卒業研究報告書

題目

リバーシの評価関数について

指導教員

石水 隆　講師

報告者

09-1-037-0133

塩田 好

近畿大学理工学部情報学科

平成 24 年 1 月 31 日提出

概要

リバーシは 1888 年イギリスで考案されたゲームである。盤面は 8x8 のマスで構成され白と黒の二つの駒で交互に駒を打ち、相手の駒をはさんで裏返えし、最終的に駒数が多いプレイヤーが勝利と誰にでも覚えやすいルールである。しかしその反面奥が深く知能ゲームの要素を持ち世界中に広まった。「覚えるのは一分、極めるのは一生」と言われており現在の技術を駆使してさえ完全解析されていないゲームの一つである。現在のコンピュータに思考させることは不可能なため、「思考の手順」を教え、その手順に従って最善の手を計算させる、これが評価関数である。

本研究では、リバーシにおける評価関数がもつパラメタに付加された重みを変化させたときの評価関数の勝率がどのように変化するか観測し、最適な重みの組み合わせを求める。対戦相手は着手可能手からランダムで手を決定するコンピュータとする。

目次

1 序論 1

1.1 二人零和有限確定完全情報ゲーム 1

1.2 リバーシ 1

1.3 リバーシに関する既知の結果 1

1.3.1 リバーシの完全解析 2
1.3.2 リバーシの定石 2
1.3.3 局面の評価 2
1.3.4 先読み 2
1.3.5 既存のリバーシプログラム 2

1.4 本研究の目的 3

1.5 本報告書の構成 3

2 リバーシプログラムの一般的手法 3

2.1 定石データベース・対戦データベース 3

2.2 モンテカルロ法 3

2.3 先読みと評価関数 4

3 研究内容 4

3.1. 評価関数 4

3.1.1 盤位置(BP) 4
3.1.2 確定石(FS) 4
3.1.3 候補数(CN) 5
3.1.4 評価関数 5
3.2 リバーシプログラム 5
3.2.1 クラス Reversi 6

3.3 対戦実験の前提条件 6

4 結果および考察 6

4.1 各パラメタの効力 6

4.2　BP と FS の比較 7
4.3　BP と CN の比較 7
4.4　FS と CN の比較 7
4.5　BP と FS と CN の比較 7
5　結論・今後の課題 13
謝辞 14
参考文献 15
付録 16

1　序論

1.1 二人零和有限確定完全情報ゲーム

将棋やチェス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。二人零和有限確定完全情報ゲームとは、二人または二チームでゲームを行い、ゲーム終了時双方のプレイヤーの利得合計が零、双方のプレイヤーの着手可能手が有限、プレイヤーの着手以外がゲームに影響を与える偶然の要素が入らず、そして各プレイヤーの着手の意思決定の情報が知ることができるゲームである。二人零和有限確定完全情報ゲームに分類するゲームの特徴として、理論上完全な先読みが可能であり、双方のプレイヤーが最善手を打てば、先手必勝か後手必勝か引分が決まる。二人零和有限完全情報ゲームは、その性質上解析を行い易いため、ゲーム理論において様々な研究がなされてきた。また、人工知能の分野においても広く研究がなされている。

1.2 リバーシ

リバーシは 8x8 のマスを使用する。各マスには、横に a〜b, 縦に 1〜8 の座標が付いている。図 1 にリバーシの盤面と石の初期配置を示す。自石で相手石を挟めるマスに石を置くことができ、挟んだ石は反転させ自石となる。先手、後手交互に打ち進め、一方が打てるマスがない場合パスとなる。盤面が全て埋まる、もしくは先手、後手ともに石を挟めなくなった時点でゲームが終了となる。勝敗判定は石の数が多いプレイヤが勝利、同数の場合は引き分けとなる。

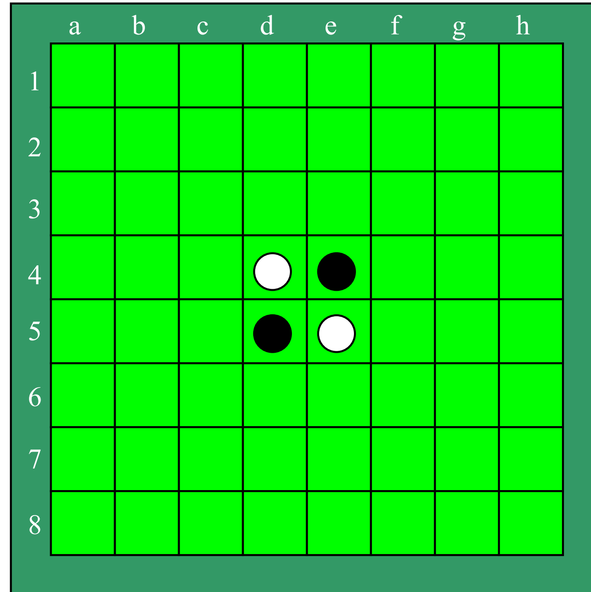

図 **1** リバーシの盤面と石の初期配置

1.3 リバーシに関する既知の結果

本節ではリバーシに関する既知の結果を示す。

1.3.1 リバーシの完全解析

リバーシは 1 ゲームにつき最大 60 手順しかないことから、対戦型ゲームとしては手順がかなり少ないゲームである。しかし、それでも初期配置の中央 4 マスは白黒の 2 通り、それ以外 60 マスは白黒空の 3 通り、可能な局面の組み合わせは $2^4*3^{60}=6.78*10^{19}$ 通り存在する。このためリバーシは現時点ではスーパーコンピュータを駆使してなお完全解析されていない。しかし盤面のサイズを小さくしたものについては可能な局面数が少なく、完全解析も可能がなされている。リバーシの縮小版 6x6 のリバーシだと $2^4*3^{32}=2.96*10^{16}$ 存在する。6x6 リバーシは 1994 年イギリスの研究者 Feinstein,によって後手必勝であること、また後手が最善を打ったとき、先手は最大 16 個しか取れないことが証明されている[7]。

1.3.2 リバーシの定石

前節で述べた通り、リバーシの完全解析は現時点ではまだであり、特に序盤においてはどのような手が最善となるかを決定することはできない。しかし、序盤においてどのような手が有利になり易いかは定石として確立している。代表的な序盤の定石には、縦取り兎定石、斜め取り牛定石、並び取り鼠定石がある[8]。

局面が様々に変化する中盤においても、定石がいくつか確立されている。代表的な中盤の定石には、中割り、引っ張り等がある[8]。中割りは「周りが全て石に囲まれている石のみを返す」事を意味し、相手の打てるマスを増やさない(減らす)ための手筋である[8]。引っ張りは壁を作り、相手はこちらの壁を崩さなければ石が置けない状態を作ることで、相手の石を誘導したいときに用いる手筋である[8]。

1.3.3 局面の評価

ゲーム中盤における局面の状況は様々であり、定石が存在しない局面も存在する。また、序盤であっても相手が定石以外の手を打った場合、同様に定石が存在しない局面となる。そのような場合に良い手を発見するために用いられるのが、その局面を評価することである。ある局面で盤上に置かれた石の並び方を入力とする評価関数を設定し、その関数の値によって先手後手のどちらがどの程度有利なのかを判定する。

どのような評価関数を用いるのが良いかは未解決の問題であり、様々な評価関数が提案されている[2][3] [4] [8] 。

1.3.4 先読み

現在の局面から数手先の局面を先読みし、それを下に評価値を決定することで評価関数の精度を上げられる。一般に先読み手数を増やすにつれ評価関数の精度は上がり、最終局面まで読むことができれば、勝敗を完全に決定できる完全な評価関数になる。しかしながら 1.3.1 節で述べたように、先読み手数が増えるにつれ可能な局面数は指数的に増えるため、序盤・中盤では完全先読みは不可能である。このため、序盤・中盤では一定手数先まで読み、得られた局面の各評価値を元に評価値を求める手法が一般によく使われる。一方、終盤では残りの手が少なくなるので、その盤面から最終局面まで全て先読みすることが可能となる。現在の計算機性能では、残り手数が 25～30 手程度でも最終局面までの完全先読みが可能となっている。

1.3.5 既存のリバーシプログラム

リバーシプログラムは、定石データベースの利用、局面の評価値計算、序盤・中盤での先読み、終盤の完全先読みのどれか、もしくは各手法組み合わせを使用することが多い。

Edax[9],MasterReversi[10],WZebra[11]等のリバーシプログラムの評価値は基本的には、

盤面に置かれた石のパターンによって算出されている。また、過去の対局データを用いて遺伝的アルゴリズムによって評価値のバランス調整がされているものもある。

1.4 本研究の目的

リバーシにおいて完全解析がなされてない為、最善手が存在しない。そこで、ゲームを有利に進める為に局面の評価をする評価関数を作成する。評価関数のパラメタに付加する重みを変化させ最も有効であると考えられる評価関数を求める。

1.5 本報告書の構成

本論文の構成は以下の通りである。まず第 2 章でリバーシプログラムの一般的手法について説明する。続く第 3 章で本研究で用いた局面の評価関数ついて説明する。第 4 章で評価関数を用いた対戦結果を示す。第 5 章で結論と課題を示す。

2 リバーシプログラムの一般的手法

1 章で述べたとおり、リバーシはまだ完全解析はできていないため、常に最善な手を選択することはできない。そこで本章では、リバーシプログラムで用いられる一般的手法について述べる。

現在、強いと評価されているプログラム[9][10][11]は、序盤は定石データベースに従って打ち、中盤、あるいは相手が定石から外れた手を打ったときの序盤では、一定手数先読みし、先読み後の局面に対して評価関数を用いて評価値を求め、その値から打つ手を決定する。そして終盤、残り手数がある一定数以下になると完全読みを行い、最善手を打つ。

2.1 定石データベース・対戦データベース

定石データベースとは、リバーシの定石をデータベース化し、各局面で有効な定石があればそれに従って打つという手法である。定石データベースを使用することで強いリバーシプログラムとなる。しかし、相手があえて定石以外の手を打つなどして、データベースに無い局面が出てきたときにはこの手法は使えない。

繰り返し対戦を行う場合は、それまでの対戦記録をデータベース化しておくという手法も考えられる。過去の対戦において、その手が有効であったかどうかを対戦結果から判定し、データベースに蓄える。数多く対戦することで、その手が有効かどうかより精度が高い判定をすることができる。対戦データベースを用いることで、対戦経験が増えるにつれて強くなる人工知能型のリバーシプログラムになる。対戦データベースを使うためには、事前に繰り返し対戦して学習しておく必要がある。しかし、学習が足りなかったり、事前の対戦でなかった手を打たれたりした場合には使えないという欠点がある。

2.2 モンテカルロ法

オセロプログラムではあまり使われないが、モンテカルロ法 [8]の利用も考えられる。モンテカルロ法とは、各着手可能手に対し、その手から先終局までをランダムに打ち勝敗判定を行うという作業を数千～数万回繰り返し、最も勝率の高い着手可能手を採用すると

いうものである。この手法は局面数が極めて多い囲碁プログラムでは最近主流になっている[13]。

2.3 先読みと評価関数

前述の通り、定石データベースは序盤のみ、完全読みは終盤のみ使用できる。そこで、一般的には評価関数を用いて現在の局面、または数手先の局面を評価する。評価関数として、一般的に初心者でも組みやすいとされているのが盤面評価値である。盤面 1 マスずつに重みをつけてある局面をその値で評価する。ただ、盤面評価値では強いオセロプログラムを作ることはできない。これは盤面の重みだけで見るため全体の局面を見ることができないといった問題がある。

また、ある局面の評価値を求める評価関数は、現在の局面のみを考慮する現局面評価と、数手先の局面を先読みし、先読みした局面に対して現局面評価を行い、その評価値を元に現在の局面の評価値を求める先読み局面評価の 2 つに分けられる。従って先読み局面評価を行うためには、まず現局面評価を行える必要がある。よってまず現局面評価について述べる。

現局面評価の評価関数の計算に用いられる評価基準については、先に記述した盤面評価、直線性、確定石、直線性、駒数、候補数等様々なものが提案されている[13] [13] [13] [13]。

3 研究内容

2.3 節で述べたように、評価関数としてどのようなパラメタを用いれば良いかは自明ではない。本研究では評価関数のパラメタとして盤面に存在する石の位置から評価する盤位置、ひっくり返される可能性が無い位置に置かれた確定石の数、ある局面で次に打てる手の候補数の三つを用いる。

3.1. 評価関数

本節では、本研究で用いる評価関数について述べる。

3.1.1 盤位置(BP)

盤位置(以下 BP とする)の評価は、8x8 のマス全てに価値を持たせ、自石が置かれていればその値を加算、相手石が置かれていれば減算しその合計値を盤位置の評価値とする。各マスの価値はあらゆる実践データの統計と分析が必要となるため、オリジナルの評価値の作成はしない。各マスの価値は様々なものが提案されている[1][2][4]。本研究では図 1 に示す評価値を用いる[2]。この評価を用いる理由として他のパラメタで設定する値より差が大きくなり過ぎない為用いた。盤位置の評価値 BP は以下の式で与えられる。ただし $board(i,j)$はマス(i,j)が自石なら 1, 相手石なら-1, 空マスなら 0 となり、$BP(i,j)$は各マス目の評価値である。また、以下 rnd は 0 から 1 までの一様乱数とする。

$$BP = \sum_{i=0}^{7}\sum_{j=0}^{7} BP(i,j) * board(i,j) * rnd * 3$$

3.1.2 確定石(FS)

確定石とは一度取ると絶対に相手に取られることのない石の事を指す。確定石はその後の展開に左右されず最後まで残るため、確定石が多いほど有利と考えられる。本研究で

は全ての確定石を求めるアルゴリズムの作成が困難なため、四つの辺における確定石のみを評価した。確定石の評価値 FS は以下の式で与えられる。

$$FS = ((\text{自分の確定石数} - \text{相手の確定石数}) + rnd * 3) * 11$$

	a	b	C	d	e	f	g	h
1	45	−11	4	−1	−1	4	−11	45
2	−11	−16	−1	−3	−3	2	−16	−11
3	4	−1	2	−1	−1	2	−1	4
4	−1	−3	−1	0	0	−1	−3	−1
5	−1	−3	−1	0	0	−1	−3	−1
6	4	−1	2	−1	−1	2	−1	4
7	−11	−16	−1	−3	−3	−1	−16	−11
8	45	−11	4	−1	−1	4	−11	45

図 1.盤位置の評価(BP)

	a	b	c	d	e	f	g	h
1								
2			○					
3	●	●	○	○	○	○		
4	●	●	○	●	●	●		
5	●	●	○	●	○	●		
6	●	○	○	○	●	○		
7	●		○	○	●	○		
8	●	○	○	○	○	○	○	

図 2.局面の例

図 2 に示す盤面における確定石は a-3 から a-8 の黒石 6 個が確定石となる。

3.1.3 候補数(CN)

候補数とは、ある局面で次に自分、もしくは相手が着手可能なマスの数を表す。一般的に自分の候補数が多いほどよく、相手の候補数が少ないほどよいとされている。候補数の評価値 CN は以下の式で与えられる。

$$CN = (\text{着手可能な候補数} + \mathrm{rnd} * 2) * 10$$

3.1.4 評価関数

本研究では、上記の盤位置 BP, 確定石 FS, 候補数 CN の 3 つを評価関数のパラメタとして用いる。本研究で用いる評価関数 f は以下の式で与えられる。ただし W_{BP}, W_{FS}, W_{CN} は各パラメタの重みである。

$$f = BP * W_{BP} + FS * W_{FS} + CN * W_{CN}$$

各パラメタに付加する重みの範囲は以下とした。

$$0 \leqq W_{BP} \leqq 5$$

$$0 \leqq W_{FS} \leqq 5$$

$$0 \leqq W_{CN} \leqq 1$$

3.2 リバーシプログラム

本研究では、3.1 節で述べた評価関数の各要素のどれがより正確に各局面の評価値を反映させているかを検討するために、各パラメタに付加された重みを変化させ、計算機実験を行う。付録 1 に本研究で作成したプログラムを示す。

3.2.1 クラス Reversi

クラス Reversi は本研究で作成したりリバーシプログラムの中心部である。

Reversi では int evaluateBoard(),int evaluateFinalStone(),int evaluateCN()の 3 つの関数を用いて評価値の計算を行っている。

int[][] board が盤面の情報を記憶し、int turn がどちらの手番かを記憶する。boolean[][] pboard は着手可能なマスを記憶する変数であり、可能なマスを true、それ以外を false として記憶する。int[][] pList は pBoard の true のマスの座標をリストとして保持する。

3 つの評価値を組み合わせて実際に手を決定するメソッドが

int[] valueMapComputer(), int[] valueFinalComputer(), int[] valueCNComputer(),

int[] valueMapFinalComputer, int[] valueMapCNComputer(),

int[] valueFinalCNComuter(), int[] valueMapFinalCNcomputer()

の 7 つである。

また、以下が実際に評価値を計算する 3 つの関数について示す。

int evaluateBoard():現局面における盤位置 BP を計算し、それを戻り値として持つ。8x8 マスを 3.1.1 章で示した方法で計算する。

int evaluateFinalStone():現局面における 4 辺上の確定石を計算し、それを戻り値として持つ。

int evaluateCN():現局面における着手可能なマスの数を計算し、それを戻り値として持つ。pboard より true の数が CN となる。

3.3 対戦実験の前提条件

本研究では3.1節で示した評価関数の各パラメタに付加する最適な重みの値を検討するために、各パラメタに付加された重みを変化させた場合の勝率を求める。対戦相手は着手可能手からランダムで手を決定するコンピュータである。対戦の条件は次の通りである。

① ：各重みにつき対戦回数は 1000 回とする。
② ：先手、後手の両方で対戦する。

この条件をもとに対戦を行い､勝率を比較することによって最適であると考えられる重みを求める。

4 結果および考察

本章では､各パラメタに重みに対し､先手および後手でランダムプログラムと対戦したときの対戦結果、およびその結果から得られるパラメタの最適な重みについて考察する。

4.1 各パラメタの効力

評価関数のパラメタとして、各パラメタを単独で用いた場合の、対戦結果を表 1.に示す。表 1 より、先手、後手に関わらず、ほとんどの場合でパラメタ FS が最も効力を表し

ていることがわかる。表 1. より以下の効力の優先順位が予測できる。

$$FS > BP > CN$$

4.2 BP と FS の比較

BP と FS の二つのパラメタを用いたときの対戦結果を表 2.に、重みの変化に伴う勝数の推移を図 3.に示す。表 2 より、W_{BP}の値に関わらず W_{FS}が上がるにつれ勝数が増えていることがわかる。図 3.より

$$1 \leqq W_{BP} \leqq 2,\ 3 \leqq W_{FS} \leqq 5$$

の範囲が有効であると予測できる。

4.3 BP と CN の比較

BP と CN の二つのパラメタを用いたときの対戦結果を表 3.に、重みの変化に伴う勝数の推移を図 4.に示す。図 4.より

$$2 \leqq W_{BP} \leqq 5,\ 1 \leqq W_{CN} \leqq 2$$

の範囲が有効であると予測できる。

4.4 FS と CN の比較

FS と CN の二つのパラメタを用いたときの対戦結果を表 4.に、重みの変化に伴う勝数の推移を図 5.に示す。表 4 より、W_{FS} の値に左右されず、W_{CN}の上昇に伴い勝数の著しい低下が見られる。この図 5.より

$$1 \leqq W_{BP} \leqq 5\ ,\ 0 < W_{CN} \leqq 1$$

の範囲が有効であると予測できる。

4.5 BP と FS と CN の比較

BP と FS と CN の三つのパラメタを用いたときの対戦結果を表 5.示す。表 5 より、先手、後手ともに[2:5:1](=[W_{BP}, W_{FS}, W_{CN})が有効であることが示される。また表 5.から先手、後手ともに勝数の上位 6 位までを抽出し、その重みの分布を表 6.に示す。表 6.からも最も有効である重みが[2:5:1] (=[W_{BP}, W_{FS}, W_{CN})であるとわかる。

表 1. 各パラメタの勝率

	先手			後手		
	勝	負	引分	勝	負	引分
BP	749	215	36	739	215	46
FS	832	139	29	820	154	26
CN	659	326	15	612	367	21

表 2. $F=BP*W_{BP}+FS*W_{FS}$ の対戦結果

BP	FS	先手			後手		
		勝	負	引	勝	負	引
1	1	946	40	14	946	46	8
	2	971	18	11	968	19	13
	3	982	12	6	981	13	6
	4	980	14	6	978	16	6
	5	975	19	6	975	19	6
2	1	911	73	16	922	60	18
	2	956	36	8	946	46	8
	3	968	27	5	974	21	5
	4	980	16	4	975	19	6
	5	972	21	7	978	14	8
3	1	896	78	26	874	99	27
	2	938	50	12	940	48	12
	3	954	37	9	953	35	12
	4	969	24	7	968	21	11
	5	971	18	11	965	26	9
4	1	910	62	28	880	86	34
	2	931	59	10	928	53	19
	3	938	49	13	937	50	13
	4	942	46	12	944	43	13
	5	961	31	8	959	33	8
5	1	856	116	28	875	104	21
	2	895	79	26	893	84	23
	3	935	51	14	918	62	20
	4	940	41	19	929	48	23
	5	949	42	9	951	32	17

表 3. $F=BP*W_{BP}+CN*W_{CN}$ の対戦結果

BP	CN	先手			後手		
		勝	負	引	勝	負	引
1	1	770	185	45	759	204	37
	2	665	295	40	682	281	37
	3	632	324	44	619	330	51
	4	581	370	49	536	415	49
	5	493	463	44	521	431	48
2	1	790	171	39	822	143	35
	2	762	196	42	751	219	30
	3	733	223	44	719	244	37
	4	709	255	36	696	249	55
	5	650	305	45	656	290	54
3	1	824	145	31	785	182	33
	2	788	176	36	774	196	30
	3	737	219	44	744	216	40
	4	689	262	49	746	219	35
	5	711	245	44	698	261	41
4	1	806	164	30	799	165	36
	2	788	174	38	816	145	39
	3	808	157	35	785	175	40
	4	758	206	36	752	210	38
	5	742	219	39	740	219	41
5	1	781	184	35	785	173	42
	2	828	138	34	807	159	34
	3	790	185	25	806	166	28
	4	769	203	28	778	188	34
	5	745	225	30	756	204	40

表 4. $F=FS*W_{FS}+CN*W_{CN}$の対戦結果

FS	CN	先手			後手		
		勝	負	引	勝	負	引
1	1	780	189	31	849	128	23
	2	734	237	29	769	200	31
	3	663	295	42	708	262	30
	4	592	352	56	653	315	32
	5	552	387	61	622	347	31
2	1	788	194	18	846	126	28
	2	721	247	32	762	205	33
	3	647	309	44	713	249	38
	4	613	339	48	645	305	50
	5	562	394	44	660	303	37
3	1	782	199	19	852	126	22
	2	717	258	25	783	191	26
	3	637	320	43	755	213	32
	4	616	349	35	678	289	33
	5	598	364	38	633	328	39
4	1	793	171	36	840	139	21
	2	698	264	38	776	199	25
	3	624	334	42	727	239	34
	4	629	326	45	637	322	41
	5	569	391	40	601	355	44
5	1	813	166	21	842	140	18
	2	722	245	33	800	180	20
	3	671	298	31	721	246	33
	4	639	323	38	624	326	50
	5	578	379	43	604	355	41

表 4. $F=BP*W_{BP}+FS*W_{FS}+CN*W_{CN}$ の対戦結果

BP	FS	CN	先手			後手			BP	FS	CN	先手			後手		
			勝	負	引	勝	負	引				勝	負	引	勝	負	引
1	1	1	915	65	20	919	62	19	2	4	1	959	31	10	973	21	6
		2	850	124	26	855	120	25			2	935	43	22	947	44	9
		3	800	162	38	794	175	31			3	938	49	13	922	60	18
		4	752	222	26	735	233	32			4	910	68	22	914	67	19
		5	741	223	36	731	233	36			5	887	88	25	879	100	21
	2	1	944	43	13	950	41	9		5	1	981	11	8	977	18	5
		2	904	74	22	902	79	19			2	940	50	10	955	39	6
		3	885	84	31	865	107	28			3	943	43	14	925	59	16
		4	835	130	35	833	146	21			4	893	86	21	921	60	19
		5	809	167	24	788	187	25			5	907	77	16	919	69	12
	3	1	961	26	13	956	36	8	3	1	1	885	93	22	868	95	37
		2	923	65	12	924	59	17			2	851	116	33	847	121	32
		3	888	100	12	891	91	18			3	839	135	26	833	136	31
		4	879	100	21	878	100	22			4	796	171	33	791	169	40
		5	864	109	27	830	147	23			5	772	192	36	778	185	37
	4	1	951	39	10	958	36	6		2	1	931	59	10	948	39	13
		2	921	64	15	932	53	15			2	913	74	13	934	51	15
		3	888	83	29	911	68	21			3	890	88	22	901	84	15
		4	880	97	23	867	116	17			4	860	119	21	865	108	27
		5	881	97	22	868	105	27			5	838	126	36	847	124	29
	5	1	957	32	11	961	32	7		3	1	952	39	9	950	37	13
		2	934	47	19	916	68	16			2	929	55	16	928	53	19
		3	925	62	13	926	57	17			3	915	66	19	918	67	15
		4	913	75	12	894	88	18			4	913	64	23	892	88	20
		5	874	107	19	864	118	18			5	882	93	25	891	87	22
2	1	1	909	71	20	926	56	18		4	1	959	31	10	963	28	9
		2	871	107	22	860	116	24			2	946	42	12	948	38	14
		3	825	140	35	823	148	29			3	937	50	13	937	48	15
		4	814	161	25	803	167	30			4	903	81	16	917	72	11
		5	766	202	32	751	216	33			5	888	88	24	902	83	15
	2	1	940	42	18	953	39	8		5	1	963	27	10	963	25	12
		2	927	61	12	889	95	16			2	963	28	9	951	38	11
		3	891	95	14	886	90	24			3	924	61	15	945	44	11
		4	850	131	19	850	124	26			4	915	72	13	929	55	16
		5	828	152	20	822	156	22			5	906	77	17	895	86	19
	3	1	960	30	10	966	26	8	4	1	1	845	125	30	845	126	29
		2	917	62	21	943	45	12			2	865	103	32	853	111	36
		3	894	81	25	910	76	14			3	835	129	36	805	173	22
		4	885	98	17	893	91	16			4	817	158	25	809	156	35
		5	880	97	23	859	114	27			5	814	154	32	794	173	33

BP	FS	CN	先手			後手		
			勝	負	引	勝	負	引
4	2	1	920	63	17	907	69	24
		2	894	82	24	907	73	20
		3	865	107	28	886	87	27
		4	854	122	24	861	127	12
		5	829	140	31	835	138	27
	3	1	952	40	8	943	48	9
		2	923	61	16	936	53	11
		3	914	73	13	913	65	22
		4	884	92	24	908	79	13
		5	876	105	19	885	94	21
	4	1	953	30	17	964	27	9
		2	953	40	7	955	33	12
		3	948	38	14	930	53	17
		4	922	60	18	919	67	14
		5	906	74	20	915	64	21
	5	1	962	24	14	958	24	18
		2	956	29	15	959	30	11
		3	933	51	16	934	53	13
		4	933	55	12	916	69	15
		5	912	76	12	915	65	20
5	1	1	842	126	32	834	130	36
		2	833	139	28	848	115	37
		3	856	124	20	826	139	35
		4	789	168	43	801	167	32
		5	814	157	29	826	137	37
	2	1	896	84	20	914	62	24
		2	900	78	22	904	76	20
		3	885	91	24	876	100	24
		4	863	113	24	870	110	20
		5	847	115	38	839	125	36
	3	1	940	51	9	940	43	17
		2	926	56	18	918	65	17
		3	900	82	18	908	72	20
		4	886	93	21	907	77	16
		5	873	104	23	877	97	26

BP	FS	CN	先手			後手		
			勝	負	引	勝	負	引
5	4	1	942	39	19	946	34	20
		2	933	51	16	935	53	12
		3	919	58	23	926	54	20
		4	922	63	15	907	74	19
		5	898	87	15	906	81	13
	5	1	960	30	10	970	23	7
		2	947	35	18	943	38	19
		3	940	47	13	937	45	18
		4	918	67	15	924	59	17
		5	922	64	14	913	74	13

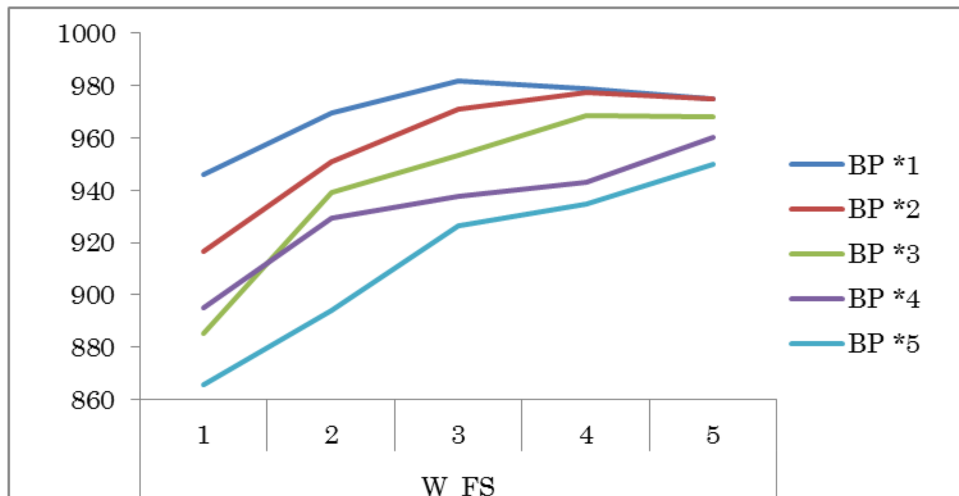

図 3. F= BP*W_{BP}+FS*W_{FS} の勝数推移

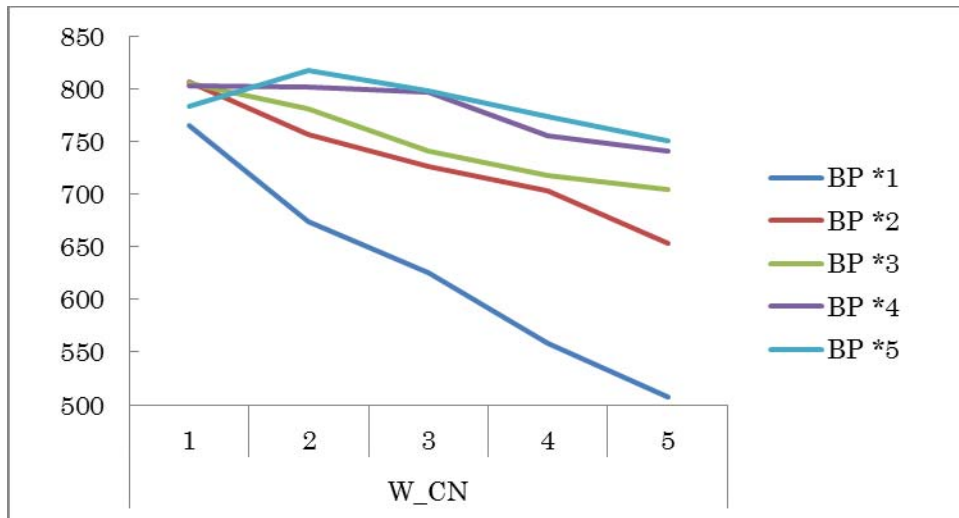

図 4. F= BP*W_{BP}+CN*W_{CN} の勝数推移

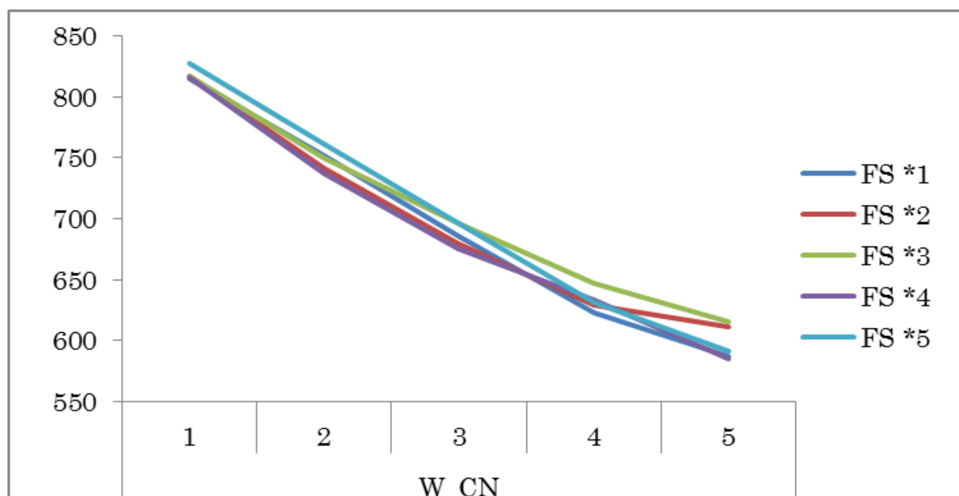

図 5. F= FS*W_{FS}+CN*W_{CN} の勝数推移

5　結論・今後の課題

本研究では、リバーシの局面の評価値を定める最適な評価関数を得るために、評価関数の各パラメタの重みを変えて計算機実験を行った。

本研究により局面の評価値を求める評価関数が 3 つのパラメタ盤位置 BP,確定石 FS,候補数 CN を持つとき各パラメタに付加する重みは以下の値が最適であることが示された。

$$W_{BP}= 2,\ W_{FS}= 5,\ W_{CN}= 1$$

この結果は 4 章で推測された 4 つの推測

$$FS > BP > CN$$

$$1 \leqq W_{BP} \leqq 2,\ 3 \leqq W_{FS} \leqq 5$$

$$2 \leqq W_{BP} \leqq 5,\ 1 \leqq W_{CN} \leqq 2$$

$$1 \leqq W_{BP} \leqq 5,\ 0 < W_{CN} \leqq 1$$

をすべて満たす。

本研究では先読みなしにて検証を行ったため、評価関数の精度には限りがある。先読みと組み合わせて検証することが重要である。また他に挙げられたパラメタを利用することで精度があがるかもしれない。先読みをすることによって計算時間が長くなるため、いかに時間を短縮するか、今後の重要な課題である。

謝辞

本研究を行うにあたり、直接指導して頂いた近畿大学理工学部情報学科情報論理工学研究室石水講師には大変お世話になりました。日頃の研究に関する議論や研究のサポート、研究へのアドバイス、論文指導に対し適切なご助言と励ましを頂きましたので、ここに感謝の意を表します。

参考文献

[1] Seal software,リバーシのアルゴリズム C++&Java 対応,工学社(2003)

[2] Koso Sato, 評価関数を考える, プログラミングティーショップ(2003)
http://www.geocities.co.jp/SiliconValley-Bay/4543/Osero/Value/Value.html.

[3] 保田和隆, オセロ・リバーシプログラミング講座(2011)
http://uguisu.skr.jp/othello

[4] 大筆豊, オセロプログラムの評価関数の改善について, 情報処理学会研究報告 2004-G1-11, pp.15-20(2004)

[5] リバーシ/オセロ,http://www.mix-zone.net/

[6] Yuichi, Sundry Street(2012), http://www2u.biglobe.ne.jp/~yuichi/

[7] Joel Feinstein, Amenor Wins World 6x6 Championships!, Forty billion noted under the tree (July 1993), pp.6-8, British Othello Federation's newsletter., (1993), http://www.britishothello.org.uk/fbnall.pdf

[8] 谷田邦彦, 図解早わかりオセロ これが必勝のコツだ！！ , 日東書院, (2003).

[9] Richard Delrme, Ohello programing, (2012), http://abulmo.perso.neuf.fr/index.htm

[10] 石井隆, Master Reversi, (2011), http://homepage2.nifty.com/t_ishii/mr/index.html

[11] Gunnar Andersson, WZebra, (2006), http://radagast.se/othello/

[12] 橋本剛, 上田徹, 橋本隼一, オセロ求解へ向けた取り組み, 組合せゲーム・パズル ミニプロジェクト, 第3回ミニ研究会, (2008), http://www.lab2.kuis.kyoto-u.ac.jp/~itohiro/Games/Game080307.html#anchor

[13] 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).

[14] Tetsuya Nakajima, Othello!JAPAN(2013), http://www.othello.org/

[15] 作田 誠,人工知能 コンピュータゲームの実装:リバーシ(2012), http://www.ci.sys.fit.ac.jp/ai/

付録

以下に本研究で作成したリバーシのプログラムのソースを示す.

```
import java.util.Scanner;

public class Reversi {
	private final static int s = 5; // 先読み数
	private final int N = 8; // N目リバーシ
	private int[][] board = new int[N + 2][N + 2];
	private int[][][] bak = new int[99999][N + 2][N + 2];
	private boolean[][] pboard = new boolean[N + 2][N + 2];
	private int[][] pList;
	int pSize;
	final static int empty = 0;
	public final static int white = 1;
	final static int black = -1;
	final static int wall = 2;
	int turn = 1;
	int time = 0;

	// 評価マップ
	private final int[][] valueMap = {{  45, -11,   4,  -1,  -1,   4, -11,  45 },
	                                  { -11, -16,  -1,  -3,  -3,  -1, -16, -11 },
	                                  {   4,  -1,   2,  -1,  -1,   2,  -1,   4 },
	                                  {  -1,  -3,  -1,   0,   0,  -1,  -3,  -1 },
	                                  {  -1,  -3,  -1,   0,   0,  -1,  -3,  -1 },
	                                  {   4,  -1,   2,  -1,  -1,   2,  -1,   4 },
	                                  { -11, -16,  -1,  -3,  -3,  -1, -16, -11 },
	                                  {  45, -11,   4,  -1,  -1,   4, -11,  45 } };

	/*
	* boardをリセット
	*/
	public void resetBoard() {
		// wallとemptyの設定
		for (int y = 0; y < N + 2; y++) {
			for (int x = 0; x < N + 2; x++) {
				if (x == 0) {
```

```
                        board[y][x] = wall;
                } else if (x == N + 1) {
                        board[y][x] = wall;
                } else if (y == 0) {
                        board[y][x] = wall;
                } else if (y == N + 1) {
                        board[y][x] = wall;
                } else {
                        board[y][x] = empty;
                }
            }
        }
        // 初期石の設定
        board[N / 2][N / 2] = white;
        board[N / 2 + 1][N / 2 + 1] = white;
        board[N / 2 + 1][N / 2] = black;
        board[N / 2][N / 2 + 1] = black;
}

/**
 * 配置可能なマスか返す
 * @param x
 * @param y
 * @return pboard[y][x]
 */
public boolean isPossible(int x, int y) {
        setPossibleBoard();
        return pboard[y][x];
}

/**
 * 配置可能なマスが存在するかどうか
 */
public boolean isPossible() {
        setPossibleBoard();

        for (int y = 0; y < N + 2; y++) {
                for (int x = 0; x < N + 2; x++) {
                        if (pboard[y][x]) {
```

```
                        return pboard[y][x];
                    }
                }
            }
            return false;
        }

        /*
         * pSize を求める
         */
        public void setPSize() {
            pSize = 0;
            // 配置可能なマスの個数を求める
            for (int y = 0; y < N + 2; y++) {
                for (int x = 0; x < N + 2; x++) {
                    if (pboard[y][x]) {
                        pSize += 1;
                    }
                }
            }
        }

        public int getPSize() {
            return pSize;
        }

        /*
         * 配置可能なマスをすべて探し, pboard に True,False を入れる
         */
        private void setPossibleBoard() {
            // pBoard の初期化(全て false)
            for (int y = 0; y < N + 2; y++) {
                for (int x = 0; x < N + 2; x++) {
                    pboard[y][x] = false;
                }
            }

            // 判断基準を自分に合わせる　判断基準:piece
            // 相手の基準:cPiece
```

```
int piece = getTurn();
int cPiece = getTurn() * (-1);
// System.out.println("自分:" + piece + ",相手:" + cPiece);
// 探索変数
int sX, sY;
// 判断基準と同一のマスから
// 八方に配置可能なマスを探索する
for (int y = 0; y < N + 2; y++) {
        for (int x = 0; x < N + 2; x++) {
                if (board[y][x] == piece) {
                        // 左上方向を探索 x--,y--
                        sX = x;
                        sY = y;
                        if (board[sY - 1][sX - 1] == cPiece) {
                                do {
                                        sX--;
                                        sY--;
                                } while (board[sY][sX] == cPiece);
                                if (board[sY][sX] == empty) {
                                        pboard[sY][sX] = true;
                                }
                        }
                        // 上方向を探索 y--
                        sX = x;
                        sY = y;
                        if (board[sY - 1][sX] == cPiece) {
                                do {
                                        sY--;
                                } while (board[sY][sX] == cPiece);
                                if (board[sY][sX] == empty) {
                                        pboard[sY][sX] = true;
                                }
                        }
                        // 右上方向を探索 x++,y--
                        sX = x;
                        sY = y;
                        if (board[sY - 1][sX + 1] == cPiece) {
                                do {
                                        sX++;
```

```
				sY--;
			} while (board[sY][sX] == cPiece);
			if (board[sY][sX] == empty) {
				pboard[sY][sX] = true;
			}
		}
		// 左方向を探索 x--
		sX = x;
		sY = y;
		if (board[sY][sX - 1] == cPiece) {
			do {
				sX--;
			} while (board[sY][sX] == cPiece);
			if (board[sY][sX] == empty) {
				pboard[sY][sX] = true;
			}
		}
		// 右方向を探索 x++
		sX = x;
		sY = y;
		if (board[sY][sX + 1] == cPiece) {
			do {
				sX++;
			} while (board[sY][sX] == cPiece);
			if (board[sY][sX] == empty) {
				pboard[sY][sX] = true;
			}
		}
		// 左下方向を探索 x--,y++
		sX = x;
		sY = y;
		if (board[sY + 1][sX - 1] == cPiece) {
			do {
				sX--;
				sY++;
			} while (board[sY][sX] == cPiece);
			if (board[sY][sX] == empty) {
				pboard[sY][sX] = true;
			}
```

```
				}
				// 下方向を探索 y++
				sX = x;
				sY = y;
				if (board[sY + 1][sX] == cPiece) {
					do {
						sY++;
					} while (board[sY][sX] == cPiece);
					if (board[sY][sX] == empty) {
						pboard[sY][sX] = true;
					}
				}
				// 右下方向を探索 x++,y++
				sX = x;
				sY = y;
				if (board[sY + 1][sX + 1] == cPiece) {
					do {
						sX++;
						sY++;
					} while (board[sY][sX] == cPiece);
					if (board[sY][sX] == empty) {
						pboard[sY][sX] = true;
					}
				}
			}
		}
	}
}

/**
 * @return turn
 */
public int getTurn() {
	return turn;
}

/*
 * 盤面を表示 white：○ black：● empty:□
 */
```

```
public void printBoard() {
        for (int i = 1; i <= N; i++) {
                System.out.print(" " + i);
        }
        System.out.println();
        for (int y = 1; y < N + 1; y++) {
                System.out.print(y);
                for (int x = 1; x < N + 1; x++) {
                        if (board[y][x] == white) {
                                System.out.print("○");
                        } else if (board[y][x] == black) {
                                System.out.print("●");
                        } else {
                                System.out.print("□");
                        }
                }
                System.out.println();
        }
}

/*
 * 現在の盤面に配置可能なマスを加え表示
 */
public void printBoardPlusP() {
        for (int i = 1; i <= N; i++) {
                System.out.print(" " + i);
        }
        System.out.println();
        for (int y = 1; y < N + 1; y++) {
                System.out.print(y);
                for (int x = 1; x < N + 1; x++) {
                        if (board[y][x] == white) {
                                System.out.print("○");
                        } else if (board[y][x] == black) {
                                System.out.print("●");
                        } else if (isPossible(x, y)) {
                                if (getTurn() == white) {
                                        System.out.print("☆");
                                } else {
```

```
                    System.out.print("★");
                }
            } else {
                System.out.print("□");
            }
        }
        System.out.println();
    }
}

/*
 * マスを反転
 */
public void reversiPiece(int[] input) {
    int ix = input[0];
    int iy = input[1];

    board[iy][ix] = getTurn();
    int piece = getTurn();
    int cPiece = getTurn() * (-1);
    int sx, sy;

    // 左上探索
    if (board[iy - 1][ix - 1] == cPiece) {
        sx = ix - 1;
        sy = iy - 1;
        do {
            sx--;
            sy--;
        } while (board[sy][sx] == cPiece);
        if (board[sy][sx] == piece) {
            sx = ix - 1;
            sy = iy - 1;
            do {
                board[sy][sx] = piece;
                sx--;
                sy--;
            } while (board[sy][sx] == cPiece);
        }
```

```
}
// 上探索
if (board[iy - 1][ix] == cPiece) {
        sx = ix;
        sy = iy - 1;
        do {
                sy--;
        } while (board[sy][sx] == cPiece);
        if (board[sy][sx] == piece) {
                sx = ix;
        sy = iy - 1;
                do {
                        board[sy][sx] = piece;
                        sy--;
                } while (board[sy][sx] == cPiece);
        }
}
// 右上探索
if (board[iy - 1][ix + 1] == cPiece) {
        sx = ix + 1;
        sy = iy - 1;
        do {
                sx++;
                sy--;
        } while (board[sy][sx] == cPiece);
        if (board[sy][sx] == piece) {
                sx = ix + 1;
                sy = iy - 1;
                do {
                        board[sy][sx] = piece;
                        sx++;
                        sy--;
                } while (board[sy][sx] == cPiece);
        }
}
// 左探索
if (board[iy][ix - 1] == cPiece) {
        sx = ix - 1;
        sy = iy;
```

```
		do {
			sx--;
		} while (board[sy][sx] == cPiece);
		if (board[sy][sx] == piece) {
			sx = ix - 1;
			sy = iy;
			do {
				board[sy][sx] = piece;
				sx--;
			} while (board[sy][sx] == cPiece);
		}
	}
	// 右探索
	if (board[iy][ix + 1] == cPiece) {
		sx = ix + 1;
		sy = iy;
		do {
			sx++;
		} while (board[sy] [sx] == cPiece);
		if (board[sy][sx] == piece) {
			sx = ix + 1;
			sy = iy;
			do {
				board[sy][sx] = piece;
				sx++;
			} while (board[sy][sx] == cPiece);
		}
	}
	// 左下探索
	if (board[iy + 1][ix - 1] == cPiece) {
		sx = ix - 1;
		sy = iy + 1;
		do {
			sx--;
			sy++;
		} while (board[sy][sx] == cPiece);
		if (board[sy][sx] == piece) {
			sx = ix - 1;
			sy = iy + 1;
```

```
				do {
					board[sy][sx] = piece;
					sx--;
					sy++;
				} while (board[sy][sx] == cPiece);
			}
		}
		// 下探索
		if (board[iy + 1][ix] == cPiece) {
			sx = ix;
			sy = iy + 1;
			do {
				sy++;
			} while (board[sy][sx] == cPiece);
			if (board[sy][sx] == piece) {
				sx = ix;
				sy = iy + 1;
				do {
					board[sy][sx] = piece;
					sy++;
				} while (board[sy][sx] == cPiece);
			}
		}
		// 右下探索
		if (board[iy + 1][ix + 1] == cPiece) {
			sx = ix + 1;
			sy = iy + 1;
			do {
				sx++;
				sy++;
			} while (board[sy][sx] == cPiece);
			if (board[sy][sx] == piece) {
				sx = ix + 1;
				sy = iy + 1;
				do {
					board[sy][sx] = piece;
					sx++;
					sy++;
				} while (board[sy][sx] == cPiece);
```

```
				}
			}
		}

		/*
		 * ターン交代 A
		 */
		public void turnChange() {
			turn *= (-1);
		}

		/*
		 * ターン交代 B
		 */
		public void consectiveTurnChange() {
			time += 1;
			turn *= (-1);
		}

		/*
		 * タイムリセット
		 */
		public void timeReset() {
			time = 0;
		}

		/*
		 * 2 回以上ターン交代 B をしていないか していたら false
		 */
		public boolean timeWatcher() {
			if (time >= 2) {
				return false;
			} else {
				return true;
			}
		}

		/*
		 * White を数える
```

```
 */
public int countWhite() {
        int count = 0;
        for (int y = 0; y < N + 2; y++) {
                for (int x = 0; x < N + 2; x++) {
                        if (board[y][x] == white) {
                                count += 1;
                        }
                }
        }
        return count;
}

/*
 * black を数える
 */
public int countBlack() {
        int count = 0;
        for (int y = 0; y < N + 2; y++) {
                for (int x = 0; x < N + 2; x++) {
                        if (board[y][x] == black) {
                                count += 1;
                        }
                }
        }
return count;
}

/**
 * 配置可能な座標を pList に格納
 */
public void setPList() {
        setPossibleBoard();
        setPSize();
        pList = new int[pSize][2];
        int i = 0;
        // 配置可能なマスをリストアップする
        for (int y = 0; y < N + 2; y++) {
                for (int x = 0; x < N + 2; x++) {
```

```
                        if (pboard[y][x]) {
                                pList[i][0] = x;
                                pList[i][1] = y;
                                i++;
                        }
                }
        }
}

/**
 * Playerの動作
 *
 * @return
 */
public int[] player() {
        int[] input = new int[2];
        int inputX;
        int inputY;
        Scanner kbs = new Scanner(System.in);

        do {
                System.out.print("x:");
                inputX = kbs.nextInt();
                if (inputX <= 0 || inputX >= 9) {
                        System.out.println("xは1-8で入力してください");
                }
        } while (inputX <= 0 || inputX >= 9);
        do {
                System.out.print("y:");
                inputY = kbs.nextInt();
                if (inputY <= 0 || inputY >= 9) {
                        System.out.println("yは1-8で入力してください");
                }
        } while (inputY <= 0 || inputY >= 9);
        input[0] = inputX;
        input[1] = inputY;
        return input;
}
```

```
/**
 * 盤面評価のみ 評価値の高い座標を返すコンピュータ
 * @return (x, y)
 */
public int[] valueMapComputer() {
	setPList();

	int MaxValue[] = { -999999, -1 };
	int value;

	for (int i = 0; i < pSize; i++) {
		value = evaluateMap(pList[i]);
		if (MaxValue[0] < value) {
			MaxValue[0] = value;
			MaxValue[1] = i;
		}
	}
	return pList[MaxValue[1]];
}

/**
 * 確定石のみ 確定石が0のときランダム 評価値の高い座標を返すコンピュータ
 * @return (x, y)
 */
public int[] valueFinalComputer() {
	setPList();

	int MaxValue[] = { -999999, -1 };
	int value;
	int zero = 0;
	for (int i = 0; i < pSize; i++) {
		value = evaluateFinal(pList[i]);
		zero += value;
		if (MaxValue[0] < value) {
			MaxValue[0] = value;
			MaxValue[1] = i;
		}
	}
```

```
		if (zero == 0) {
			return pList[(int) Math.floor(Math.random() * (pSize))];
		}
		return pList[MaxValue[1]];
	}

	/**
	 * 候補数のみ 評価値の高い座標を返すコンピュータ
	 *
	 * @return (x, y)
	 */
	public int[] valueCNComputer() {
		setPList();

		int MaxValue[] = { -999999, -1 };
		int value;

		for (int i = 0; i < pSize; i++) {
			value = evaluateCN(pList[i]);
			if (MaxValue[0] < value) {
				MaxValue[0] = value;
				MaxValue[1] = i;
			}
		}
		return pList[MaxValue[1]];
	}

	/**
	 * 盤面評価と確定石 評価値の高い座標を返すコンピュータ
	 * @return (x, y)
	 */
	public int[] valueMapFinalComputer(int a, int b) {
		setPList();

		int MaxValue[] = { -999999, -1 };
		int value;

		for (int i = 0; i < pSize; i++) {
			value = evaluateMapFinal(pList[i], a, b);
```

```
				if (MaxValue[0] < value) {
					MaxValue[0] = value;
					MaxValue[1] = i;
				}
			}
			return pList[MaxValue[1]];
		}

		/**
		 * 盤面評価と候補数 評価値の高い座標を返すコンピュータ
		 * @return (x, y)
		 */
		public int[] valueMapCNComputer(int a, int b) {
			setPList();

			int MaxValue[] = { -999999, -1 };
			int value;

			for (int i = 0; i < pSize; i++) {
				value = evaluateMapCN(pList[i], a, b);
				if (MaxValue[0] < value) {
					MaxValue[0] = value;
					MaxValue[1] = i;
				}
			}
			return pList[MaxValue[1]];
		}

		/**
		 * 確定石と候補数 評価値の高い座標を返すコンピュータ
		 * @return (x, y)
		 */
		public int[] valueFinalCNComputer(int a, int b) {
			setPList();

			int MaxValue[] = { -999999, -1 };
			int value;

			for (int i = 0; i < pSize; i++) {
```

```
			value = evaluateFinalCN(pList[i], a, b);
			if (MaxValue[0] < value) {
				MaxValue[0] = value;
				MaxValue[1] = i;
			}
		}
		return pList[MaxValue[1]];
	}

	/**
	 * 盤面評価と確定石と候補数 評価値の高い座標を返すコンピュータ
	 * @return (x, y)
	 */
	public int[] valueMapFinalCNComputer(int a,int b,int c) {
		setPList();

		int MaxValue[] = { -999999, -1 };
		int value;

		for (int i = 0; i < pSize; i++) {
			value = evaluateMapFinalCN(pList[i],a,b,c);
			if (MaxValue[0] < value) {
				MaxValue[0] = value;
				MaxValue[1] = i;
			}
		}
		return pList[MaxValue[1]];
	}

	public int evalueateboard() {
		int value = 0;
		for (int y = 1; y < N + 1; y++) {
			for (int x = 1; x < N + 1; x++) {
				if (board[y][x] != 0) {
					value += (board[y][x] * valueMap[y - 1][x - 1]) * getTurn();
				}
			}
		}
		return value;
```

```
}

/**
 * 引数で与えた座標においた時の評価値を返す 盤面評価
 * @param piece
 *            ={x,y}
 * @return value
 */
public int evaluateMap(int[] piece) {
         int value = 0;
         int[][] boardbak = new int[N + 2][N + 2];
         int myTurn = getTurn();

         // board のバックアップ作成
         copyBoard(board, boardbak);

         // マスの反転
         reversiPiece(piece);
         // 評価値計算

         // 盤面評価
         for (int y = 1; y < N + 1; y++) {
                  for (int x = 1; x < N + 1; x++) {
                           if (board[y][x] != 0) {
                                    value += (board[y][x] * (valueMap[y - 1][x - 1]
                                             + (int) Math.floor(Math.random() * 3)));
                           }
                  }
         }
         if (myTurn == -1) {
                  value *= -1;
         }

         // board の復元
         copyBoard(boardbak, board);

         // System.out.printf("(%d , %d ) %d ¥n", piece[0], piece[1], value);
         return value;
}
```

```
/**
 * 引数で与えた座標においた時の評価値を返す 確定石
* @param piece
 *            ={x,y}
 * @return value
 */
public int evaluateFinal(int[] piece) {
         int value = 0;
         int[][] boardbak = new int[N + 2][N + 2];

         // boardのバックアップ作成
         copyBoard(board, boardbak);

         // マスの反転
         reversiPiece(piece);

         // 評価値計算
         // 確定石
         value = evaluateFinalStone();

         // boardの復元
         copyBoard(boardbak, board);

         // System.out.printf("(%d , %d ) %d ¥n", piece[0], piece[1], value);
         return value;
}

/**
 * 引数で与えた座標においた時の評価値を返す 盤面評価と確定石
 * @param piece
 *            ={x,y}
 * @return value
 */
public int evaluateMapFinal(int[] piece, int a, int b) {
         int value = 0;
         int[][] boardbak = new int[N + 2][N + 2];
         int myTurn = getTurn();
         int FS = 0, BP = 0;
```

```
		// boardのバックアップ作成
		copyBoard(board, boardbak);

		// マスの反転
		reversiPiece(piece);
		// 評価値計算

		// 盤面評価
		for (int y = 1; y < N + 1; y++) {
			for (int x = 1; x < N + 1; x++) {
				if (board[y][x] != 0) {
					BP += (board[y][x] * (valueMap[y - 1][x - 1]
						+ (int) Math.floor(Math.random() * 3)));
				}
			}
		}
		if (myTurn == -1) {
			BP *= -1;
		}
		// 確定石
		FS = evaluateFinalStone();

		// boardの復元
		copyBoard(boardbak, board);

		value = (a * BP) + (b * FS);
		// System.out.printf("(%d , %d ) %d ¥n", piece[0], piece[1], value);
		return value;
	}

	/**
	 * 評価値を計算 盤面評価と候補数
	 * @param piece
	 * @return
	 */
	public int evaluateMapCN(int[] piece, int a, int b) {
		int CN = 0;
		int[][] boardbak = new int[N + 2][N + 2];
		int myTurn = getTurn();
```

```
int value = 0, BP = 0;

// boardのバックアップ作成
copyBoard(board, boardbak);

// マスの反転
reversiPiece(piece);

// 盤面評価
for (int y = 1; y < N + 1; y++) {
        for (int x = 1; x < N + 1; x++) {
                if (board[y][x] != 0) {
                        BP += (board[y][x] * (valueMap[y - 1][x - 1]
                                + (int) Math.floor(Math.random() * 3)));
                }
        }
}
if (myTurn == -1) {
        BP *= -1;
}

// 仮にターンをチェンジ
turnChange();
// 仮のppList,ppSizeを設定
setPossibleBoard();
setPSize();
// 配置可能な石の数
CN = pSize + (int) Math.floor(Math.random() * 2);

// boardの復元
copyBoard(boardbak, board);

// ターンの復元
turnChange();
// ppList,ppSizeの復元
setPossibleBoard();
setPSize();

value = (BP * a) + (CN * 10 * b);
```

```
		return value;
	}

	/**
	 * 評価値を計算 確定石と候補数
	 * @param piece
	 * @return 評価値
	 */
	public int evaluateFinalCN(int[] piece, int a, int b) {
		int[][] boardbak = new int[N + 2][N + 2];
		int value = 0;
		int FS = 0, CN = 0;

		// board のバックアップ作成
		copyBoard(board, boardbak);

		// マスの反転
		reversiPiece(piece);
		FS = evaluateFinalStone();
		// 仮にターンをチェンジ
		turnChange();
		// 仮の ppList,ppSize を設定
		setPossibleBoard();
		setPSize();
		// 配置可能な石の数
		CN = pSize + (int) Math.floor(Math.random() * 2);

		// board の復元
		copyBoard(boardbak, board);

		// ターンの復元
		turnChange();
		// ppList,ppSize の復元
		setPossibleBoard();
		setPSize();

		value = (FS * 1) + (CN * 10 * b);
		return value;
```

```
}

/**
 * 評価値を計算しそれを返す。 盤面評価と確定石と候補数
 * @param piece
 * @return 評価値
 */
public int evaluateMapFinalCN(int[] piece,int a,int b,int c) {
	int[][] boardbak = new int[N + 2][N + 2];
	int myTurn = getTurn();
	int value = 0;
	int BP=0,FS=0,CN=0;

	// board のバックアップ作成
	copyBoard(board, boardbak);

	// マスの反転
	reversiPiece(piece);

	// 盤面評価
	for (int y = 1; y < N + 1; y++) {
		for (int x = 1; x < N + 1; x++) {
			if (board[y][x] != 0) {
				BP += ((board[y][x] * valueMap[y - 1][x - 1])
					+(int) Math.floor(Math.random() * 3));
			}
		}
	}
	if (myTurn == -1) {
		BP *= -1;
	}

	// 確定石
	FS = evaluateFinalStone();

	// 仮にターンをチェンジ
	turnChange();
	// 仮の ppList,ppSize を設定
	setPossibleBoard();
```

```
        setPSize();
        // 配置可能な石の数
        CN = pSize + (int) Math.floor(Math.random() * 2);

        // boardの復元
        copyBoard(boardbak, board);

        // ターンの復元
        turnChange();
        // ppList,ppSizeの復元
        setPossibleBoard();
        setPSize();

        value = (a*BP)+(b*FS)+(CN*10*c);

        return value;
}

/**
 * 引数で与えた座標においた時の評価値を返す 候補が一つにつき-10point 相手の候補がないときはそこを返す
 * @param piece
 *            ={x,y}
 * @return value
 */
public int evaluateCN(int[] piece) {
        int CN = 0;
        int[][] boardbak = new int[N + 2][N + 2];

        // boardのバックアップ作成
        copyBoard(board, boardbak);

        // マスの反転
        reversiPiece(piece);

        // 仮にターンをチェンジ
        turnChange();
        // 仮のppList,ppSizeを設定
        setPossibleBoard();
        setPSize();
```

```
		// 配置可能な石の数
		CN = pSize;

		// board の復元
		copyBoard(boardbak, board);

		// ターンの復元
		turnChange();
		// ppList,ppSize の復元
		setPossibleBoard();
		setPSize();
		return -((CN+(int) Math.floor(Math.random() * 2)) * 10);
	}

	/*
	 * 盤面をコピー ( a を b にコピー)
	 */
	public void copyBoard(int[][] a, int[][] b) {
		for (int y = 0; y < N + 2; y++) {
			for (int x = 0; x < N + 2; x++) {
				b[y][x] = a[y][x];
			}
		}
	}

	public int getNullBoard() {
		int num = 0;
		for (int y = 1; y < N + 2; y++) {
			for (int x = 1; x < N + 2; x++) {
				if (board[y][x] == 0) {
					num++;
				}
			}
		}
		return num;
	}

	/**
	 * 先読み数 s の評価コンピュータ
```

```
* minlevel と maxlevel でミニマックス法を実施
* @return 最も評価の高かった座標
*/
public int[] search_computer(int s) {
	int[] answer = new int[2];
	int value, value_max = -999999;
	// 配置可能な手を生成
	setPList();
	int[][] list = new int[pSize][2];
	for (int y = 0; y < pSize; y++) {
		for (int x = 0; x < 2; x++) {
			list[y][x] = pList[y][x];
		}
	}
	int size = pSize;
	int nullNum = getNullBoard();

	// 空きマスより先読み数が多いとき先読み数を空きマスの数に合わせる
	if (s > nullNum) {
		s = nullNum;
	}

	System.out.println(nullNum);
	for (int i = 0; i < size; i++) {
		copyBoard(board, bak[s]);
		reversiPiece(list[i]);
		turnChange();
		value = minlevel(s - 1);
		System.out.println("x:" + list[i][0] + ",y:" + list[i][1]
				+ ",value:" + value);
		turnChange();
		copyBoard(bak[s], board);
		if (value > value_max) {
			answer[0] = list[i][0];
			answer[1] = list[i][1];
		}
	}
	return answer;
}
```

```
public int maxlevel(int limit) {
        if (limit == 0) {
                return evalueateboard();
        }

        // 配置可能な手を生成
        setPList();
        int[][] list = new int[pSize][2];
        for (int y = 0; y < pSize; y++) {
                for (int x = 0; x < 2; x++) {
                        list[y][x] = pList[y][x];
                }
        }
        int score, score_max = -99999;

        for (int i = 0; i < list.length; i++) {
                copyBoard(board, bak[limit]);// バックアップ作成
                reversiPiece(list[i]);
                turnChange();
                score = minlevel(limit - 1);
                turnChange();
                copyBoard(bak[limit], board);// バックアップに戻す
                if (score > score_max) {
                        score_max = score;
                }
        }

        return score_max;
}

public int minlevel(int limit) {
        if (limit == 0) {
                return evalueateboard();
        }

        // 配置可能な手を生成
        setPList();
        int[][] list = new int[pSize][2];
```

```
		for (int y = 0; y < pSize; y++) {
			for (int x = 0; x < 2; x++) {
				list[y][x] = pList[y][x];
			}
		}
		int score, score_min = 99999;
		for (int i = 0; i < list.length; i++) {
			copyBoard(board, bak[limit]);// バックアップ作成
			reversiPiece(list[i]);
			turnChange();
			score = maxlevel(limit - 1);
			turnChange();
			copyBoard(bak[limit], board);// バックアップに戻す
			if (score < score_min) {
			score_min = score;
			}
		}
		return score_min;
	}

	public int[] randomComputer() {
		setPList();

		return pList[(int) Math.floor(Math.random() * (pSize))];
	}

	public int evaluateFinalStone() {
		int valueOfFinalStone = 0;
		int myTurn = getTurn();
		int i;
		boolean full;
		int stonePoint = 11;
		int HL = board[1][1], // 左上
		HR = board[1][N], // 右上
		LL = board[N][1], // 左下
		LR = board[N][N];// 右下
		// 四隅に１つ以上石があるか
		if (HL != empty || HR != empty || LL != empty || LR != empty) {
			/** 上辺 */
```

```
full = true;
for (int j = 0; j < N + 1; j++) {
        // empty があれば false
        if (board[1][j] == empty) {
                full = false;
        }
}

if (full) {// 全て埋まっている
        for (int j = 1; j < N + 1; j++) {
                valueOfFinalStone += ((board[1][j] * myTurn * stonePoint)
                        + (int) Math.floor(Math.random() * 3));
        }
} else {// 全ては埋まっていない
                // 左上は埋まっている
        if (HL != empty) {
                i = 1;
                while (board[1][i] == HL) {
                        valueOfFinalStone += ((board[1][i] * myTurn * stonePoint)
                                + (int) Math.floor(Math.random() * 3));
                        i++;
                }
        }
        // 右上は埋まっている
        if (HR != empty) {
                i = N;
                while (board[1][i] == HR) {
                        valueOfFinalStone += ((board[1][i] * myTurn * stonePoint)
                                 + (int) Math.floor(Math.random() * 3));
                        i--;
                }
        }
}
/** 下辺 */
full = true;
for (int j = 1; j < N + 1; j++) {
        // empty があれば false
        if (board[N][j] == empty) {
                full = false;
```

```
		}
	}
	if (full) {/* 全て埋まっている */
		for (int j = 1; j < N + 1; j++) {
			valueOfFinalStone += ((board[N][j] * myTurn * stonePoint)
				+ (int) Math.floor(Math.random() * 3));
		}
	} else {/* 全ては埋まっていない */
		// 左下は埋まっている
		if (LL != empty) {
			i = 1;
			while (board[N][i] == LL) {
				valueOfFinalStone += ((board[N][i] * myTurn * stonePoint)
				 + (int) Math.floor(Math.random() * 3));
				i++;
			}
		}
		// 右下は埋まっている
		if (LR != empty) {
			i = N;
			while (board[N][i] == LR) {
				valueOfFinalStone += ((board[N][i] * myTurn * stonePoint)
				 + (int) Math.floor(Math.random() * 3));
				i--;
			}
		}
	}
	/** 左辺 */
	full = true;
	for (int j = 1; j < N + 1; j++) {
		// emptyがあればfalse
		if (board[j][1] == empty) {
			full = false;
		}
	}
	if (full) {/* 全て埋まっている */
		for (int j = 1; j < N + 1; j++) {
			valueOfFinalStone += ((board[j][1] * myTurn * stonePoint)
				+ (int) Math.floor(Math.random() * 3));
```

```
		}
} else {/* 全ては埋まっていない */
		// 左上は埋まっている
		if (HL != empty) {
			i = 1;
			while (board[i][1] == HL) {
				valueOfFinalStone += ((board[i][1] * myTurn * stonePoint)
				 + (int) Math.floor(Math.random() * 3));
				i++;
			}
		}
		// 左下は埋まっている
		if (LL != empty) {
			i = N;
			while (board[i][1] == LL) {
			valueOfFinalStone += ((board[i][1] * myTurn * stonePoint)
				+ (int) Math.floor(Math.random() * 3));
				i--;
			}
		}
}
/** 右辺 */
full = true;
for (int j = 1; j < N + 1; j++) {
		// empty があれば false
		if (board[j][N] == empty) {
			full = false;
		}
}
if (full) {/* 全て埋まっている */
		for (int j = 1; j < N + 1; j++) {
			valueOfFinalStone += ((board[j][N] * myTurn * stonePoint)
				+ (int) Math.floor(Math.random() * 3));
		}
} else {/* 全ては埋まっていない */
		// 右上は埋まっている
		if (HR != empty) {
			i = 1;
			while (board[i][N] == HR) {
```

```
						valueOfFinalStone += ((board[i][N] * myTurn * stonePoint)
						 + (int) Math.floor(Math.random() * 3));
						i++;
					}
				}
				// 右下は埋まっている
				if (LR != empty) {
					i = N;
					while (board[i][1] == LR) {
						valueOfFinalStone += ((board[i][N] * myTurn * stonePoint)
						 + (int) Math.floor(Math.random() * 3));
						i--;
					}
				}
			}
		}
		valueOfFinalStone -= ((HL + HR + LL + LR) * 10);
		return valueOfFinalStone;
	}

/*****************************************************************************************************/
	/**
	 * randomComputer ランダム valueMapComputer 盤面評価 valueFinalComputer 確定石
	 * valueCNComputer 候補数 valueMapFinalComputer 盤面評価と確定石 valueMapCNComputer
	 * 盤面評価と候補数 valueFinalCNComuter 確定石と候補数
	 */

	public static void main(String[] args) {
		Date before = new Date();
		Reversi board = new Reversi();
		int input[];

		for (int a = 1; a < 6; a++) {
			for (int b = 1; b < 6; b++) {
			/*
				for (int c = 1; c < 6; c++) {
			*/
				int Win = 0, False = 0, Drow = 0;
```

```
// System.out.printf("Value:%s,Random:%s¥n",
// (board.getTurn() == white) ? "○" : "●",
// (board.getTurn() == white) ? "●" : "○");
// board.printBoard();
for (int i = 0; i < 1000; i++) {
        board.resetBoard();
        do {
                while (board.isPossible()) {
                        board.timeReset();
                        // System.out.printf("%s のターン¥n",
                        // (board.getTurn() == white) ? "Value(○)"
                        // : "Random(●)");
                        // board.printBoardPlusP();
                        do {
                                // 先手
                                // ValueComputer turn
                                if (board.getTurn() == white) {
                                // input =
                                // board.valueMapFinalComputer();
                                        input
                                        = board.randomComputer();
                                        // RandomComputer turn
                                } else {
                                        // 後手
                                        input
                                        = board.valueMapCNComputer
                                                (a, b);
                                        // System.out.printf
                                        //      ("x:%d, y:%d¥n",
                                        // input[0],
                                        // input[1]);
                                }

                                if(!board.isPossible(input[0], input[1])) {
                                        System.out.println
                                        ("その座標は打てません.");
                                }
                        } while (!(board.isPossible(input[0], input[1])));
                        board.reversiPiece(input);
```

```
								board.turnChange();
							}
							// System.out.printf
							//("%s は打てる場所がありません。¥n",(board.getTurn()
							// ==
							// white) ? "Value(○)" : "Random(●)");
							board.consectiveTurnChange();
						} while (board.timeWatcher());
						// System.out.println("GameOver");
						// board.printBoard();
						// System.out.printf("Value(%s):%d,Random(%s):%d¥n",
						// (board.getTurn() == white) ? "○" : "●",
						// board.countWhite(),
						// (board.getTurn() == white) ? "●" : "○",
						// board.countBlack());

						if (board.countWhite() > board.countBlack()) {
							Win += 1;
							// System.out.println("win");
						} else if (board.countWhite() < board.countBlack()) {
							False += 1;
							// System.out.println("false");
						} else {
							Drow += 1;
						}
					}
					System.out.println(/* "["+a+":"+b+"]"¥t"+ */False + "¥t"
							+ Win + "¥t" + Drow);
				}
			}
			/*
		}
		*/
	}
}
```