卒業研究報告書

題目

盤面の評価値によるオセロプログラム

指導教員

石水　隆　講師

報告者

10-1-037-0140

沖垣　駿

近畿大学理工学部情報学科

平成 27 年 1 月 31 日提出

概要

オセロは、8×8＝64 のマス目で構成された盤面を用い、交互に白と黒の駒を打ち、相手の駒を挟み、挟まれると裏返し、全盤面が埋まったとき駒数が多いほうが勝利となる単純明快なゲームで、また奥深いゲーム性を持ち世界に広まった。また、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。思考手順を数え、その手順に従い最もよい手を計算させる。

本研究では、オセロにおける盤面の評価関数を変えたとき、勝率がどのように変動するかを観測し最もよい組み合わせを求める。

目次

1序論

1.1 二人零和有限確定完全情報ゲーム

将棋やチェスなど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、二人零和有限確定完全情報ゲームとは、二人あるいは二人以上のチームでゲームを行い、ゲーム終了時お互いのプレイヤーの利得合計が零、お互い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、各プレイヤーの着手の意思決定の情報を知ることが可能なゲームである。二人零和有限確定完全情報ゲームに分類され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、お互いのプレイヤーが最善手を打つこ

とにより、先手必勝か後手必勝か引き分けが決定する。ゲームの理論の中で二人零和有限確定完全情報ゲームは、最も単純なゲームといえ、ゲーム理論の研究の最初期から研究されてきた。現在では研究の中心はゲームの性質についての研究から、人工知能を用いた具体的なゲームにおける戦略の研究にその中心が移っている。

二人零和有限確定完全情報ゲームは双方最善手を打った場合、先手勝ち、後手勝ち、引き分けのど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大きいため、完全解析を行うことは不可能である。例を挙げれば、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おり、現在の計算機の性能を越えている。

オセロは 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とても少ないゲームである。だがしかし、それでも初期配置の中央 4 マスは白黒の二通り、他の 60 マスは白黒空の 3 通りであるので、可能な局面の組み合わせは大雑把に見積もって $2_4*3_{60}=6.78*10_{19}$ 通り存在する。このためオセロは現時点でスーパーコンピュータを駆使してもなお完全解析されていない。

一方、可能な局面数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[1]。チェッカーは双方最善手を指すと引き分けとなる[2]。

局面数が大きいゲームについては、ゲーム盤をより小さいサイズに限定した場合の解析も行われている。囲碁については、サイズ 4x4 の囲碁は双方最善手を打つと持碁(引き分け)[3]、5x5 の囲碁は黒の 25 目勝ちとなる[4]。オセロについては、J.Feinstein がサイズ 6x6 のミニオセロの完全解析を行い、双方最善手を打つと 16 対 20 で後手勝ちとなることを示した[5]。

1.2 ゲーム AI の手法

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、定跡・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。局面の評価値計算は、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値を計算させることにより着手と決定する手法である。定石データベースは、一つの流れで両者が最善の手を打ち、互角の分れを得る打ち方のデータを集め管理し、容易に探索･抽出などの再利用をできるようにしたものである。打ち方の勝敗のみを読みきることを必勝読み、石差まで読みきることを完全読みという。必勝読みの方が計算時間が少なくてもすむため、一般にまず必勝読みで勝ちを確定させた上で、残り手数が少なくなると完全読みに切り替えてより点数の高い勝ちを目指すことが多い。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を判定するという作業を数千～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。

以上の手法を用いることにより、完全解析を行わなくても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。

1.3 オセロ

オセロは 8x8 のマスを使用する。横に a～h、縦に 1～8 の座標が各マスにある。図 1 にオセロの盤面と石の初期配置を示す。石を打つとき、縦・横・斜め方向に相手色の石を自色で挟み、挟まれた石を自色に返す。相手の石を返すことができない升に石を打つことはできない。打てる升がない場合はパスとなり、パスの回数に制限はない。返せる石がある場合、パスをすることは認められない。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終了となる。判定は石の数が多いほうが勝利、同数の場合引き分けとなる。

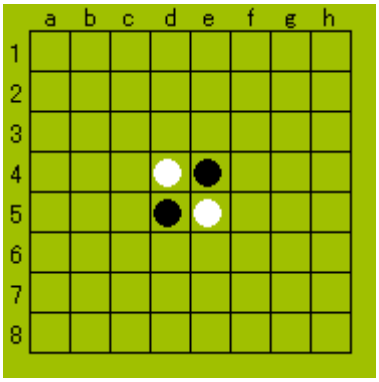

図 1 オセロの盤面と石の初期配置

1.4 オセロに関する既知の結果

本節ではオセロに関する既知の結果を示す。

1.4.1 オセロの定石

前節で述べたとおり、いまだオセロの完全解析はされていない。特に序盤においてどのような手が最善となるかは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やす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。図 2, 3, 4 に各定石を示す。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。中割りは、「周りが全て石に囲わ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ための手法である。引っ張りは、壁をつくり、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法である。図 5, 6 に中割り，引っ張りを示す。

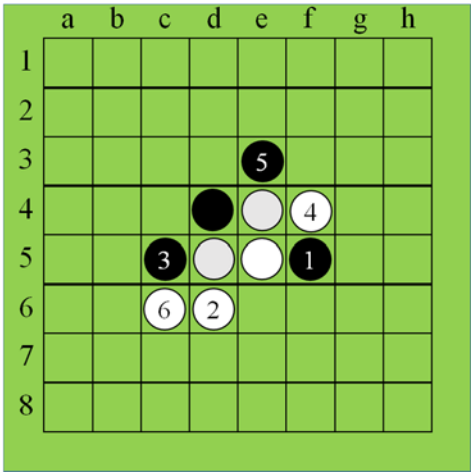

図 2　縦取り兎定石

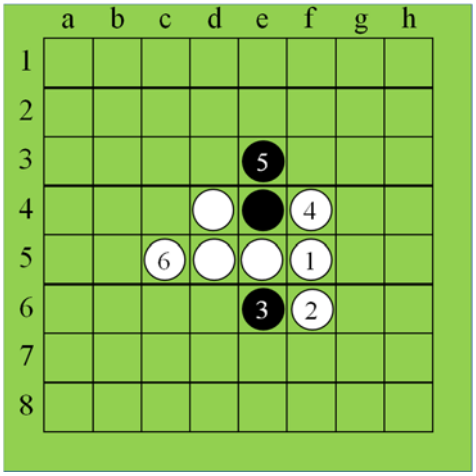

図 3　斜め取り牛定石

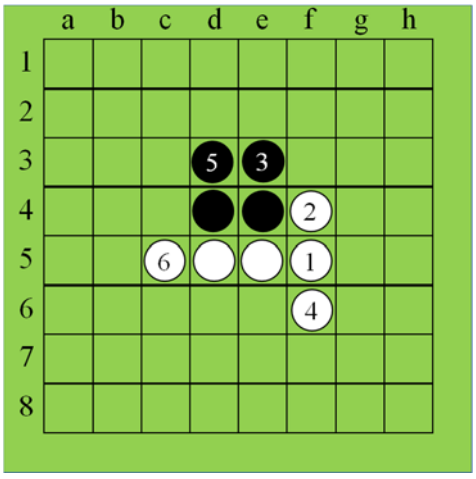

図 4　並び取り鼠定石

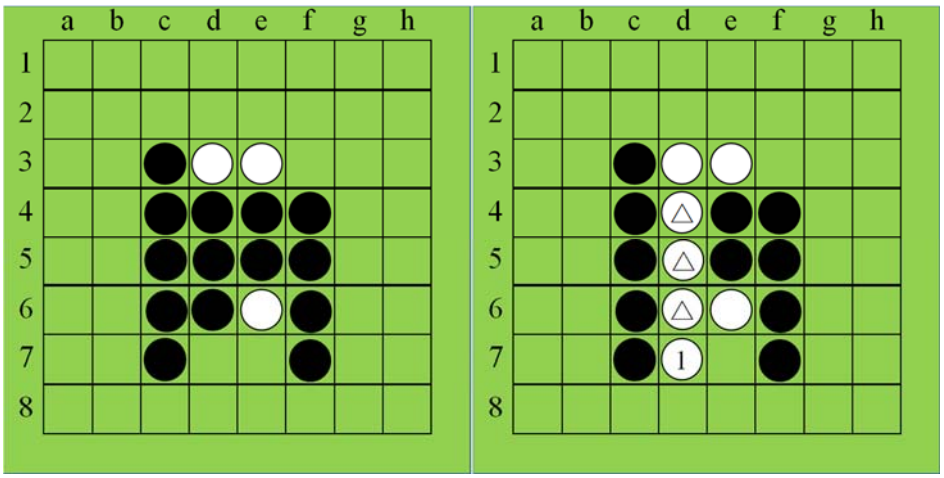

図 5　中割り

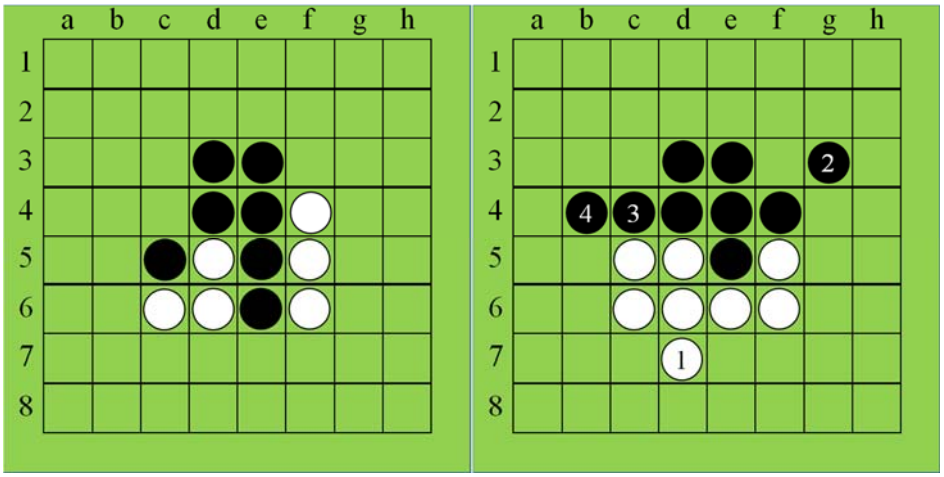

図 6　引っ張り

1.4.2 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、定石が存在しない局面となる。その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決定付けられない問題であり、様々な評価関数が考えられている。

1.4.3 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あげることができる。先読みの手数を増やす度、評価関数の精度はあがり、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、1.3 節で述べたように、先読み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。

1.4.4 既知のオセロプログラム

オセロプログラムは、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、もしくは各手法の組み合わせを使用することが多い。

手法として定石データベースのみを用いたプログラムとして、Logistello[7]がある。Logistello は 1997 年には村上健八段(当時の世界チャンピオン)に勝利しており[8]、定石データベースを使うことでそれなりの強さのプログラムとなることが示された。しかし、Logistello は定石データベースのみを使用するため、定石以外の手を打たれた場合には対処できない。また、その他に Zebra[9]や Edax[10]などもある。これらについても book と呼ばれる序盤・中盤の定石データや対戦データベースを使用しているが、中盤の先読みと終盤の完全読みを行っている。特に Edax はマルチコア対応で高速処理が可能となっているが、book 非使用の時は勝率が下がるというのが一般的な見方である[11]。

最新のオセロ AI は、オセロはルールが単純であるため、古くからプログラミングの教材として、あるいは実際の製品としてコンピュータ上で開発されてきた。1980 年には、家庭用ゲーム機である Atari2600 用のオセロが発売されている。また、アスキーによりオセロプログラムを対局させる「マイクロオセロリーグ」が企画され、その模様は記事として掲載された。1986 年には同社からオセロを題材とした思考ゲームのプログラミング解説書も出版された（森田ら（共著）『思考ゲームプログラミング』）。当初はコンピュータの性能が低かったため人間は容易にコンピュータに勝つことができた。しかし徐々に、特に終盤でコンピュータに読み切られて圧倒されるようになった。現在では、最高性能のオセロプログラムには人間はまず勝つことができず、世界チャンピオンさえも敗れている。

歴史順としては、探索空間の狭い順に、チェッカー・オセロ・チェスにおいてコンピュータが人間のチャンピオンに勝利し、現在将棋が焦点となっている。チェッカーについては双方の最善手が解明されている。

1.5.本研究の目的

オセロにおいて完全解析がなされていない為、最善手が存在しない。局面に評価をする評価関数をより有利にゲームを進めるために作成する。

1.6.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 章でオセロプログラムを説明し、次に第 3 章で本研究で用いた局面の評価関数について説明する。第 4 章で評価関数を用いた対戦結果を示す。第 5 章で結論と今後の課題を示す。

２　オセロプログラムの一般的手法

１章で述べたとおり、オセロはまだ完全解析はできていないため、常に最善な手を選ぶことはできない。そこで本章はオセロプログラムで用いられる一般的手法について述べる。

いま現在、強いと評価されているプログラムは、序盤は定石データベースにしたがって打ち、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オセロプログラムとなる。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、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。何回も対戦することにより、より精度が高い手段がわかる。対戦データベースを用いることにより、対戦経験が増えるにつれて、強くなる人工知能型のオセロプログラムができる。

2.2 先読みと評価関数

1.4.4 で述べたように、定石データベースは序盤のみ、完全読みは終盤のみ使用可能。そこで一般的には評価関数を用いて現在の局面、もしくは数手先の局面を評価できる。評価関数として、一般的に初心者でもやりやすい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。だが、盤面評価値では強いオセロプログラムを作成することはできない。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はできないからである。

また、ある局面の評価値を求める評価値は、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、その評価値をもとに、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には、まず現局面評価を行う必要がある。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標準基準については、先に記述した盤面評価、直線性、確定石、駒数、候補数などが提案されている。

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よいかははっきりしていない。本研究では、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、ひっくり返らな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)

盤位置での評価は 8×8 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その値を減算し、その合計値を盤位置の評価値とする．各マスの価値は色々なものが提案されており、本研究では図.1 に示している評価値を用いる．盤位置の評価値 BP は、以下の式で得られる．ただし、board(i,j)はマス(i,j)が自石なら 1、相手石なら-1,空きマスなら 0 とし、BP(i,j)は各マスの評価値である．

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

3.1.2 確定石(FS)

確定石は、絶対にひっくりかえらない石のことを指す。勝敗が決まるまで残るので確定石の数は、多い方が有利とされている。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であるため、4 つの辺における確定石のみで評価した。確定石の評価値 FS は以下の式で与えられる．

FS = (自分の確定石の数－相手の確定石の数) + rnd * 33

3.1.3 候補数(CN)

候補数は各局面での次に着手可能な手の数である.一般的に自分の手の候補数が多ければよく、相手の候補数がなければよいとされている．候補数の評価値 CN は以下の式で与えられる．

CN = (着手可能な候補数+rnd * 2) * 10

これらのパラメタにより、本研究で用いる評価関数 F は以下の式で与えられる．また、

Wbp, Wfs, Wcn は各重みのパラメタである.

F=BP* Wbp+FS*Wfs+CN*Wcn

100	-40	20	5	5	20	-40	100
-40	-80	-1	-1	-1	-1	-80	-40
20	-1	5	1	1	5	-1	20
5	-1	1	0	0	1	-1	5
5	-1	1	0	0	1	-1	5
20	-1	5	1	1	5	-1	20
-40	-80	-1	-1	-1	-1	-80	-40
100	-40	20	5	5	20	-40	100

図 7. 盤位置の評価

各パラメタに付加する重みの範囲は以下とする。

$0 \leqq Wbp \leqq 5 \qquad 0 \leqq Wfs \leqq 5 \qquad 0 \leqq Wcn \leqq 1$

3.2 オセロプログラム

本研究では 3.1 節で述べたように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せた重みを変化させて、計算機実験を行う。付録にて、作成したプログラムを示す。

3.2.1 クラス Osero

Osero 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う。

int[ ][ ] board が盤面の情報を記憶、int turn がどちらの番かを記憶、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手を決定するメソッドが以下の 7 つである。

int [ ] valueMapComputer(), int [ ] valueFinalComputer(), int [ ] valueCNComputer(), int [ ] MapFinalComputer, int [ ] valueMapCNComputer(),
int [ ] valueFinalCNComputer,(), int [ ] valueMapFinalCNComputer()

また、以下に評価値を計算する 3 つの関数について示す。

int evaluateBoard() 現局面において盤位置 BP を計算し、それを戻り値として持つ。

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() 現局面において着手可能なマスの数を計算し、それを戻り値として持つ。

3.3. 対戦条件

本研究では 3.1 節で示した通り評価関数の各パラメタに付加する最もよい重みの値を求める。そのため、各パラメタに付加された重みを変えた場合の勝率を求める。対戦相手は着手可能な手からランダムで手を打つコンピュータとする。対戦条件は以下に示す。

- 各重みにつき対戦回数は 500 回。
- 先手、後手の両方行う。

以上の条件で対戦をし、最もよいと思われる重みを求める。

4. 結果および考察

本章では、各パラメタの重みに対し、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結果、およびその結果から得られるパラメタの最もよい重みについて示す。

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結果を表 1 に示す。表 1 を見て分かるように、先手後手関係なく FS が最も効力があるとわかる。したがって、以下の式が予想される。

FS＞BP＞CN

4.2. BP と FS，BP と CN，FS と CN，BP と FS と CN の比較

BP と FS，BP と CN，FS と CN，BP と FS と CN を各々用いたときの結果を表 2，表 3，表 4，表 5 に、示す。

表1. 各パラメタの勝率

	先手			後手		
	勝	負	引き分	勝	負	引き分
BP	375	107	18	370	107	23
FS	416	70	14	410	77	13
CN	330	163	7	306	184	10

表2.　F=BP*Wbp+FS*Wfs

BP	FS	先手			後手		
		勝	負	引き分	勝	負	引き分
1	1	468	22	10	473	22	5
	2	485	10	5	484	10	6
	3	491	6	3	490	7	3
	4	490	7	3	489	7	4
	5	488	9	3	481	12	7
2	1	456	36	8	461	30	9
	2	478	18	4	435	50	15
	3	484	14	2	429	54	17
	4	490	8	2	468	23	9
	5	488	9	3	478	17	5
3	1	450	37	13	432	55	12
	2	470	24	6	477	18	5
	3	477	18	5	472	18	10
	4	485	12	3	483	10	7
	5	484	10	6	489	8	3
4	1	455	31	14	449	38	13
	2	465	30	5	432	48	20
	3	469	24	7	422	50	28
	4	471	23	6	438	40	22
	5	480	16	4	450	38	12
5	1	428	58	14	432	49	21
	2	448	39	13	443	48	9
	3	468	25	7	453	29	18
	4	470	21	9	419	62	9
	5	475	20	5	425	49	26

表3.　F=BP*Wbp+CN*Wcn

BP	CN	先手			後手		
		勝	負	引き分	勝	負	引き分
1	1	380	90	30	379	101	20
	2	332	150	18	341	140	19
	3	390	92	18	309	170	21
	4	290	180	30	260	207	33
	5	250	219	31	262	217	21
2	1	390	80	30	411	70	19
	2	381	98	21	376	109	15
	3	384	110	106	359	122	19
	4	305	127	68	348	125	27
	5	325	152	23	328	148	24
3	1	412	72	16	393	91	18
	2	399	88	13	387	98	15
	3	380	109	21	372	108	20
	4	342	131	27	373	110	17
	5	355	122	23	349	136	15
4	1	402	82	16	399	85	16
	2	389	89	22	408	72	20
	3	404	73	23	492	87	21
	4	320	129	51	376	105	19
	5	328	139	33	370	110	20
5	1	340	109	51	392	87	21
	2	419	60	21	403	80	17
	3	390	78	32	403	83	14
	4	384	94	78	389	99	10
	5	372	102	26	378	101	21

表4.　F=FS*Wfs+CN*Wcn

FS	CN	先手			後手		
		勝	負	引き分	勝	負	引き分
1	1	390	90	20	389	101	10
	2	362	120	18	341	140	19
	3	390	92	18	309	170	21
	4	290	180	30	260	207	33
	5	250	219	31	262	217	21
2	1	390	80	30	421	70	9
	2	381	98	21	376	109	15
	3	384	110	106	359	122	19
	4	305	127	68	348	125	27
	5	325	152	23	328	148	24
3	1	392	92	16	393	91	18
	2	399	88	13	387	98	15
	3	380	109	21	372	108	20
	4	342	131	27	353	130	17
	5	325	122	53	349	136	15
4	1	402	82	16	399	85	16
	2	389	89	22	408	72	20
	3	384	93	23	492	87	21
	4	320	129	51	376	105	19
	5	328	139	33	370	110	20
5	1	400	79	21	392	87	21
	2	419	60	21	403	80	17
	3	390	78	32	403	83	14
	4	384	94	78	361	129	10
	5	372	102	26	358	121	21

表 2～表 4 より以下の範囲が推測できる。

$1 \leqq Wbp \leqq 2,\ 3 \leqq Wfs \leqq 5 \quad 2 \leqq Wbp \leqq 5,\ 1 \leqq Wcn \leqq 2 \quad 1 \leqq Wbp \leqq 5,\ 0 < Wcn \leqq 1$

5. 結論・今後の課題

本研究では、オセロ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重みを変え、計算実験を行った。

本研究により、局面の評価値を求める評価関数が、盤位置 BP，確定石 FS，候補数 CN の 3 つを持つとき以下の値が最もよい各パラメタに付加する重みであることがわかった。

$Wbp = 2,\ Wfs = 5,\ Wcn = 1$

この結果は 4 節で述べた 4 つの推測である。

今後の課題は、今回実行したとき、結果が全部でるまで約 40 秒ほどかかるのでもっと早いプログラムを作成する必要がある。そのために、数式をすべて足し算にするのも一つの手だと思う。また、盤面の評価値のみでなくもっと他のところにも着目してより強いプログラムを作成していきたい。

謝辞

本研究を行うにあたって、近畿大学理工学部情報学科情報論理工学研究室の石水講師には大変お世話になりました。研究に関する指摘やサポート、アドバイスなど適切な指導していただいたことを、ここに感謝をこめて記します。

参考文献

[1]Janos Wagner and Istvan Virag, Solving renju, ICGA Journal, Vol.24, No.1, pp.30-35 (2001), http://www.sze.hu/~gtakacs/download/wagnervirag_2001.pdf

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

[3]清慎一, 川嶋俊：探索プログラムによる四路盤囲碁の解, 情報処理学会研究報告, GI 2000(98), pp.69--76 (2000), http://id.nii.ac.jp/1001/00058633/

[4]Eric C.D. van der Welf, H.Jaap van den Herik, and Jos W.H.M.Uiterwijk, Solving Go on Small Boards, ICGA Journal, Vol.26, No.2, pp.92-107 (2003).

[5]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

[6]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)

[7]Michael Buro,Logistello,https://skatgame.net/mburo/log.html,1997

[8]Gunnar Andersson, WZebra, 2006,http://radagast.se/othello/

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

[10]T.Ishii,MasterReversi,他アプリとの対局結果
http://homepage2.nifty.com/t_ishii/mr/gameresult.html,200

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

[12]オセロ（リバーシ）の作り方（アルゴリズム） ～石の位置による評価～(2001)
http://uguisu.skr.jp/othello/5-1.html

付録

以下に本研究で作成したオセロプログラムのソースを示す。

```
package sotsuron;

import java.sql.Date;
import java.util.Scanner;

        public class Osero {
          private final static int s = 5; // 先読み数
          private final static 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 = {{ 100, -40, 20, 5, 5, 20, -40, 100 },
            { -40, -80, -1, -1, -1, -1, -80, -40},
            { 20, -1, 5, 1, 1, 5, -1, 20 },
            { 5, -1, 1, 0, 0, 1, -1, 5 },
            { 5, -1, 1, 0, 0, 1, -1, 5 },
            { 20, -1, 5, 1, 1, 5, -1, 20 },
            { -40, -80, -1, -1, -1, -1, -80, -40},
            { 100, -40, 20, 5, 5, 20, -40, 100 }};
          /*
          * 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() {
   eturn turn;
/*
* 盤面を表示 white：○ black：● empty:□
*/

public void printBoard() {
   or (int i = 1; i <= N; i++) {
      stem.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
28
*/
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];
I t 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
42
* 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(N, N, N);
            Reverce board = new Reverce();
            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);
```

```
            }
            // 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);
            }
          }
          /*
        }
        */
        }

}
```