "d"; break;
                case 5:   score += "e"; break;
                case 6:   score += "f"; break;
                case 7:   score += "g"; break;
                case 8:   score += "h"; break;
                default: score += "?"; break;
                }

                switch (nextY) {
                case 1:   score += "8 "; break;
                case 2:   score += "7 "; break;
                case 3:   score += "6 "; break;
                case 4:   score += "5 "; break;
                case 5:   score += "4 "; break;
                case 6:   score += "3 "; break;
                case 7:   score += "2 "; break;
                case 8:   score += "1 "; break;
                default: score += "? " ; break;
                }
        }

        if (board[nextY][nextX] != EMPTY) { //  移動先に駒がある場合
                removePiece(nextX, nextY); //  移動先にある駒を取り除く
        }

        board[piece.y()][piece.x()] = EMPTY; //  移動前のマスを空白に
        piece.move(nextX, nextY);                    //  駒を移動

        board[piece.y()][piece.x()] = type;   //  移動後のマスを指定した駒に

        recordBoard(); //  移動後の盤面を記録
        return score;
}
```

```
// 指定した位置にある駒を盤から取り除く
public void removePiece(int nextX, int nextY) {
        switch (board[nextY][nextX]) {
        case PAWN:                      // 移動先にポーン
                pawn = null;            // ポーンを取り除く
                break;
        case KING:                      // 移動先にキング(先手)
                king = null;            // キングを取り除く
                break;
        case KING2:                     // 移動先にキング(後手)
                king2 = null;           // キング(後手)を取り除く
                break;
        }
}

/**
 * 勝負がついたか
 * @param int playerNum プレイヤー番号
 * @return 勝負がついたか
 */
public boolean checkWin(int playerNum) {
        if (playerNum == 0) {                   // 先手の手番
                if (king2 == null) {            // キング(後手)を取った
                        System.out.println("先手の勝利");
                        winner = 1;
                        return true;
                } else if (isMate (1)) { // キング(後手)が詰んだ
                        if (isChecked (1)) { // キング(後手)にチェックがかかっている
                                System.out.println("チェックメイト");
                                System.out.println("先手の勝利");
                                winner = 1;
                        } else {
                                System.out.println("スティールメイト");
                                System.out.println("引き分けです");
                                winner = 0;
                        }
                        return true;
                } else if (resign) {                    // 先手が投了した
                        System.out.println("後手の勝利");
                        winner = -1;
                        return true;
                } else if (checkPerpetual()) {
                        System.out.println("千日手");
                        System.out.println("引き分けです");
                        winner = 0;
                        return true;
                } else if (pawn.y() == 1) {     // ポーンが昇格した
```

```
                    if(pawn.y()+1 == king2.y()) {
                        if(pawn.x()-1 == king2.x() || pawn.x() == king2.x() ||
pawn.x()+1 == king2.x()) {
                            if(pawn.y()+1 == king.y()) {
                                if(pawn.x()-1 != king.x() || pawn.x() !=
king.x() || pawn.x()+1 != king.x()) {
                                    System.out.println("ポーン
が昇格した");
                                    System.out.println(" 先 手
の勝利");
                                    winner = 1;
                                }
                            } else if (pawn.y() == king.y()) {
                                if(pawn.x()-1 != king.x() ||
pawn.x()+1 != king.x()) {
                                    System.out.println("ポーン
が昇格した");
                                    System.out.println(" 先 手
の勝利");
                                    winner = 1;
                                }
                            } else
                                System.out.println("昇格したポーン
が次の番とられます");
                                System.out.println("引き分けです");
                                winner = 0;
                        } else {
                            System.out.println("ポーンが昇格した");
                            System.out.println("先手の勝利");
                            winner = 1;
                        }
                    } else if(pawn.y() == king2.y()) {
                        if(pawn.x()-1 == king2.x() || pawn.x()+1 == king2.x()) {
                            if(pawn.y()+1 == king.y()) {
                                if(pawn.x()-1 != king.x() || pawn.x() !=
king.x() || pawn.x()+1 != king.x()) {
                                    System.out.println("ポーン
が昇格した");
                                    System.out.println(" 先 手
の勝利");
                                    winner = 1;
                                }
                            } else if (pawn.y() == king.y()) {
                                if(pawn.x()-1 != king.x() ||
pawn.x()+1 != king.x()) {
                                    System.out.println("ポーン
が昇格した");
```

```
                                        System.out.println(" 先 手の勝利");
                                        winner = 1;
                                    }
                                } else
                                    System.out.println("昇格したポーンが次の番とられます");
                                    System.out.println("引き分けです");
                                    winner = 0;
                            } else {
                                System.out.println("ポーンが昇格した");
                                System.out.println("先手の勝利");
                                winner = 1;
                            }
                        } else {
                            System.out.println("ポーンが昇格した");
                            System.out.println("先手の勝利");
                            winner = 1;
                        }
                        return true;
                    } else {
                        return false;
                    }
                } else {                                    // 後手の手番
                    if (isMate (0)) { // 先手が詰んだ
                        if (isChecked (0)) { // キング(先手)にチェックがかかっている
                            System.out.println("チェックメイト");
                            System.out.println("後手の勝利");
                            winner = -1;
                        } else {
                            System.out.println("スティールメイト");
                            System.out.println("引き分けです");
                            winner = 0;
                        }
                        return true;
                    } else if (resign) {                    // 後手が投了した
                        System.out.println("先手の勝利");
                        winner = 1;
                        return true;
                    } else if (checkPerpetual()) {
                        System.out.println("千日手");
                        System.out.println("引き分けです");
                        winner = 0;
                        return true;
                    } else if (pawn == null) {
                        System.out.println("ポーンをとりました");
                        System.out.println("引き分けです");
```

```
                        winner = 0;
                        return true;
                    } else {
                        return false;
                    }
                }
            }

            /**
             * 現在チェックがかかっているか
             * @param int playerNum プレイヤー番号
             * @return チェックがかかっているか
             */
            public boolean isChecked(int playerNum) {
                int x, y;
                if (playerNum == 0) {
                    x = king.x(); // キング(先手)の座標
                    y = king.y();
                    if(board[y + 1][x - 1] == KING2) return true;              // キング(後手)
がチェック
                    else if(board[y + 1][x] == KING2) return true;       // キング(後手)がチェ
ック
                    else if(board[y + 1][x + 1] == KING2) return true;       // キング(後手)が
チェック
                    else if(board[y- 1][x] == KING2) return true;       // キング(後手)がチェ
ック
                    else if(board[y - 1][x - 1] == KING2) return true;       // キング(後手)が
チェック
                    else if(board[y - 1][x + 1] == KING2) return true;       // キング(後手)が
チェック
                    else if(board[y][x - 1] == KING2) return true;       // キング(後手)がチェ
ック
                    else if(board[y][x + 1] == KING2) return true;       // キング(後手)がチェ
ック
                    else return false;
                } else {
                    x = king2.x(); // キング(後手)の座標
                    y = king2.y();
                    if(board[y + 1][x - 1] == PAWN) return true;            // ポーンがチェック
                    else if(board[y + 1][x + 1] == PAWN) return true;    // ポーンがチェック
                    else return false;
                }
            }

            /**
             * 詰みの判定
             * 移動可能な手が無ければ詰み(チェックメイトまたはスティールメイト)
```

```
	 * @param int playerNum プレイヤー番号
	 * @return 詰んだか
	 */
	public boolean isMate(int playerNum) {
		return (movableList.size() == 0); //  可能な手が無ければ詰み
	}

	/**
	 * 候補手の作成
	 * また, 移動可能な手のリストを movableList に保持する
	 * @param int playerNum プレイヤー番号
	 */
	public void createMovableList(int playerNum) {
		if (playerNum == 0) { //  先手の場合
			movableList = king.movableList(board);			// キングが移動可能な手
			if (pawn != null) {					// 盤上にまだポーンがある場合
				movableList.addAll (pawn.movableList (board)); // ポーンが移動可能な手
			}
			if (isChecked(0)) { //  キング(先手)にチェックが掛かっている場合
				int nextPlayerNum = (playerNum == 0) ? 1 : 0;	// 次のプレイヤー番号
				for(int i = 0; i < movableList.size();) {
					Board	nextBoard	=	nextBoard(movableList.get(i), playerNum);
					if (nextBoard.isChecked (0)) { //  移動後もチェックが掛かっている手は無効
						movableList.remove(i);		// 移動後もチェックが掛かっている手を取り除く
					} else {
						++i;
					}
				}
			}
		} else {			// 後手の場合
			movableList = king2.movableList(board);			// キング(後手)が移動可能な手
			if (isChecked (1)) { //  キング(後手)にチェックが掛かっている場合
				for (int i = 0; i < movableList.size();) {
					Board	nextBoard	=	nextBoard(movableList.get(i), playerNum);
					if (nextBoard.isChecked (1)) { //  移動後もチェックが掛かっている手は無効
						movableList.remove(i);		// 移動後もチェックが掛かっている手を取り除く
```

```
                                        } else {
                                                ++i;
                                        }
                                }
                        }
                }
}

/**
 * 現在の盤の評価値を表示
 * 先読みは行わず現在の盤面のみで評価する
 * @param int playerNum プレイヤー番号
 * @return 評価値
 */
public int value(int playerNum) {
        checkmate = false;
        stalemate = false;

        /* 現時点ですでに詰んでいるかどうかのチェック */
        if (playerNum == 0) { // 先手の手番
                if (king == null) {                     // キング(先手)が取られた
                        checkmate = true;
                        return Integer.MIN_VALUE; // 評価値無限小
                } else if (isMate (0)) { // キング(先手)が詰んだ
                        if (isChecked (0)) { // キング(先手)にチェックがかかっている
                                checkmate = true;
                                return Integer.MIN_VALUE; // 評価値無限小
                        } else { // ステールメイト
                                stalemate = true;
                                return 0; // 評価値 0
                        }
                } else if (resign) {                    // 後手が投了した
                        return Integer.MAX_VALUE; // 評価値無限大
                } else if(pawn == null) {     // ポーンを取られたら引き分け
                        stalemate = true;
                        return 0; // 評価値 0
                } /*else if(pawn != null && pawn.y() == 1) {
                        checkmate = true;
                        return Integer.MAX_VALUE; // 評価値無限小
                }*/
        } else { // 後手の手番
                if (king2 == null) {                    // キング(後手)が取られた
                        checkmate = true;
                        return Integer.MAX_VALUE; // 評価値無限大
                } else if (isMate (1)) { // キング(後手)が詰んだ
                        if (isChecked (1)) { // キング(後手)にチェックがかかっている
                                checkmate = true;
```

```
                        return Integer.MAX_VALUE; //  評価値無限大
                    } else { //  ステールメイト
                        stalemate = true;
                        return 0; //  評価値 0
                    }
                } else if (resign) {                    //  先手が投了した
                    return Integer.MIN_VALUE; //  評価値無限小
                }/* else if(pawn == null) {     //  ポーンを取れたら引き分け
                    stalemate = true;
                    return 0; //  評価値 0
                }*/ else if(pawn != null && pawn.y() == 1) {
                    checkmate = true;
                    return Integer.MAX_VALUE; //  評価値無限小
                }
            }

            /* 現時点ではまだ詰んでいない場合 */
            int value = 0;

            /* 先手の評価値 */


            // ポーン y 座標 1 までの距離よりポーンとキング(後手)との距離が短いときポーンの
評価値は高い
        //      if(pawn.y() < Math.abs(pawn.x() - king2.x()) + 1) {
                switch (pawn.y()) { // ポーンは y 座標が 1 に近い方が高評価
                case 1:    // ポーンの y 座標した場合
                    return  Integer.MAX_VALUE; // すでにチェックしているのでここ
に処理が移ることはありえない
                case 2:
                    value += 13;
                    break;
                case 3:
                    value += 11;
                    break;
                case 4:
                    value += 9;
                    break;
                case 5:
                    value += 7;
                    break;
                case 6:
                    value += 5;
                    break;
                case 7:
                    value += 3;
                    break;
```

```
                    case 8:
                        value += 1;
                        break;
                    }
//          }

                    switch(Math.abs(pawn.x() - king.x())) {          // キング(先手)は x 軸がポーンに近いほど評価値が高い
                    case 0:
                        value += 15;
                        break;
                    case 1:
                        value += 13;
                        break;
                    case 2:
                        value += 11;
                        break;
                    case 3:
                        value += 9;
                        break;
                    case 4:
                        value += 7;
                        break;
                    case 5:
                        value += 5;
                        break;
                    case 6:
                        value += 3;
                        break;
                    }

                    switch(pawn.y() - king.y()) {          // キング(先手)は y 軸がポーンにより前にあるほど評価値が高い
                    case 0:
                        value += 1;
                        break;
                    case 1:
                        value += 3;
                        break;
                    case 2:
                        value += 5;
                        break;
                    case 3:
                        value += 7;
                        break;
                    case 4:
                        value += 9;
```

```
                    break;
                case 5:
                    value += 11;
                    break;
                case 6:
                    value += 13;
                    break;
                case 7:
                    value += 15;
                    break;
                }

//                  if(Math.abs(pawn.y() - king.y()) < Math.abs(pawn.y() - king2.y())) {    // ポーンが取られない距離を保つ
//          }


                    /* 後手の評価値    */
                    switch(Math.abs(pawn.y() - king2.y())) {            // キング(後手)は y 軸がポーンに近いほど評価値が低い
                case 0:
                    value -= 15;
                    break;
                case 1:
                    value -= 13;
                    break;
                case 2:
                    value -= 11;
                    break;
                case 3:
                    value -= 9;
                    break;
                case 4:
                    value -= 7;
                    break;
                case 5:
                    value -= 5;
                    break;
                case 6:
                    value -= 3;
                    break;
                }

                    switch(Math.abs(pawn.x() - king2.x())) {            // キング(後手)は x 軸がポーンに近いほど評価値が低い
                case 0:
                    value -= 15;
```

```
                break;
        case 1:
                value -= 13;
                break;
        case 2:
                value -= 11;
                break;
        case 3:
                value -= 9;
                break;
        case 4:
                value -= 7;
                break;
        case 5:
                value -= 5;
                break;
        case 6:
                value -= 3;
                break;
        }


        /* 決まり手 */
        if(playerNum == 1) {
                if(king2.y() - pawn.y() >= 1) { // キング(後手)がポーンより後ろに行くと追いつけないため評価値最大
                        value = Integer.MAX_VALUE;
                }
        } else {
                if(king2.y() - pawn.y() >= 2) { // キング(後手)がポーンより後ろに行くと追いつけないため評価値最大
                        value = Integer.MAX_VALUE;
                }
        }


        if (playerNum == 1) {
                if(pawn.y() <= Math.abs(pawn.x()) - king2.x()) {   // キング(後手)がポーンが昇格するまでに追いつけないため評価値最大
                        value = Integer.MAX_VALUE;
                }
        } else {
                if(pawn.y() < Math.abs(pawn.x()) - king2.x()) {   // キング(後手)がポーンが昇格するまでに追いつくため評価値最小
                        value = Integer.MAX_VALUE;
                }
        }
```

```
		return value;
	}

	/**
	 * 現在の盤の評価値を表示
	 * @param int playerNum プレイヤー番号
	 * @param int depth 先読みする手数
	 * @return 評価値
	 */
	public int value(int playerNum, int depth) {
		createMovableList(playerNum);	// 移動可能な手のリストを生成

		int value = value(playerNum); // 先読み無しの現在の盤面の評価値を求める
		if (depth == 0) {	// 一定手数まで読んでいる場合はそれ以上先読みしない
			return value;
		}
		if (checkmate || stalemate || resign) return value; // すでに詰んでいるときはそのまま
値を返す
		if (precheckPerpetual()) {
			// System.out.println (moves + "+" + lookAheadMoves + ":" + "parpetual");
			return 0; // 先読み開始後同一局面になる場合は評価を停止する
		}
		int bestValue;				// 最も良い評価値
		NextMove bestMove = null;		// 最も良い手
		int nextPlayerNum;			// 次の手番プレイヤー

		++lookAheadMoves; // 先読みのために手数を 1 増やす
		if (playerNum == 0) { // 先手の場合 評価値が高いほど良い手と見做す
			nextPlayerNum = 1;			// 次の手番プレイヤー
			bestValue = Integer.MIN_VALUE;		// 盤面の評価値の初期値

			for (int i = 0; i < movableList.size(); ++i) {
				NextMove nextMove = movableList.get(i);	// i 番目の候補手
				Board nextBoard = nextBoard(nextMove, nextPlayerNum);	//
次の盤面を生成
				value = nextBoard.value(nextPlayerNum, depth - 1);// 盤面の評
価値を計算
				if (value > bestValue) {			// 高評価の手を発見
した
					bestMove = nextMove;			// 高評価
の手を記憶
					bestValue = value;			// 評価値を
記憶
				}
				if (bestValue == Integer.MAX_VALUE) {	// 評価無限大の手
```

```
＝必勝の手を発見
                                        --lookAheadMoves; // 先読みが終了したので手数を 1
戻す
                                        return Integer.MAX_VALUE;
                                }
                        }
                } else { // 後手チームの場合 評価値が低いほど良い手と見做す
                        nextPlayerNum = 0;                      // 次のプレイヤー番号
                        bestValue = Integer.MAX_VALUE; // 盤面の評価値の初期値

                        for (int i = 0; i<movableList.size(); ++i) {
                                NextMove nextMove = movableList.get(i);
                        // i 番目の候補手
                                Board nextBoard = nextBoard(nextMove, nextPlayerNum);
        // 次の盤面を生成
                                value = nextBoard.value(nextPlayerNum, depth - 1);
                // 盤面の評価値を計算
                                if (value < bestValue) {
                        // 高評価の手を発見した
                                        bestMove = nextMove;
                                        // 高評価の手を記憶
                                        bestValue = value;
                                // 評価値を記憶
                                }
                                if (bestValue == Integer.MIN_VALUE) {
                        // 評価無限小の手＝必勝の手を発見
                                        --lookAheadMoves;
                                                        // 先読みが終了したので手数を 1
戻す
                                        return Integer.MIN_VALUE;
                                }
                        }
                }
                --lookAheadMoves; // 先読みが終了したので手数を 1 戻す
                Random rnd = new Random();
                int ran = rnd.nextInt(3); // 0～2 の乱数を生成
                if (bestValue != Integer.MAX_VALUE && bestValue != Integer.MIN_VALUE) {
                        if (playerNum == 0) {
                                bestValue = bestValue + ran - 1; // 評価値に乱数を加える
                        } else {
                                //bestValue = bestValue + ran - 1; // 評価値に乱数を加える
                        }

                        //bestValue = bestValue + ran - 1; // 評価値に乱数を加える
                }

                //System.out.println (moves + "+" + lookAheadMoves + ":" + bestMove + "[" +
```

```
bestValue + "]");
                return bestValue;
        }

        /**
         * 指定した駒を指定した位置に動かした後の盤を得る
         * @param NextMove nextMove 次の手
         * @param int playerNum プレイヤー番号
         * @return 移動した後の盤
         */
        public Board nextBoard(NextMove nextMove, int playerNum) {
                Board nextBoard = new Board (board);
                int movingType =   nextMove.type(); // 移動する駒の種類
                Piece movingPiece = null; // 移動する駒
                switch (movingType) {
                case PAWN :
                        movingPiece = nextBoard.pawn;
                        break;
                case KING :
                        movingPiece = nextBoard.king;
                        break;
                case KING2 :
                        movingPiece = nextBoard.king2;
                        break;
                }

                int nextX = nextMove.nextX(); // 移動先の X 座標
                int nextY = nextMove.nextY(); // 移動先の Y 座標
                nextBoard.movePiece(movingPiece, movingType, nextX, nextY); // 駒を移動させる

                return nextBoard;
        }

        /**
         * 駒名を返す
         * @param int type 駒の種類
         * @return 駒名
         */
        public String name(int type) {
                switch (type) {
                case PAWN:
                        return "ポーン";
                case KING:
                        return "キング";
                case KING2:
                        return "キング 2";
                default:
```

```
                return "？";
            }
        }

        /**
         * 現在の盤面を記憶
         * 現在の盤面を配列 boardRecord に記憶する
         */
        void recordBoard() {
            if (moves + lookAheadMoves < maxMove) {
                for (int y = 0; y < 8; ++y) {
                    for (int x = 0; x < 8; ++x) {
                        boardRecord[moves + lookAheadMoves][y][x] = board[y
+ 1][x + 1];
                    }
                }
            }
        }

        /**
         * 千日手になったか判定
         * 同じ局面が 3 回出てくれば千日手と見做す
         * @return 千日手か
         */
        boolean checkPerpetual() {
            int counter = 0;
            for (int i = moves - 3; i >= 0; i -= 2) {
                int match = 0;
                for (int y = 0;   y < 8; ++y) {
                    for (int x = 0; x < 3; ++x) {
                        if (boardRecord [i][y][x] == board[y + 1][x + 1]) {
                            ++match;
                        }
                    }
                }
                if (match == 15) {
                    ++counter;
                    if (counter >= 3) return true;
                }
            }
            return false;
        }

        /**
         * 先読み時に同じ局面になったか判定
         * 先読み開始以降同じ局面が出てくればそれ以上先読みはしない
         * @return 同一局面か
```

```
	 */
	boolean precheckPerpetual() {
		int m = moves - 6;
		if (m < 0) m = 0;
		for (int i = moves + lookAheadMoves - 1; i >= m; --i) {
			int match = 0;
			for (int y = 0; y < 8; ++y) {
				for (int x = 0; x < 8; ++x) {
					if (boardRecord [i][y][x] == board[y + 1][x + 1]) {
						++match;
					}
				}
			}
			if (match == 15) {
				// System.out.print (moves + "+" + lookAheadMoves + ":" + i +
"parpetual");
				return true;
			}
		}
		return false;
	}

	/**
	 * 先読みの上限数を指定
	 * @param d 先読みの上限数
	 */
	public void setMaxDepth(int d) {
		maxDepth = d;
	}

	/**
	 * 手数をリセット
	 */
	public void resetMoves() {
		moves = 0;
		lookAheadMoves = 0;
	}

	/**
	 * 勝者の番号を帰す
	 * 1:先手勝ち -1:後手勝ち 0:引き分け
	 * @return 勝者の番号
	 */
	public int winner() {
		return winner;
	}
}
```

・Piece クラス

```
package chess;

import java.util.ArrayList;

public class Piece {
	final static int KING	= 1;
				// キング(先手)
	final static int PAWN	= 2;
			// ポーン
	final static int KING2	= -1;
				// キング(後手)
	final static int EMPTY	= 0;
	// 空白
	final static int BORDER = Integer.MAX_VALUE;
	// 盤外

	final static int[] PAWN_MOVABLE_X_VECTOR	= {0};			// ポーンの移動可能X方向
	final static int[] PAWN_MOVABLE_Y_VECTOR	= {-1};			// ポーンの移動可能Y方向
	final static int[] KING_MOVABLE_X_VECTOR	= {-1, -1,  0,  1,  1, -1,  0,  1}; // キング(先手)の移動可能X方向
	final static int[] KING_MOVABLE_Y_VECTOR	= { 0, -1, -1, -1,  0,  1,  1,  1}; // キング(先手)の移動可能Y方向
	final static int[] KING2_MOVABLE_X_VECTOR	= {-1, -1,  0,  1,  1, -1,  0,  1}; // キング(後手)の移動可能X方向
	final static int[] KING2_MOVABLE_Y_VECTOR	= { 0, -1, -1, -1,  0,  1,  1,  1}; // キング(後手)の移動可能Y方向

	boolean isOnBoard;
					// 盤上に駒があるか
	int x;
				// 駒のX座標
	int y;
				// 駒のY座標
	int type;
					// 駒の種類
	int movableYVector[], movableXVector[];			// 移動可能方向(駒の現在位置を(0,0)とした場合の相対位置)

	/**
	 * コンストラクタ
	 * 駒を生成し初期位置に置く
	 * @param int type 駒の種類
	 */
```

```
public Piece(int type) {
		this.type = type;
		setMovableVector();
		setInitialPosition();
}

/**
 * コンストラクタ
 * 駒を生成し指定した位置に置く
 * @param int type  駒の種類
 * @param int
 */
public Piece(int type, int x, int y) {
		this.type = type;
		setMovableVector();
		this.x = x;
		this.y = y;
		this.isOnBoard = true;
}

/**
 * 駒の移動可能方向をセット
 */
private void setMovableVector() {
		switch (type) { //  駒の種類により分岐
		case PAWN: //  ポーンの場合
				movableXVector = PAWN_MOVABLE_X_VECTOR;
				movableYVector = PAWN_MOVABLE_Y_VECTOR;
				break;
		case KING: //  キング（先手）の場合
				movableXVector = KING_MOVABLE_X_VECTOR;
				movableYVector = KING_MOVABLE_Y_VECTOR;
				break;
		case KING2: //   キング（後手）の場合
				movableXVector = KING2_MOVABLE_X_VECTOR;
				movableYVector = KING2_MOVABLE_Y_VECTOR;
				break;
		default: //  それ以外の場合
				System.out.println ("駒の種類を認識できません");
		}
}

/**
 * 駒を盤上の初期位置にセット
 */
private void setInitialPosition(){
		switch (type) {
```

```
            case PAWN:      // ポーンの場合
                    y = 7;
                    x = 5;
                    break;
            case KING: // キング（先手）の場合
                    y = 8;
                    x = 5;
                    break;
            case KING2: // キング（後手）の場合
                    y = 1;
                    x = 5;
                    break;
            }
            isOnBoard = true;
}

/**
 * 駒を盤上の指定した座標にセット
 * @param int x X座標
 * @param int y Y座標
 */
private void setPosition(int x, int y) {
            this.x = x;
            this.y = y;
            isOnBoard = true;
}

/**
 * 駒を盤上に置く
 */
public void setOnBoard() {
            isOnBoard = true;
}

/**
 * 駒を盤上から削除
 */
public void removeFromBoard() {
            isOnBoard = false;
}

/**
 * 盤上に駒が存在するか
 * @return 駒が存在するか
 */
public boolean isOnBoard() {
            return isOnBoard;
```

```
}

/**
 * 指定した座標に駒が存在するか
 * @param int x X座標
 * @param int y Y座標
 * @return 駒が存在するか
 */
public boolean isThere(int x, int y) {
        if(isOnBoard) {
                return ((this.x == x) && (this.y == y));
        } else
                return false;
}

/**
 * X座標を返す
 * @return X座標
 */
public int x(){
        return this.x;
}

/**
 * Y座標を返す
 * @return Y座標
 */
public int y(){
        return this.y;
}

/**
 * 駒の種類を返す
 * @return 駒の種類
 */
public int type(){
        return this.type;
}

/**
 * 駒名を返す
 * @return 駒名
 */
public String name() {
        switch (type) {
        case PAWN:
                return "ポーン";
```

```
            case KING:
                return "キング";
            case KING2:
                return "キング 2";
            default:
                return "？";
            }
}

/**
 * 駒を指定した座標に移動する
 * @param nextX 移動するX座標
 * @param nextY 移動するY座標
 */
public void move(int nextX, int nextY) {
            x = nextX;
            y = nextY;
}

/**
 * 駒が指定した座標に移動できるか
 * 他の駒は無視して自分が移動できるかのみを判断する
 * @param int nextX 移動したいX座標
 * @param int nextY 移動したいY座標
 * @return 移動できるか
 */
public boolean movable(int nextX, int nextY) {
            if (!isOnBoard) // すでに取られている場合
                return false;
            if (nextX < 1 || nextX > 8 || nextY < 1 || nextY > 8) // 盤外を指定した場合
                return false;
            for (int i = 0; i < movableXVector.length; ++i) {
                if ((nextX == x + movableXVector[i]) && (nextY == y + movableYVector[i]))
                    return true;
            }
            return false;
}

/**
 * 駒が指定した座標に移動できるか
 * 他の駒も考慮して移動できるかを判断する
 * @param int[][] board 現在の盤
 * @param int nextX 移動したいX座標
 * @param int nextY 移動したいY座標
 * @return 移動できるか
 */
public boolean movable(int[][] board, int nextX, int nextY) {
```

```
		if (!isOnBoard) //  すでに取られている場合
			return false;
		ArrayList<NextMove> movableList = movableList(board);
		for (int i = 0; i < movableList.size(); ++i) {
			if ((movableList.get(i).nextX() == nextX) && (movableList.get(i).nextY() ==
nextY)) {
				return true;
			}
		}
		return false;
	}

	/**
	 * 駒を指定した座標に移動する
	 * 他の駒は無視して自分が移動できるかのみを判断し、可能なら移動する
	 * @param int nextX  移動したいX座標
	 * @param int nextY  移動したいY座標
	 * @return  移動できたか
	 */
	public boolean checkAndMove(int nextX, int nextY) {
		if (movable (nextX, nextY)) {
			x = nextX;
			y = nextY;
			return true;
		} else return false;
	}

	/**
	 * 駒を指定した座標に移動する
	 * 他の駒の位置も考慮して移動できるか判断し、可能なら移動する
	 * @param int[][] board  現在の盤
	 * @param int nextX  移動したい  x座標
	 * @param int nextY  移動したい  y座標
	 * @return  移動できたか
	 */
	public boolean checkAndMove(int[][] board, int nextX, int nextY) {
		if (movable (board, nextX, nextY)) {
			x = nextX;
			y = nextY;
			return true;
		} else return false;
	}

	/**
	 * 移動可能な座標のリストを返す
	 * @param int[][] board  現在の盤
	 * @return  移動可能な座標のリスト
```

```
         */
        public ArrayList<NextMove> movableList(int[][] board) {
                ArrayList<NextMove> movableList = new ArrayList<NextMove>();
                boolean[][] isMovable = {{false, false, false, false, false, false, false, false, false, false},
// 移動可能な位置
                                        {false, true,   true,  true,  true,  true,  true,  true,  true,
false},
                                        {false, true,   true,  true,  true,  true,  true,  true,  true,
false},
                                        {false, true,   true,  true,  true,  true,  true,  true,  true,
false},
                                        {false, true,   true,  true,  true,  true,  true,  true,  true,
false},
                                        {false, true,   true,  true,  true,  true,  true,  true,  true,
false},
                                        {false, true,   true,  true,  true,  true,  true,  true,  true,
false},
                                        {false, true,   true,  true,  true,  true,  true,  true,  true,
false},
                                        {false, true,   true,  true,  true,  true,  true,  true,  true,
false},
                                        {false, false, false, false, false, false, false, false, false, false}};

                switch (type) { // 他の駒による移動不可能な位置の判定
                case KING: // キング(先手)の場合
                        for (int y = 1; y <= 8; ++y) {
                                for (int x = 1; x <= 8; ++x) {
                                        switch (board[y][x]) {
                                        case KING:      // 自分の駒がある位置へは移動不
可
                                        case PAWN:
                                                isMovable[y][x] = false;
                                                break;
                                        case KING2:     // キング(後手)に取られる位置へ
は移動不可
                                                isMovable[y][x - 1] = false;
                                                isMovable[y + 1][x - 1] = false;
                                                isMovable[y + 1][x] = false;
                                                isMovable[y + 1][x + 1] = false;
                                                isMovable[y][x + 1] = false;
                                                isMovable[y - 1][x - 1] = false;
                                                isMovable[y - 1][x] = false;
                                                isMovable[y - 1][x + 1] = false;
                                                break;
                                        }
                                }
                        }
```

```
                case PAWN: // ポーンの場合
                        for(int y = 1; y <= 8; ++y) {
                                for(int x = 1; x <= 8; ++x) {
                                        switch (board[y][x]) {
                                        case KING:      // 自分の駒がある位置へは移動不
可
                                        case PAWN:
                                                isMovable[y][x] = false;
                                                break;
                                        case KING2:     // キング(後手)が前にあっても進め
ない
                                                //isMovable[y - 1][x] = false;
                                                isMovable[y][x] = false;
                                                break;
                                        }
                                }
                        }
                        break;
                case KING2: // キング(後手)の場合
                        for (int y = 1; y <= 8; ++y) {
                                for (int x = 1; x <= 8; ++x) {
                                        switch (board[y][x]) {
                                        case KING2:     // 自分の駒がある位置へは移動
不可
                                                isMovable[y][x] = false;
                                                break;
                                        case KING:      // キング(先手)に取られる位置へは
移動不可
                                                isMovable[y][x - 1] = false;
                                                isMovable[y][x + 1] = false;
                                                isMovable[y - 1][x - 1] = false;
                                                isMovable[y - 1][x] = false;
                                                isMovable[y - 1][x + 1] = false;
                                                isMovable[y + 1][x - 1] = false;
                                                isMovable[y + 1][x] = false;
                                                isMovable[y + 1][x + 1] = false;
                                                break;
                                        case PAWN:      // ポーンに取られる位置へは移動
不可
                                                isMovable[y - 1][x + 1] = false;
                                                isMovable[y - 1][x - 1] = false;
                                                break;
                                        }
                                }
                        }
                        break;
                }
```

```
                for (int i = 0; i < movableXVector.length; ++i) { // 移動可能かチェック
                        int nextX = x + movableXVector[i];
                        int nextY = y + movableYVector[i];
                        if (isMovable [nextY][nextX]) {
                                NextMove nextMove = new NextMove(type, nextX,
nextY); // 移動可能リストに挿入
                                movableList.add(nextMove);
                        }
                }
        return movableList;
    }

    /**
     * 移動可能な座標を表示する
     * @param int[][] board 現在の盤
     */
    public void showMovable(int[][] board) {
        ArrayList<NextMove> movableList = movableList (board); // 移動可能な座標の判
定

        if (movableList.size() == 0) { // 駒が移動可能な位置が無い場合
            System.out.println(name() + "が進める位置はありません");
        } else {
            System.out.print (name() + "は現在(" + x + "," + y + ")にいて");
            for (int i = 0; i < movableList.size(); ++i) {
                int nextX = movableList.get(i).nextX();
                int nextY = movableList.get(i).nextY();
                System.out.print("(" +  nextX + "," + nextY + ")");
            }
            System.out.println("へ移動できます");
        }
    }

    /**
     * クローン生成
     */
    public Piece clone() {
        Piece newPiece = new Piece (this.type, this.x, this.y);
        if (!isOnBoard) newPiece.removeFromBoard();
        return newPiece;
    }
}
```

・NextMove クラス

```
package chess;
```

```
/**
 * 駒の移動可能な位置を表すクラス
 */
public class NextMove {
	final static int KING     = 1;                                    // キング(先手)
	final static int PAWN     = 2;                                          // ポーン
	final static int KING2    = -1;                                   // キング(後手)
	final static boolean isChessStyleScore = true;   // 棋譜表記チェス式
	int type;                                                  // 駒の種類
	int nextX;                                                 // 移動先のX座標
	int nextY;                                                 // 移動先のY座標
	int value;                                                 // 移動した場合の盤面の評価値

	/**
	 * コンストラクタ
	 * @param int type  駒の種類
	 * @param int nextX  移動先のX座標
	 * @param int nextY  移動先のY座標
	 */
	public NextMove(int type, int nextX, int nextY) {
		this.type  = type;
		this.nextX = nextX;
		this.nextY = nextY;
	}

	/**
	 * 駒の種類を返す
	 * @return 駒の種類
	 */
	public int type() {
		return type;
	}

	/**
	 * 移動先のX座標を返す
	 * @return X座標
	 */
	public int nextX() {
		return nextX;
	}

	/**
	 * 移動先のY座標を返す
	 * @return Y座標
	 */
	public int nextY() {
		return nextY;
```

```
}

/**
 * 移動した場合の盤面の評価値を返す
 * @return 評価値
 */
public int value() {
        return value;
}

/**
 * 評価値をセットする
 * @param value 評価値
 */
public void setValue(int value) {
        this.value = value;
}

/**
 * 棋譜を返す
 * 変数isChessStyleによりチェス風の棋譜あるいは将棋風の棋譜か変化する
 * @return 棋譜の文字列表現
 */
public String toString() {
        String score; // 棋譜
        if (isChessStyleScore) { // チェス風の棋譜を作成
                switch (type) {
                case PAWN:      score = "P"; break;
                case KING:      score = "K"; break;
                case KING2:     score = "K2"; break;
                default:        score = "?"; break;
                }

                switch (nextX) {
                case 1:   score += "a"; break;
                case 2:   score += "b"; break;
                case 3:   score += "c"; break;
                case 4:   score += "d"; break;
                case 5:   score += "e"; break;
                case 6:   score += "f"; break;
                case 7:   score += "g"; break;
                case 8:   score += "h"; break;
                default: score += "?"; break;
                }

                switch (nextY) {
                case 1:   score += "8 "; break;
```

```
                    case 2:   score += "7 "; break;
                    case 3:   score += "6 "; break;
                    case 4:   score += "5 "; break;
                    case 5:   score += "4 "; break;
                    case 6:   score += "3 "; break;
                    case 7:   score += "2 "; break;
                    case 8:   score += "1 "; break;
                    default: score += "? " ; break;
                    }
                }
                return score;
        }
}
```