卒業研究報告書

題目

「ハルマ」アプリの開発

指導教員

石水 隆　講師

報告者

10-1-037-0022

段野　健太

近畿大学理工学部情報学科

平成 26 年 1 月 31 日提出

概要

本研究では、1883 年ないし 1884 年に、バーバード・メディカルスクールの形成外科医ジョージ・ハワード・モンク(George Howard Monks)によって発明された、「ハルマ」[2]と呼ばれる 2 人または 4 人用のボードゲームをプレイできるプログラムを製作した。ハルマとはギリシャ語で「跳ぶ」という意味である。

本研究テーマのハルマは二人零和有限確定完全情報ゲームなので可能な局面全ての最善手をデータベースに持てば無敵の AI を作成できる。しかしハルマの可能な局面数は極めて大きいため、現在の計算機では完全解析は不可能である。そこで、本研究では各局面に対して評価値を設定し、数手先を読んで高い評価を選択する AI を作成することを目標とする。

目次

1　序論

1.1 本研究の背景

囲碁、将棋やチェス、チェッカー等に代表されるボードゲームは、二人零和有限確定完全情報ゲームに分類される。零和とは、ゲーム上プレイしているプレイヤーの利得が常に零、または個々のプレイヤーの指す手の組み合わせに対する利得が常に一定の数値となることである。利得とは、プレイヤーがゲーム終了時、またはターン終了時に獲得する状況に対する評価である。有限とは、そのゲームにおける各プレイヤーの可能な手の組み合わせの総数が有限であることである。確定とは、プレイヤーの着手以外にゲームに影響を与える偶然が全く入り得ないという意味である。完全情報ゲームとは、各プレイヤーが自分の手番において、これまで各プレイヤーが取った選択、または意思決定についての全ての情報を完全に知ることができるゲームのことである。二人零和有限完全情報ゲームは、その性質上、解析を行いやすいため、ゲーム理論において様々な研究がなされてきた。また、人口知能の分野においても広く研究がなされている

1.2 二人零和有限確定完全情報ゲームの完全解析に関する既知の結果

二人零和有限確定完全情報ゲームは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で決定しており、理論上、全ての可能な局面を解析することができれば最善の手を打つことができる。しかし多くのボードゲームでは、可能な局面の総数が極めて多いため、完全解析を行うことは不可能である。例を挙げれば、リバーシが 10^{28} 通り、チェスが 10^{50} 通り、将棋が 10^{69} 通り、囲碁が 10^{170} 通り程度あるとされている。これは現在の計算機の性能を超えてしまっている。

一方、可能な局面が少ないゲームでは完全解析されているものもある。連珠は双方最善手を打った場合 47 手で先手が勝つ[3]。チェッカーは双方最善手を指すと引き分けとなる[4]。その他にも「シンペイ」と呼ばれるバンダイが発売したゲームは、勝ちに要する最長手数が 49 手で後手が勝つことがわかっている[5]など、多数のゲームで完全解析が行われている。

また、局面数が大きいゲームについては、ゲーム盤を小さいサイズに限定した場合の解析も行われている。本研究でも本来 16 路盤を用いるハルマを 8 路盤に限定して研究を行っている。完全解析されて例として、6 路盤のリバーシでは双方最善手を打つと 16 対 20 で後手勝ちとなる[6]。将棋では、盤のサイズが 3×4 で駒の種類が 4 つの「どうぶつしょうぎ」[7]が双方最善手を指すと 78 手で後手勝ちとなることが判明している[8]。

1.3 完全解析されていない二人零和有限確定完全情報ゲームの完全解析に対する手法

可能な局面数が多いゲームに対して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確実な最適手を得ることはできないが、局面の評価値計算、定石データベース、一定手数の先読み、終盤での必勝読みと完全読み、モンテカルロ法などを用いて、より有利だと思われる手を選択することができる。囲碁ではモンテカルロ法を用いた AI が成果を出している。モンテカルロ法とは、終盤までシミュレーションをして勝率の最も高い手を選択する方法である。より差を広げて勝つことよりも、いかに負けないよう、無難に勝てるかという考え方である。これらの方法を用いることにより、完全解析を行わずにある程度の強さのプログラムを作ることが可能であり、ゲームによってはプロに勝つこともできる。

チェスでは、1997 年 5 月にチェスプログラム Deep Blue[9]が世界チャンピオン Garry Kimovich Kasparov と対戦を行い 2 勝 1 敗 3 引き分けで勝利した[10]。将棋では 2012 年 1 月に将棋プログラムボンクラ―ズ[11]が元プロ棋士の米長邦雄永世棋聖と対戦しボンクラ―ズ先手 113 手で勝利した[12]。また、2013 年 3 月から 4 月にかけて行われた、第 2 回将棋電王戦[13]では全 5 局の内、第 2 局で先手「ponanza」が後手佐藤慎一四段に 141 手で勝利[14]。第 3 局で後手「ツツカナ」が先手船

江恒平五段に 184 手で勝利[14]。第 5 局で後手「GPS 将棋」が先手三浦弘行八段に 102 手で勝利し、結果全 5 局で 3 勝 1 敗 1 引き分け[14]と将棋プログラムがプロ棋士に勝利したことは一般社会にも衝撃を与えた。

1.4 本研究の目的

本研究テーマであるハルマは、世界中で発売されているダイヤモンドゲーム(チャイニーズ・チェッカー)の原型であるにも関わらず、ダイヤモンドゲームに比べても知名度も無く、研究、解析もされていない。そこで、本研究ではハルマ解析の先駆けとして AI の作成を目指した。

ハルマは可能な局面数が約10^{57}通りと極めて大きい為､深い先読みすることは膨大な時間がかかり、また完全解析は現在の計算機の性能では不可能である。そこで、現在の盤面に対して様々な要素から評価値を設定し、高い評価を選択する AI を作成した。

1.5 本報告書の構成

本報告書の第 2 節ではまずハルマとは何なのか、ルールを説明する。第 3 章では作成したアプリケーションの内容について説明する。第 4 章では実験結果について述べる。第 5 章では今回の研究の結論をまとめ、今後の課題を挙げる。

2　ハルマについて

本章ではハルマについて説明する。

2.1 ハルマとは

ハルマとはヴィクトリア朝後期(19 世紀末)のイギリスで大流行したボードゲームである。日本でも発売されているダイヤモンドゲームの原型である。自分の陣地の駒を全て相手の陣地へ入れる事がゲームの目的の競争ゲームである。

2.2 ハルマのルール

ハルマの基本的なルールは以下の通りである。

- 16 路盤を使用し、それぞれ 19 個の駒を持つ。図 1 にハルマの初期配置図を示す。
- 各プレイヤーは自分の手番で自分の駒を 1 つ移動させることができる。移動する際は、各プレイヤーは隣接移動またはジャンプ移動のどちらかを行える。各移動ついてのルールは以下の通りである。
  - ある自駒に隣接する 8 マスの中で空いているマスがあれば、その駒は隣接する任意の空きマスに 1 マス移動可能である。図 3 に各駒の移動可能マスを示す。
  - ある自駒の隣接するマスに(自分、相手は問わない)駒があり、更にその先に空いているマスがあれば、その駒を飛び越えて移動することも可能である。また、駒を跳び越えた後、更に跳び越えることができる場合は連続して跳ぶことも可能である。その際、方向は変更しても良い。図 4，図 5 に駒を飛び越えて移動する例を示す。ただし、跳び越える際は必ず 1 つの駒のみで、2 つ以上連なっている場合は跳びこせない。図 6 に飛び越えることができない場合の例を示す。なお、作戦的に途中でジャンプを辞めることも可能である。必ずジャンプしなければならないというルールは無い。また、相手駒を飛び越えたとしても、チェッカーとは違い、駒を取り除くことはしない。
- 先に全ての駒を相手の陣地に移動させれば勝利である。
- 一度相手の陣地に辿り着いた駒を陣地の外に出したままターン終了してはならない。

- 他のバリエーションとして、8 路盤、10 路盤を使用することもある。その際、それぞれプレイヤーは各 10 駒、15 駒を用いてプレイする。
  - 4 人対戦も可能であり、各々13 駒を用いてプレイする。図 2 に 4 人対戦の場合の初期配置図を示す。4 人対戦の場合、ターン終了時に左隣りのプレイヤーにターンが移る。

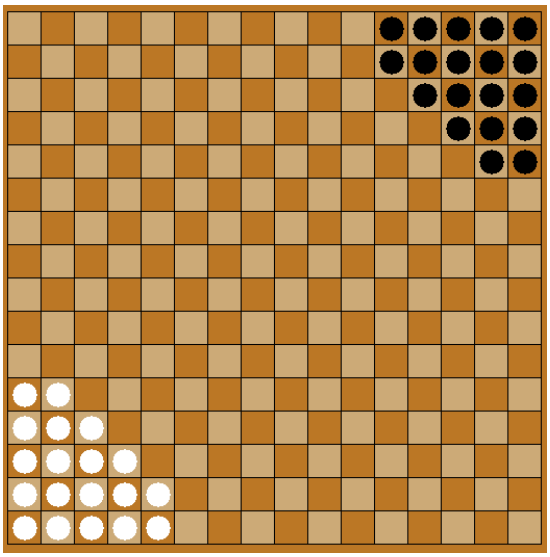

図 1　初期配置

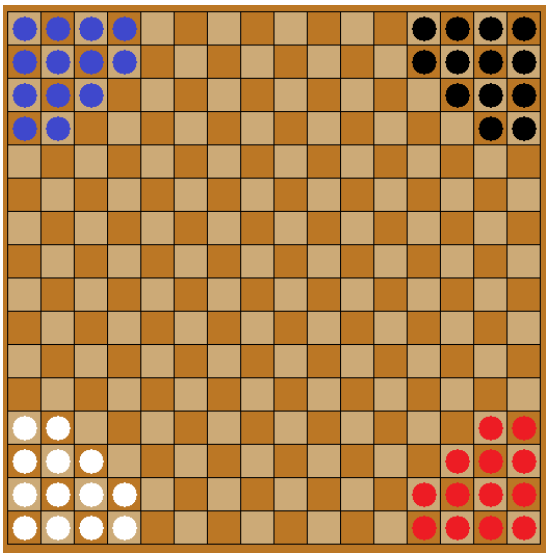

図 2　初期配置(4 人対戦時)

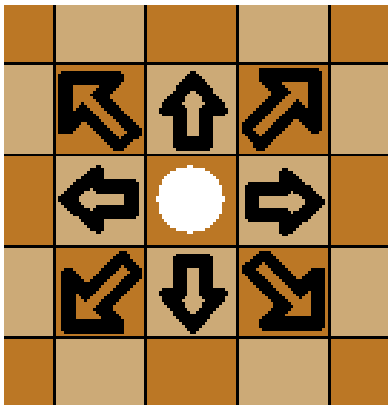

図 3　移動方向

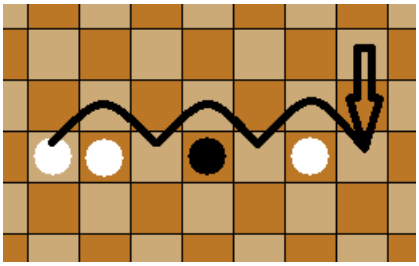

図 4　ジャンプ

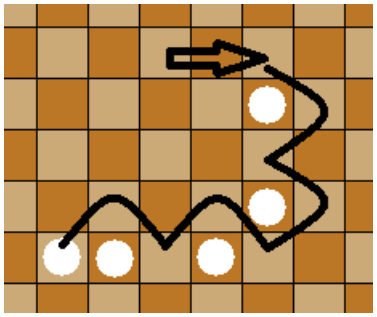

図 5　ジャンプ(方向転換)

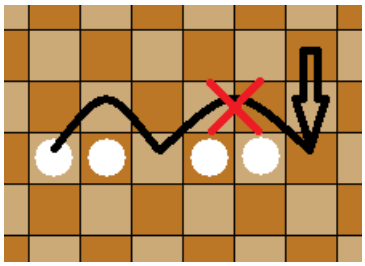

図 6　ジャンプ不可

3　研究内容

本章では本研究で作成したハルマアプリケーションについて説明する。

3.1 ハルマアプリケーションで用いた手法

1.3 節で述べたように、次に指すべき手をどのように選択するかは様々な手法がある。本節ではハルマアプリケーションで用いた手法について説明する。本研究では 1.3 節で述べた局面の評価値計算と一定手数の先読みを行う AI を作成した。本研究で作成した AI は、着手可能手から得られる局面を評価し、評価値が最大となる手を指す AI である。本研究では、以下に示す異なる 4 つの評価基準にそれぞれ従う 4 つの AI を作成する。

A)　陣地の対角線上（白駒ならば右、もしくは上方向）に高い評価値を与えなるべく最短距離を進むように設定する。遠回りして（白駒ならば左、もしくは下方向に）迂回するような経路には低い評価値を与えた。

B)　なるべく二つの駒が離れないようにする。ハルマは一つの駒だけで動くよりも二つの駒が馬跳びをするかの様な動かし方の方が移動距離が長いためである。なるべく駒同士を接近させることで、隣接する駒が空きマスで、一マスしか動けない状況を避ける。また、二つ駒が並んでいると相手プレイヤーに跳びこされることが無くなるので、相手の行動を制限する上でも有利になる。

C)　移動可能な駒の中から最も移動距離が長い駒を選択し動かす。全ての駒が相手陣地まで進まないと勝てないので、単純に移動距離の長いものから前進させる。

D)　移動可能な駒の中から最もゴール地点までの距離が遠い駒を選択し前進させる。

以上の四種類の評価値の与え方により、どの戦略を重視すれば強い AI になるのかが分かる。

3.2 ハルマアプリケーションプログラム

ハルマのアプリケーションを作成するために本研究では Java 言語を用いた。付録に本研究で作成した Java プログラムのソースを示す。アプリケーションは本研究では以下の 3 つのクラスから成る。

3.2.1　Halma クラス

このクラスでは、盤面の大きさ、駒の数、着手可能な手の表示などの設定を行う。

与えられた大きさの盤面にそれぞれの駒をセットし、初期盤面をまず表示する。それから、現在の手番は白番か黒番かどちらなのか判定する。そして、その盤面で現在着手可能手を探し出してリストへ全てセットする。設定された深さだけ先読みを行って、その結果を得る。得られた結果通りに駒を打って、その後の盤面の状態を再度表示する。ここでゴール判定を行い、ゴールしていると見なされれば、その時点でアプリケーションは終了する。ゴールしていないと見なされれば相手番に移り、再度同じ処理を繰り返す。

3.2.2　Board クラス

このクラスでは、主にゲーム盤面を表す役割を果たす。

- Board(int size)
  Halma クラスで設定した大きさの盤面を生成する。先手は白に設定する。

- void initBoard(int stoneNum)
  Halma クラスで設定した数の駒を決められた初期位置に配置する。

- void show()
  盤面を表示するメソッドである。

- void set(int x, int y, int type)
  盤面の x 座標、y 座標を type 別に設定する。set メソッドによって配置可能な type を表 1 に示す。

表 1 盤面に配置される要素

type	盤面に配置される要素
WHITE	白駒
BLACK	黒駒
EMPTY	空きマス
SELECTED	選択した駒
REASHABLE	選択した駒が移動できるマス
BORDER	盤外

- ArrayList<NextMove> createMovableList(Boolean isWhite)
  現在の盤面で着手可能な手を List に格納するメソッドである。白手番の場合で説明すると、盤面の白駒がある座標を探して、その座標を movableBoard に代入して得られた盤面を表示させる。そして、得られた盤面で到達可能マスを探して座標を得る。こうして得られた、現在の x 座標と y 座標、移動後の x 座標と y 座標、どちらの手番か true もしくは false を引数にオブジェクトを生成する。NextMove クラスで評価値を付けられた後に、movableList という名のリストへ格納してゆく。

- void addMovableList(ArrayList<NextMove> movableList, NextMove nextMove)
  着手可能手リスト movableList に評価値の高い順に並ぶように着手可能手を加えるメソッドである。NextMove クラスで付けられた評価値を比べてゆく。

- void showMovableList(ArrayList<NextMove> movableList)
  movableList、着手可能手を表示させるメソッドである。

- Board clone()
  盤面のコピーを生成するメソッドである。

- Board nextBoard(NextMove nextMove)
  指定した着手を指した後の盤面を生成するメソッドである。現在駒が存在する x 座標 y 座標を空きマスにして、移動後の x 座標 y 座標に白駒、もしくは黒駒をセットする。

- Board movableBoard(int x, int y)
  まず引数で得られた座標上に駒が存在するか確認する。存在すればコピー盤面を生成し、選択した座標の type を SELECTED にセットする。そして、その座標の周囲 8 マスを調査して EMPTY であるならば、その座標の type を REACHABLE にセットする。また、jampableBoard を用いてジャンプできるか判定を行った後、ここで生成した盤面を返す。指定した座標に駒が存在しない場合はこのメソッドを終了し、呼び出し元へ処理を戻す。

- Board jampableBoard(int x, int y, Board jBoard)
  指定した位置からジャンプで移動できる位置を記入された盤を返すメソッドである。ジャンプした先が盤面外でないことを確認し、現在の x 座標 y 座標の 8 方向を調べてどちらかの駒が存在すれば、更にその一マス先が空きマスであるかどうかを判定する。空いていればジャンプ先の座標を移動可能マスとしてセットする。そこから更にジャンプできるかを再帰的に行う。

- boolean isGoal(boolean isWhite)
  引数に現在どちらの手番かを受け取り、ゴール地点に全ての駒が存在しているかを判定する。存在していれば true を返し、プログラムは終了する。存在していなければ false を返し続行する。

- int value(int depth)
  先読みを行いそこで得られた局面の評価値を返すメソッドである。引数に何手分先読みを行うかを受け取り、depth==0 であればその時点での結果を返す。1 以上であれば、着手可能手リストに含まれる各手につき 1 手先の局面を生成し、各局面に対して depth-1 として value()メソッド自身を再帰的呼び出して評価値を計算し、最も高い評価値を持つ局面の評価値を返す。

3.2.3 NextMove クラス

このクラスでは、駒に評価値を与え、移動可能な位置を表している。

- NextMove(int curX, int curY, int nextX, int nextY, boolean isWhite, Board BOARD)
  NexeMove クラスのコンストラクタである。移動元座標、移動先座標、駒の色、局面に関する情報をセットし、その移動の評価値を設定する。ここで評価値の与え方を調整し、それぞれの戦略にふさわしい評価値を与える。

- String toString()
  着手可能手の文字列表現をするメソッドである。“(現在の x 座標, 現在の y 座標) -> (移動後の x 座標, 移動後の y 座標)：評価値”というように表示する。

- int value()
  着手の評価値を返すメソッドである。

4　結果・考察

今回の実験は 8×8 の簡易ハルマで行う。

3.1 節で述べた 4 つの戦略の有効性を評価するため、各戦略に従う AI 同士で総当たり対戦を行った。表 1 に対戦結果を示す。表 2 より、戦略 C が最も勝率が高い。

表 2　AI 同士の対戦結果(試行回数 100 回)

先手＼後手	戦略 A	戦略 B	戦略 C	戦略 D	計
戦略 A	57＼43	38＼62	41＼59	69＼31	205＼195
戦略 B	43＼57	41＼59	47＼53	61＼39	192＼208
戦略 C	55＼45	52＼48	58＼42	64＼36	229＼171
戦略 D	24＼76	24＼76	32＼68	53＼47	133＼267
計	179＼221	155＼245	178＼222	247＼153	759＼841

これより、第一に多く移動することを目標とした方が良いこと、他の戦略はそれ単独では効果が低いことが分かる。だが、他の戦略と組み合わせて評価値の条件付けには有効であると考えられる。そこで、それぞれの長所を組み合わせた新 AI を新たに作成した。この AI の評価関数は以下の式で表される。

評価値 ＝ 移動距離＋評価値マップによる評価＋条件 i , ii , iii により評価値を増加

i. (8 － 現在の x 座標＞現在の y 座標)の場合、右方向に進めば評価値を 1 増加させる。もしくは、(8 － 現在の x 座標＜現在の y 座標)の場合、上方向に進めば評価値を 1 増加させる。
ii. 移動先の進行方向に黒駒かつ、その対角線上に白駒が存在すれば、それぞれ評価値を 1 増加させる。
iii. ゴール外からゴールする場合のみ、評価値を 3 増加させる。

表 3　評価値マップ(左側：移動先、右側：移動元)

-1	-1	-1	0	6	6	7	8
-1	-1	0	0	0	5	6	7
-1	0	0	0	0	0	5	6
0	0	0	0	0	0	0	6
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	-1
0	0	0	0	0	0	-1	-1
0	0	0	0	0	-1	-1	-1

10	10	10	10	4	3	2	1
9	8	7	6	5	4	3	2
10	9	8	7	6	5	4	3
11	10	9	8	7	6	5	4
12	11	10	9	8	7	6	10
13	12	11	10	9	8	7	10
14	13	12	11	10	9	8	10
15	14	13	12	11	10	9	10

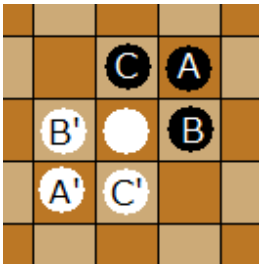

図 8　条件 ii 判定

上記の評価関数の与え方は白駒の場合である。4 つの戦略で総当たりした結果、最も重要だとされる移動距離はそのままに各戦略を調節したものを付け加えた。表 3 で示した評価値マップと条件 i で右、もしくは上方向のみに偏って移動することの無い様にし、戦略 A、D を満たすよう設定した。条件 ii で戦略 B を満たすように設定した。図 8 に対応する座標を示す。A に黒駒が存在して、かつ A'に白駒が存在していれば評価値を 1 増加させる。B、C についても同様である。2 つ駒を連ねることで相手のジャンプを防ぎ、迂回させることで相手の手を遅らせることが目的である。これは戦略 B の相手の行動を制限することを満たし、かつ相手が移動すれば後ろの駒から動かしていくことで距離も稼ぐことができる。条件 iii に関しては、ゴールした後にゴール内で動き回ったり、盤面端の駒がゴールへと向かわなかったりすることを防ぐために付け加えた。ゴールできる駒があれば優先してゴールするように設定した。

上記の通り新たに作成した AI と上記の 4 つの戦略を双方着手可能な手からランダムに選択して打つよう設定し、試行回数は 100 回で対戦を行った。結果を表 4 に示す。

表 4　新戦略 AI と旧戦略 AI の対戦結果(試行回数 100 回)

	戦略 A	戦略 B	戦略 C	戦略 D
新 AI	69-31	85-15	74-26	77-23

新たに作成した AI が 4 つ全ての戦略に対して勝ち越すことができた。ランダムに選択した手であるため、多少結果に違いが出るかもしれないが、3.1 節で述べた戦略よりも強い AI が出来上がったと言える。

上記の表 4 の結果では着手可能手からランダムに選択させたため、7 割程の勝率となった。そこで更に勝率を上げるために新 AI を先読みさせて対戦を行う。まず、その際に必要となるおおよその所要時間を計測する。できる限り深く先読みを行った方が良いが、その分処理に時間がかかる。先読み手数 2～6 手で先読みを行った際の一手目にかかる所要時間を表 5 に記す。

表 5　先読みを行った場合の一手目の所要時間

先読み手数	2	3	4	5	6
所要時間(ms)	16	135	2983	112513	4613077

表 5 より 5 手以上読んだ場合、一気に処理に必要な時間が増加する。4 手までなら 3 秒前後で処理が終わるため、対人戦に用いても AI の思考時間に待たされるストレスは少ないように思える。ただ、5 手でも 2 分弱、6 手先読みを行おうとすると 70 分以上待たされることになるので、処理を並列化するなりして改善を行わないと使えない。

これにより先読みを行う際のおおよその所要時間が分かったので、今度は試行回数を 100 回、新 AI を 2 手先読みを行わせて上記の 4 つの戦略を対戦させたところ、全てに勝利した。これは、それぞれの 4 つ分の戦略が合わさった新戦略と 1 つだけの戦略が戦った結果であり、当然の結果と言える。つまり、これだけでは新 AI がいかに強くなったのか十分な証明にはなっておらず、他の完成された AI と対戦させて初めて本研究で作成した AI の実力が分かる。例えば、現在モンテカルロ法は強い囲碁 AI を作成するには有望だとされている。そのモンテカルロ法を用いた AI と対戦してみれば、本研究で作成した AI の実力はどの程度のものか分かると考えられる。

5　結論・今後の課題

本研究では、ハルマアプリケーションプログラムを作成した。本研究で作成したプログラムは、ハルマにおける様々な戦略を組み合わせ、先読みで得られた局面の評価値を用いて打つ手を決定する。評価には盤面の座標と、自分と相手のそれぞれの駒の位置関係を用いた。ハルマは基本的には移動距離の多いものから選択する方が勝ちに繋がりやすく、なるべくジャンプするか対角線上の斜め方向へ移動させる方が良い。そこに相手の動きを制限させるための動きなど組み合わせた結果、ひたすら距離を稼ぐ行動よりも勝率は高くなった。しかし、対戦時に深く先読みを行うと深さに比例して処理時間が非常に伸びてしまい、対戦として成り立たなくなってしまった。そこで今後の課題として、処理を並列化させるなどの処理時間の改善、戦略そのものの強化、次ターン以降動き易くするために布石を打つのか、相手の動きを制限するのか、やはり距離を稼ぐことを最優先とするのかといった状況に応じた戦略の変更、4 人対戦に対応することが挙げられる。

謝辞

本研究書を作成するにあたり、私を担当して下さった石水隆先生には、多大なご迷惑をおかけしたことのお詫びと、最後までご指導して頂きました感謝しきれないほどの気持ちをこの場を借りて心より申し上げます。本当にありがとうございました。

参考文献

[1] 松田道弘：世界のゲーム辞典 pp204,205, 東京堂出版(1989)

[2] Halma - アブストラクトゲーム博物館, http://www.nakajim.net/index.php?Halma

[3] 美添一樹, 山下宏, 松原仁：コンピュータ囲碁―モンテカルロ法の理論と実践―,共立出版(2012)

[4] Jonathan Schaeffer, Neil Burch, Yngvi Bjorsson, Akihiro Kishimoto, Martin Muller, Robert Lake, Paul Lu, and Steve Suphen, Checkers is Science Vol.317, No.5844, pp1518-1522 (2007) http://www.sciencemag.org/content/317/5844/1518.full.pdf

[5] 田中哲郎,ボードゲーム「シンペイ」完全解析(修正版), http://media.itc.u-tokyo.ac.jp/ktanaka/simpei/20060307-rev.pdf

[6] Joel Feinstein, Amenor Wins World 6×6 Championships!, Forty billion noted under the tree(July 1993), pp6-8, British Fedrations newsletter,(1993), http://britishothello.org.uk/fbnall.pdf

[7] 北尾まどか, 藤田麻衣子, どうぶつしょうぎねっと, (2010), http://dobutsushogi.net/

[8] 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-GI-22 No.3, pp1-8(2009), http://id.nii.ac.jp/1001/00062415/

[9] IBM100-DeepBlue,IBM,(1997),http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue

[10] Michael Khodarkovsky and Leonid Shamkvoich, 人間対機械―チェス世界チャンピオンとスーパーコンピューターの闘いの記録, 毎日コミュニケーションズ,(1998)

[11] Puellaαソース公開, A級リーグ指し手1号,(2013) http://aleag.cocolog-nifty.com/blog/2013/08/puella-620e.html

[12] コンピュータ将棋vs米長邦雄永世棋聖の対局が決着,富士通,(2013), http://pr.fujitsu.com/jp/news/2012/01/16-1.html

[13] 第2回将棋　電王戦　HUMAN　VS　COMPUTER,　niconico, (2013), http://ex.nicovideo.jp/denousen2013/index.html

[14] 対戦結果―第2回将棋　電王戦　HUMAN VS COMPUTER, niconico, (2013) http://ex.nicovideo.jp/denousen2013/result.html

[15] 山本一成のHP, http://www.graco.c.u-tokyo.ac.jp/~issei/

[16] WCSC21 ツツカナ アピール文書, (2011), http://www.computer-shogi.org/wcsc21/appeal/tsutsukana/WCSC21_tsutsukana_20110327.pdf

[17] GPSshogi, (2013), http://gps.tanaka.ecc.u-tokyo.ac.jp/gpsshogi/

[18] Seal Software: リバーシのアルゴリズム, 工学社(2007)

付録

Halma.java

```
package Halma;

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

public class Halma {
        static final boolean isShowMovableBoard = false; // true にすると選択した石の移動
                                                          可能な位置が表示される
        static final boolean isShowMovableList = false;  // true にすると移動可能な位置の
                                                          リストが表示される
        Board board; // 盤
        static final int boardSize = 8;  // 盤の大きさ
        static final int stoneNum = 10;  // 配置する石の数 (10,13,19 から選択)
        static final int maxMove = 1000; // 最大手数 (これを越えると引き分けと見做す)
        static int maxDepth = 2; // 先読みする手数の上限;

        public static void main (String[] args) {
                Board board = new Board (boardSize); // 盤
                Random rnd = new Random(); // 乱数発生用
                int rndVal; // 乱数用

                board.initBoard (stoneNum);          // 石を初期配置
                board.show();

                ArrayList<NextMove> movableList;
                NextMove nextMove;

                for (int i=1; i<maxMove; ++i) {
                        // 白番
                        movableList = board.createMovableList (true);
                        if (isShowMovableList) board.showMovableList (movableList);
                        Val = board.value(maxDepth);

//                      rndVal = rnd.nextInt ( movableList.size() / 10);
                         // リストの前の方にある手(評価値の高い手)から選択

                        nextMove = movableList.get(Val);
                        board = board.nextBoard (nextMove);

                        board.show();
                        if (board.isGoal (true)) {
                                System.out.println (i+"手目白勝利");
                                break;
                        }
```

```
                        // 黒番
                        movableList = board.createMovableList (false);

                        rndVal = rnd.nextInt (movableList.size() / 10);
                        // リストの前の方にある手(評価値の高い手)から選択

                        nextMove = movableList.get(rndVal);
                        board = board.nextBoard (nextMove);
                        if (isShowMovableList) board.showMovableList (movableList);
                        board.show();
                        if (board.isGoal (false)) {
                                System.out.println (i+"手目黒勝利");
                                break;
                        }
                }
        }
}
```

Board.java

```
package Halma;

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

/**
 * ゲーム盤を表すクラス
 */
public class Board {
        int[][] board; // 盤
        boolean isWhite;
        static int size;      // 盤サイズ
        static int stoneNum;  // 配置する石の数
        static final int WHITE = 1;     // 白石
        static final int BLACK = -1;    // 黒石
        static final int EMPTY = 0;     // 空マス
        static final int SELECTED = 2; // 選択石
        static final int REACHABLE = 3; // 到達可能マス
        static final int BORDER = Integer.MAX_VALUE; // 盤端
        static int value;

        public Board (int size) {
                board = new int[size+2][size+2];
                this.size = size;
```

```
		for (int j=0; j<size+2; ++j)
			board[0][j] = BORDER;
		for (int i=1; i<size+1; ++i) {
			board[i][0] = BORDER;
			for (int j=1; j<size+1; ++j)
				board[i][j] = EMPTY;
			board[i][size+1] = BORDER;
		}
		for (int j=0; j<size+2; ++j)
			board[size+1][j] = BORDER;
		isWhite = true;
	}

	/**
	 * 初期配置盤面を作成する
	 */
	public void initBoard(int stoneNum) {
		this.stoneNum = stoneNum;
		board[1][size]   = BLACK;
		board[1][size-1] = BLACK;
		board[1][size-2] = BLACK;
		board[1][size-3] = BLACK;
		board[2][size]   = BLACK;
		board[2][size-1] = BLACK;
		board[2][size-2] = BLACK;
		board[3][size]   = BLACK;
		board[3][size-1] = BLACK;
		board[4][size]   = BLACK;
		board[size]  [1] = WHITE;
		board[size]  [2] = WHITE;
		board[size]  [3] = WHITE;
		board[size]  [4] = WHITE;
		board[size-1][1] = WHITE;
		board[size-1][2] = WHITE;
		board[size-1][3] = WHITE;
		board[size-2][1] = WHITE;
		board[size-2][2] = WHITE;
		board[size-3][1] = WHITE;
		if (stoneNum >= 13) { // 13 個以上石を置く場合に追加配置
			board[2][size-3] = BLACK;
			board[3][size-2] = BLACK;
			board[4][size-1] = BLACK;
			board[size-1][4] = WHITE;
			board[size-2][3] = WHITE;
			board[size-3][2] = WHITE;
		}
		if (stoneNum >= 19) { // 19 個以上石を置く場合に追加配置
			board[1][size-4] = BLACK;
```

```
                    board[2][size-4] = BLACK;
                    board[3][size-3] = BLACK;
                    board[4][size-2] = BLACK;
                    board[5][size-1] = BLACK;
                    board[5][size]   = BLACK;
                    board[size]  [5] = WHITE;
                    board[size-1][5] = WHITE;
                    board[size-2][4] = WHITE;
                    board[size-3][3] = WHITE;
                    board[size-4][2] = WHITE;
                    board[size-4][1] = WHITE;
          }
}

/**
 * 盤面表示
 */
public void show() {
          for (int j=0; j<size+2; ++j)
                    System.out.print ("■");
          System.out.println();
          for (int i=1; i<size+1; ++i) {
                    System.out.print ("■");
                    for (int j=1; j<size+1; ++j) {
                              switch (board[i][j]) {
                              case WHITE :     System.out.print ("○"); break; // 白石
                              case BLACK :     System.out.print ("●"); break; // 黒石
                              case SELECTED :  System.out.print ("◎"); break; // 選択石
                              case REACHABLE : System.out.print ("☆"); break; // 移動可能マス
                              default :        System.out.print ("  "); break; // 空マス
                              }
                    }
                    System.out.print ("■");
                    System.out.println();
          }
          for (int j=0; j<size+2; ++j)
                    System.out.print ("■");
          System.out.println();
}

/**
 * 指定した座標に石を配置する
 * @param int x : 石のｘ座標
 * @param int y : 石のｙ座標
 * @param int type : 石の種別
 */
public void set (int x, int y, int type) {
          board[x][y] = type;
```

```
	}

	public ArrayList<NextMove> createMovableList (boolean isWhite) {
		ArrayList movableList = new ArrayList<NextMove>(); // 着手可能手のリスト
		NextMove nMove; // 着手可能手
		Board mBoard;  // 移動可能位置記入用盤

		if (isWhite) {
			for (int i=1; i<size+2; ++i) {
				for (int j=1; j<size+2; ++j) {
					if (board[i][j] == WHITE) {

					// (i, j)から移動可能な位置を記入した盤を得る
						mBoard = movableBoard (i, j);

					// (i, j)から移動可能な位置を表示
					if (Halma.isShowMovableBoard) mBoard.show();
					for (int u=1; u<size+2; ++u) {
						for (int v=1; v<size+2; ++v) {

							// (i, j)から(u, v)へ移動可能
							if (mBoard.board[u][v] == REACHABLE) {
							nMove = new NextMove (i, j, u, v, true, mBoard);
								addMovableList (movableList, nMove);
									}
							}
						}
					}
				}
			}
		} else {
			for (int i=1; i<size+2; ++i) {
				for (int j=1; j<size+2; ++j) {
					if (board[i][j] == BLACK) {

							// (i, j)から移動可能な位置を記入した盤を得る
							mBoard = movableBoard (i, j);

							// (i, j)から移動可能な位置を表示
							if (Halma.isShowMovableBoard) mBoard.show();

							for (int u=1; u<size+2; ++u) {
								for (int v=1; v<size+2; ++v) {
									if    (mBoard.board[u][v]    ==
REACHABLE) { // (i, j)から(u, v)へ移動可能
										nMove = new NextMove
(i, j, u, v, false, mBoard);
										addMovableList
```

```
(movableList, nMove);
                                                                                    }
                                                                                }
                                                                            }
                                                                        }
                                                                    }
                                                                }
                                                            }
                                                            return movableList;
                                                    }

                                                    /**
                                                     * 着手可能手リストに評価値の高い順に並ぶように着手可能手を加える
                                                     * @param ArrayList movableList: 着手可能手のリスト
                                                     * @param NextMove nextMove : 着手可能手
                                                     */
                                                    private void addMovableList (ArrayList<NextMove> movableList, NextMove nextMove) {
                                                            for (int i=0; i<movableList.size(); ++i) {
                                                                    NextMove e = movableList.get(i);
                                                                    if (nextMove.value() > e.value()) { // 評価値の比較
                                                                            movableList.add (i, nextMove);  // リストの比較した位置への挿入
                                                                            return;
                                                                    }
                                                            }
                                                            movableList.add (nextMove); // リストの末尾への挿入
                                                    }

                                                    /**
                                                     * 着手可能手の表示
                                                     */
                                                    public void showMovableList (ArrayList<NextMove> movableList) {
                                                            for (NextMove nextMove : movableList) {
                                                                    System.out.println (nextMove);
                                                            }
                                                    }

                                                    /**
                                                     * 盤のコピー生成
                                                     */
                                                    public Board clone() {
                                                            Board cloneBoard = new Board (size);
                                                            for (int i=0; i<size+2; ++i)
                                                                    for (int j=0; j<size+2; ++j)
                                                                            cloneBoard.board[i][j] = board[i][j];
                                                            cloneBoard.size = size;
                                                            return cloneBoard;
                                                    }
```

```
/**
 * 指定した着手を指した後の盤を生成
 */
public Board nextBoard (NextMove nextMove) {
          Board nextBoard = clone();
          nextBoard.isWhite = !isWhite; // 手番入れ替え
          nextBoard.set (nextMove.curX, nextMove.curY, EMPTY);
          if (nextMove.isWhite) nextBoard.set (nextMove.nextX, nextMove.nextY, WHITE);
          else nextBoard.set (nextMove.nextX, nextMove.nextY, BLACK);
          return nextBoard;
}


/**
 * 指定した石から移動可能な位置を記入された盤を返す
 * @param int x : 石のx座標
 * @param int y : 石のy座標
 * @return Board : 指定した石が移動可能な位置が記入された盤
 */
public Board movableBoard (int x, int y) {
          if (board[x][y] != BLACK && board[x][y] != WHITE) {
                    System.out.println ("指定の位置に石はありません");
                    return this;
          }
          Board mBoard = clone();
          mBoard.set (x, y, SELECTED);
          if (board[x-1][y-1] == EMPTY) mBoard.set (x-1, y-1, REACHABLE);
          if (board[x-1][y]   == EMPTY) mBoard.set (x-1, y,   REACHABLE);
          if (board[x-1][y+1] == EMPTY) mBoard.set (x-1, y+1, REACHABLE);
          if (board[x]  [y-1] == EMPTY) mBoard.set (x,   y-1, REACHABLE);
          if (board[x]  [y+1] == EMPTY) mBoard.set (x,   y+1, REACHABLE);
          if (board[x+1][y-1] == EMPTY) mBoard.set (x+1, y-1, REACHABLE);
          if (board[x+1][y]   == EMPTY) mBoard.set (x+1, y,   REACHABLE);
          if (board[x+1][y+1] == EMPTY) mBoard.set (x+1, y+1, REACHABLE);
          mBoard = jampableBoard (x, y, mBoard); // ジャンプで移動できる位置の探索
          return mBoard;
}

/**
 * 指定した位置からジャンプで移動できる位置を記入された盤を返す
 * @param int x : ジャンプ開始位置のx座標
 * @param int y : ジャンプ開始位置のy座標
 * @param Board jBoard : 位置を記入する盤
 * @return Board : 指定した位置からジャンプで移動できる位置を記入した盤
 */
private Board jampableBoard (int x, int y, Board jBoard) {
          if (jBoard.board[x-1][y-1] != BORDER) { // (-2, -2) 方向へのジャンプ
                    if  ((jBoard.board[x-1][y-1]  ==  WHITE  ||  jBoard.board[x-1][y-1]  ==  BLACK)  &&
```

```
jBoard.board[x-2][y-2] == EMPTY) {
                                        jBoard.set (x-2, y-2, REACHABLE);
                                        jBoard = jampableBoard (x-2, y-2, jBoard); // さらにジャンプできる位置を
探す
                                }
                        }
                        if (jBoard.board[x-1][y]   != BORDER) { // (-2,  0) 方向へのジャンプ
                                if ((jBoard.board[x-1][y]    == WHITE || jBoard.board[x-1][y]    == BLACK) &&
jBoard.board[x-2][y]   == EMPTY) {
                                        jBoard.set (x-2, y,   REACHABLE);
                                        jBoard = jampableBoard (x-2, y,   jBoard); // さらにジャンプできる位置を
探す
                                }
                        }
                        if (jBoard.board[x-1][y+1] != BORDER) { // (-2, +2) 方向へのジャンプ
                                if ((jBoard.board[x-1][y+1] == WHITE || jBoard.board[x-1][y+1] == BLACK) &&
jBoard.board[x-2][y+2] == EMPTY) {
                                        jBoard.set (x-2, y+2, REACHABLE);
                                        jBoard = jampableBoard (x-2, y+2, jBoard); // さらにジャンプできる位置を
探す
                                }
                        }
                        if (jBoard.board[x][y-1]   != BORDER) { // ( 0, -2) 方向へのジャンプ
                                if ((jBoard.board[x][y-1]    == WHITE || jBoard.board[x][y-1]    == BLACK) &&
jBoard.board[x][y-2]   == EMPTY) {
                                        jBoard.set (x,   y-2, REACHABLE);
                                        jBoard = jampableBoard (x,   y-2, jBoard); // さらにジャンプできる位置を
探す
                                }
                        }
                        if (jBoard.board[x][y+1]   != BORDER) { // ( 0, +2) 方向へのジャンプ
                                if ((jBoard.board[x][y+1]    == WHITE || jBoard.board[x][y+1]    == BLACK) &&
jBoard.board[x][y+2]   == EMPTY) {
                                        jBoard.set (x,   y+2, REACHABLE);
                                        jBoard = jampableBoard (x,   y+2, jBoard); // さらにジャンプできる位置を
探す
                                }
                        }
                        if (jBoard.board[x+1][y-1] != BORDER) { // (+2, -2) 方向へのジャンプ
                                if ((jBoard.board[x+1][y-1] == WHITE || jBoard.board[x+1][y-1] == BLACK) &&
jBoard.board[x+2][y-2] == EMPTY) {
                                        jBoard.set (x+2, y-2, REACHABLE);
                                        jBoard = jampableBoard (x+2, y-2, jBoard); // さらにジャンプできる位置を
探す
                                }
                        }
                        if (jBoard.board[x+1][y]   != BORDER) { // (+2,  0) 方向へのジャンプ
                                if ((jBoard.board[x+1][y]    == WHITE || jBoard.board[x+1][y]    == BLACK) &&
```

```
jBoard.board[x+2][y]   == EMPTY) {
                                        jBoard.set (x+2, y,    REACHABLE);
                                        jBoard = jampableBoard (x+2, y,    jBoard); // さらにジャンプできる位置を
探す
                                }
                        }
                        if (jBoard.board[x+1][y+1] != BORDER) { // (+2, +2) 方向へのジャンプ
                                if  ((jBoard.board[x+1][y+1]  ==  WHITE  ||  jBoard.board[x+1][y+1]  ==  BLACK)  &&
jBoard.board[x+2][y+2] == EMPTY) {
                                        jBoard.set (x+2, y+2, REACHABLE);
                                        jBoard = jampableBoard (x+2, y+2, jBoard); // さらにジャンプできる位置を
探す
                                }
                        }
                        return jBoard;
                }

                /**
                 * ゴールしたか？
                 * @param boolean isWhite : 白番か？
                 * @return boolean : 指定した手番がゴールしたか？
                 */
                public boolean isGoal (boolean isWhite) {
                        if (isWhite) { // 白番
                                if (board[1][size]   != WHITE
                                 || board[1][size-1] != WHITE
                                 || board[1][size-1] != WHITE
                                 || board[1][size-2] != WHITE
                                 || board[1][size-3] != WHITE
                                 || board[2][size]   != WHITE
                                 || board[2][size-1] != WHITE
                                 || board[2][size-2] != WHITE
                                 || board[3][size]   != WHITE
                                 || board[3][size-1] != WHITE
                                 || board[4][size]   != WHITE) {
                                        return false;
                                } else if (stoneNum >= 13) { // 13 個以上石を置く場合に追加配置
                                        if (board[2][size-3] != WHITE
                                         || board[3][size-2] != WHITE
                                         || board[4][size-1] != WHITE) {
                                                return false;
                                        } else if (stoneNum >= 19) { // 19 個以上石を置く場合に追加配置
                                                if (board[1][size-4] != WHITE
                                                 || board[2][size-4] != WHITE
                                                 || board[3][size-3] != WHITE
                                                 || board[4][size-2] != WHITE
                                                 || board[5][size-1] != WHITE
                                                 || board[5][size]   != WHITE) {
```

```
                                        return false;
                                } else return true;
                        } else return true;
                } else return true;
        } else { // 黒番
                if (board[size]  [1] != BLACK
                 || board[size]  [2] != BLACK
                 || board[size]  [3] != BLACK
                 || board[size]  [4] != BLACK
                 || board[size-1][1] != BLACK
                 || board[size-1][2] != BLACK
                 || board[size-1][3] != BLACK
                 || board[size-2][1] != BLACK
                 || board[size-2][2] != BLACK
                 || board[size-3][1] != BLACK) {
                        return false;
                } else if (stoneNum >= 13) { // 13個以上石を置く場合に追加配置
                        if (board[size-1][4] != BLACK
                         || board[size-2][3] != BLACK
                         || board[size-3][2] != BLACK) {
                                return false;
                        } else if (stoneNum >= 19) { // 19個以上石を置く場合に追加配置
                                if (board[size]  [5] != BLACK
                                 || board[size-1][5] != BLACK
                                 || board[size-2][4] != BLACK
                                 || board[size-3][3] != BLACK
                                 || board[size-4][2] != BLACK
                                 || board[size-4][1] != BLACK) {
                                        return false;
                                } else return true;
                        }else return true;
                } else return true;
        }
}

/**
 * 先読みを行う
 * @param int depth : 先読み数
 */
public int value (int depth) {
        NextMove move;
        int maxValue = -1000;
        NextMove bestMove = null;

        if (depth == 0) {
                return value;
        }
        ArrayList<NextMove> List = createMovableList(isWhite);
```

```
                for (int i=0; i<List.size(); ++i) {
                        move = List.get(i);
                        Board nextBoard = nextBoard(move);
                        int value = nextBoard.value(depth-1);
                        if (value > maxValue){
                                maxValue = value;
                                bestMove = List.get(i);
                        }
                }
                return value;
        }
}
```

NextMove.java

```
package Halma;

import Halma.Board;


/**
 * 駒の移動可能な位置を表すクラス
 */
public class NextMove {
        int curX;        // 現在の x 座標
        int curY;        // 現在の y 座標
        int nextX;       // 移動先の x 座標
        int nextY;       // 移動先の y 座標
        boolean isWhite; // 白石か？
        int value;       // 着手の評価値
        Board BOARD;

        private static final int valueOfPlaceW[][] = {  //  盤面の評価値
              // y 0  1  2  3  4  5  6  7  8  9      x
                   {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 0
                   {0,-1,-1,-1, 0, 6, 6, 7, 8, 0}, // 1
                   {0,-1,-1, 0, 0, 0, 5, 6, 7, 0}, // 2
                   {0,-1, 0, 0, 0, 0, 0, 5, 6, 0}, // 3
                   {0, 0, 0, 0, 0, 0, 0, 0, 6, 0}, // 4
                   {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 5
                   {0, 0, 0, 0, 0, 0, 0, 0,-1, 0}, // 6
                   {0, 0, 0, 0, 0, 0, 0,-1,-1, 0}, // 7
                   {0, 0, 0, 0, 0, 0,-1,-1,-1, 0}, // 8
                   {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 9
        };
        private static final int valueOfPlaceB[][] = {  //  盤面の評価値
```

```
    // y 0  1  2  3  4  5  6  7  8  9      x
          {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 0
          {0,-1,-1,-1, 0, 0, 0, 0, 0, 0}, // 1
          {0,-1,-1, 0, 0, 0, 0, 0, 0, 0}, // 2
          {0,-1, 0, 0, 0, 0, 0, 0, 0, 0}, // 3
          {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 4
          {0, 6, 0, 0, 0, 0, 0, 0, 0, 0}, // 5
          {0, 6, 5, 0, 0, 0, 0, 0,-1, 0}, // 6
          {0, 7, 6, 5, 0, 0, 0,-1,-1, 0}, // 7
          {0, 8, 7, 6, 6, 0,-1,-1,-1, 0}, // 8
          {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 9
};
private static final int valueOfPlaceW2[][] = {  //  盤面の評価値
    // y 0  1  2  3  4  5  6  7  8  9      x
          {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 0
          {0,10,10,10,10, 4, 3, 2, 1, 0}, // 1
          {0, 9, 8, 7, 6, 7, 4, 3, 2, 0}, // 2
          {0,10, 9, 8, 7, 6, 7, 4, 3, 0}, // 3
          {0,11,10, 9, 8, 7, 6, 7, 4, 0}, // 4
          {0,12,11,10, 9, 8, 7, 6,10, 0}, // 5
          {0,13,12,11,10, 9, 8, 7,10, 0}, // 6
          {0,14,13,12,11,10, 9, 8,10, 0}, // 7
          {0,15,14,13,12,11,10, 9,10, 0}, // 8
          {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 9
};
private static final int valueOfPlaceB2[][] = {  //  盤面の評価値
    // y 0  1  2  3  4  5  6  7  8  9      x
          {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 0
          {0,10, 9,10,11,12,13,14,15, 0}, // 1
          {0,10, 8, 9,10,11,12,13,14, 0}, // 2
          {0,10, 7, 8, 9,10,11,12,13, 0}, // 3
          {0,10, 6, 7, 8, 9,10,11,12, 0}, // 4
          {0, 4, 5, 6, 7, 8, 9,10,11, 0}, // 5
          {0, 3, 4, 5, 6, 7, 8, 9,10, 0}, // 6
          {0, 2, 3, 4, 5, 6, 7, 8, 9, 0}, // 7
          {0, 1, 2, 3, 4,10,10,10,10, 0}, // 8
          {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 9
};


public NextMove (int curX, int curY, int nextX, int nextY, boolean isWhite, Board BOARD) {
          this.curX = curX;
          this.curY = curY;
          this.nextX = nextX;
          this.nextY = nextY;
          this.isWhite = isWhite;
          this.BOARD = BOARD;
```

```
                    if (isWhite) {
                              value    =    curX-nextX    +    nextY-curY    +    valueOfPlaceW[nextX][nextY]    +
valueOfPlaceW2[curX][curY];
                              int counter = 0;
                              if (8 - curX > curY){
                                        if (nextY > curY) counter += 1;
                              } else if (8 - curX < curY) {
                                        if (nextX < curX) counter += 1;
                              }

                              counter += (curX - 1) + (8 - curY);

                              if (BOARD.board[nextX-1][nextY]   == Board.BLACK)
                                        if (BOARD.board[nextX+1][nextY]   == Board.WHITE) counter += 1;
                              if (BOARD.board[nextX-1][nextY+1] == Board.BLACK)
                                        if (BOARD.board[nextX+1][nextY-1]   == Board.WHITE) counter += 1;
                              if (BOARD.board[nextX]  [nextY+1] == Board.BLACK)
                                        if (BOARD.board[nextX][nextY-1]   == Board.WHITE) counter += 1;

                              if (nextX == 1) if (nextY >= 5){
                                        if (curX == 1) if (curY < 5)  counter += 3;
                                        if (curX == 2) if (curY < 6)  counter += 3;
                                        if (curX == 3) if (curY < 7)  counter += 3;
                                        if (curX == 4) if (curY == 8) counter += 3;
                                        if (curX >  4) counter += 3;
                              }
                              if (nextX == 2) if (nextY >= 6) {
                                        if (curX == 1) if (curY < 5)  counter += 3;
                                        if (curX == 2) if (curY < 6)  counter += 3;
                                        if (curX == 3) if (curY < 7)  counter += 3;
                                        if (curX == 4) if (curY == 8) counter += 3;
                                        if (curX >  4) counter += 3;
                              }
                              if (nextX == 3) if (nextY >= 7) {
                                        if (curX == 1) if (curY < 5)  counter += 3;
                                        if (curX == 2) if (curY < 6)  counter += 3;
                                        if (curX == 3) if (curY < 7)  counter += 3;
                                        if (curX == 4) if (curY == 8) counter += 3;
                                        if (curX >  4) counter += 3;
                              }
                              if (nextX == 4) if (nextY == 8) {
                                        if (curX == 1) if (curY < 5)  counter += 3;
                                        if (curX == 2) if (curY < 6)  counter += 3;
                                        if (curX == 3) if (curY < 7)  counter += 3;
                                        if (curX == 4) if (curY == 8) counter += 3;
                                        if (curX >  4) counter += 3;
                              }
                              value += counter;
```

```
} else {
            value = nextX-curX + curY-nextY + valueOfPlaceB[nextX][nextY] +
                                                valueOfPlaceB2[curX][curY];
            int counter = 0;
            if (8 - curX > curY){
                        if (nextX > curX) counter += 1;
            } else if (8 - curX < curY) {
                        if (nextY < curY) counter += 1;
            }

            counter += (curY - 1) + (8 - curX);

            if (BOARD.board[nextX][nextY-1]   == Board.WHITE)
                        if (BOARD.board[nextX][nextY+1]   == Board.BLACK) counter += 1;
            if (BOARD.board[nextX+1][nextY-1] == Board.WHITE)
                        if (BOARD.board[nextX-1][nextY+1] == Board.BLACK) counter += 1;
            if (BOARD.board[nextX+1][nextY]   == Board.WHITE)
                        if (BOARD.board[nextX-1][nextY]   == Board.BLACK) counter += 1;

            if (nextX == 8) if (nextY <= 4){
                        if (curX == 8) if (curY > 4)  counter += 3;
                        if (curX == 7) if (curY > 3)  counter += 3;
                        if (curX == 6) if (curY > 2)  counter += 3;
                        if (curX == 5) if (curY == 1) counter += 3;
                        if (curX >  5) counter += 3;
            }
            if (nextX == 7) if (nextY <= 3) {
                        if (curX == 8) if (curY > 4)  counter += 3;
                        if (curX == 7) if (curY > 3)  counter += 3;
                        if (curX == 6) if (curY > 2)  counter += 3;
                        if (curX == 5) if (curY == 1) counter += 3;
                        if (curX >  5) counter += 3;
            }
            if (nextX == 6) if (nextY >= 2) {
                        if (curX == 8) if (curY > 4)  counter += 3;
                        if (curX == 7) if (curY > 3)  counter += 3;
                        if (curX == 6) if (curY > 2)  counter += 3;
                        if (curX == 5) if (curY == 1) counter += 3;
                        if (curX >  5) counter += 3;
            }
            if (nextX == 5) if (nextY == 1) {
                        if (curX == 8) if (curY > 4)  counter += 3;
                        if (curX == 7) if (curY > 3)  counter += 3;
                        if (curX == 6) if (curY > 2)  counter += 3;
                        if (curX == 5) if (curY == 1) counter += 3;
                        if (curX >  5) counter += 3;
            }
            value += counter;
```

```
                }
        }

        /**
         * @return String 着手の文字列表現
         */
        public String toString() {
                String color;
                if (isWhite) color = "W";
                else color = "B";
                return (color+":("+curX+","+curY+")->("+nextX+","+nextY+") : "+value);
        }

        /**
         * @return int 着手の評価値
         */
        public int value() {
                return value;
        }
}
```