卒業研究報告書

題目

縮小版ニップの完全解析

指導教員

石水　研　講師

報告者

19–1–037–0143

山本 巧

近畿大学理工学部情報学科

令和５年２月１日提出

概要

円形オセロとも呼ばれるニップはオセロのバリエーションの一つである．ニップの基本的なルールはオセロと同じであるが，オセロと異なり，ニップは角がない円形の盤を使用する．そのため，最後の１マスで全ての石が置き換わる可能性があり，オセロよりも終局まで逆転が起きやすい特徴を持つ．ニップはオセロと同じく二人零和有限確定完全情報ゲームに分類され，理論上双方が最善手を打った場合に初期局面で勝敗が確定している．通常オセロ（８ｘ８盤）は可能な局面数が膨大なため，初期局面で勝敗はまだ解析が至っていない．一方，盤面を小さくしたミニオセロに対して，いくつかの盤面サイズは初期局面で勝敗が判明している．ニップは通常サイズ，縮小サイズどちらもまだ初期局面で解析が至っていない．

そこで本研究ではニップのゲーム AI を作成し，サイズを十分に小さくした盤面に対して完全解析による初期局面で勝敗を求め，完全解析が不可能なサイズの盤に対しては探索により初期局面が先手有利か後手有利かを検証する．

目 次

1　序論

1.1　本研究の背景

ニップは円形オセロとも呼ばれ，オセロのバリエーションの一つである．ニップの基本的なルールはオセロと同じであるが，ニップとオセロは使用する盤に違いがある．通常のオセロは正方形の盤面に石を並べていくが，ニップは円形の盤に石を置く．オセロは角の重要性が高く，角を獲れるかが勝敗にも直結してしまう．しかし，ニップは盤の外周が円形になっている [1][2]．そのため，角がないので終局までひっくり返すことが不可能なマスがない．最後の 1 マスに石を配置すると，全ての石が置き換わる可能性があり，オセロよりも終局まで逆転が起きやすい特徴を持つ．オセロや三目並べ，囲碁などのボードゲームと同じくニップも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は対戦人数が二人（二人），双方の得点を合計すると常に 0（零和），可能な局面数が有限（有限），ランダム性が無く（確定），ゲームの情報は全て公開されている（完全情報）ゲームのことである．この種類のゲームでは理論上，双方が最善手を打った場合に初期局面からすでに先手必勝・後手必勝・引き分けのいずれかが確定している．しかし，通常のオセロや囲碁では可能な局面数が膨大なため，初期局面の勝敗はまだ分かっていない．一方，ミニオセロやミニ囲碁などゲームの盤を小さくしたゲームや総局面数を少ないゲームに対して，全局面の勝敗を求めることができ，初期局面での勝敗が判明している．

1.2　二人零和有限確定完全情報ゲームの完全解析に関する既知の結果

二人零和有限確定完全情報ゲームは全局面を解析することができれば，最善手を打つができる．しかし，可能な局面数が膨大なボードゲームが多く，完全解析することは難しい．例としてはオセロが 10^{30} 通り，チェスが 10^{50} 通り，将棋が 10^{69} 通り，囲碁が 10^{170} 通りとされている．しかし，総局面数が少ないゲームは完全解析がされているものがある．連珠は双方最善手を打つと 47 手で先手が勝利，チェッカーは双方最善手を指すと引き分け，コネクト 4 は 41 手で先手が勝利となる [3][4][5]．双方最善手を打つと 5 x 5 のミニ囲碁は黒の 24 目勝ちとなる [6]．オセロは可能な局面数が膨大なため完全解析はされていないが、盤面をより小さいサイズに縮小したミニオセロでは完全解析されているものもある．表 1 にミニオセロに対する双方最善手を打った場合の既知の結果を示す [7][8][9]．表中の初期配置の交差とは黒石と白石が交差する配置，平行とは黒石と白石を平行に並べる配置を表す．また，交差 (1) と交差 (2) は盤面サイズが奇数のため対称性が異なる配置である．ニップについては現在のところ完全解析は行われておらず，小さなサイズに限定したミニニップについても完全解析されていない．

1.3　本研究の目的

1.2 節で述べた通り，ミニオセロではいくつかの縮小サイズに対して，完全解析が行われている．ニップは通常サイズ（８ｘ８盤）の可能な局面数が 10^{25} 通り（マスが初期配置を除き４８）と膨大であり，まだ完全解析はされていない．また，ニップの縮小サイズもまだ初期局面の完全解析はされていない．そこで，本研究ではサイズを十分に小さくした盤に対しては完全解析により初期局面での勝敗を求め，それが不可能なサイズの盤に対しては探索により初期局面が先手有利か後手有利かを検証する．また，解析と検証を行うためのゲーム AI プログラムを作成する．

表 1: ミニオセロの完全解析の関する既知の結果

４ｘ４	交差	後手必勝	3	11	[7] [9]
	平行	後手必勝	6	9	
４ｘ６	交差	先手必勝	20	4	
	平行	先手必勝	21	3	
４ｘ８	交差	先手必勝	28	0	
	平行	先手必勝	28	0	
４ｘ１０	交差	先手必勝	39	0	
	平行	先手必勝	32	0	
６ｘ６	交差	後手必勝	16	20	
	平行	後手必勝	17	9	

３ｘ３	交差 (1)	先手必勝	0	9	[8]
	交差 (2)	後手必勝	9	0	
４ｘ３	交差	先手必勝	12	0	
６ｘ３	交差	先手必勝	18	0	
５ｘ４	交差	先手必勝	18	2	
５ｘ５	交差 (1)	先手必勝	19	6	
	交差 (2)	先手必勝	22	2	
７ｘ４	交差	先手必勝	25	1	

1.4 本報告書の構成

本報告書の構成を以下に示す．2 章でニップについて，3 章で研究内容，4 章で解析と考察，5 章でプログラムの構造，6 章で結論と今後の課題について述べる．

2 ニップの概要

2.1 ニップについて

まずは，ニップというゲームについて解説する．1.1 節で述べた通り，ニップは盤面の外周が円形になっている．円形オセロとも呼ばれることから，盤面以外のルールは外周部分を除き，オセロと同じである． 通常のオセロは縦横斜め方向に自石で敵石を挟めばひっくり返す．ニップはそれに加えて円周上で挟んだ場合もひっくり返すことができる．他のルールは同じく，黒石が先行，パスは何回でも可（着手可能手がある場合はパス不可），終了条件は全てのマスを埋め尽くすこと，もしくは双方とも置けるマスが無いことである．

続いて，オセロとの大きな違いである外周部分についてである．オセロの外周は四隅にある角とそれに隣接する辺に石を置くことができれば，終局までひっくり返すことは不可能な確定石とすることができる．しかし，ニップでは外周は外周の石を全て埋めつくさない限りは確定石になることはない．外周の変化の例を図 1，図 2，図 3 の順に示す．外周の大半を黒石で埋めた状態で（図 1），外周の残り 1 マスに白石を置くと（図 2），外周の石が黒→白へと置き換わる（図 3）．この例から，外周を全て埋めるまで必ず有利とはならず，終局まで逆転が起きやすい．そして，勝敗が分かりにくいゲームである．

ニップの盤面と石の初期配置は図 4 に示す．本研究では，アプリケーションで見やすくするために簡素化した盤面を使用する．図 5 では図 4 の簡素化した盤面を示す．

縮小版の盤面は正方形版の図 6 と長方形版の図 7 に示す．図 6 と図 7 は一辺の長さが 4 を含む盤面（４ｘ４，４ｘ６，６ｘ４など）であり，角の 1 マスを削除した盤面とする．それ以外は角の 3 マスを削除した盤面とする．図 6，図 7 中の石を置いてあるマスは外周となるマスである．

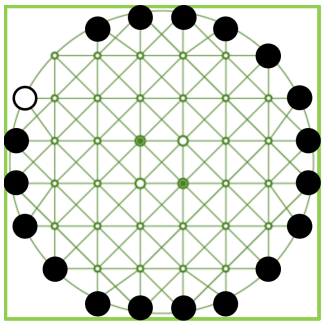

図 1: 外周の石の変化 (1)

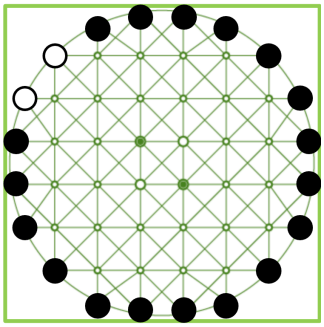

図 2: 外周の石の変化 (2)

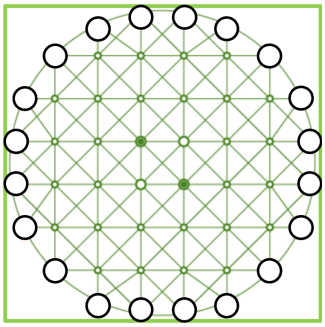

図 3: 外周の石の変化 (3)

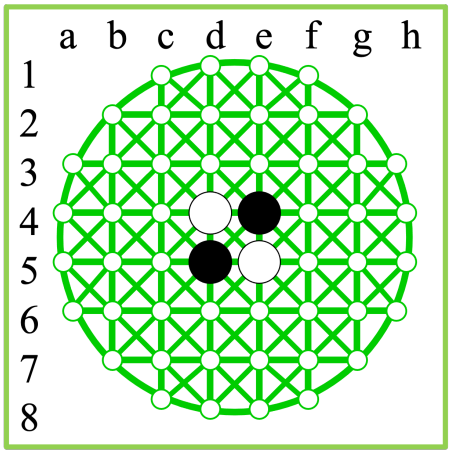

図 4: ニップの盤面＆初期配置

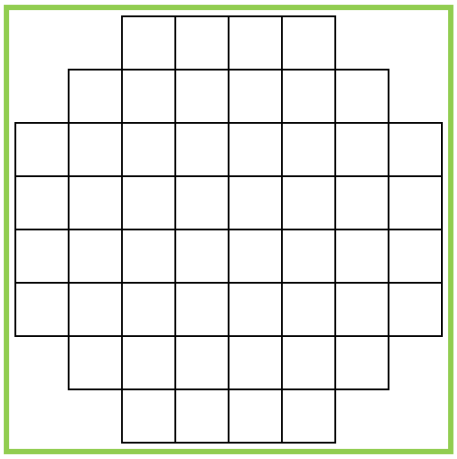

図 5: 8x8 の盤面 [簡素化版]

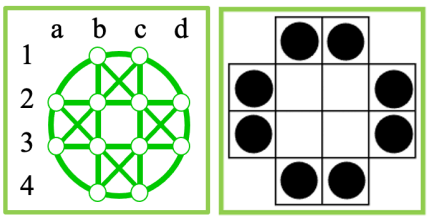

図 6: 4x4 の盤面 [盤面図・簡素版]

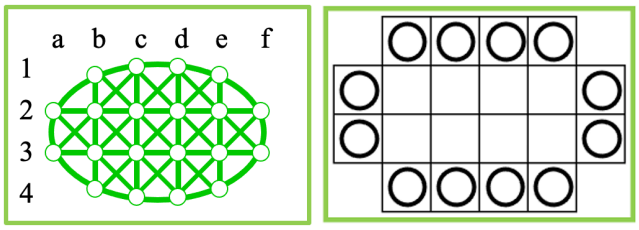

図 7: 4x6 の盤面 [盤面図・簡素版]

2.2　ニップの総局面数

オセロは，本来のサイズである８ｘ８盤の総局面数は 10^{30} 通りであり完全解析はされていない．一方、縮小サイズの６ｘ６盤の総局面数は 10^{17} 通りであり，完全解析が行われている．

ニップの本来のサイズである ８ｘ８ 盤の総局面数は 10^{25} 通りである．この総局面数は，オセロよりは少ないものの非常に大きい値であるため，８ｘ８のニップの場合でも完全解析を行うには難しいと思われる．一方，縮小サイズの４ｘ４や６ｘ６のサイズの場合はそれぞれ 10^{4} 通りと 10^{11} 通りであり，完全解析を行うことが可能な値である．

3　研究内容

本研究では２つの課題を行う．

- 解析＆検証を行うためのゲーム AI を作成する．
- 異なる盤サイズのニップの解析＆検証を行う．

3.1　探索アルゴリズム：アルファベータ法（Alpha-beta Pruning）

本研究では Java 言語を用いてアルファベータ法による探索で最善手を求めるゲーム AI を作成する．アルファベータ法はミニマックス法を改良した探索法である．ミニマックス法は全ての局面を探索し最善手を求められる．アルファベータ法は絶対に選ばれない局面を枝切りすることによりミニマックス法よりも短時間で最善手が求められる．

3.2　着手可能手の判断方法

ある局面において着手可能手の判断方法は以下の手順で探索を行う．

1. その局面における，盤面の空白のマスを判別する．
2. 空白のマスにおいて，隣接する全方向（8 方向）のマスが相手側のプレイヤーの石（以下他石とする）であるか探索する．

   2-1. 手順 2 で他石が存在した場合，手順 3 に進む．

   2-2. 手順 2 で他石が存在しない場合，手順 2 に戻る．
3. 他石がある方角へ，空白マス，手番側のプレイヤーの石（以下自石とする），盤外のいずれかにぶつかるまで探索する．

   3-1. 手順 3 で自石にぶつかって探索を終了した場合，手順 4 に進む．

   3-2. 手順 3 で空白マスまたは盤外にぶつかって探索を終了した場合，手順 2 に戻る．
4. 着手可能手の選択肢として追加する．

以上を 1 で判別した空白のマスがなくなるまで繰り返すことで着手可能手を判断する．着手可能手が得られない場合は打つ手なしと判断し，パスとなる．

3.3　初手着手可能手の縮小

盤のサイズが大きくなれば，探索する局面数も膨大になる．そこで，少しでも探索不要な局面を削減するため，初期局面の着手可能手数を縮小する．４ｘ４の通常盤と通常盤を回転させた盤を図 8 に示す．図 8 の通常盤より，先手が黒なので，着手可能手はマスの番号：2・5・12・15 の 4 通りである．例えば，初手をマスの番号：2 に打つ場合の盤と同じように残り 3 通りを表す．

- 180 度回転させた盤（初手をマスの番号：15）
- 各左右に 90 度回転させ上下対称にした盤（初手をマスの番号：5・12）

この 3 通りが同じ盤になることが分かる．図 8 の回転させた盤より，初手着手可能手を 1/4 探索するのみで残り 3/4 は探索不要となる．条件としては縦と横の長さが同じ正方形版であり，中央に初期配置が設置できる時に限る．

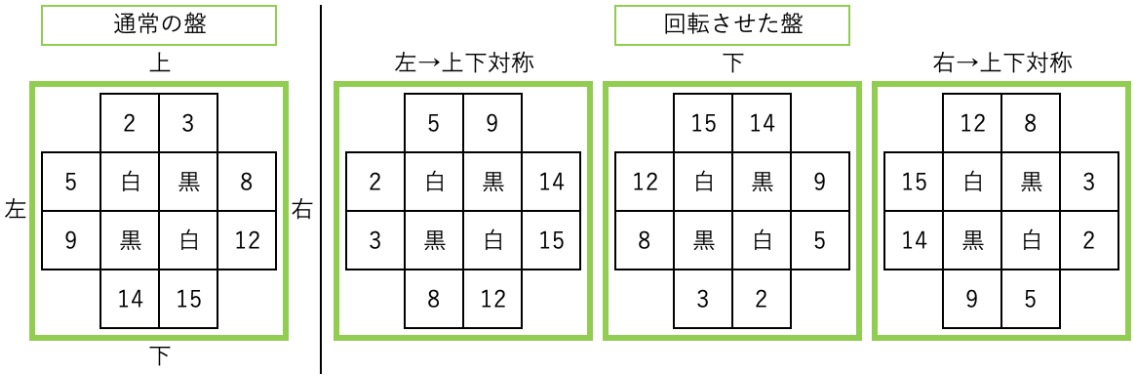

図 8: 通常の盤 (４ｘ４盤) &回転させた盤 (４ｘ４盤)

次に，縦と横の長さが異なる盤については４ｘ６の通常盤と 180 度回転させた盤を図 9 に示す．図 9 より,「180 度回転させた盤が同じ盤」になることが分かる．縦と横の長さが異なる長方形版の場合は初手着手可能手の 1/2 探索し，残り 1/2 は探索不要となる．

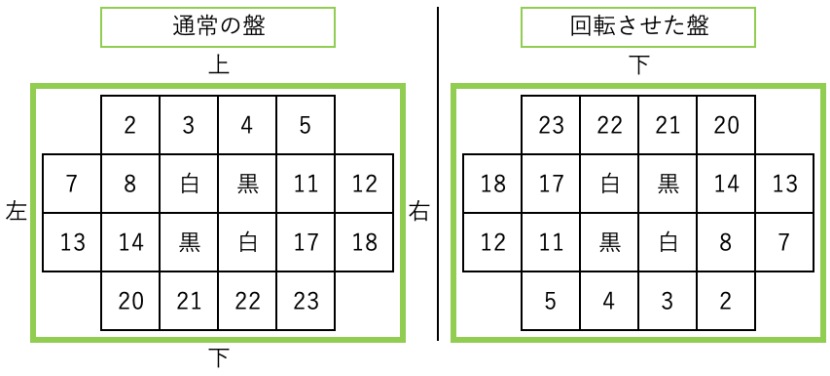

図 9: 通常の盤 (４ｘ６盤) &回転させた盤 (４ｘ６盤)

3.4　解析＆検証手法

本研究では最善手を求める手法を 3 つ使用して解析と検証を行う．

完全読み切り

全局面を探索しゲーム終了時に自石の数が最も多くなる手を求める．

必勝読み切り

探索内容を勝敗のみに絞って全局面を探索し勝つ手を求める．

一定手数の先読み

一定手数先まで読んでその局面を評価し有望な手を求める．
発見した有望な手が複数ある場合は、それら手に対してさらに深く探索し最も優れていると思われる手を決定する．

サイズが十分に小さい盤面に対しては、完全読み切りにより勝敗だけでなくゲーム終了時に自石の最も多くなる最善手を求める．完全読み切りが不可能なサイズの盤面に対しては、勝敗のみに絞って勝つ手を求める．それも不可能なサイズの大きい盤面に対しては、一定手数の先読みし有望そうな手を求める．

一定手数の先読みは以下の手順で行う．任意の整数 m, n に対して、まず m 手先までの局面を探索し，各局面の評価値を求めてアルファベータ法で優れた局面を評価する．優れた局面の条件は途中経過で勝つ手を評価する．優れた各局面に対して，さらに n 手先 (現在の局面から m+n 手先) まで探索を行って評価値を求め，アルファベータ法により最も優れた手を選択する．この操作を、以下 ［m,n］手探索とする．

以上の 3 つの手法で盤面のサイズを変化させたニップの解析および検証を行う．

4　解析＆検証と考察

4.1　解析＆検証結果

表 2 にサイズ４ｘ４および４ｘ６，６ｘ６，４ｘ８の盤面で完全読み切りを行った結果を示す．表 2 より、４ｘ４と６ｘ６では後手必勝、４ｘ６，４ｘ８では先手必勝となる．

表 2: 結果：完全読み切り

盤面	勝敗	結果		探索数	処理時間
４ｘ４	後手必勝	Ｂ：　２	Ｗ：１０	168	0.09s
４ｘ６	先手必勝	Ｂ：１２	Ｗ：　８	2433070	1m10s
６ｘ６	後手必勝	Ｂ：　８	Ｗ：１６	163759963	29m3s
４ｘ８	先手必勝	Ｂ：１８	Ｗ：１０	36063860210	133h5m

表はここまで

サイズ４ｘ４および４ｘ６，６ｘ６，４ｘ８の盤において双方が最善手を打った場合の手順を図 10 に示す．黒が先手のため，図中の奇数は黒番の手，偶数は白番の手を表す．ただし，６ｘ６盤において，15 手に黒はパスになるため，それ以降は黒番と白番が入れ替わる．

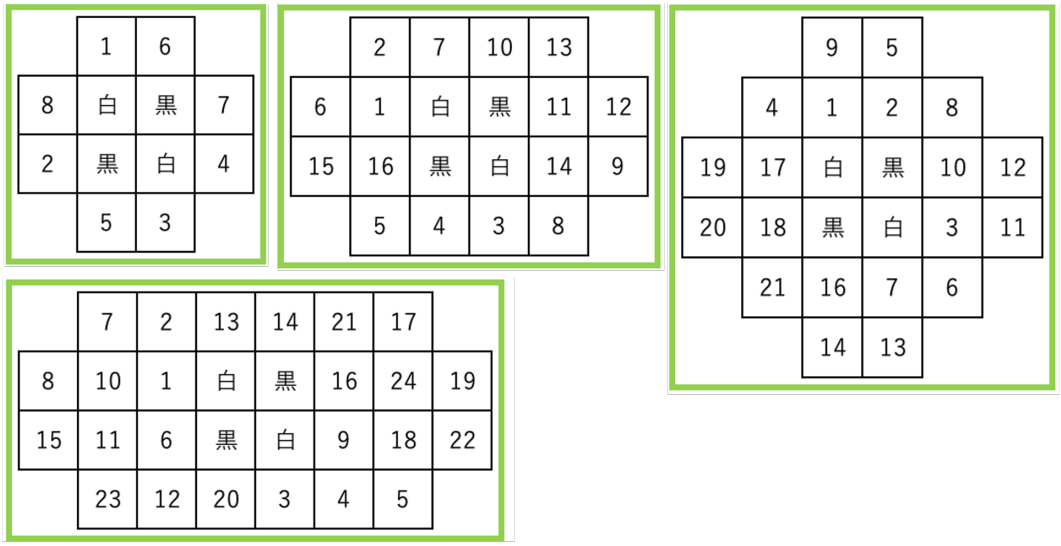

図 10: 双方が最善手を打った場合の手順

表 3 にサイズ４ｘ４および４ｘ６，６ｘ６，４ｘ８，４ｘ１０の盤面で必勝読み切りを行った結果を示す．表 3 より、４ｘ４と６ｘ６では後手必勝、４ｘ６，４ｘ８，４ｘ１０では先手必勝となる．

表 3: 結果：必勝読み切り

盤面	勝敗	探索数	処理時間
４ｘ４	後手必勝	74	0.04s
４ｘ６	先手必勝	63272	3s
６ｘ６	後手必勝	2240980	1m8s
４ｘ８	先手必勝	228806832	2h20m
４ｘ１０	先手必勝	10924512385	108h55m

表はここまで

表 4 にサイズ４ｘ４および４ｘ６，６ｘ６，４ｘ８，４ｘ１０，６ｘ８の盤面で一定手数の探索を行った結果を示す．表 4 より、４ｘ４，６ｘ６では後手有利，４ｘ６，４ｘ８，４ｘ１０では先手有利となる．表 2，表 3 に示される完全読み切り・必勝読み切りにより勝敗が確定している４ｘ１０までのサイズの勝敗と，表 4 に示される一定手数の探索の勝敗予測を比較すると両者は一致するため，一定手数の探索の勝敗予想は妥当な結果であると言える．

表 4: 結果：一定手数の探索

盤面	勝敗	先手有利	後手有利	盤面	勝敗	先手有利	後手有利
４ｘ４	後手必勝	1	5	４ｘ８	先手必勝	12	10
４ｘ６	先手必勝	8	6	４ｘ１０	先手必勝	16	12
６ｘ６	後手必勝	7	11				

表はここまで

一定手数の探索詳細はサイズ４ｘ４を表5，４ｘ６を表6，６ｘ６を表7，４ｘ８を表8，４ｘ１０を表9，６ｘ８を表10に示す．

表5〜表8より、サイズ４ｘ４、４ｘ６，６ｘ６，４ｘ８の処理時間は表3の必勝読み切りの処理時間未満である．探索数も一部を除いて下回っている．探索数の上回った例外は４ｘ４が１通り，４ｘ６が２通り，６ｘ６が４通り，４ｘ８が１通りで表3の探索数を1〜2割程度上まっている．

表 5: 結果：サイズ４ｘ４の探索詳細

勝敗	先読数	勝敗	先読数	勝敗	先読数
後手有利	2,6	後手有利	4,4	先手有利	6,2
後手有利	3,5	後手有利	5,3	先手有利	7,1

表はここまで

表 6: 結果：サイズ４ｘ６の探索詳細

勝敗	先読数	勝敗	先読数	勝敗	先読数	勝敗	先読数
後手有利	2,14	後手有利	6,10	後手有利	10,6	先手有利	13,3
先手有利	3,13	先手有利	7,9	先手有利	11,5	先手有利	14,2
後手有利	4,12	後手有利	8,8	後手有利	12,4	先手有利	15,1
先手有利	5,11	先手有利	9,7				

表はここまで

表 7: 結果：サイズ６ｘ６の探索詳細

勝敗	先読数	勝敗	先読数	勝敗	先読数	勝敗	先読数
後手有利	2,18	先手有利	7,13	先手有利	12,8	後手有利	16,4
先手有利	3,17	後手有利	8,12	後手有利	13,7	先手有利	17,3
後手有利	4,16	先手有利	9,11	後手有利	14,6	後手有利	18,2
先手有利	5,15	後手有利	10,10	後手有利	15,5	先手有利	19,1
後手有利	6,14	後手有利	11,9				

表はここまで

表 8: 結果：サイズ４ｘ８の探索詳細

勝敗	先読数	勝敗	先読数	勝敗	先読数	勝敗	先読数
後手有利	2,22	先手有利	8,16	先手有利	14,10	後手有利	19,5
先手有利	3,21	後手有利	9,15	先手有利	15,9	先手有利	20,4
後手有利	4,20	後手有利	10,14	後手有利	16,8	後手有利	21,3
後手有利	5,19	先手有利	11,13	後手有利	17,7	先手有利	22,2
先手有利	6,18	後手有利	12,12	後手有利	18,6	先手有利	23,1
後手有利	7,17	後手有利	13,11				

表はここまで

表 9 より，４ｘ１０の勝敗結果は 3.3 節で述べた通り，初手探索数は 1/2 の 2 つである．4 通りの例外がある．先読数［2，30］･［4,28］は探索数が膨大なため，探索不可とする．先読数［3,29］･［5,27］は初手探索の 1 つ目で先手有利が確定したため，初手探索の 2 つ目は探索不要とする．

表 9: 結果：サイズ４ｘ１０の探索詳細

勝敗	先読数	探索数	処理時間	勝敗	先読数	探索数	処理時間
/	2,30	探索難	/	先手有利	17,15	103873 2363	6s
先手有利	3,29	7 17202983	1m5s	先手有利	18,14	85437745 32431	49m14s
/	4,28	探索難	/	先手有利	19,13	747189 507	32s
先手有利	5,27	19 17202983	1m3s	先手有利	20,12	191800731 89	2h0m
先手有利	6,26	572 17171656	9m39s	後手有利	21,11	14602014 681	9m4s
先手有利	7,25	56 303402627	2h59m	後手有利	22,10	523276672 46	4h39m
後手有利	8,24	3496 97764804	59m33s	先手有利	23,9	47865116 333	27m28s
先手有利	9,23	206 13912074	8m10s	先手有利	24,8	1620813026 153	12h15m
後手有利	10,22	24133 5468365	3m12s	後手有利	25,7	299838335 13	2h31m
先手有利	11,21	525 8419668	5m6s	先手有利	26,6	2364077252 13	31h0m
後手有利	12,20	186529 31541228	17m16s	後手有利	27,5	1066451924 9	9h10m
先手有利	13,19	3469 368563	15s	後手有利	28,4	8501326974 3	55h42m
後手有利	14,18	1175358 471729	1m7s	後手有利	29,3	2164523541 1	14h52m
先手有利	15,17	17550 32491	3s	先手有利	30,2	12991710142 2	90h53m
後手有利	16,16	9351453 15378	6m0s	先手有利	31,1	4364710743 1	33h29m

表はここまで

表 10 より，６ｘ８の勝敗結果は 3.3 節で述べた通り，初手探索数は 1/2 の 2 つである．初期局面からの必勝読み切りができなかったため，初期局面から［7,25］～［23,9］（[20,12],[22,10］は除く）までの探索を行った．3 通りの例外がある．先読数［8,24］･［9,23］･［10,22］は初手探索の 1 つ目で先手有利が確定したため，初手探索の 2 つ目は探索不要とする．途中経過では先手有利が

９，後手有利が６より，先手がやや優勢である傾向が見られた．

表 10: 結果：サイズ６ｘ８の探索詳細

勝敗	先読数	探索数	処理時間	勝敗	先読数	探索数	処理時間
後手有利	7,25	217 4101995596	24h12m	先手有利	15,17	2564361 1919568	2m56s
後手有利	8,24	21117 63402476	21m20s	後手有利	16,16	936600203 489425	7h7m
先手有利	9,23	1526 61058316	21m40s	先手有利	17,15	6779670 116412	5m44s
後手有利	10,22	387286 2236366	56s	後手有利	18,14	10610751451 14063	72h34m
先手有利	11,21	14954 216187526	2h	先手有利	19,13	51977313 20461	37m42s
後手有利	12,20	4359919 180048179	2h28m	先手有利	21,11	357837202 1439	4h23s
先手有利	13,19	190475 3801839	2m18s	先手有利	23,9	1519577163 629	14h10m
先手 有利	14,18	72279574 6598594	42m31s				

表はここまで

4.2 考察

本研究では，縮小版ニップの完全解析＆一定手数の探索の検証を行った．

完全解析の結果として，完全読み切りではサイズ４ｘ４と６ｘ６は後手必勝、４ｘ６，４ｘ８は先手必勝と分かった．必勝読み切りでは サイズ４ｘ４と６ｘ６で後手必勝、４ｘ６と４ｘ８，４ｘ１０で先手必勝と分かった．盤のサイズの違いに依る先手有利，後手有利の差は見られなかったが，正方形盤だと後手有利，長方形盤だと先手有利の傾向が見られた．縮小版のオセロでも同様に正方形盤では後手有利，長方形盤では先手有利の傾向が見られたため（サイズ４ｘ４と６ｘ６は後手必勝，４ｘ６と４ｘ８，４ｘ１０は先手必勝），ニップとオセロの違いである外周部分と円周を挟むことに違いはあるが，傾向が似ていることは分かる．以上より，縮小版のニップは盤の形で先手有利か後手有利の性質がある．したがって，４ｘ１０以降の盤サイズは６ｘ８（マス目の総数：３６）と４ｘ１２（マス目の総数：４４）が先手有利の可能性があり，通常盤のサイズ８ｘ８（マス目の総数：５２）が後手有利の可能性が高くなると推測できる．

次に，一定手数の探索を，５つの盤サイズで行い勝敗予測を行ったところ，必勝読み切りの結果と全て一致した．必勝読み切りで何日かかるか分からない盤サイズを一定の先読数で区切り探索数を絞ることで探索が難しい盤サイズを徐々に解析することができる．しかし，分割した先読数の通りを全て探索する場合，分割せずに探索するより総処理時間が約３倍（４ｘ８の場合）かかる．しかし，先手有利か後手有利かが過半数をとればその時点で探索不要になるので，効率よく探索するためには探索順序が重要であることを確認できた．

5　プログラムの構造

本章では，本研究 Java 言語で作成したアルファベータ法による探索で最善手を求めるゲーム AI のプログラム構造について述べる．

付録に本研究で作成したプログラムのソースを示す．本研究で作成したプログラムは以下のクラスから成る．

- Board クラス
- BoardSetting クラス
- Game クラス
- CheckBoard クラス
- ChangeBoard クラス
- ConstantAndStatic クラス
- Repetition クラス
- Technique クラス
- Exploratory クラス
- UpdateData クラス
- ShowBoard クラス
- Test クラス

以下では，各クラスについて述べる．

5.1　Board クラス

Board クラスは盤面ごとの変数の情報，石を打った変化，指定した手数先まで再帰的に先読み，手番の変更，得た石の計算などの抽象クラスに関するクラスである．ConstantAndStatic クラスのサブクラスである．表 11 に Board クラスのクラス図を示す．

表 11: Board クラス

Board	
#board[]:int	盤面
− turn[]:int	手番 [白・黒]
− pass[]:boolean	パス実行時用 [直前・直後]
− emptySquares:int	空升
− previousPlace:int	直前に打った石の位置
− stoneGet[]:int	得た石
− stoneDiffer:int	得た石の差

Board は次ページに続く

前ページからの続き

Board	
− nextChoice:int	盤面ごとの着手可能手数
− alphaBeta:int	アルファベータ法用の評価値
#repe:Repetition	参照型変数
#check:CheckBoard	参照型変数
− setting:BoardSetting	参照型変数
+Board()	コンストラクタ，盤面の情報の初期化.
+getTurn():int[]	変数 turn の getter.
+setTurn(setTurn:int[]):void	変数 turn の setter.
+getPass():boolean[]	変数 pass の getter.
+setPass(setPass:boolean[]):void	変数 pass の setter.
+setAfterPass(setPass:boolean[]):void	変数 pass の setter(直後のみ).
+getEmptySquares():int	変数 emptySquares の getter.
+setEmptySquares(setEmptySquares:int):void	変数 emptySquares の setter.
+getPreviousPlace():int	変数 previousPlace の getter.
+setPreviousPlace(setPreviousPlace:int):void	変数 previousPlace の setter.
+getStoneGet():int[]	変数 stoneGet の getter.
+setStoneGet(setStoneGet:int[]):void	変数 stoneGet の setter.
+getStoneDiffer():int	変数 stoneDiffer の getter.
+setStoneDiffer(setStoneDiffer:int):void	変数 stoneDiffer の setter.
+getNextChoice():int	変数 nextChoice の getter.
+setNextChoice(setNextChoice:int):void	変数 nextChoice の setter.
+getAlphaBeta():int	変数 alphaBeta の getter.
+setAlphaBeta(setAlphaBeta:int):void	変数 alphaBeta の setter.
#player():void	人間が一手打つ.
#com():void	COM が一手打つ.
#setStone(checkBoard:int[], checkTurn:int, place:int):void	石を打った変化.
#comInsight(depth:int, futureDepth:int, selectNectBoard:int, fullDepth:int):void	指定した手数先まで再帰的に先読み.
#nextBoard(place:int):Board	石を打った新たな盤面を作成.
#copy():Board	盤面のコピーを作成.
#nextTurn(futureTurn:int):int	手番の変更.
#calculateStone(futureBoard:int[]):int[]	得た石の計算

Board クラスはここまで

5.2 BoardSetting クラス

BoardSetting クラスは Board クラスの初期化が複雑な変数に関するクラスである． ConstantAndStatic クラスのサブクラスである．表 12 に BoardSetting クラスのクラス図を示す．

表 12: BoardSetting クラス

BoardSetting	
+boardSetting():int[]	盤面の初期化.
+emptySquaaresSetting:int	空升の初期化.
+circleInitializaation(board:int[]):void	外周の盤面のマス目の番号をリストで作成.

BoardSetting クラスはここまで

5.3 Game クラス

Game クラスはゲームを開始するクラスである．Board クラスのサブクラスである．表 13 に Game クラスのクラス図を示す.

表 13: Game クラス

Game	
+Game()	コンストラクタ.
#player():void	人間が一手打つ.
#com():void	COM が一手打つ.
#setStone(checkBoard:int[]，checkTurn:int，place:int):void	石を打った変化.
#comInsight(depth:int，futureDepth:int，selectNect-Board:int，fullDepth:int):void	指定した手数先まで再帰的に先読み.
#nextBoard(place:int):Board	石を打った新たな盤面を作成.
#copy():Board	盤面のコピーを作成.
#nextTurn(futureTurn:int):int	手番の変更.
#calculateStone(futureBoard:int[]):int[]	得た石の計算.

Game クラスはここまで

5.4 CheckBoard クラス

CheckBoard クラスは指定したマス目に石を打つことが可能かチェック，指定した位置と周辺の石を変化させる抽象クラスである．ConstantAndStatic クラスのサブクラスである．表 14 に CheckBoard クラスのクラス図を示す.

表 14: CheckBoard クラス

CheckBoard	
#yesDireCount:int	石を打てる方角の累計.
#yesCircleCount:int	石を打てる外周の位置の累計.
#circleAround[]:int	外周に石を打てる位置を記録.
#direRecord:ArrayList<Integer>	石を打てる方角を記録.

CheckBoard は次ページに続く

前ページからの続き

CheckBoard	
#circleRecord:ArrayList<Integer> #check_circleLine[]：ArrayList<Integer>	石を打てる外周の位置を記録. 変化する外周の石の位置を記録.
+CheckBoard() +currentFuture(checkBoard:int[]，checkTurn:int，place:int):boolean − checkStone(checkBoard:int[]，checkTurn:int，place:int，checkColor:int):boolean − allDireCheck(checkBoard:int[]，place:int，checkColor:int):void − allCircleCheck(checkBoard:int[]，place:int，checkColor:int，leftRight:int):void − oneDireCheck(checkBoard:int[]，checkTurn:int，place:int，dire:int，checkColor:int):boolean − oneCircleCheck(checkBoard:int[]，checkTurn:int，place:int，leftRight:int，checkColor:int):boolean *#changeStone(checkBoard:int[]，cTurn:int，place:int，changeBoard:int[]):void* *#oneDireChange(changeBoard:int[]，turn:int，place:int，dire:int):int[]* *#circleChange(changeBoard:int[]，turn:int，place:int，leftRight:int):int[]*	コンストラクタ. 指定した位置の石を打つことが可能かチェック. 指定した位置の石を打つことが可能かチェック. 指定した位置に対して，石を打てる方角の累計と記録，外周の位置を記録. 指定した位置に対して，石を打てる外周の位置の累計と記録. 記録した方角に対して石の変化が可能かチェック. 記録した外周の位置に対して石の変化が可能かチェック. 指定した位置に石を打つ. 盤面を置換する (外周を除く). 盤面を置換する (外周).

CheckBoard クラスはここまで

5.5 ChangeBoard クラス

ChangeBoard クラスは指定した位置と周辺の石を変化させるクラスである．CheckBoard クラスのサブクラスである．表 15 に ChangeBoard クラスのクラス図を示す.

表 15: ChangeBoard クラス

ChangeBoard	
#changeStone(checkBoard:int[]，cTurn:int，place:int，changeBoard:int[]):void #oneDireChange(changeBoard:int[]，turn:int，place:int，dire:int):int[] #circleChange(changeBoard:int[]，turn:int，place:int，leftRight:int):int[]	指定した位置に石を打つ. 盤面を置換する (外周を除く). 盤面を置換する (外周).

ChangeBoard クラスはここまで

5.6 ConstantAndStatic クラス

ConstantAndStatic クラスは static 変数と定数に関するクラスである．表 16 に ConstantAndStatic クラスのクラス図を示す.

表 16: ConstantAndStatic クラス

ConstantAndStatic	
#TECHNIQUE：int = 0	解析手法の番号
#SWITCHING：int = 3	解析手法の移行番号
#VERTICAL:int = 4	縦の長さ (盤面)
#BESIDE:int = 4	横の長さ (盤面)
#ALL_BESIDE:int = BESIDE + 2	横の長さ (盤外も含む)
#ALL_SIZE:int = VERTICAL * ALL_BESIDE	盤面のサイズ (盤外も含む)
#WHITE:int = − 1	白石
#BLACK:int = − 1	黒石
#EMPTY:int = 0	空白
#BORDER:int = Integer.MAX_VALUE	盤外
#DEBUG:boolean = true	デバグ用
#depth：int = 8	完全・必勝読み切りの先読数.
#depth1：int = 6	一定数の先読みの先読数（１回目の深さ）.
#depth2：int = depth-depth1+1	一定数の先読みの先読数（２回目の深さ）.
#circle_Line:ArrayList<Integer>[]	外周のマス目の番号のリスト作成用.
#board_Pattern:ArrayList<Board>	初期盤面～終了盤面までを記憶.
#judge:int	枝切り位置（探索中の先読みの深さ）.
#full_Data:ArrayList<ArrayList<Board>>[]	先読みの深さごとに一時記憶させる初期～終局までの盤面.
#value:int[]	先読みの深さごとに一時記憶させる評価値.
#num012:int	探索数１（１回目の探索時）.
#num2:int	探索数２（２回目の探索時）.
#full_Result1:ArrayList<ArrayList<Board>>	探索後の最善手の初期～終局までの盤面（１回目の探索時）.
#alphabeta_Result1:int	探索後の最善手の評価値（１回目の探索時）.
#methodSwitch：boolean	手法の変更可能かチェック.
#full_Result2:ArrayList<ArrayList<Board>>	探索後の最善手の初期～終局までの盤面（２回目の探索時）.

ConstantAndStatic は次ページに続く

前ページからの続き

ConstantAndStatic	
#alphabeta_Result2:int	探索後の最善手の評価値（２回目の探索時）.
#alphabeta_Result3:ArrayList<Integer>	最善手の評価値.

ConstantAndStatic クラスはここまで

5.7 Repetition クラス

Repetition クラスは先読みまで探索を繰り返しながら比較や枝切りを行い，最善手を更新するクラスである．ConstantAndStatic クラスのサブクラスである．表 17 に Repetition クラスのクラス図を示す.

表 17: Repetition クラス

Repetition	
#MAX:int = Integer.MAX_VALUE	最大値.
#MIN:int = Integer.MIN_VALUE	最小値.
− text:ShowText	参照型変数
-show:ShowBoard	参照型変数
− ldtStart1:LocalDateTime	実行時間の開始日時を格納.
− ldtEnd1:LocalDateTime	実行時間の終了日時を格納 (探索１回目).
− ldtEnd2:LocalDateTime	実行時間の終了日時を格納 (探索２回目).
− ramdom:Random	乱数発生用
+Repetition()	コンストラクタ.
+firstRepetition(nextBoardList1:ArrayList¡Board¿):Board	先読み後に選択した１手を返す.
− secondRepetition():void	１回目の指定した先読数で有望な手をさらに先読みして最も優れている手を求める（２回目の探索）.
+allRepetition(nextBoardList1，futureDepth:int，depth:int，fullDepth:int):void	指定した手数先まで再帰的に先読みする.
− reset_Initial():void	全体の先読み開始前に static 変数の定義する.
− reset_Repetition():void	初手着手可能手ごとの先読み開始前に static 変数のリセットを行う.
#tech_Perfect():void	手法：完全読み切り.
#tech_Victory(search:int):void	手法：必勝読み切り.
#searchData(search:int):void	探索を行う.
#delete(search:int):void	探索中の先読み位置の盤面データを削除.

Repetition は次ページに続く

前ページからの続き

Repetition	
#delete2(deleteLine:int):void	指定した先読み位置まで盤面データを削除.
endUpData(first:boolean)	初手着手可能手ごとに結果を更新 (1 回目の探索).
endUpData2(count:int)	初手着手可能手ごとに結果を更新 (2 回目の探索).
− information():void	探索結果を表示.
− resultS():void	探索結果から最善手の盤面・評価値を表示.

Repetition クラスはここまで

5.8 Technique クラス

Technique クラスは探索手法に関する抽象クラスである．Repetition クラスのサブクラスである．表 18 に Technique クラスのクラス図を示す．

表 18: Technique クラス

Technique	
+Technique()	コンストラクタ
#tech_Perfect():void	手法：完全読み切り.
#tech_Victory(search:int):void	手法：必勝読み切り.
− pruning(prun:int):void	枝切り.

Technique クラスはここまで

5.9 Exploratory クラス

Exploratory クラスは探索中の盤面データの更新を行う抽象クラスである．Technique クラスのサブクラスである．表 19 に Exploratory クラスのクラス図を示す．

表 19: Exploratory クラス

Exploratory	
+Exploratory()	コンストラクタ.
#update(back:int, front:int):void	更新 1，盤面データの上書き.
#update2(back:int, front:int):void	更新 2，盤面データの追加.
#update3(back:int):void	更新 3，探索中の盤面データを削除.
#searchData(search:int):void	探索を行う.

Exploratory は次ページに続く

前ページからの続き

Exploratory	
#delete(search:int):void	探索中の先読み位置の盤面データを削除.
#delete2(deleteLine:int):void	指定した先読み位置まで盤面データを削除.

Exploratory クラスはここまで

5.10 UpdateData クラス

UpdateData クラスは探索中の盤面データを更新するクラスである．Exploratory クラスのサブクラスである．表 20 に UpdateData クラスのクラス図を示す.

表 20: UpdateData クラス

UpdateData	
+UpdateData()	コンストラクタ.
#update(back:int，front:int):void	更新 1，盤面データの上書き.
#update2(back:int，front:int):void	更新 2，盤面データの追加.
− update2_1(position:int):void	更新 2 − 1，盤面データを 1 つに縮小.
#update3(back:int):void	更新 3，探索中の盤面データを削除.
#endUpData(first:boolean)	初手着手可能手ごとに結果を更新 (1 回目の探索).
#endUpData2(count:int)	初手着手可能手ごとに結果を更新 (2 回目の探索).

UpdateData クラスはここまで

5.11 ShowBoard クラス

ShowBoard クラスはニップの盤面を表示するクラスである．ConstantAndStatic クラスのサブクラスである．表 21 に ShowBoard クラスのクラス図を示す.

表 21: ShowBoard クラス

ShowBoard	
− interval:int = BESIDE	表示する 2 つの盤面の間隔.
− digit:int = String.valueOf(ALL_SIZE).length()	マス目の番号の最大値の桁数.
+show(board:int[]):void	ニップの盤面を表示.
− showUp(up:int):void	ニップの盤面 [上部] の表示.

ShowBoard は次ページに続く

前ページからの続き

ShowBoard	
− showDown (down:int):void	ニップの盤面 [下部] の表示.
− highControl(space:int，differ:int):int	角の高さを返す.
− centralPart(board:int[]，high:int，adjust:int):void	ニップの盤面 [中心部] の表示.
− square(board:int[]，position:int):String	盤面の升目の文字列表現を返す.
− position(posi:int):void	盤面のマス目の位置を表現し，桁数に合わせて空白を足して表示.

ShowBoard クラスはここまで

5.12 Test クラス

Test クラスは COM のレベルを決定と実行するクラスである．表 22 に Test クラスのクラス図を示す.

表 22: Test クラス

Test	
− board:Board	参照型変数
− show:ShowBoard	参照型変数
− beforePassW:boolean	パス実行用 (白石の直前用)
− beforePassB:boolean	パス実行用 (黒石の直前用)
− comLevel:int[] = 0,0	COM のレベル (− 1 は人間用)
+main(args[]:String):void	メインメソッド.
− setComLevel():void	COM のレベルを決定.

Test クラスはここまで

6 結論・今後の課題

本研究では縮小版ニップの完全解析を行った．完全解析を行うためにゲーム AI を作成し，完全読み切りは正方形盤が６ｘ６，長方形盤が４ｘ８まで求めることができ，必勝読み切りは正方形盤が６ｘ６，長方形盤が４ｘ１０まで求めることができた．また，一定数の探索は必勝読み切りと同じ結果を求めることができた．本研究の結果から，４ｘ４と６ｘ６の正方形盤では先手必勝，４ｘ６，４ｘ８，４ｘ１０の長方形盤では後手必勝となることが示された．このことから，縮小版ニップは正方形盤だと後手有利，長方形盤だと先手有利の傾向だと推測される.

今後の課題としてはサイズ４ｘ１０の完全読み切り，サイズ６ｘ８，４ｘ１２の必勝読み切りである．盤をより拡張させれば，探索数も増加し，処理時間も増加するため，コンピュータの性能も大切だが，いかに分散させて処理するかが重要となってくる．より拡張させた盤の完全解析を実現するには並列化が重要になる．並列化が実現できれば，より拡張させた盤の完全解析が可能になってくる.

謝辞

本研究に行うにあたり，指導してくださった石水先生にはとても感謝しています．本当にありがとうございました．

参考文献

[1] Nipp – アブストラクトゲーム博物館，https://www.nakajim.net/index.php?Nipp

[2] ニップ。円型の盤で戦う角が無いリバーシ。円形オセロ。 - 月の方舟，http://noah.n43foto.com/archives/3043

[3] 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

[4] 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

[5] Victor Allis, A Knowledge-based Approach of Connect-Four, The Game is Solved: White Wins, Vrije Universiteit, Subfaculteit Wiskunde en Informatica, (1998)，http://www.informatik.uni-trier.de/ fernau/DSL0607/Masterthesis-Viergewinnt.pdf

[6] 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)，https://www.researchgate.net/publication/2925531_Solving_Go_On_Small_Boards/link/0fcfd511dfb651482c000000/download

[7] 竹下拓輝. 池田論, 坂本眞人, 伊藤隆夫, 縮小盤オセロにおける完全解析, 情報処理学会九州支部火の国情 報シンポジウム, No.1A-2, pp.1-6 (2015)，https://www.ipsj-kyushu.jp/page/ronbun/hinokuni/1004/1A/1A-2.pdf

[8] 中村和樹：奇数マスを含む縮小版オセロの完全解析，サレジオ工業高等専門学校 2016 年度卒業研究概要集，p.135 (2017)，https://www.salesio-sp.ac.jp/papers/sotsuken/sotsuken2016.pdf

[9] 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

A　付録について

以下に本研究で作成したプログラムのソースコードを示す.

Board クラス

```
1  package experiment.last.suspend;
2
3  /**
4   * 正方形のニップ自動化
5   */
6  public abstract class Board extends ConstantAndStatic {
7    protected int board[]; // 盤面
8    private int turn[]; // 白・黒の手番 [更新前：0・更新後：1]
9    // パス実行時用（直前 [白]：0・直前 [黒]：1・直後 [白]：2・直後 [黒]：3）
10   private boolean pass[];
11   private int emptySquares; // 空升
12   private int previousPlace; // 直前に打った石の位置
13   private int stoneGet[]; // 得た石（[白]：0・[黒]：1）
14   private int stoneDiffer; // 得た石の差
15   private int nextChoice; // 盤面ごとの着手可能手数
16   private int alphaBeta; // アルファベータ法用の評価値
17   /* 参照型変数の定義 */
18   protected Repetition repe;
19   protected CheckBoard check;
20   private BoardSetting setting;
21
22   /**
23   * コンストラクタ，盤面の情報の初期化.
24   */
25   public Board(){
26     this.repe = new UpdateData(); // オブジェクト生成
27     this.check = new ChangeBoard(); // オブジェクト生成
28     this.setting = new BoardSetting(); // オブジェクト生成
29     this.board = this.setting.boardSetting(); // 盤面の初期化
30     // 盤面による空升の設定
31     this.emptySquares = this.setting.emptySquaresSetting();
32     this.setting.circleInitialization(board); // 円周の盤面の番号をリストで作成
33
34     this.turn = new int[2]; // 配列の生成
35     this.turn[0] = BLACK; // 更新前の手番の初期化 [先行：黒]
36     this.turn[1] = BLACK; // 更新後の手番の初期化 [先行：黒]
37     this.pass = new boolean[4]; // 配列の生成
38     for(int i=0;i<4;i++)
```

```
39        this.pass[i] = false; // 初期状態：未パス
40        this.previousPlace = 0; // 初期状態：直前の手はない
41        this.stoneGet = new int[2]; // 配列の生成
42        this.stoneGet[0] = 0; // 初期状態：0
43        this.stoneGet[1] = 0; // 初期状態：0
44        this.stoneDiffer = 0; // 初期状態：0
45        this.nextChoice = 0; // 初期状態：0
46        this.alphaBeta = 0; // 初期状態：0
47      }
48
49      /**
50      * 変数 turn の getter.
51      * @return int[]：手番 [更新前・更新後]
52      */
53      public int[] getTurn() {
54        return this.turn;
55      }
56
57      /**
58      * 変数 turn の setter.
59      * @param setTurn：更新前・更新後の手番
60      */
61      public void setTurn(int[] setTurn) {
62        for(int i=0;i<2;i++)
63          this.turn[i] = setTurn[i];
64      }
65
66      /**
67      * 変数 pass の getter.
68      * @return int[]：パス実行時用（直前・直後）
69      */
70      public boolean[] getPass() {
71        return pass;
72      }
73
74      /**
75      * 変数 pass の setter.
76      * @param setPass：パス実行時用（直前・直後）
77      */
78      public void setPass(boolean[] setPass) {
79        for(int i=0;i<4;i++)
80          this.pass[i] = setPass[i];
81      }
```

```
82
83      /**
84      * 変数 pass の setter.
85      * @param setPass : パス実行時用（直後）
86      */
87      public void setAfterPass(boolean[] setPass) {
88        for(int i=2;i<4;i++)
89          this.pass[i] = setPass[i-2];
90      }
91
92      /**
93      * 変数 emptySquares の getter.
94      * @return int : 空升
95      */
96      public int getEmptySquares() {
97        return this.emptySquares;
98      }
99
100     /**
101     * 変数 emptySquares の setter.
102     * @param setEmptySquares : 空升
103     */
104     public void setEmptySquares(int setEmptySquares) {
105       this.emptySquares = setEmptySquares;
106     }
107
108     /**
109     * 変数 previousPlace の getter.
110     * @return int : 直前に打った石の位置
111     */
112     public int getPreviousPlace() {
113       return this.previousPlace;
114     }
115
116     /**
117     * 変数 previousPlace の setter.
118     * @param setPreviousPlace : 直前に打った石の位置
119     */
120     public void setPreviousPlace(int setPreviousPlace) {
121       this.previousPlace = setPreviousPlace;
122     }
123
124     /**
```

```
125     * 変数 stoneGet の getter.
126     * @return int[]：得た石 [白・黒]
127     */
128     public int[] getStoneGet() {
129       return this.stoneGet;
130     }
131
132     /**
133     * 変数 stoneGet の setter.
134     * @param setStoneGet：得た石 [白・黒]
135     */
136     public void setStoneGet(int[] setStoneGet) {
137       for(int i=0;i<2;i++)
138         this.stoneGet[i] = setStoneGet[i];
139     }
140
141     /**
142     * 変数 stoneDiffer の getter.
143     * @return int：得た石の差
144     */
145     public int getStoneDiffer() {
146       return this.stoneDiffer;
147     }
148
149     /**
150     * 変数 stoneDiffer の setter.
151     * @param setStoneDiffer：得た石の差
152     */
153     public void setStoneDiffer(int setStoneDiffer) {
154       this.stoneDiffer = setStoneDiffer;
155     }
156
157     /**
158     * 変数 nextChoice の getter.
159     * @return int：盤面の着手可能手数
160     */
161     public int getNextChoice() {
162       return this.nextChoice;
163     }
164
165     /**
166     * 数 nextChoice の setter.
167     * @param setNextChoice：盤面の着手可能手数
```

```
168     */
169     public void setNextChoice(int setNextChoice) {
170       this.nextChoice = setNextChoice;
171     }
172
173     /**
174     * 変数 alphaBeta の getter.
175     * @return int：アルファ・ベータ手法用の評価値
176     */
177     public int getAlphaBeta() {
178       return this.alphaBeta;
179     }
180
181     /**
182     * 変数 alphaBeta の setter.
183     * @param setAlphaBeta：アルファ・ベータ手法用の評価値
184     */
185     public void setAlphaBeta(int setAlphaBeta) {
186       this.alphaBeta = setAlphaBeta;
187     }
188
189     /**
190     * 人間が一手打つ.
191     */
192     protected abstract void player();
193
194     /**
195     * COM が一手打つ.
196     */
197     protected abstract void com();
198
199     /**
200     * 指定した位置に石を置き，周りの石を変化.
201     * @param checkBoard：チェックする盤面
202     * @param checkTurn：チェックする手番
203     * @param place：石を置く位置
204     */
205     protected abstract void setStone(int[] checkBoard,int checkTurn,int place);
206
207     /**
208     * 指定した手数先まで再帰的に先読みする.
209     * @param depth：先読み数（現状）
210     * @param futureDepth：先読みする盤面の手数
```

```
211     * @param selectNextBoard：先読みする直前の盤面
212     * @param furdepth ：先読み数（初期）
213     */
214     protected abstract void comInsight(int depth,int futureDepth,Board selectNextBoard
215                                        ,int fullDepth);
216
217     /**
218     * 指定した位置に石を打った新たな盤面を作成.
219     * @param place：石を打つ位置
220     * @return Board：新たな盤面
221     */
222     protected abstract Board nextBoard(int place);
223
224     /**
225     * 盤面のコピーを作成.
226     * @return Board：盤面のコピー
227     */
228     protected abstract Board copy();
229
230     /**
231     * 手番を変更.
232     * @param futureTurn：直前の手番
233     * @return int：手番（1=白，-1=黒）
234     */
235     protected abstract int nextTurn(int futureTurn);
236
237     /**
238     * 得た石の計算.
239     * @param futureBoard：盤面
240     * @return int：得た石 [白・黒]
241     */
242     protected abstract int[] calculateStone(int[] futureBoard);
243   }
244
```

BoardSetting クラス

```
1   package experiment.last.suspend;
2
3   import java.util.ArrayList;
4
5   /**
6    * 複雑な変数 (Board クラス) の初期化をするクラス.
7    */
```

```
8  public class BoardSetting extends ConstantAndStatic{
9
10   /**
11   * 盤面の初期化.
12   * @return int[]：初期化した盤面
13   */
14   public int[] boardSetting() {
15     int settingBoard[] = new int[ALL_SIZE + 2]; // 盤外も含めた盤面
16     settingBoard[0] = BORDER; // 盤外扱い
17     // 初期配置の左上（白）
18     int leftCenter = (ALL_BESIDE / 2) + (ALL_BESIDE * ((VERTICAL / 2) - 1));
19     for(int i=1;i<=ALL_SIZE;i++) {
20       int leftBoard = 0,rightBoard = 0,a12 = 0;
21       if((VERTICAL == 4 || BESIDE == 4)) { // 4*4, 4*6, 6*4
22         // VERTICAL の 1 段目・最終段目
23         if((i / ALL_BESIDE) == 0 || (i / ALL_BESIDE) == (VERTICAL - 1))
24           a12 = 1;
25       }
26       else {
27         if((i / ALL_BESIDE) == 0 || (i / ALL_BESIDE) == (VERTICAL - 1))
28           a12 = 2;  // VERTICAL の 1 段目・最終段目
29         else if((i / ALL_BESIDE) == 1 || (i / ALL_BESIDE) == (VERTICAL - 2))
30           a12 = 1; // VERTICAL の 2 段目・下 2 段目
31       }
32       leftBoard = 2 + ALL_BESIDE * (i / ALL_BESIDE) + a12; // 盤内の左端
33       rightBoard = ALL_BESIDE * (i / ALL_BESIDE + 1) - a12; // 盤内の右端
34
35       if(i == leftCenter || i == (leftCenter + ALL_BESIDE) + 1)
36         settingBoard[i] = WHITE; // 盤面（盤内）中央左斜め 2 つ：白
37       else if(i == (leftCenter + ALL_BESIDE)  || i == leftCenter + 1)
38         settingBoard[i] = BLACK; // 盤面（盤内）中央右斜め 2 つ：黒
39       else if(leftBoard <= i && i < rightBoard)
40         settingBoard[i] = EMPTY; // 盤面（盤内）を空白
41       else
42         settingBoard[i] = BORDER; // 盤外扱い
43     }
44     settingBoard[ALL_SIZE + 1] = BORDER; // 盤外扱い
45     return settingBoard;
46   }
47
48   /**
49   * 空升を初期化. 初期状態：4*4 の場合→ 16-8=8（中央 4 つ・角 4 つ),
50   * 6*6 の場合→ 36-16=20（中央 4 つ・角 1 2 つ）
```

```
51      * @return int：空升
52      */
53      public int emptySquaresSetting() {
54        return ((VERTICAL * BESIDE) - ((VERTICAL == 4 || BESIDE == 4) ? 8 : 16));
55      }
56
57      /**
58      * 円周の盤面の番号をリストで作成（時計回り：要素０・反時計回り：要素１）
59      * @param board：盤面
60      */
61      public void circleInitialization(int[] board) {
62        ArrayList<Integer> circle0 = new ArrayList<>();
63        ArrayList<Integer> circle1 = new ArrayList<>();
64        for(int i=1;i<ALL_BESIDE;i++) {
65          if(board[i]==EMPTY)
66            circle0.add(i); // 盤面：１段目
67        }
68        for(int i=0;i<VERTICAL-2;i++) {
69          int n=0,r=0;
70          for(int j=(ALL_BESIDE*2 + ALL_BESIDE*i); j>(ALL_BESIDE * (i+1)); j--) {
71            if(board[j]==EMPTY && n==0) {
72              circle0.add(j); // 盤面：円周の右側（１段目・最終段目を除く）
73              n++;
74            }
75            else if(board[j] == BORDER && board[j+1] == EMPTY && r==0) {
76              circle1.add(j+1); // 盤面：円周の左側（１段目・最終段目を除く）
77              r++;
78            }
79          }
80        }
81        for(int i=ALL_SIZE;i>(VERTICAL-1)*ALL_BESIDE+1;i--) {
82          if(board[i]==EMPTY)
83            circle0.add(i); // 盤面：最終段目
84        }
85        for(int i=0;i<circle1.size();i++)
86          circle0.add(circle1.get(circle1.size()-1-i));
87
88        circle_Line = new ArrayList[2];
89        circle_Line[0] = new ArrayList<>();
90        circle_Line[1] = new ArrayList<>();
91
92        for(int i=0;i<circle0.size();i++)
93          circle_Line[0].add(circle0.get(i));
```

```
94      for(int i=circle0.size()-1;i>=0;i--)
95        circle_Line[1].add(circle0.get(i));
96    }
97  }
98
```

Game クラス

```
1   package experiment.last.suspend;
2
3   import java.util.ArrayList;
4   import java.util.Scanner;
5
6   /**
7    * ボードゲーム：ニップの完全解析
8    */
9   public class Game extends Board{
10
11    /**
12    * コンストラクタ，親クラスのコンストラクタを実行.
13    */
14    public Game() {
15      super();
16    }
17
18    /**
19    * 人間が一手打つ
20    */
21    protected void player() {
22      int place; // 打つ位置
23      Scanner keyBoardScanner = new Scanner(System.in);
24      System.out.println((this.getTurn()[0] == WHITE) ? "白石の番です"
25                                                      : "黒石の番です");
26      while (true) { // 適切な位置が選択されるまでループ
27        System.out.print ("隣盤面の選択番号を選んでください (0=パス): ");
28        String inputString = keyBoardScanner.next();
29        try {
30          place = Integer.parseInt (inputString);
31        }
32        catch (NumberFormatException e) { // 整数値以外が入力された場合
33          System.out.println ("選択番号を入力してください");
34          continue;
35        }
36
```

```
37          if (place <0 || ALL_SIZE<place) {
38            System.out.println ("選択番号を入力してください");
39            continue;
40          }
41          if (place == 0) {
42            boolean nselect = false;
43            for(int i=1;i<=ALL_SIZE;i++) { // 各升に対して着手可能手の生成を行う.
44              if(this.board[i] == EMPTY) { // 空升なら着手可能.
45                // 指定の升が空升か判定.
46                if(check.currentFuture(this.board, this.getTurn()[1], i))
47                  nselect = true; // 0 は投了
48              }
49            }
50            System.out.println((nselect == false) ? "パスを受理" : "パスを却下");
51            if(nselect == false) {
52              this.getPass()[(this.getTurn()[1] == WHITE) ? 2 : 3] = true;
53              break;
54            }
55            else
56             continue;
57          }
58          if (board[place] != EMPTY
59              || !check.currentFuture(this.board, this.getTurn()[1], place)) {
60            System.out.println ("その位置には置けません");
61            continue;
62          }
63          break;
64        }
65        this.setStone(this.board, this.getTurn()[1], place); // 盤面に石を置く
66      }
67
68      /**
69      * com が一手打つ.
70      */
71      protected void com() {
72        // 着手可能手のオブジェクトをリストで作成.
73        ArrayList<Board> nextBoardList = new ArrayList();
74        Board nextBoard; // 着手可能手な新しい盤面を格納..
75        int possibleCount = 0; // 着手可能手の数.
76        /* 着手可能手を打った後の盤面を生成するし、リストに挿入 */
77        for(int i=1;i<=ALL_SIZE;i++) { // 各升に対して着手可能手の生成を行う.
78          if(this.board[i] == EMPTY) { // 空升なら着手可能.
79            // 指定の升が空升か判定.
```

```
80            if(check.currentFuture(this.board, this.getTurn()[1], i)) {
81              possibleCount += 1; // 着手可能手の数を＋１.
82              nextBoard = this.nextBoard(i); // その升に打った後の盤面を生成，格納.
83              nextBoardList.add(nextBoard); // リストの末尾に追加.
84            }
85          }
86        }
87        if(DEBUG) { // 着手可能手を表示用
88          for(int i=0;i<nextBoardList.size();++i)
89            System.out.println("着手:" + nextBoardList.get(i).getPreviousPlace());
90        }
91
92        if(possibleCount == 0) // パスの場合
93          this.getPass()[(this.getTurn()[1] == WHITE) ? 2 : 3] = true;
94        else {//着手可能の場合
95          // ランダムに 1 手選択.
96          nextBoard = this.repe.firstRepetition(nextBoardList);
97          /* 指定した位置に石を置く */
98          if(this.getEmptySquares() != 0) // 空升が有であるか判定.
99            this.setStone(this.board, this.getTurn()[1], nextBoard.getPreviousPlace());
100         System.out.println("選択の石：" + nextBoard.getPreviousPlace());
101       }
102       System.out.println("残り升目：" + this.getEmptySquares());
103     }
104
105     /**
106     * 指定した位置に石を置き，周りの石を変化.
107     * @param checkBoard：チェックする盤面
108     * @param checkTurn：チェックするターン
109     * @param place：石を置く位置
110     */
111     protected void setStone(int[] checkBoard,int checkTurn,int place) {
112       if(place != 0) {
113         this.board[place] = this.getTurn()[1]; // 指定した位置に手番側の石を置く
114         int empty = this.getEmptySquares();
115         this.setEmptySquares(--empty); // 空升の数を減少
116         this.setPreviousPlace(place); // 打った位置を記憶
117         // 指定した位置の周りを変化させる.
118         check.changeStone(checkBoard, checkTurn, place, this.board);
119       }
120       else
121         this.setPreviousPlace(place); // 打った位置を記憶
122     }
```

```
123
124     /**
125     * 指定した手数先まで再帰的に先読みする.
126     * @param depth：先読み数（現状）
127     * @param futureDepth：先読みする盤面の手数
128     * @param selectNextBoard：先読みする直前の盤面
129     * @param furdepth ：先読み数（初期）
130     */
131     protected void comInsight(int depth,int futureDepth,Board selectNextBoard
132                                                             , int fullDepth) {
133       // 手数の上限は空升数なので、先読み手数も空升数に合わせる
134       if (futureDepth > this.getEmptySquares())
135         futureDepth = this.getEmptySquares();
136       // 着手可能手のオブジェクトをリストで作成.
137       ArrayList<Board> nextBoardList = new ArrayList();
138       Board nextBoard = selectNextBoard; // 着手可能手を打つ直前前の盤面を格納.
139       int possibleCount = 0; // 着手可能手の数.
140       /* 着手可能手を打った後の盤面を生成するし、リストに挿入 */
141       for (int i=1; i<=ALL_SIZE; ++i) { // 各升に対して着手可能手の生成を行う.
142         nextBoard = selectNextBoard; // 先読みする直前の盤面を格納.
143         if (nextBoard.board[i] == EMPTY) { // 空升なら着手可能.
144           // 指定の升が空升か判定.
145           if(check.currentFuture(nextBoard.board, nextBoard.getTurn()[1], i)) {
146             possibleCount += 1; // 着手可能手の数を＋１.
147             nextBoard = this.nextBoard(i); // その升に打った後の盤面を生成，格納.
148             nextBoardList.add(nextBoard); // リストの末尾に追加.
149           }
150         }
151       }
152       /* 着手可能手がない場合 */
153       if(possibleCount == 0) {
154         nextBoard.getPass()[(nextBoard.getTurn()[1] == WHITE) ? 2 : 3] = true;
155         // 再帰中にパス状態を解除するために格納.
156         int beOutTurn = nextBoard.getTurn()[1];
157         // 直前・直後のパスの状態を判定.（true →直前のターンで終了，false →再帰継続）
158         if((nextBoard.getPass()[0] == true && nextBoard.getPass()[3] == true)
159             || (nextBoard.getPass()[1] == true && nextBoard.getPass()[2] == true)){
160             nextBoard = this.nextBoard(0); // その升に打った後の盤面を生成，格納.
161             nextBoardList.add(nextBoard); // リストの末尾に追加.
162             int substitute = board_Pattern.get(board_Pattern.size()-1).getStoneDiffer();
163             board_Pattern.get(board_Pattern.size()-1).setAlphaBeta(substitute);
164             ArrayList<Board> temporary = new ArrayList<>();
165             for(int i0=0;i0<board_Pattern.size();i0++)
```

```
166             temporary.add(board_Pattern.get(i0));
167           if(judge == 0) {
168             value[board_Pattern.size()-1] = board_Pattern.get(board_Pattern.size()-1)
169                                                                 .getStoneDiffer();
170             full_Data[board_Pattern.size()-1].add(temporary);
171           }
172           num012++;
173       }
174       else {
175         if(futureDepth >= 1) {
176           // パス状態を作成.
177           boolean settingPass[] = {nextBoard.getPass()[2],nextBoard.getPass()[3]
178                                    ,false,false};
179           nextBoard.setPass(settingPass); // パス状態を更新.
180           nextBoard = this.nextBoard(0); // その升に打った後の盤面を生成，格納.
181           // 先読み（再帰）.
182           nextBoard.comInsight(depth,futureDepth,nextBoard,fullDepth);
183         }
184       }
185       // 再帰後，パス状態を解除するためにターン状態を作成
186       int settingTurn[] = {nextBoard.getTurn()[0],beOutTurn};
187       nextBoard.setTurn(settingTurn); // ターン状態を更新
188     }
189     else // 先読み（再帰）.
190       this.repe.allRepetition(nextBoardList, futureDepth, futureDepth,fullDepth);
191   }
192
193   /**
194   * 指定した位置に石を打った新たな盤面を作成.
195   * @param place：石を打つ位置
196   * @return Board：新たな盤面
197   */
198   protected Board nextBoard(int place) {
199     Board nextBoard = copy();
200     nextBoard.setStone(nextBoard.board,nextBoard.getTurn()[1],place);
201     int settingStoneGet[] = {this.calculateStone(nextBoard.board)[0]
202                              ,this.calculateStone(nextBoard.board)[1]};
203     nextBoard.setStoneGet(settingStoneGet);
204     if(TECHNIQUE == 0)
205       nextBoard.setStoneDiffer((settingStoneGet[1] - settingStoneGet[0]));
206     else if(TECHNIQUE == 1) {
207       if((settingStoneGet[1] - settingStoneGet[0]) > 0)
208         nextBoard.setStoneDiffer(1);
```

```
209         else if((settingStoneGet[1] - settingStoneGet[0]) < 0)
210           nextBoard.setStoneDiffer(-1);
211         else
212           nextBoard.setStoneDiffer(0);
213       }
214       else if(TECHNIQUE == 2) {
215         if(SWITCHING == 0)        // 完全→完全
216           nextBoard.setStoneDiffer((settingStoneGet[1] - settingStoneGet[0]));
217         else if(SWITCHING == 1) {
218           if(methodSwitch) {
219             if((settingStoneGet[1] - settingStoneGet[0]) > 0)
220               nextBoard.setStoneDiffer(1);
221             else if((settingStoneGet[1] - settingStoneGet[0]) < 0)
222               nextBoard.setStoneDiffer(-1);
223             else
224               nextBoard.setStoneDiffer(0);
225           }
226           else
227             nextBoard.setStoneDiffer((settingStoneGet[1] - settingStoneGet[0]));
228         }
229         else if(SWITCHING == 2) {
230           if(methodSwitch)
231             nextBoard.setStoneDiffer((settingStoneGet[1] - settingStoneGet[0]));
232           else {
233             if((settingStoneGet[1] - settingStoneGet[0]) > 0)
234               nextBoard.setStoneDiffer(1);
235             else if((settingStoneGet[1] - settingStoneGet[0]) < 0)
236               nextBoard.setStoneDiffer(-1);
237             else
238               nextBoard.setStoneDiffer(0);
239           }
240         }
241         else if(SWITCHING == 3) {
242           if((settingStoneGet[1] - settingStoneGet[0]) > 0)
243             nextBoard.setStoneDiffer(1);
244           else if((settingStoneGet[1] - settingStoneGet[0]) < 0)
245             nextBoard.setStoneDiffer(-1);
246           else
247             nextBoard.setStoneDiffer(0);
248         }
249       }
250       int settingTurn[] = {nextBoard.getTurn()[0]
251                            ,nextBoard.nextTurn(nextBoard.getTurn()[1])};
```

```
252       nextBoard.setTurn(settingTurn);
253       int choiceNum = 0;
254       if(nextBoard.getEmptySquares() > 0) {
255         for(int i=1;i<=ALL_SIZE;i++) { // 各升に対して着手可能手の生成を行う.
256           if(nextBoard.board[i] == EMPTY) { // 空升なら着手可能.
257             // 指定の升が空升か判定.
258             if(check.currentFuture(nextBoard.board, nextBoard.getTurn()[1], i))
259               choiceNum += 1;
260           }
261         }
262       }
263       nextBoard.setNextChoice(choiceNum);
264       return nextBoard;
265     }
266
267     /**
268     * 盤面のコピーを作成.
269     * @return Board：盤面のコピー
270     */
271     protected Board copy() {
272       Board newBoard = new Game();
273       for(int i=1;i<=ALL_SIZE;i++)
274         newBoard.board[i] = this.board[i];
275       int settingTurn[] = {this.getTurn()[1],this.getTurn()[1]};
276       newBoard.setTurn(settingTurn);
277       newBoard.setPass(this.getPass());
278       newBoard.setEmptySquares(this.getEmptySquares());
279       newBoard.setPreviousPlace(this.getPreviousPlace());
280       newBoard.setAlphaBeta((settingTurn[0] == WHITE) ? Integer.MAX_VALUE
281                                                       : Integer.MIN_VALUE);
282       return newBoard;
283     }
284
285     /**
286     * 手番を変更.
287     * @param turn：直前のターン
288     * @return int：手番（1=白, -1=黒）
289     */
290     protected int nextTurn(int futureTurn) {
291       return futureTurn == WHITE ? BLACK : WHITE;
292     }
293
294     /**
```

```
295     * 得た石の計算..
296     * @param nextBoard 盤面
297     * @return int：得た石 [白・黒]
298     */
299     protected int[] calculateStone(int[] futureBoard) {
300       int[] stoneGet = new int[2];
301       for(int i=1;i<=ALL_SIZE;i++) {
302         if(futureBoard[i] == WHITE || futureBoard[i] == BLACK)
303           stoneGet[futureBoard[i] == WHITE ? 0 : 1] += 1;
304       }
305       return stoneGet;
306     }
307   }
308
```

CheckBoard クラス

```
1   package experiment.last.suspend;
2
3   import java.util.ArrayList;
4
5   /**
6    * 指定した位置の石を置くことが可能かチェック.
7    */
8   public abstract class CheckBoard extends ConstantAndStatic{
9
10    protected int yesDireCount; // 石を置ける方角の数（第一段階）
11    protected int yesCircleCount; // 石を置ける円周の数
12    // 外周に石を置く位置（変数：circleLine の場合）
13    protected int circleAround[] = new int[2];
14    // 石を置ける方角を記録（初期化：メソッド allDireCheck）
15    protected ArrayList<Integer> direRecord;
16    // 石を置ける外周の位置を記録（初期化：メソッド allDireCheck）
17    protected ArrayList<Integer> circleRecord;
18    // 変化する外周の石の位置を記録.
19    protected ArrayList<Integer>[] check_circleLine;
20
21    /**
22    * コンストラクタ，変数の初期化.
23    */
24    public CheckBoard() {
25      check_circleLine = new ArrayList[2];
26      this.yesDireCount = 0; // 初期状態：0
27      this.yesCircleCount =0; // 初期状態：0
```

```
28      }
29
30      /**
31      * 指定した位置の石を置く
32      * @param checkBoard：チェックする盤面
33      * @param cTurn：チェック・チェンジするターン
34      * @param place：チェック・チェンジするする石の位置
35      * @param changeBoard：チェンジする盤面
36      */
37      protected abstract void changeStone(int[] checkBoard,int cTurn,int place
38                                          , int[] changeBoard);
39
40      /**
41      * 外側を除く盤面を置換させる
42      * @param changeBoard：チェンジする盤面
43      * @param turn：チェンジするターン
44      * @param place：チェンジする石の位置
45      * @param dire：方角（指定した石から８方角）
46      * @return int[]：新たな盤面
47      */
48      protected abstract int[] oneDireChange(int[] changeBoard,int turn,int place
49                                             ,int dire);
50
51      /**
52      * 外側の盤面を置換させる
53      * @param changeBoard：チェンジする盤面
54      * @param turn：チェンジするターン
55      * @param place：チェンジする石の位置
56      * @param leftRight：外側の盤面（0:時計回り，1:反時計回り）
57      * @return int[]：新たな盤面
58      */
59      protected abstract int[] circleChange(int[] changeBoard,int turn,int place
60                                            ,int leftRight);
61
62      /**
63      * 指定した位置の石を置くことが可能かチェック.
64      * @param checkBoard：チェックする盤面
65      * @param checkTurn：チェックする手番
66      * @param place：チェックする石の位置
67      * @return boolean：指定した位置の石を置くことが可能か
68      */
69      public boolean currentFuture(int[] checkBoard, int checkTurn, int place) {
70        return (checkTurn == WHITE ? this.checkStone(checkBoard,WHITE,place,BLACK)
```

```
71                                  : this.checkStone(checkBoard,BLACK,place,WHITE));
72      }
73
74      /**
75      * 指定した位置の石を置くことが可能かチェック.（最終段階）
76      * @param checkBoard：チェックする盤面
77      * @param checkTurn：チェックする手番
78      * @param place：石を打つ位置
79      * @param checkColor：敵の石（白のターン→黒をチェック）
80      * @return boolean：指定した位置と指定した方角に石を置くことが可能か
81      */
82      private boolean checkStone(int[] checkBoard,int checkTurn,int place,int checkColor) {
83        boolean direJudge = false,circleJudge = false;
84        this.allDireCheck(checkBoard,place, checkColor);
85        if(this.yesDireCount != 0) {
86          for(int i=0;i<this.yesDireCount;i++) {
87            if(this.oneDireCheck(checkBoard,checkTurn,place,
88                this.direRecord.get(i), checkColor))
89               direJudge = true;
90            else {
91              this.direRecord.remove(i);
92              this.yesDireCount--;
93              i--;
94            }
95          }
96          if(this.yesCircleCount != 0) {
97            for(int i=0;i<this.yesCircleCount;i++) {
98              if(this.oneCircleCheck(checkBoard,checkTurn,place,this.circleRecord.get(i)
99                                     ,checkColor))
100               circleJudge = true;
101             else {
102               this.circleRecord.remove(i);
103               this.yesCircleCount--;
104               i--;
105             }
106           }
107         }
108       }
109       return ((direJudge || circleJudge) ? true : false);
110     }
111
112     /**
113     * 指定した位置の石を置くことが可能かチェック.（第一段階：外側の盤面を除く）
```

```
114     * @param checkBoard：チェックする盤面
115     * @param place：石を打つ位置
116     * @param checkColor：敵の石（白のターン→黒をチェック）
117     */
118     private void allDireCheck(int[] checkBoard,int place,int checkColor) {
119       this.direRecord = new ArrayList<>();
120       this.yesDireCount = 0;
121       for(int i=-(ALL_BESIDE + 1);i<=(ALL_BESIDE + 1);i++) {
122         if(place+i > 0 && place+i <= ALL_SIZE) {
123           if(checkBoard[place + i] == checkColor &&
124              ((i==-(ALL_BESIDE + 1) || i==-ALL_BESIDE || i==-(ALL_BESIDE - 1)
125               || i==-1 || i==1 || i==(ALL_BESIDE - 1) || i==ALL_BESIDE
126               || i==ALL_BESIDE + 1))) {
127              this.direRecord.add(i);
128              this.yesDireCount++;
129           }
130         }
131       }
132       this.circleRecord = new ArrayList<>();
133       this.yesCircleCount = 0;
134       for(int j=0;j<circle_Line.length;j++) {
135         for(int i=0;i<circle_Line[j].size();i++) {
136           if(circle_Line[j].get(i) == place) {
137             this.circleAround[j] = i;
138             this.allCircleCheck(checkBoard, place, checkColor,j);
139             break;
140           }
141         }
142       }
143     }
144
145     /**
146     * 指定した位置の石を置くことが可能かチェック.（第一段階：外側の盤面）
147     * @param checkBoard：チェックする盤面
148     * @param place：石を打つ位置
149     * @param checkColor：敵の石（白のターン→黒をチェック）
150     * @param leftRight：外側の盤面（0:時計回り，1:反時計回り）
151     */
152     private void allCircleCheck(int[] checkBoard,int place,int checkColor,int leftRight) {
153       check_circleLine[leftRight] = new ArrayList<>();
154       for(int i=0;i<circle_Line[leftRight].size();i++)
155         check_circleLine[leftRight].add(circle_Line[leftRight].get(i));
156       for(int i=0;i<this.circleAround[leftRight];i++)
```

```
157         check_circleLine[leftRight].add(circle_Line[leftRight].get(i));
158       if(checkBoard[check_circleLine[leftRight].get(this.circleAround[leftRight]+1)]
159          == checkColor) {
160         this.circleRecord.add(leftRight);
161         this.yesCircleCount++;
162       }
163     }
164
165     /**
166     * 指定した位置に石を置くことが可能かチェック.(第二段階：外側の盤面を除く)
167     * @param checkBoard：チェックする盤面
168     * @param checkTurn：チェックするターン
169     * @param place：石を打つ位置
170     * @param dire：方角（指定した石から８方角）
171     * @param checkColor：敵の石（白のターン→黒をチェック）
172     * @return boolean：指定した位置と指定した方角に石を置くことが可能か
173     */
174     private boolean oneDireCheck(int[] checkBoard,int checkTurn,int place,int dire
175                                  , int checkColor) {
176       int highSize = (VERTICAL >= BESIDE ? VERTICAL : BESIDE);
177       for(int j=2;j<=highSize-1;j++) {
178         if(place + dire * j > 0 && place + dire * j <= ALL_SIZE) {
179           if(checkBoard[place + dire * j] == checkTurn)
180             return true;
181           else if(checkBoard[place + dire * j] == checkColor)
182             continue;
183           else
184             break;
185         }
186       }
187       return false;
188     }
189
190     /**
191     * 指定した位置に石を置くことが可能かチェック.(第二段階：外側の盤面)
192     * @param checkBoard：チェックする盤面
193     * @param checkTurn：チェックするターン
194     * @param place：石を打つ位置
195     * @param leftRight：外側の盤面（0:時計回り，1:反時計回り）
196     * @param checkColor：敵の石（白のターン→黒をチェック）
197     * @return boolean：指定した位置と指定した方角に石を置くことが可能か
198     */
199     private boolean oneCircleCheck(int[] checkBoard,int checkTurn,int place,int leftRight
```

```
                                    ,int checkColor) {
    for(int i=this.circleAround[leftRight];i<check_circleLine[leftRight].size()-2;i++) {
      if(checkBoard[check_circleLine[leftRight].get(i+2)] == checkTurn)
        return true;
      else if(checkBoard[check_circleLine[leftRight].get(i+2)] == checkColor)
        continue;
      else
        break;
    }
    return false;
  }
}
```

ChangeBoard クラス

```
package experiment.last.suspend;

/**
* 指定した位置の石を置く.
*/
public class ChangeBoard  extends CheckBoard {

  /**
  * 指定した位置の石を置く
  * @param checkBoard：チェックする盤面
  * @param cTurn：チェック・チェンジするターン
  * @param place：チェックする石の位置
  * @param changeBoard：チェンジする盤面
  */
  public void changeStone(int[] checkBoard,int cTurn,int place, int[] changeBoard) {
    if(this.currentFuture(checkBoard, cTurn, place)) {
      if(this.yesDireCount != 0) {
        for(int i=0;i<this.yesDireCount;i++)
          changeBoard = this.oneDireChange(changeBoard,cTurn, place
                                          , this.direRecord.get(i));
      }
      if(this.yesCircleCount != 0) {
        for(int i=0;i<this.yesCircleCount;i++)
          changeBoard = this.circleChange(changeBoard,cTurn,place
                                         , this.circleRecord.get(i));
      }
    }
  }
```

```
29
30      /**
31      * 外側を除く盤面を置換させる
32      * @param changeBoard：チェンジする盤面
33      * @param turn：チェンジするターン
34      * @param place：チェンジする石の位置
35      * @param dire：方角（指定した石から 8 方角）
36      * @return int[]：新たな盤面
37      */
38      public int[] oneDireChange(int[] changeBoard,int turn,int place,int dire) {
39        int highSize = (VERTICAL >= BESIDE ? VERTICAL : BESIDE);
40        for(int j=2;j<=highSize - 1;j++) {
41          if(place + dire * j > 0 && place + dire * j <= ALL_SIZE) {
42            if(changeBoard[place + dire * j] == turn) {
43              for(int i=1;i<j;i++)
44                changeBoard[place + dire * (i)] = turn;
45            }
46            else if(changeBoard[place + dire * j] == EMPTY || changeBoard[place + dire * j]
47                    == BORDER)
48              break;
49          }
50        }
51        return changeBoard;
52      }
53
54      /**
55      * 外側の盤面を置換させる
56      * @param changeBoard：チェンジする盤面
57      * @param turn：チェンジするターン
58      * @param place：チェンジする石の位置
59      * @param leftRight：外側の盤面（0:時計回り，1:反時計回り）
60      * @return int[]：新たな盤面
61      */
62      public int[] circleChange(int[] changeBoard,int turn,int place,int leftRight) {
63        for(int j=2;j<check_circleLine[leftRight].size();j++) {
64         if(changeBoard[check_circleLine[leftRight].get(circleAround[leftRight]+j)] == turn) {
65           for(int i=1;i<j;i++)
66             changeBoard[check_circleLine[leftRight].get(circleAround[leftRight]+i)] = turn;
67           break;
68         }
69         else if(changeBoard[check_circleLine[leftRight].get(circleAround[leftRight]+j)]
70                == EMPTY
71                || changeBoard[check_circleLine[leftRight].get(circleAround[leftRight]+j)]
```

```
72                == BORDER)
73            break;
74        }
75        return changeBoard;
76    }
77  }
78
```

ConstantAndStatic クラス

```
1   package experiment.last.suspend;
2
3   import java.util.ArrayList;
4
5   /**
6    * 定数と static 変数のクラス.
7    *
8    */
9
10   public class ConstantAndStatic {
11     // 定数
12     // 解析手法の番号（0：完全読み切り，1：必勝読み切り，2：一定の先読み）.
13     protected static final int TECHNIQUE = 0;
14     // 解析手法の移行番号（0:完全→完全，1:完全→必勝，2:必勝→完全，3:必勝→必勝）.
15     protected static final int SWITCHING = 3;
16     protected static final int VERTICAL = 4; // 縦の長さ（盤面）.
17     protected static final int BESIDE = 4; // 横の長さ（盤面）.
18     // 盤外も含めた横の長さ.
19     protected static final int ALL_BESIDE = BESIDE + 2;
20     // 盤外も含めた盤面サイズ.
21     protected static final int ALL_SIZE = VERTICAL * ALL_BESIDE;
22     protected static final int WHITE = 1; // 白.
23     protected static final int BLACK = -1; // 黒.
24     protected static final int EMPTY = 0; // 空白.
25     protected static final int BORDER = Integer.MAX_VALUE; // 盤外.
26     // デバグ用 true にすると実行中に盤面の評価値等のデータを表示する.
27     protected static final boolean DEBUG = true;
28     // static 変数
29     protected static int depth = 8; // 完全・必勝読み切りの先読数.
30     protected static int depth1 = 6; // 一定数の先読みの先読数（１回目の深さ）.
31     // 一定数の先読みの先読数（２回目の深さ）.
32     protected static int depth2 = depth-depth1+1;
33     // 円周の盤面の番号を記憶（要素０：時計回り，要素１：反時計回り）.
34     protected static ArrayList<Integer>[] circle_Line;
```

```
35      protected static ArrayList<Board> board_Pattern; // 初期～終局までの盤面.
36      protected static int judge; // 枝切り位置（探索中の先読みの深さ）.
37      // 先読みの深さごとに一時記憶させる初期～終局までの盤面.
38      protected static ArrayList<ArrayList<Board>>[] full_Data;
39      protected static int[] value; // 先読みの深さごとに一時記憶させる評価値.
40      protected static long num012; // 探索数１（１回目の探索時）.
41      protected static long num2; // 探索数２（２回目の探索時）.
42      // 探索後の最善手の初期～終局までの盤面（１回目の探索時）.
43      protected static ArrayList<ArrayList<Board>> full_Result1;
44      // 探索後の最善手の評価値（１回目の探索時）.
45      protected static int alphabeta_Result1;
46      protected static boolean methodSwitch; // 手法の変更可能かチェック.
47      // 探索後の最善手の初期～終局までの盤面（２回目の探索時）.
48      protected static ArrayList<ArrayList<Board>> full_Result2;
49      // 探索後の最善手の評価値（２回目の探索時）.
50      protected static int alphabeta_Result2;
51      protected static ArrayList<Integer> alphabeta_Result3; // 最善手の評価値.
52    }
53
```

Repetition クラス

```
1   package experiment.last.suspend;
2
3   import java.time.Duration;
4   import java.time.LocalDateTime;
5   import java.util.ArrayList;
6   import java.util.Random;
7
8   /**
9    * 先読みするクラス.
10   */
11   public abstract class Repetition extends ConstantAndStatic{
12     protected final int MAX = Integer.MAX_VALUE; // 最大値
13     protected final int MIN = Integer.MIN_VALUE; // 最小値
14     private ShowText text; // 参照型変数を定義
15     private ShowBoard show; // 参照型変数を定義
16     private LocalDateTime ldtStart1; // 実行時間の開始日時を格納.
17     private LocalDateTime ldtEnd1; // 実行時間の終了日時を格納（探索１回目）.
18     private LocalDateTime ldtEnd2; // 実行時間の終了日時を格納（探索２回目）.
19     private Random random; // 乱数発生用
20
21     /**
22     * コンストラクタ，変数の初期化.
```

```
23      */
24      public Repetition() {
25        this.text = new ShowText(); // オブジェクト生成
26        this.bo = new ShowBoard(); // オブジェクト生成
27        long seed = System.currentTimeMillis(); // 現在時刻から乱数の種を生成
28        this.random = new Random(seed); // 乱数発生用
29      }
30
31
32      /**
33      * 指定した先読み後に選択した 1 手の盤面を返す.
34      * @param nextBoardList1：先読みする直前の盤面のリスト.
35      * @return Board1：ランダムに選択した盤面.
36      */
37      public Board firstRepetition(ArrayList<Board> nextBoardList1) {
38        ArrayList<Board> nextBoardList = nextBoardList1;
39        Board nextBoard;
40        this.reset_Initial();
41        alphabeta_Result1 = 0;
42        full_Result1 = new ArrayList<>();
43        board_Pattern = new ArrayList<>();
44        /* 先読みして着手する手を選択 */
45        this.ldtStart1 = LocalDateTime.now();
46        int useDepth = (TECHNIQUE <= 1) ? depth : depth1;
47        if(useDepth > 1) {
48          for (int i=0; i<2; ++i) { // 最大：nextBoardList.size()
49            if(TECHNIQUE == 2) {
50              if((depth/2) <= depth1) {
51                if(i != 0) {
52                  this.ldtEnd2 = LocalDateTime.now();
53                  this.information2();
54                }
55              }
56            }
57            else {
58              if(i != 0) {
59                this.ldtEnd2 = LocalDateTime.now();
60                this.information2();
61              }
62            }
63            System.out.println("・" + nextBoardList.get(i).getPreviousPlace());
64            nextBoard = nextBoardList.get(i);
65            this.reset_Repetition();
```

```
66            board_Pattern.add(nextBoardList.get(i));
67            nextBoard.comInsight(useDepth,useDepth-1,nextBoard,useDepth);
68            this.endUpdata(true);
69          }
70        }
71        this.ldtEnd1 = LocalDateTime.now();
72        if(TECHNIQUE == 2)
73          this.secondRepetition();
74        this.ldtEnd2 = LocalDateTime.now();
75        this.information();
76        return nextBoardList.get(this.random.nextInt(nextBoardList.size()));
77      }
78
79      /**
80      * 1回目の指定した先読数で有望な手をさらに先読みして
81      * 最も優れている手を求める（2回目の探索）.
82      */
83      private void secondRepetition() {
84        Board nextBoard;
85        num2 = num012;
86        this.reset_Initial();
87        full_Result2 = new ArrayList<>();
88        alphabeta_Result2 = 0;
89        alphabeta_Result3 = new ArrayList<>();
90        methodSwitch = true;
91        System.out.println("第二段階：先読み");
92        if(depth2 > 1) {
93          for (int i=0; i<full_Result1.size(); ++i) {
94            if((depth/2) < depth2) {
95              if(i != 0) {
96                this.ldtEnd2 = LocalDateTime.now();
97                this.information3();
98              }
99            }
100           int size = full_Result1.get(i).size()-1;
101           nextBoard = full_Result1.get(i).get(size);
102           this.reset_Repetition();
103           board_Pattern.add(full_Result1.get(i).get(size));
104           board_Pattern.get(0).setAlphaBeta((board_Pattern.get(0).getTurn()[0] == BLACK)
105                                             ? MIN : MAX);
106           nextBoard.comInsight(depth2,depth2-1,nextBoard,depth2);
107           this.endUpdata2(i);
108         }
```

```
109         }
110       }
111
112
113       /**
114       * 指定した手数先まで再帰的に先読みする.
115       * @param nextBoardList1：先読みする直前の盤面のリスト.
116       * @param futureDepth：先読みする手数
117       * @param depth：先読み数（現状）
118       * @param furdepth：先読み数（初期）
119       */
120       public void allRepetition(ArrayList<Board> nextBoardList1, int futureDepth
121                                 , int depth,int fullDepth) {
122         ArrayList<Board> nextBoardList = nextBoardList1;
123         Board nextBoard;
124         /* 先読み（再帰）*/
125         for (int i=0; i<nextBoardList.size(); ++i) { // 盤面の昇順から先読み
126           if(TECHNIQUE == 0) {
127             if(i==1)
128               this.tech_Perfect();
129           }
130           else if(TECHNIQUE == 1) {
131             if(i!=0)
132               this.tech_Victory(fullDepth -futureDepth);
133           }
134           else if(TECHNIQUE == 2) {
135             if(SWITCHING == 0) {
136               if(i==1)
137               this.tech_Perfect();
138             }
139             else if(SWITCHING == 1) {
140               if(methodSwitch) {
141                 if(i!=0)
142                   this.tech_Victory(fullDepth -futureDepth);
143               }
144               else {
145                 if(i==1)
146                 this.tech_Perfect();
147               }
148             }
149             else if(SWITCHING == 2) {
150               if(methodSwitch) {
151                 if(i==1)
```

```
152                 this.tech_Perfect();
153             }
154             else {
155               if(i!=0)
156                 this.tech_Victory(fullDepth -futureDepth);
157             }
158           }
159           else if(SWITCHING == 3) {
160             if(i!=0)
161               this.tech_Victory(fullDepth -futureDepth);
162           }
163         }
164         if(judge == 0) {
165           nextBoard = nextBoardList.get(i);
166           board_Pattern.add(nextBoardList.get(i));
167           if(futureDepth==1) {
168             int substitute = board_Pattern.get(board_Pattern.size()-1).getStoneDiffer();
169             board_Pattern.get(board_Pattern.size()-1).setAlphaBeta(substitute);
170             ArrayList<Board> temporary = new ArrayList<>();
171             for(int i0=0;i0<board_Pattern.size();i0++)
172               temporary.add(board_Pattern.get(i0));
173
174             if(judge == 0) {
175               value[board_Pattern.size()-1] = board_Pattern
176                                               .get(board_Pattern.size()-1).getStoneDiffer();
177               full_Data[board_Pattern.size()-1].add(temporary);
178             }
179             num012++;
180           }
181           else if(futureDepth > 1) // 1手進めて再帰
182             nextBoard.comInsight(depth,futureDepth-1,nextBoard,fullDepth);
183         }
184         if(judge > 0) {
185           if((fullDepth -futureDepth) == judge) {
186             judge = 0;
187             if(i == nextBoardList.size()-1)
188               this.delete(fullDepth - futureDepth);
189           }
190         }
191         else  // JUDGE == 0
192           this.searchData(fullDepth - futureDepth);
193       }
194     }
```

```
195
196     /**
197     * 全体の先読み開始前に static 変数の定義する.
198     */
199     private void reset_Initial() {
200       num012 = 0;
201       methodSwitch = false;
202       full_Data = new ArrayList[depth];
203       for(int i=0;i<full_Data.length;i++)
204         full_Data[i] = new ArrayList<>();
205       value = new int[depth];
206     }
207
208     /**
209     * 初手着手可能手ごとの先読み開始前に static 変数のリセットを行う.
210     */
211     private void reset_Repetition() {
212       for(int i0=0;i0<full_Data.length;i0++) {
213         full_Data[i0].clear();
214         value[i0] = MAX;
215       }
216       judge = 0;
217       board_Pattern.clear();
218     }
219
220     /**
221     * 手法：完全読み切り.
222     */
223     protected abstract void tech_Perfect();
224
225     /**
226     * 手法：必勝読み切り.
227     * @param search：探索位置
228     */
229     protected abstract void tech_Victory(int search);
230
231     /**
232     * 探索を行う.
233     * @param search：探索位置
234     */
235     protected abstract void searchData(int search);
236
237     /**
```

```
238     * 探索中の盤面を削除.
239     * @param search：探索中の位置
240     */
241     protected abstract void delete(int search);
242
243     /**
244     * 選択した盤面をまとめて削除.
245     * @param deleteLine：先読みの削除位置
246     */
247     protected abstract void delete2(int deleteLine);
248
249     /**
250     * 初手着手可能手ごとの探索終了時に結果を更新（1回目の探索）.
251     * @param first：分割しているか
252     */
253     protected abstract void endUpdata(boolean first);
254
255     /**
256     * 初手着手可能手ごとの探索終了時に結果を更新（2回目の探索）.
257     * @param count：更新回数（0：初回）
258     */
259     protected abstract void endUpdata2(int count);
260
261     /**
262     * 先読みパターン数，探索結果（完全解析した結果），実行時間を表示.
263     */
264     private void information() {
265       System.out.println("＜結果＞");
266       System.out.println("・" + ((TECHNIQUE <= 1) ? ("探索数1：" + num012)
267                           : ("探索数1：" + num2 + "，探索数2：" + num012)));
268       System.out.println("･" + ((TECHNIQUE <= 1) ? ("ALPHABETA:" + alphabeta_Result1)
269              : ("ALPHABETA1:" + alphabeta_Result1 + ", ALPHABETA2:" + alphabeta_Result2)));
270       System.out.println("FULL_DATA：");
271       text.resultshow();
272       System.out.println("FULL_RESULT：");
273       this.resultS();
274       System.out.println("＜先読み終了＞");
275       System.out.print("実行時間：\n" + "・開始時刻：" + this.ldtStart1 + "\n"
276                       + "・終了時刻：" + this.ldtEnd2 + "\n");
277       Duration summerVacationDuration = Duration.between(this.ldtStart1, this.ldtEnd2);
278       Duration summerVacationDuration2 = Duration.between(this.ldtEnd1, this.ldtEnd2);
279       System.out.print("・処理時間1：" + (summerVacationDuration) + "\n");
280       System.out.print("・処理時間2：" + (summerVacationDuration2) + "\n");
```

```
281         System.out.println("＜石の選択＞");
282       }
283
284       /**
285       * 先読みパターン数，探索結果（完全解析した結果），実行時間を表示.
286       */
287       private void information2() {
288         System.out.println("＜途中経過＞");
289         System.out.println("・" + ("探索数１：" + num012));
290         System.out.println("・" +("ALPHABETA1：" + alphabeta_Result1));
291         System.out.println("＜先読み終了＞");
292         System.out.print("実行時間：\n" + "・開始時刻：" + this.ldtStart1 + "\n"
293                          + "・終了時刻：" + this.ldtEnd2 + "\n");
294         Duration summerVacationDuration = Duration.between(this.ldtStart1, this.ldtEnd2);
295         System.out.print("・処理時間１：" + (summerVacationDuration) + "\n");
296         System.out.println();
297       }
298
299       /**
300       * 先読みパターン数，探索結果（完全解析した結果），実行時間を表示.
301       */
302       private void information3() {
303         System.out.println("＜途中経過＞");
304         System.out.println("・" +  ("探索数１：" + num2 + "，探索数２：" + num012));
305         System.out.println("・" +("ALPHABETA1：" + alphabeta_Result1
306                                  + "，ALPHABETA2：" + alphabeta_Result2));
307         System.out.println("FULL_DATA：");
308         text.resultshow();
309         System.out.println("FULL_RESULT：");
310         this.resultS();
311         System.out.println("＜先読み終了＞");
312         System.out.print("実行時間：\n" + "・開始時刻：" + this.ldtStart1 + "\n"
313                          + "・終了時刻：" + this.ldtEnd2 + "\n");
314         Duration summerVacationDuration = Duration.between(this.ldtStart1, this.ldtEnd2);
315         System.out.print("・処理時間１：" + (summerVacationDuration) + "\n");
316         System.out.println();
317       }
318
319       /**
320       * 探索結果から最善手の盤面・評価値を表示.
321       */
322       private void resultS() {
323         if(TECHNIQUE <= 1) {
```

```
324         for(int i=0;i<full_Result1.size();i++) {
325           System.out.print((i+1) +", 評価値："
326                            + full_Result1.get(i).get(0).getAlphaBeta() + "→/");
327           for(int j=0;j<full_Result1.get(i).size();j++)
328             System.out.print(full_Result1.get(i).get(j).getPreviousPlace() + "/");
329           System.out.println();
330           for(int j=0;j<full_Result1.get(i).size();j++)
331             bo.show(full_Result1.get(i).get(j).board);
332         }
333       }
334       else {
335         for(int i0=0;i0<full_Result2.size();i0++) {
336           System.out.print((i0+1) +", ");
337           for(int j0=0;j0<full_Result2.get(i0).size();j0++)
338             System.out.print(full_Result2.get(i0).get(j0).getPreviousPlace() + "/");
339           System.out.print(", 評価値：" + alphabeta_Result1 + "→"
340                                                    + alphabeta_Result3.get(i0));
341           System.out.println();
342         }
343       }
344     }
345   }
346
```

Technique クラス

```
1    package experiment.last.suspend;
2
3    public abstract class Technique extends Repetition {
4
5      public Technique() {
6        super();
7      }
8
9      /**
10     * 完全読み切り.
11     */
12     protected void tech_Perfect() {
13       int series =0;
14       for(int i0=0;i0<full_Data.length;i0++) {
15         if(full_Data[i0].size() > 0)
16           series += 1;
17       }
18       if(series > 1) {
```

```
19          for(int i1=value.length-1;i1>0;i1--) {
20            if(value[i1] != MAX && value[i1-1] !=MAX) {
21              int dataSize1 = full_Data[i1-1].get(0).size()
22                , dataSize2 = full_Data[i1].get(0).size();
23              for(int i2=0;i2<(dataSize1 < dataSize2 ? dataSize1 : dataSize2);i2++) {
24                if(full_Data[i1-1].get(0).get(i2).getPreviousPlace()
25                   != full_Data[i1].get(0).get(i2).getPreviousPlace()) {
26                  if(full_Data[i1].get(0).get(i2-1).getTurn()[0]  == WHITE) {
27                    if(full_Data[i1].get(0).get(i2).getTurn()[0]  == BLACK) {
28                      if(full_Data[i1-1].get(0).get(i2).getAlphaBeta()
29                         < full_Data[i1].get(0).get(i2).getAlphaBeta())
30                      this.pruning(i1); // 枝切り
31                    }
32                  }
33                  else {
34                    if(full_Data[i1].get(0).get(i2).getTurn()[0]  == WHITE) {
35                      if(full_Data[i1-1].get(0).get(i2).getAlphaBeta()
36                         > full_Data[i1].get(0).get(i2).getAlphaBeta())
37                      this.pruning(i1); // 枝切り
38                    }
39                  }
40                  break;
41                }
42              }
43              break;
44            }
45          }
46        }
47      }
48
49
50      /**
51      * 必勝読み切り.
52      * @param search：探索位置
53      */
54      protected void tech_Victory(int search) {
55        boolean p = false;
56        if(board_Pattern.get(search- 1).getTurn()[0] == WHITE) {
57          if(board_Pattern.get(search- 1).getAlphaBeta() == -1) {
58            p = true;
59            this.pruning(search);
60          }
61        }
```

```
62          else {
63            if(board_Pattern.get(search- 1).getAlphaBeta() == 1) {
64              p = true;
65              this.pruning(search);
66            }
67          }
68          if(!p) {
69            this.tech_Perfect();
70          }
71        }
72
73
74        /**
75        * 枝切りを行い，static 変数 FULL_DATA，VALUE，DISPLAY から削除.
76        * @param prun : 枝切り位置
77        */
78        private void pruning(int prun) {
79          judge = prun;
80          for(int i3=prun;i3<full_Data.length;i3++) {
81            full_Data[i3].clear();
82            value[i3] = MAX;
83          }
84          this.delete2(prun);
85        }
86      }
87
```

Exploratory クラス

```
1   package experiment.last.suspend;
2
3   public abstract class Exploratory extends Technique{
4
5     /**
6     * コンストラクタ.
7     */
8     public Exploratory() {
9       super();
10    }
11
12    /**
13    * 記憶する盤面の更新 1
14    * @param back : 探索中の位置
15    * @param front : 探索中の 1 段階前の位置
```

```
16      */
17      protected abstract void update(int back, int front);
18
19      /**
20      * 記憶する盤面の更新２
21      * @param back：探索中の位置
22      * @param front：探索中の１段階前の位置
23      */
24      protected abstract void update2(int back, int front);
25
26      /**
27      * static 変数 FULL_DATA の要素ごとに２以上の時は１つに縮小する.
28      * @param position：static 変数 FULL_DATA の要素番号
29      */
30      protected abstract void update3(int back);
31
32      /**
33      * 探索を行う.
34      * @param search：探索位置
35      */
36      protected void searchData(int search) {
37        if(board_Pattern.get(search-1).getAlphaBeta() == MAX
38           || board_Pattern.get(search-1).getAlphaBeta() == MIN)
39           board_Pattern.get(search-1).setAlphaBeta(board_Pattern.get(search)
40                                                   .getAlphaBeta());
41        if(board_Pattern.get(search-1).getTurn()[0] == WHITE) {
42          if(board_Pattern.get(search-1).getAlphaBeta()
43             > board_Pattern.get(search).getAlphaBeta()) {
44            board_Pattern.get(search-1).setAlphaBeta(board_Pattern.get(search)
45                                                     .getAlphaBeta());
46            this.update(search,search-1);
47          }
48          else if(board_Pattern.get(search-1).getAlphaBeta()
49                  == board_Pattern.get(search).getAlphaBeta())
50            this.update2(search,search-1);
51          else
52            this.update3(search);
53        }
54        else {
55          if(board_Pattern.get(search-1).getAlphaBeta()
56             < board_Pattern.get(search).getAlphaBeta()) {
57            board_Pattern.get(search-1).setAlphaBeta(board_Pattern.get(search)
58                                                     .getAlphaBeta());
```

```
59            this.update(search,search-1);
60          }
61          else if(board_Pattern.get(search-1).getAlphaBeta()
62                  == board_Pattern.get(search).getAlphaBeta())
63            this.update2(search,search-1);
64          else
65            this.update3(search);
66        }
67        this.delete(search);
68      }
69
70      /**
71      * 探索中の盤面を削除.
72      * @param search：探索中の位置
73      */
74      protected void delete(int search) {
75        for(int j=(search);j<board_Pattern.size();j++)
76          board_Pattern.remove(j);
77      }
78
79      /**
80      * 選択した盤面をまとめて削除.
81      * @param deleteLine：先読みの削除位置
82      */
83      protected void delete2(int deleteLine) {
84        for(int j=board_Pattern.size()-1;j>=deleteLine;j--)
85          board_Pattern.remove(j);
86      }
87    }
88
```

UpdateData クラス

```
1   package experiment.last.suspend;
2
3   import java.util.ArrayList;
4
5   public class UpdateData extends Exploratory{
6
7     private boolean judgeFirst1 = true;
8     private boolean judgeFirst2 = true;
9     /**
10    * コンストラクタ.
11    */
```

```
12    public UpdateData() {
13      super();
14    }
15
16    /**
17    * 記憶する盤面の更新 1
18    * @param back : 探索中の位置
19    * @param front : 探索中の 1 段階前の位置
20    */
21    protected void update(int back, int front) {
22      boolean judge1 = true;
23      if(full_Data[front].size() > 0) {
24        for(int i0=0;i0<full_Data[front].size();i0++) {
25          for(int i1 = 0;i1<board_Pattern.size();i1++) {
26            if(board_Pattern.get(i1).getAlphaBeta()
27               != full_Data[front].get(i0).get(i1).getAlphaBeta())
28              judge1 = false;
29          }
30          if(judge1)
31            break;
32          else
33            judge1 = true;
34        }
35      }
36      if(judge1)
37        this.update2(back,front);
38    }
39
40    /**
41    * 記憶する盤面の更新 2
42    * @param back : 探索中の位置
43    * @param front : 探索中の 1 段階前の位置
44    */
45    protected void update2(int back, int front) {
46      if(full_Data[back].size() == 1) {
47        full_Data[front].add(full_Data[back].get(0));
48        full_Data[back].clear();
49        value[front] = full_Data[front].get(0).get(front).getAlphaBeta();
50        value[back] = MAX;
51      }
52      else {
53        int win = 0;
54        for(int j=0;j<full_Data[back].size();j++) {
```

```
55          for(int i=0;i<full_Data[back].get(j).size();i++) {
56            if(full_Data[back].get(win).get(i).getPreviousPlace()
57               != full_Data[back].get(j).get(i).getPreviousPlace()) {
58              if(full_Data[back].get(win).get(i-1).getTurn()[0] == WHITE) {
59                if(full_Data[back].get(win).get(i).getAlphaBeta()
60                   > full_Data[back].get(j).get(i).getAlphaBeta())
61                  win = j;
62              }
63              else {
64                if(full_Data[back].get(win).get(i).getAlphaBeta()
65                   < full_Data[back].get(j).get(i).getAlphaBeta())
66                  win = j;
67              }
68              break;
69            }
70          }
71        }
72        full_Data[front].add(full_Data[back].get(win));
73        full_Data[back].clear();
74        value[front] = full_Data[front].get(0).get(front).getAlphaBeta();
75        value[back] = MAX;
76      }
77      for(int i=0;i<full_Data.length;i++) {
78        if(full_Data[i].size() > 1) {
79          this.update2_1(i);
80        }
81      }
82    }
83
84
85    /**
86    * static 変数 FULL_DATA の要素ごとに 2 以上の時は 1 つに縮小する.
87    * @param position : static 変数 FULL_DATA の要素番号
88    */
89    private void update2_1(int position){
90      int win = 0;
91      for(int j=0;j<full_Data[position].size();j++) {
92        for(int i=0;i<full_Data[position].get(j).size();i++) {
93          if(full_Data[position].get(win).get(i).getPreviousPlace()
94             != full_Data[position].get(j).get(i).getPreviousPlace()) {
95            if(full_Data[position].get(win).get(i-1).getTurn()[0] == WHITE) {
96              if(full_Data[position].get(win).get(i).getAlphaBeta()
97                 > full_Data[position].get(j).get(i).getAlphaBeta())
```

```
98                win = j;
99              else if(full_Data[position].get(win).get(i).getAlphaBeta()
100                     == full_Data[position].get(j).get(i).getAlphaBeta())
101               win = j;
102           }
103           else {
104             if(full_Data[position].get(win).get(i).getAlphaBeta()
105                < full_Data[position].get(j).get(i).getAlphaBeta())
106               win = j;
107             else if(full_Data[position].get(win).get(i).getAlphaBeta()
108                      == full_Data[position].get(j).get(i).getAlphaBeta())
109               win = j;
110           }
111         }
112       }
113     }
114     ArrayList<ArrayList<Board>>[] kari = new ArrayList[1];
115     kari[0] = new ArrayList<>();
116     kari[0].add(full_Data[position].get(win));
117     full_Data[position].clear();
118     full_Data[position].add(kari[0].get(0));
119     value[position] = full_Data[position].get(0).get(position).getAlphaBeta();
120   }
121
122   /**
123   * 記憶する盤面の更新３
124   * @param back：探索中の位置
125   */
126   protected void update3(int back) {
127     full_Data[back].clear();
128     value[back] = MAX;  //  System.out.println("[更新３]fullData：");
129   }
130
131   /**
132   * 初手着手可能手ごとの探索終了時に結果を更新（１回目の探索）.
133   * @param first：分割しているか
134   */
135   protected void endUpdata(boolean first) {
136     ArrayList<Board> temporary = new ArrayList<>();
137     for(int f =0;f<full_Data[0].get(0).size();f++)
138       temporary.add(full_Data[0].get(0).get(f));
139     if(judgeFirst1 == first) {
140       alphabeta_Result1 = full_Data[0].get(0).get(0).getAlphaBeta();
```

```
141         full_Result1.add(temporary);
142         judgeFirst1 =false;
143       }
144       else {
145         if(full_Data[0].get(0).get(0).getTurn()[0] == BLACK) {
146           if(alphabeta_Result1 < full_Data[0].get(0).get(0).getAlphaBeta()) {
147             alphabeta_Result1 = full_Data[0].get(0).get(0).getAlphaBeta();
148             full_Result1.clear();
149             full_Result1.add(temporary);
150           }
151           else if(alphabeta_Result1 == full_Data[0].get(0).get(0).getAlphaBeta())
152             full_Result1.add(temporary);
153         }
154         else {
155           if(alphabeta_Result1 > full_Data[0].get(0).get(0).getAlphaBeta()) {
156             alphabeta_Result1 = full_Data[0].get(0).get(0).getAlphaBeta();
157             full_Result1.clear();
158             full_Result1.add(temporary);
159           }
160           else if(alphabeta_Result1 == full_Data[0].get(0).get(0).getAlphaBeta())
161             full_Result1.add(temporary);
162         }
163       }
164     }
165
166     /**
167     * 初手着手可能手ごとの探索終了時に結果を更新（２回目の探索）.
168     * @param count：更新回数（0：初回）
169     */
170     protected void endUpdata2(int count) {
171       ArrayList<Board> temporary = new ArrayList<>();
172       for(int b =0;b<full_Result1.get(count).size();b++)
173         temporary.add(full_Result1.get(count).get(b));
174       if(temporary.get(temporary.size()-1).getPreviousPlace()
175          == full_Data[0].get(0).get(0).getPreviousPlace()) {
176          for(int f =1;f<full_Data[0].get(0).size();f++)
177            temporary.add(full_Data[0].get(0).get(f));
178       }
179       full_Result2.add(temporary);
180       alphabeta_Result3.add(full_Data[0].get(0).get(0).getAlphaBeta());
181       if(count == 0) {
182         alphabeta_Result2 = full_Data[0].get(0).get(0).getAlphaBeta();
183       }
```

```
184       else {
185         if(full_Data[0].get(0).get(0).getTurn()[0] == BLACK) {
186           if(alphabeta_Result2 < full_Data[0].get(0).get(0).getAlphaBeta())
187             alphabeta_Result2 = full_Data[0].get(0).get(0).getAlphaBeta();
188         }
189         else {
190           if(alphabeta_Result2 > full_Data[0].get(0).get(0).getAlphaBeta())
191             alphabeta_Result2 = full_Data[0].get(0).get(0).getAlphaBeta();
192         }
193       }
194     }
195   }
196
```

ShowBoard クラス

```
1     package experiment.last.suspend;
2
3     /**
4     * ニップの盤面を表示するクラス.
5     */
6     public class ShowBoard extends ConstantAndStatic{
7
8       private int interval = BESIDE; // 2 つの盤を表示するための間隔.
9       // 盤面の桁数の最大値.
10      private int digit = String.valueOf(ALL_SIZE).length();
11
12      /**
13      * ニップの盤面を表示.
14      * @param board : 盤面
15      */
16      public void show(int[] board) {
17        System.out.println();
18        if(VERTICAL == 4 && BESIDE== 4) {
19          for(int u=0;u<BESIDE / 2-1;u++)
20            System.out.print("    ");
21          System.out.print("／-＼");
22        }
23        else
24          this.showUp((VERTICAL <= BESIDE) ? (VERTICAL == 4 ? 2 : 3)
25                                           : (BESIDE == 4 ? 2 : 3));
26
27        System.out.println();
28
```

```
29          int half = (VERTICAL <= BESIDE) ? BESIDE / 2 : VERTICAL / 2;
30          int allHalf = ALL_SIZE /2;
31          for(int h=2;h<ALL_SIZE;h+=ALL_BESIDE) {
32            int adjust = 0;
33            for(int z0=0;z0<half;z0++) {
34              if(((h >= (allHalf - BESIDE - z0 * ALL_BESIDE)
35              && (allHalf - 1 - z0 * ALL_BESIDE) >= h))
36              || ((h >= (allHalf + 2 + z0 * ALL_BESIDE)
37              && (allHalf + ALL_BESIDE -1+ z0 * ALL_BESIDE) >= h))) {
38                int differ = 0;
39                if(VERTICAL == 4 && BESIDE== 4) {
40                  for(int z1=0;z1<z0;z1++)
41                    System.out.print(" ");
42                }
43                else {
44                  int shortLength = (VERTICAL < BESIDE) ? VERTICAL : BESIDE;
45                  differ = (shortLength == 4) ? VERTICAL/2-2 : VERTICAL /2-3;
46                  if(z0 == 1 + differ || z0 == 2 + differ)
47                    adjust = this.highControl(z0, differ);
48                }
49              }
50            }
51            this.centralPart(board, h, adjust);
52          }
53          if(VERTICAL == 4 && BESIDE== 4) {
54            for(int a=0;a<BESIDE / 2-1;a++)
55              System.out.print("    ");
56            System.out.print("＼-／");
57          }
58          else
59            this.showDown((VERTICAL <= BESIDE) ? (VERTICAL == 4 ? 2 : 3)
60                                               :(BESIDE == 4 ? 2 : 3));
61          System.out.println();
62        }
63
64        /**
65        * ニップの盤面 [上部] の表示（４＊４の盤面を除く）.
66        * @param up：初期空間数
67        */
68        private void showUp(int up) {
69          for(int u=0;u<up;u++)
70            System.out.print("   ");
71          System.out.print("／-");
```

```
72          for(int u=0;u<BESIDE-up*2;u++)
73            System.out.print("ー-");
74          System.out.print("＼");
75        }
76
77        /**
78        * ニップの盤面 [下部] の表示（４＊４の盤面を除く）.
79        * @param down：初期空間数
80        */
81        private void showDown(int down) {
82          for(int d=0;d<down;d++)
83            System.out.print("   ");
84          System.out.print("＼-");
85          for(int d=0;d<BESIDE-down*2;d++)
86            System.out.print("ー-");
87          System.out.print("／");
88        }
89
90        /**
91        * 角の高さを返す.
92        * @param z0：角の高さ
93        * @param differ：変化性
94        * @return int：角の高さ
95        */
96        private int highControl(int space,int differ) {
97          for(int h=0;h<space-differ;h++)
98            System.out.print("   ");
99          return (space == 1 + differ) ? 1 : 2;
100       }
101
102       /**
103       * ニップの盤面 [中心部] の表示.
104       * @param board：盤面
105       * @param high：縦の辺の位置.
106       * @param adjust：角の高さ
107       */
108       private void centralPart(int[] board,int high,int adjust) {
109         if(VERTICAL == 4 && BESIDE == 4) {
110           if(high==2 || high==20) {
111             System.out.print(high==2 ? "／" : "＼");
112             for(int p=high+1 ;p<high+BESIDE-1;p++)
113               System.out.print("|" + square(board , p));
114             System.out.print(high==2 ? "|＼" : "|／");
```

```
115
116             for(int p=high ;p<high+BESIDE-1+interval;p++) {
117               if(p<high+1+interval)
118                 System.out.print("   ");
119               else {
120               if(p==high+1+interval)
121                 System.out.print(high==2 ? "／" : "＼");
122               this.position(p-interval);
123             }
124           }
125           System.out.print(high==2 ? "|＼" : "|／");
126         }
127         else {
128           for(int p=high+adjust ;p<high+BESIDE-adjust;p++)
129             System.out.print("|" + square(board , p));
130           System.out.print("|");
131
132           for(int p=high+adjust ;p<high+BESIDE-adjust+interval;p++) {
133             if(p<high+adjust+interval)
134               System.out.print("   ");
135             else
136               this.position(p-interval);
137           }
138           System.out.print("|");
139         }
140       }
141       else {
142         if(ALL_SIZE / 2 > high) {
143           System.out.print(adjust != 0 ? "／" : "  ");
144           for(int p=high+adjust ;p<high+BESIDE-adjust;p++)
145             System.out.print("|" + square(board , p));
146           System.out.print(adjust != 0 ? "|＼" : "|");
147
148           for(int p=high-adjust ;p<high+BESIDE-adjust+interval;p++) {
149             if(p<high+adjust+interval)
150               System.out.print("  ");
151             else {
152               if(p==high+adjust+interval)
153                 System.out.print(adjust != 0 ? (adjust==1 ? " ／":"    ／") : "  ");
154               this.position(p-interval);
155             }
156           }
157           System.out.print(adjust != 0 ? "|＼" : "|");
```

```
158         }
159         else {
160           System.out.print(adjust != 0 ? "＼" : "  ");
161           for(int p=high+adjust ;p<high+BESIDE-adjust;p++)
162             System.out.print("|" + square(board , p));
163           System.out.print(adjust != 0 ? "|／" : "|");
164
165           for(int p=high-adjust ;p<high+BESIDE-adjust+interval;p++) {
166             if(p<high+adjust+interval)
167               System.out.print("  ");
168             else {
169               if(p==high+adjust+interval)
170                 System.out.print(adjust != 0 ? (adjust==1 ? " ＼":"    ＼") : "  ");
171               this.position(p-interval);
172             }
173           }
174           System.out.print(adjust != 0 ? "|／" : "|");
175         }
176       }
177       System.out.println();
178     }
179
180     /**
181     * 盤面の升目の文字列表現を返す.
182     * @param i：盤面の番号
183     * @return String：升目の文字列
184     */
185     private String square(int[] board,int position) {
186       switch(board[position]) {
187         case WHITE:
188           return "白";
189         case BLACK:
190           return "黒";
191         case EMPTY:
192           return "  ";
193         default :
194           return "？";
195       }
196     }
197
198     /**
199     * 盤面のマス目の位置を表現し，桁数に合わせて空白を足して表示.
200     * @param posi：マス目の位置
```

```
201     */
202     private void position(int posi) {
203       System.out.print("|");
204       if(digit == 3) {
205         if(String.valueOf(posi).length() == 2)
206           System.out.print(" ");
207         else if(String.valueOf(posi).length() == 1)
208           System.out.print("  ");
209       }
210       if(digit == 2) {
211         if(String.valueOf(posi).length() == 1)
212           System.out.print(" ");
213       }
214       System.out.print(posi);
215     }
216   }
217
```

Test クラス

```
1   package experiment.last.suspend;
2
3   import java.util.Scanner;
4
5   /**
6    * Main メソッド
7    * @author yamamoto
8    */
9   public class Test {
10    private static Board board;
11    private static ShowBoard show;
12    private static boolean beforePassW = false;
13    private static boolean beforePassB = false;
14    /* コンピュータのレベル
15    -1 : 人間が打つ
16    0 : 完全解析用
17    */
18    private static int[] comLevel = {0,0};
19
20    public static void main(String args[]) {
21      setComLevel();
22      board = new Game();
23      show = new ShowBoard();
24      show.show(board.board);
```

```
25
26      int i=0;
27      System.out.println("深さ：" + ConstantAndStatic.depth);
28      if(ConstantAndStatic.TECHNIQUE==2) {
29        if(ConstantAndStatic.SWITCHING == 0)
30          System.out.println("切替:完全読み切り（深さ:" +ConstantAndStatic.depth1 + ")
31                              →完全読み切り（深さ:" +(ConstantAndStatic.depth2-1) + ")");
32        else if(ConstantAndStatic.SWITCHING == 1)
33          System.out.println("切替:完全読み切り（深さ:" +ConstantAndStatic.depth1 + ")
34                              →必勝読み切り（深さ:" +(ConstantAndStatic.depth2-1) + ")");
35        else if(ConstantAndStatic.SWITCHING == 2)
36          System.out.println("切替:必勝読み切り（深さ:" +ConstantAndStatic.depth1 + ")
37                              →完全読み切り（深さ:" +(ConstantAndStatic.depth2-1) + ")");
38        else if(ConstantAndStatic.SWITCHING == 3)
39          System.out.println("切替:必勝読み切り（深さ:" +ConstantAndStatic.depth1 + ")
40                              →必勝読み切り（深さ:" +(ConstantAndStatic.depth2-1) + ")");
41      }
42      while(i < 1) {
43        if(board.getTurn()[1] == 1)
44          System.out.println("ーーーーーターン" + (i+1) + ":W" +  ":ーーーーー");
45        else // -1
46          System.out.println("ーーーーーターン" + (i+1) + ":B" +  ":ーーーーー");
47        if(comLevel[(board.getTurn()[1] == 1) ? 0 : 1] == -1)
48          board.player();
49        else
50          board.com();
51        if(ConstantAndStatic.TECHNIQUE <= 1)
52          ConstantAndStatic.depth--;
53        else {
54          if(ConstantAndStatic.depth1 >= 1)
55            ConstantAndStatic.depth1--;
56          else
57            ConstantAndStatic.depth2--;
58        }
59
60        show.show(board.board);
61        if(!(board.getPass()[3] == true || board.getPass()[2] == true))
62          i++;
63
64        if((beforePassW == true && board.getPass()[3] == true)
65           || (beforePassB == true && board.getPass()[2] == true)
66           || board.getEmptySquares() == 0)
67           break;
```

```
68          else {
69            int settingTurn[] = {board.getTurn()[0],board.nextTurn(board.getTurn()[1])};
70            board.setTurn(settingTurn);
71            beforePassW = board.getPass()[2];
72            beforePassB = board.getPass()[3];
73            boolean settingAfterPass[] = {false,false};
74            board.setAfterPass(settingAfterPass);
75          }
76        }
77        System.out.println();
78        System.out.println("<終了>");
79        int stone[] = new int[2];
80        stone = board.calculateStone(board.board);
81        if(stone[0] > stone[1])
82          System.out.println("結果：白の勝ち");
83        else if(stone[0] < stone[1])
84          System.out.println("結果：黒の勝ち");
85        else
86          System.out.println("結果：引き分け");
87      }
88
89      /**
90      * COM のレベルを決定する
91      */
92      private static void setComLevel() {
93        Scanner keyBoardScanner = new Scanner(System.in);
94
95        System.out.print ("白石は COM が持ちますか？ (Y/N) ");
96        String inputString = keyBoardScanner.next();
97        if (inputString.equals ("Y") || inputString.equals ("y")) {
98          while (true) { // 適切な値が選択されるまでループ
99            System.out.print ("COM のレベルは？ (0〜9) ");
100           inputString = keyBoardScanner.next();
101           try {
102             comLevel[0] = Integer.parseInt (inputString);
103           }
104           catch (NumberFormatException e) { // 整数値以外が入力された場合
105             System.out.println ("0〜9 を入力してください");
106             continue;
107           }
108           if (comLevel[0] <0 || 9<comLevel[0]) {
109             System.out.println ("0〜9 を入力してください");
110             continue;
```

```
111           }
112           break;
113         }
114       }
115       System.out.print ("黒石は COM が持ちますか？ (Y/N) ");
116       inputString = keyBoardScanner.next();
117       if (inputString.equals ("Y") || inputString.equals ("y")) {
118         while (true) { // 適切な値が選択されるまでループ
119           System.out.print ("COM のレベルは？ (0〜9) ");
120           inputString = keyBoardScanner.next();
121           try {
122             comLevel[1] = Integer.parseInt (inputString);
123           }
124           catch (NumberFormatException e) { // 整数値以外が入力された場合
125             System.out.println ("0〜9 を入力してください");
126             continue;
127           }
128           if (comLevel[1] <0 || 9<comLevel[1]) {
129             System.out.println ("0〜9 を入力してください");
130             continue;
131           }
132           break;
133         }
134       }
135     }
136   }
137
```