卒業研究報告書

題目

リバーシの並列化

指導教員

石水　隆　講師

報告者

09-1-037-0164

前田　昂寛

近畿大学理工学部情報学科

平成 24 年 1 月 31 日提出

概要

リバーシは二人零和有限確定完全情報ゲームである。そしてリバーシプログラムは一般に数手先を先読みし、その中で最も評価関数が高い手を採用する。しかし、先読み手数増加に伴い、可能な盤面の組み合わせは指数的に増大するため、先読み数を 10 手程度に制限しても探索には膨大な時間がかかる。

そこで、本研究では、先読み数にかかる時間を短縮するためにマルチスレッド処理を用いる Java プログラムを作成する。マルチスレッド処理は先読みにおける探索作業をサブスレッドで処理することにより並列計算が可能であり、一定時間内に行える先読み数を増やす事ができる。探索作業のみをマルチスレッド化することで先読み数を増やし、並列計算にする有効性について考える。

先読みによる探索は $\alpha - \beta$ 法を用いて行い、対戦相手をランダムに動く AI として、シングルスレッドのプログラム、マルチスレッドのプログラムの先読みにかかる時間と勝率の変化を比較した。この結果、シングルスレッドを用いた場合に比べて探索における先読み数は大幅に拡大でき、実行時間の短縮に成功した。

目次

1　序論

1.1 本研究の背景

リバーシは二人零和有限確定完全情報ゲームである。リバーシプログラムを組む場合、この様な特性から完全読みが望ましいが、指数的に増大する探索数が問題となり困難である。そこで一般的に評価関数による評価が行われる。評価基準は様々で、盤面上の１マスずつに重みをつけた盤面評価や、ゲーム中確実に自分の手として確保できる確定石、自分や相手の候補手を増やしたり減らしたりすることを目的とした着手可能手数や開放度、C 打ち判定や X 打ち判定等挙げきれないほどの評価基準がある。しかし、評価関数単体では現在の局面のみを評価するため手の精度が低い。というのも、今は局面的にその手が有利でも 2 手 3 手先ではウィング等作ってはいけない形に追い込まれる可能性があるからである。このため、先読みを行う必要がある。先読みでは数手先までを読み、数手先での局面を考慮して評価関数による評価を行う。この先読み数は大きければ大きい程評価関数による手の選択の精度が上がるとされているが、完全読み同様探索数が指数的に増大するため、動作に不具合がでる等の理由から一般的に 10 手程で限界である。

1.2 マルチスレッド

通常のプログラムは1行ずつコードを実行していく。仮に for 文や if 文、オブジェクト指向等あってもこれは変わらない。こういったプログラムの一連の流れのことをスレッドという。

また、このように 1 行ずつだけを実行していくスレッドの事をシングルスレッドのプログラムという。一般的にはシングルスレッドのプログラムが主流である。

しかし、リバーシの様なボードゲームの場合は探索等に負荷が大きいことからシングルスレッドの動作では長期稼動、もしくは動作停止等先読みの機能を充分に生かせないことがある。こういったときに役立つのがマルチスレッドである。通常のスレッドはあくまで逐次的に 1 行ずつ実行されていくが、マルチスレッドではマルチスレッド化されたプログラムについて並行して処理が行われている。つまり、シングルスレッドが 1 行ずつ処理している間に、マルチスレッドでは複数のスレッドが 1 行ずつ実行していくため、より高速で高機能なプログラムを組めるといった利点がある。

ただ注意点として、マルチスレッド処理ではマシンスペックへの依存が高い。マルチスレッドではサブスレッドの起動、サブスレッドの処理開始、サブスレッドの同期等シングルスレッドに比べて処理数が多くなるからである。例えば、コア数の少ない低スペックのコンピュータにマルチスレッド処理をさせると余計に時間がかかる可能性すらある。また、複数のコンピュータを用いた MPI による並列処理とは違い、故障を補完するような機能は一切ないため、扱いには慎重さが求められる。

1.3 リバーシの盤面

リバーシは8x8のマスを使用する。各マスには、横にa～b, 縦に1～8の座標が付いている。図 1 にリバーシの盤面を示す。

リバーシにおいて角を取れば有利に進められることが多くなるのは一般的に知られている。このことから、角と隣接するマスは危険なマスと考えられている。角のマスと辺で隣接する a2, a7, b1, b8, g1, g8, h2, h7 の 8 個のマスを C マスと言う。また角のマスと頂点で接する b2, b7, g2, g7 の 4 個のマスを X マスと言う。

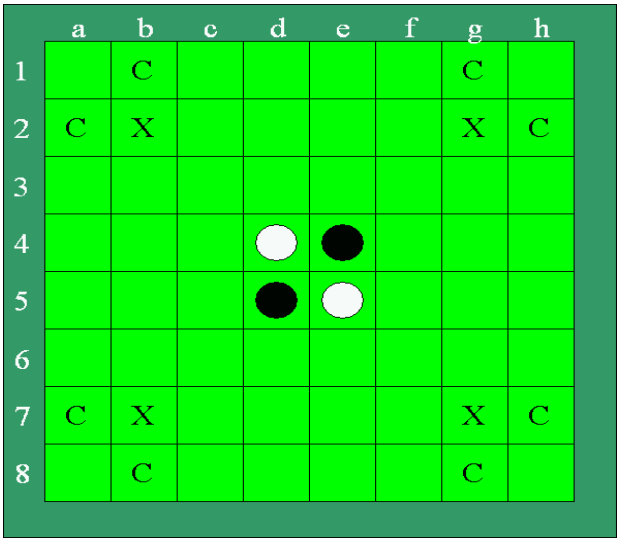

図 1: リバーシの盤面

1.4 リバーシに関する既知の結果

本節ではリバーシに関する既知の結果を示す。

1.4.1 リバーシの完全解析

リバーシは 60 手以内に必ず勝負が付く。よって、初期状態から 60 手先までを先読みし、起こり得る盤面を全て解析することができれば、常に最善の手を打つことができる。しかし、先読みする手数が増えると、起こり得る局面の数は指数的に増大する。リバーシのマスは 64 マスあり、中央の 4 マスは白黒の 2 通り、それ以外のマスは白黒空の 3 通りの可能性があることから、リバーシの起こり得る局面の総数は、大ざっぱに見積もって 2^4*3^{60} 通り=$6.78*10^{29}$ 通りである。このため、現在の計算機の解析能力ではリバーシの完全解析を行うことはまだ不可能である。

しかし、盤面のサイズを小さくした場合は完全解析が可能である。サイズ 6x6 の場合、局面の総数は2^4*3^{32}通り=$2.96*10^{16}$通りである。この 6x6 のリバーシに対して、J.Feinstein は完全解析を行い、双方最善手を打つと 16 対 20 で後手勝ちとなることを示した[6]。

1.4.2 リバーシの定石

前節で述べた通り、リバーシの完全解析は現時点ではまだであり、特に序盤においてはどのような手が最善となるかを決定することはできない。しかし、序盤においてどのような手が有利になり易いかは定石として確立している。代表的な序盤の定石には、縦取り兎定石、縦取り虎定石、斜め取り牛定石、並び取り鼠定石がある。兎定石では初手が黒として、1 手目から黒 f3、白 d6、黒 c5、白 f4、黒 e3 というパターンである。虎定石では黒 f5、白 d6、黒 c3、白 d3、黒 c4 というパターンである。牛定石では黒 f5、白 f6、黒 e6、白 d6、黒 c5 というパターンである。鼠定石では黒 f5、白 f4、黒 e3、隘路 f6、黒 d3 というパターンである。図 2, 3, 4, 5 に兎定石、虎定石、牛定石、鼠定石の 5 手後の盤面を示す。

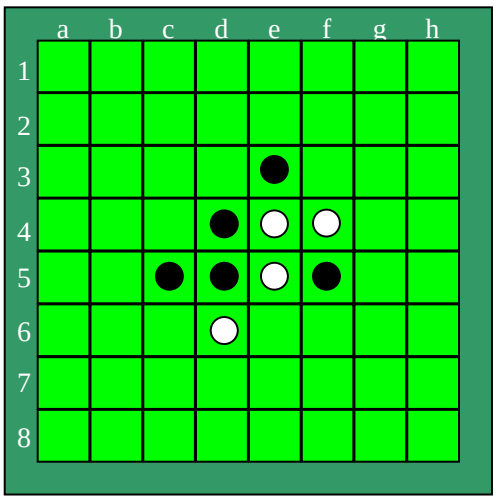

図 2 兎定石後の盤面(5 手目)

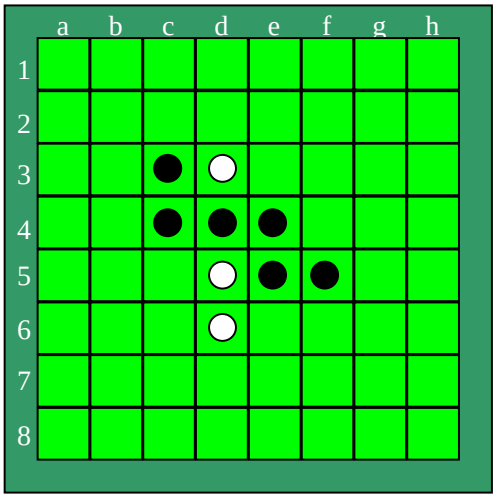

図 3 虎定石後の盤面(5 手目)

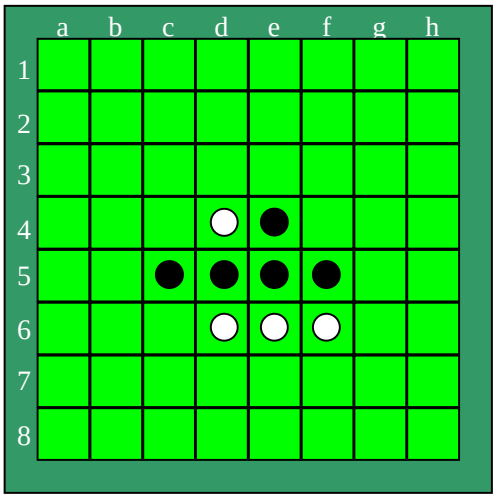

図 4 牛定石後の盤面(5 手目)

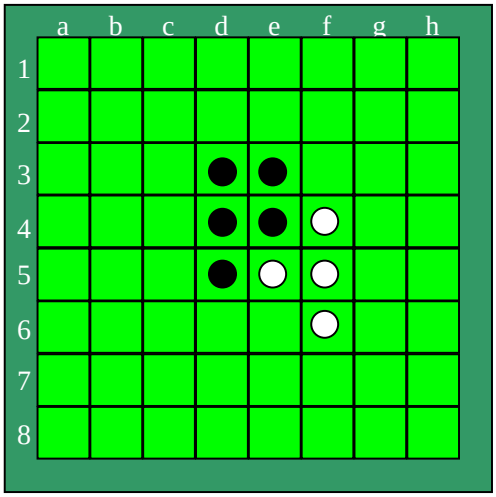

図 5 鼠定石後の盤面(5 手目)

局面が様々に変化する中盤においても、定石がいくつか確立されている。代表的な中盤の定石には、中割、引っ張り、一石返し等がある。以下これらについて説明する。

中割とはリバーシの場合、中盤で相手に囲まれる手が有利である[3]という観点から使う技である。これは自分が裏返した石の周りのマスの空きを見るものである。当然空きマスが 0 ならば確定石（章 2.6 参照）となり、良い状態である。

引っ張りとは相手が壁を作った時に使う技である。壁とは自色の石が邪魔で裏返すことができなくった盤面上の部分を指す。相手が壁を作った場合、自分の打つ手は相手の壁を壊さないようにして打っていき、相手の打つ手を制限していく。

一石返しとは一石のみ裏返すことをいう。自分が不利な状況を作ってしまった場合や相手の手が全く読めない場合、無難な場所の一石のみを返して相手の様子を見るという方法である。

1.4.3 局面の評価

ゲーム中盤における局面の状況は様々であり、定石が存在しない局面も存在する。また、序盤であっても相手が定石以外の手を打った場合、同様に定石が存在しない局面となる。そのような場合に良い手を発見するために用いられるのが、その局面を評価することである。ある局面で

盤上に置かれた石の並び方を入力とする評価関数を設定し、その関数の値によって先手後手のどちらがどの程度有利なのかを判定する。どのような評価関数を用いるのが良いかは未解決の問題であり、様々な評価関数が提案されているが、一例としてパターンに基づく評価を挙げる。

パターンに基づく局面評価では現在の局面のあるパターンを抽出し、着手可能数や確定石など必要な要素で評価する。その際、空きマスが偶数か奇数かなどまで見る。

1.4.4 先読み

現在の局面から数手先の局面を先読みし、それを下に評価値を決定することで評価関数の精度を上げられる。一般に先読み手数を増やすにつれ評価関数の精度は上がり、最終局面まで読むことができれば、勝敗を完全に決定できる完全な評価関数になる。しかしながら 1.4.1 節で述べたように、先読み手数が増えるにつれ可能な局面数は指数的に増えるため、序盤・中盤では完全先読みは不可能である。このため、序盤・中盤では一定手数先まで読み、得られた局面の各評価値を元に評価値を求める手法が一般によく使われる。

一方、終盤では残りの手が少なくなるので、その盤面から最終局面まで全て先読みすることが可能となる。現在の計算機性能では、残り手数が 25～30 手程度でも最終局面までの完全先読みが可能となっている。また、計算機の性能から高速計算を目的とした必勝読みと完全読みをわけて使う方法がある。完全読みは最後の局面で一番多く自分の色を増やそうとするのに対し、必勝読みは状態勝ち、負け、引き分けのみの判断となる。このことから、比較演算子による計算は 1,0,-1 のみで完全読みに比べて計算量が少ないというメリットがある。

1.4.5 既知のリバーシプログラム

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

手法として定石データベースのみを用いたプログラムとして、Logistello[5]がある。Logistello は 1997 年には村上健八段(当時の世界チャンピオン)に勝利しており[5]、定石データベースを使うことでそれなりの強さのプログラムとなることが示された。しかし、Logistello は定石データベースのみを使用するため、定石以外の手を打たれた場合には対処できない。また、その他に Zebra[10]や Edax[8]などもある。これらについても book と呼ばれる序盤・中盤の定石データや対戦データベースを使用しているが、中盤の先読みと終盤の完全読みを行っている。特に Edax はマルチコア対応で高速処理が可能となっているが、bool 非使用の時は勝率が下がるというのが一般的な見方である[13]。

1.5 本研究の目的

先読み手数を増やすと評価関数の精度があがり、強いリバーシプログラムとなる。しかし、先読み手数を増やすと関数値の解析に必要な時間は指数的に増大する。そこで本研究ではリバーシプログラムをマルチスレッド化することによって高速化をはかり、先読み数を大幅に増やすことを目的としている。一般的なリバーシプログラムでは先読み数が 10 手に満たない事もある為、評価関数による打つ手の精度が低い。そこで本研究では、先読み数を増やすことにより勝率が上がるのかもあわせて検証する。

前節で述べた様に、Logistello は高評価を受けているオセロプログラムである。しかし、Logistello では統計による定石データベースを使用しているため、強いオセロプログラムとはなったものの、その特性から解析には至らない。リバーシの解析には探索を高速処理することが最低条件で、これができなければ先読み数を深くすることが困難である。よって高速化を図ることは重要な位置づけを持つ。また、本研究では MPI において並列化するという手段もあったが、通信速度も考慮にいれなければならないためマルチスレッドが最適であると判断した。本研究では、

マルチスレッドによる高速化を主目的とするため、リバーシプログラムで用いる手法は先読みと評価関数のみに絞り、定石データベース等解析の妨げとなるものは使用しないものとする。

1.6 本論文の構成

本論文の構成は以下の通りである。まず第 2 章でリバーシプログラムの一般的手法について説明する。続く第 3 章で本研究で作成したリバーシプログラムについて説明する。4 章は結果を、5 章では結論と今後の課題を示す。

2 リバーシプログラムの一般的手法

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

現在、強いと評価されているプログラム[8][9][10]は、序盤は定石データベースに従って打ち、中盤、あるいは相手が定石から外れた手を打ったときの序盤では、一定手数先読みし、先読み後の局面に対して評価関数を用いて評価値を求め、その値から打つ手を決定する。そして終盤、残り手数がある一定数以下になると完全読みを行い、最善手を打つ。

2.1 定石データベース・対戦データベース

定石データベースとは、リバーシの定石をデータベース化し、各局面で有効な定石があればそれに従って打つという手法である。前章で述べた Logistello[5]のように、定石データベースを使用することで強いリバーシプログラムとなる。しかし、相手があえて定石以外の手を打つなどして、データベースに無い局面が出てきたときにはこの手法は使えない。

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

2.2 モンテカルロ法

オセロプログラムではあまり使われないが、モンテカルロ法[14]の利用も考えられる。モンテカルロ法とは、各着手可能手に対し、その手から先終局までをランダムに打ち勝敗判定を行うという作業を数千～数万回繰り返し、最も勝率の高い着手可能手を採用するというものである。この手法は局面数が極めて多い囲碁プログラムでは最近主流になっている[11]。

2.3 先読みと評価関数

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

これは盤面の重みだけで見るため全体の局面を見れないといった問題がある。

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

現局面評価の評価関数の計算に用いられる評価基準について、先に記述した盤面評価、直線性、確定石、直線性、駒数、候補数等がある[4]。本研究では評価関数の計算にどのような評価基準を用いるかについては本研究の主たる目的ではないため、なるべく計算量が大きそうな評価関数を選んだまた、本研究で作成したリバーシプログラムは、序盤・中盤では一定手数先までの先読みを行い、終盤では終局までの必勝読みと完全読みを行うように設計した。また、先読みを行う際は、どの候補から読むのか、読む順番も実行時間に影響し、予想評価値の高きものから読むことにより、実行時間が減らせる場合がある。以下、本研究で採用した評価関数について記述する。

2.4 本研究で使用した評価基準

本研究で作成したリバーシプログラムは定石データベースを使用せず、序盤と中盤で同一の評価関数を用いる。以下、本研究で用いた、盤面の評価値を決めるための評価基準について述べる。本研究では評価基準として、ウィング、C 打ち、X 打ち、山、確定石、開放度、着手可能手数を用いている。

2.4.1 C 打ち、X 打ち

C 打ち、X 打ちとは 1.4.1 章の図３におけるマス C とマス X に打つことである。一般に隣接する角が取られていない状態での C 打ち、X 打ちは危険とされている。

2.4.2 ウィング

ウィングとは一般的に端の列を左か右を 2 マス空けてもう片方を 1 マス空けた状態である。図 6 にウィングの例を示す。

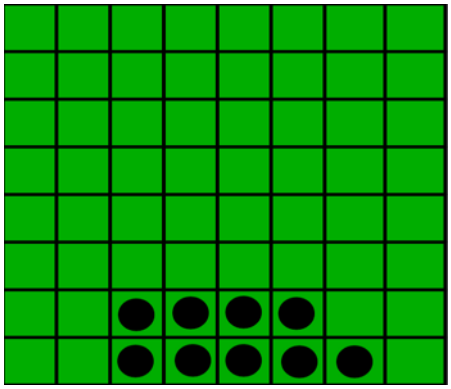

図 6：ウィング

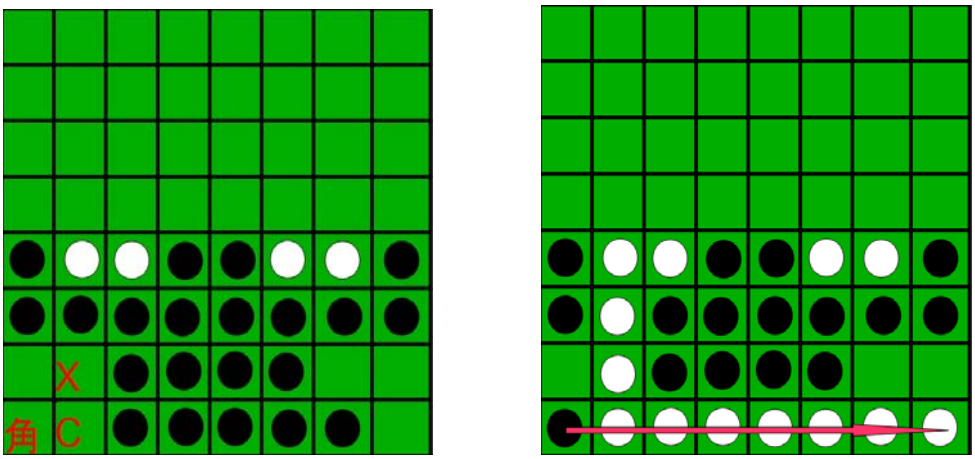

図 7：ウィングの不利な例

図 7 はウィングの例である。例えば、図 3 の状態で白番であり、他に打つ場所がなくなったという状況だったとする。b7 に白が打つと、黒は角 a8 が取れるが、次に白が C 打ち b8 を行うと入り込んだ白 b8 を黒は裏返すことができない。更に、白は次の番で反対側の角 h8 と端の一列 c8～g8 を全て取ることができる。このようにウィングを作ると、不利な状況になりかねない。仮に白 b7 の後黒が角 a8 を取らなかったとしても、C 打ち a7 または b8 を行うと b7 の石がほぼ裏返り、相手が角 a8 を取ることになるので黒は番の左下部分への着手の先延ばしをしても意味がない。

これらのことからウィングを作らないように打たなければならない。

2.4.3 山

山とは、ある辺の C マスから反対側の C マスまでの 6 マス、例えば b8～g8 の 6 マスが一色で、その上の列が自分の一色または混色で並べた状態のことを指す。図 8 に例を示す。一般的にこの形状は良い形状であると言われている。

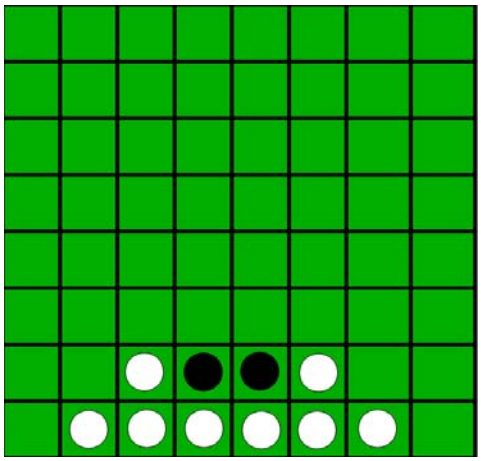
図 8：山の例

2.4.4 確定石

確定石とは、ゲーム終了まで二度と裏返らない石のことをいう。確定石は有利な局面を作るためには非常に重要で、角を取れば有利に対戦を進められるという一般的見解はこの確定石であることが大きな意味を持つ。また、章 1.4.2 で記述したような中割などで周囲の空きマスが 0 の場合には確定石が生まれやすい。周囲に空きマスがない状態で、連接する石が裏返らない時になる。

2.4.5 開放度

開放度とは現在の候補手の中からひとつを選んで打ったと仮定した時、その手によって裏返った石の周りの空きマスのことをいう。着手可能な候補手から開放度の最も小さいものを選ぶと有利になる。これは、相手に囲ませた方が自分の打てる箇所が多くなるからである[3]。

2.4.6 着手可能手数

着手可能手数とは打つ手が可能なマスの数の事である。選択肢が多ければ多い程強く、少なければ少ない程弱い。これは戦略を取りたい時に選択肢がなければそもそも戦略が取れないからである[3]。

2.5 終盤の評価関数

終盤の評価関数では必勝読みを行う評価関数と完全読みを行う評価関数の二つを使う。必勝読みとは先読みによって最終局面の各色の石の数を計算し、勝ち負け引き分けの結果のみを読む評価関数である。完全読みは最終局面で黒石-白石の値を返す関数である。完全読みは探索数が大きく必勝読みの方が探索が速いため、残り手数が一定値以下になるとまず必勝読みを行い、その後さらに残り手数がより小さい一定値以下になると完全読みを行う。

2.6 先読み局面評価

本研究で作成したリバーシプログラムは、序盤・中盤では一定手数先の局面を先読みし、その局面の現局面評価から評価値を求める。ある局面の評価値は、その局面の着手可能手を打った場合に得られる次の手番の局面の評価値から求められる。自分の手番を先読みしている場合は次の局面の評価値が最も評価値が高い手を選択し、相手の手番を先読みしている場合は最も評価値が低い手を選択する。次の手番の評価値は、さらに次の次の手番の評価値から計算される。この操作を一定手数先の局面まで再帰的に繰り返す。また、探索時間を減らすため、まず予備探索を行い各着手可能手の評価値を予想し、予想評価値の高いものから探索していくことにより、探索木の枝狩りを図る $\alpha-\beta$ 法[1]を用いる。

3 研究内容

本研究では、作成したシングルスレッドのリバーシプログラム（以下 SSAI とする）とマルチスレッドのリバーシプログラム（以下 MSAI とする）をランダムに動くリバーシプログラム（以下 RAI とする）と対戦させてその性能を調査した。付録１に SSAI,付録２に MSAI のコードを示す。

3.1 SSAI,MSAI

ここでは SSAI と MSAI の違いについて簡単に述べる。SSAI も MSAI も $\alpha-\beta$ 法によって再起的に探索するという点では全く同じで、評価関数による評価基準も全く同じである。これは性能を比較するために可能な限り同じ条件に揃える必要があったからである。この 2 つの違いは、探索を並列化しているか逐次的に処理しているかである。SSAI は探索開始から終了まで再帰的に逐次処理であるが、MSAI では、まず候補手を全てサブスレッドに処理を委託する。そうしてできたサブスレッドも再帰的にサブスレッドを作成して処理を委託している。図 5 に概要を示す。

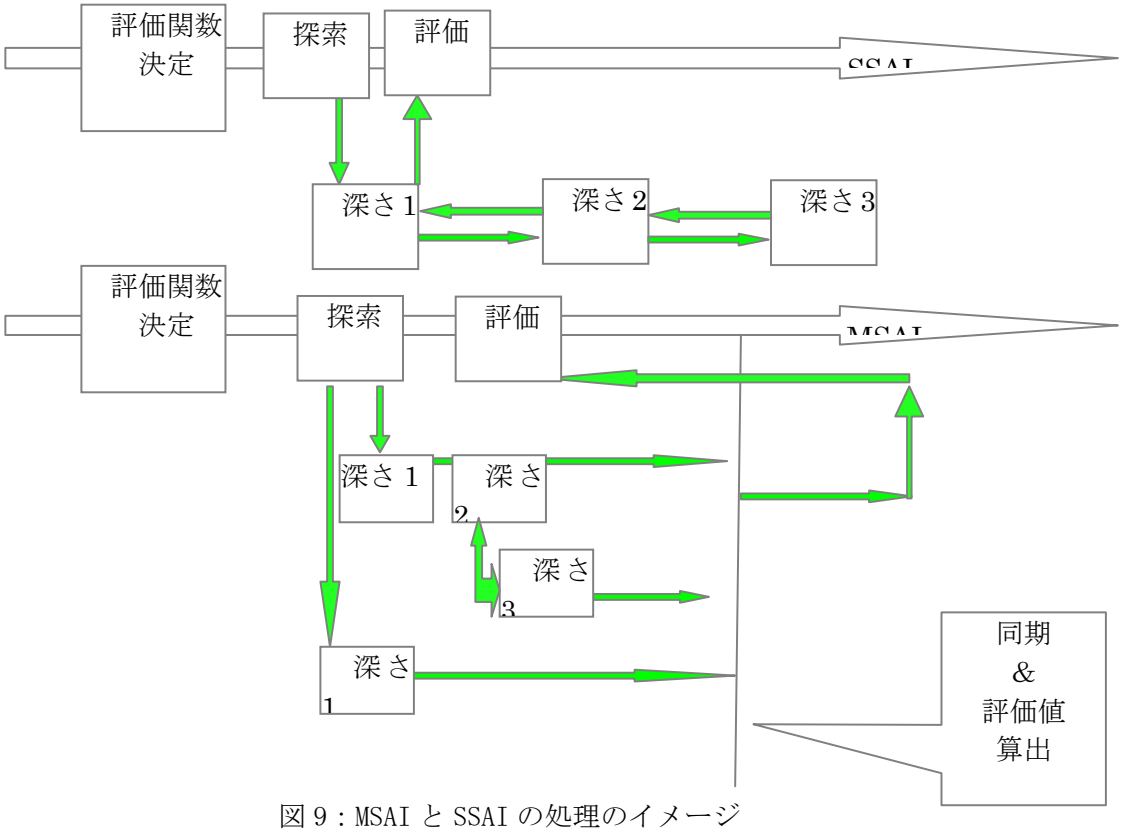

図 9：MSAI と SSAI の処理のイメージ

3.2 各評価関数の処理

本節では、各関数の処理について簡単に示す。なお、評価関数によって算出された評価値が競合した場合でも、先に選んだ評価値の方を採用する。

3.2.1 開放度

開放度を求めるには、ある位置について、その周囲の空きマスを数えればいいが、これを毎回評価関数の実行のたびに全てのマスについて行うと、かなりの計算量になる。よって、石を打った時にその周囲 8 マスの開放度を 1 ずつ減らせばよい。自分の石の開放度については負の評価を与え、相手には正の評価を与える。

3.2.2 確定石

確定石の個数を求めるには、4 隅とそれに連接する石の数を数えればよい。連続していなくても確定石となる場合はあるが、正確に求めるには計算コストが大きすぎる為、この手段を取った。自分の確定石には正の評価を、相手には負の評価を与える。

3.2.3 着手可能数

着手可能手数を求めるには縦・横・斜めのぞれぞれの横行の石の並びのパターンについて石を打てる箇所数を予め数えてテーブル化しておき、ぞれぞれの方向について打てる箇所を合計すればよい。

3.2.4 辺

本研究で評価する山とウィングのような辺のブロックを調べる場合、角が両方共空きであることや中央 6 マスに同じ色が連続して並んでいることを調べなければならない。しかし、これでは比較演算子が膨大な量になるため、計算時間が多くなる。そこで、各マスの状態を白、黒、空の 3 通りあるので、ある辺における石の並び方のパターンは、3 進法で 8 桁の値と 1 対 1 に対応できる。辺の評価値を予め全てのパターンについて計算した 3^8 = 6561 行のテーブルを作っておき、ゲーム中に辺の評価値が必要になったときはその手 0 ブルから評価値を取り出せば高速に計算が可能である。

3.3 SSAI と MSAI における先読みの定義

本節では、局面毎にわけた先読みを定義していく。α－β 法により事前に手を調べて探索順序を決める為の予備探索先読み数を presearch_depth、序盤・中盤の探索における先読み数を normal_depth、終盤の必勝読みを始める残り手数を wld_depth、終盤の完全読みを始める残り手数を perfect_depth とする。また、presearch_depth はあくまで予備探索であるため、後述するクラス MultiAI ほど並列化していない。よって先読み数は標準設定で 5 にしておく。

本研究では以下の動作環境を使用した。

- Let'sNote：プロセッサ,Intel(R)Core2 Duo CPU L7800 2.0GHZ,メモリ (RAM),2.00GB,WindowsVista
- VT64：プロセッサ, (AMD(R)Opteron,1.9GHZ 12 コア) *4 → 48 コア,メモリ,CentOS

3.4 SSAI,MSAI のプログラム

本節では、SSAI プログラムおよび MSAI プログラムについて述べる。付録に SSAI および MSAI のソースプログラムを示す。

SSAI は AI, AlphaBetaAI の 2 つのクラスから成る。また、MSAI は AI, AlphaBetaAI, MultiAI の 3 つのクラスから成る。以下各クラスについて説明する。

3.4.1 クラス AI

クラス AI は章 3.3 で定義した先読み数を扱っている抽象クラスである。また、先読み数を試行回数に合わせて増加させていくため、実行クラス ReversiGame から現在の試行回数を得る。抽象メソッド move を扱う。

3.4.2 クラス AlphaBetaAI

クラス AlphaBetaAI では、最初に BookManager クラスから現在の盤面の候補手を得る。実行の高速化を図るため、打てる箇所が一箇所しかなかった場合や打てる箇所が一箇所もない場合は先に処理しておく。ここで、現局面から使用するであろう評価関数をインタフェース Evaluator 型の変数 Eval を生成する。デフォルトでは序盤・中盤用の評価関数 MidEvaluator を生成するよう記述している。

ここから SSAI と MSAI で若干差がでてくる。次の処理ではクラス内メソッドの sort を使い、探索候補をよさそうな順にソートするが、SSAI はシングルスレッド処理を、MSAI はマルチスレッド処理を行う。次に、クラス AI の先読み変数を変数 limit に読ませるため、現局面が何手目なのかを比較演算子を用いて判断する。以降は SSAI と MSAI をわけて説明する。

SSAII

for ループ内では、候補数の数だけ再起的に α－β 法で探索する。候補を格納した配列 movables より i 番目の候補を深さ limit-1 まで同クラス内のメソッド alphabeta にて探索する。

alphaBeta メソッドは深さ制限に達した時か最終局面になった時、評価値を返す変数 Eval のメソッド evaluator により評価値を得る。また、打つ場所がなかった時はその局面をパスして通常どおり再帰的に先読みを行う。また、β 刈りを行うことで処理の高速化を意識している。この評価値は評価値を格納する為の int 型変数 eval に格納する。次のコードは変数 board の undo メソッドは局面を動かさなかったことにする為のメソッドである。最後に比較演算子により評価値の高い候補手を Point 型の変数 p に格納し、その候補手を打つ。

MSAI

基本的に処理の流れは同じである。ただ、マルチスレッド化のため、for ループの手前で候補数の数だけサブスレッドの配列 thread とクラス MultiAI の配列 multi を生成する。その後、候補手の数だけループする for ループに突入する。

for ループ内では局面を一手動かし、multi[i]にその情報を格納する。multi[i]は thread[i]に格納し、thread[i]を実行する。また、変数 board の undo メソッドを使うことで動かさなかったことにし、辻褄を合わせる。

ループを抜けた後、各スレッドを同期し各スレッド内の multi[i]の評価値を読み込む。その後評価値を比較演算子を用いて最も評価が高い候補手を選ぶ。その候補手は Point 型の変数 p に格納し、その候補手を打つ。

3.4.3 クラス MultiAI

クラス MultiAI は SSAI でいう alphaBeta メソッドの役割をするものである。ただ、Runnable を継承しているため、run メソッドにて値を返したり、値を受け取ったりすることはできない。そこで、MultiAI クラスでは生成時に、メソッド alphaBetaAI に渡されるべき値を受け取って初期化する。また、評価値を返すためにゲッターである同クラス内の getAlpha メソッドとセッターである setAlpha メソッドをを用いる。評価関数の変数 Eval は multiAI が生成された後にセットする。以上のことを行い、run メソッド内では、深さ制限に達した、または最終局面に達した時には評価値を setAlpha メソッドによりセットしておく。そこに達するまでは打つ場所がない場合でもパスで対処して深さ制限まで先読みを行う。この先読みを行い場合も、高速化するため再帰的にサブスレッドにより処理する。その後の処理は SSAI の alphaBeta メソッドと同様である。

4 結果

本章では本研究における結果について述べる。

4.1 Let'sNote の SSAI,MSAI が後攻の時の比較

SSAI,MSAI それぞれに大して、は設定した先読み数で 100 回 RAI と対戦する。その後、presearch_depth 以外の各先読み数を 1 ずつ増やした後に 100 回対戦するという作業を繰り返し行う。また、途中で探索に時間がかかりすぎて実行不可能と判断されることが予測される。実行不可能となった場合はその直前までの設定を比較する。

以下はレッツノートで eclipse を用いて行った時のものである。表 1 は各先読み数の設定、表 2 は SSAI と MSAI のある先読み設定数における 100 回実行した時の勝敗の合計である。また、表 3

はある先読み設定数で 100 回実行した時の合計所要時間である。

MSAI は先読み設定数の増加に伴い、処理時間が延びてはいるものの、先読み設定数が低いうちでは高速な動作が可能となった。対して SSAI では初期設定から 375 秒程度かかり、先読み設定数を各々6 ずつ大きくした時には 958 秒以上かかる結果となった。また全対戦の結果、MSAI は WIN=672、LOSE=293、DRAW=35 となり、SSAI は WIN=352、LOSE=225、DRAW=23 だった。各先読み設定数での勝ち負けについては表 2 の通りである。

表 1:SSAI と MSAI の先読み設定数

	preseach_depth	normal_depth	wld_depth	perfect_depth
SSAI	5	2+X	5+X	3+X
MSAI	5	2+X	5+X	3+X

表 2:先読み設定数＋X における勝敗

Thread	WLD	X= 1	X=2	X=3	X=4	X=5	X=6	X=7	X=8	X=9
SSAI	WIN	67	56	54	54	60	61	-	-	-
	LOSE	29	39	41	44	36	36	-	-	-
	DRAW	4	5	5	2	4	3	-	-	-
MSAI	WIN	68	66	72	70	73	67	61	61	72
	LOSE	28	30	27	25	23	27	38	32	26
	DRAW	4	4	1	5	4	6	1	7	2

表 3：ある先読み設定数で 100 回実行した時の合計所要時間（秒）

Thread	X=1	X=2	X=3	X=4	X=5	X=6	X=7	X=8	X=9
SSAI	375.9	395.56	417.02	437.1	479.41	958.16	-	-	-
MSAI	19.2	19.58	19.45	21.1	24.82	22.59	24.05	25.53	25.14

4.2 Let'sNote の SSAI,MSAI が先攻の時の比較

条件は章 4.1 と同様であるが、SSAI,MSAI が先攻である点が異なる。

結果は後攻の時とほとんど変化は見られなかった。また全対戦結果について、MSAI は WIN=614、LOSE=259、DRAW=27 となり、SSAI は WIN=337、LOSE=243、DRAW=20 となった。

表 4：先読み設定数＋X における勝敗

Thread	WLD	X= 1	X=2	X=3	X=4	X=5	X=6	X=7	X=8	X=9
SSAI	WIN	51	57	57	57	57	54	61	-	-
	LOSE	43	40	40	41	41	43	36	-	-
	DRAW	6	3	3	2	2	3	3	-	-
MSAI	WIN	65	68	68	75	63	60	62	62	73
	LOSE	33	29	31	24	32	38	35	35	24
	DRAW	2	3	1	1	5	2	3	3	3

表 5：ある先読み設定数で 100 回実行した時の合計所要時間(秒)

Thread	X=1	X=2	X=3	X=4	X=5	X=6	X=7	X=8	X=9
SSAI	404.95	413.71	401.5	427	490.08	948.03	-	-	-
MSAI	22.39	22.98	24.03	27.94	26.99	28.37	26.39	29.07	27.44

4.3 VT64 の SSAI,MSAI が後攻の時の比較

条件は 4.1 節と同様であるが、SSAI,MSAI が先攻である点が異なる。また、実験環境は Let'sNote とは違い、ターミナルで行った。

結果は後攻の時とほとんど変化は見られなかった。また、SSAI は実行に多大な時間を要するため、先読み設定数は 6 までしか増やしていない。処理時間的には SSAI が VT64 の恩恵を受けて高速化している。また全対戦結果について、MSAI は WIN=636、LOSE=233、DRAW=31 となり、SSAI は WIN=374、LOSE=205、DRAW=21 となった。

表 6：SSAI と MSAI の先読み設定数

	preseach_depth	normal_depth	wld_depth	perfect_depth
SSAI	5	2+X	5+X	3+X
MSAI	5	2+X	5+X	3+X

表 7：先読み設定数＋X における勝敗

Thread	WLD	X= 1	X=2	X=3	X=4	X=5	X=6	X=7	X=8	X=9
SSAI	WIN	56	60	57	71	60	70	-	-	-
	LOSE	41	35	37	26	38	28	-	-	-
	DRAW	3	5	6	3	2	2	-	-	-
MSAI	WIN	72	74	77	70	68	74	65	61	75
	LOSE	24	23	22	28	29	25	27	34	21
	DRAW	4	3	1	2	3	1	8	5	4

表 8：ある先読み設定数で 100 回実行した時の合計所要時間（秒）

Thread	X=1	X=2	X=3	X=4	X=5	X=6	X=7	X=8	X=9
SSAI	301.93	313.45	311.96	341.49	374.9	791.68	-	-	-
MSAI	29.28	27.73	27.55	27.2	28.37	28.01	28.6	28.66	28.83

4.4 VT64 の SSAI,MSAI が先攻の時の比較

条件は 4.3 節と同様であるが、SSAI,MSAI が先攻である点が異なる。

結果は後攻の時とほとんど変化は見られなかった。また、SSAI は実行に多大な時間を要するため、先読み設定数は 6 までしか増やしていない。また、VT64 を使った結果 SSAI のみは高速化している。また全対戦結果について、MSAI は WIN=648、LOSE=227、DRAW=25 となり、SSAI は WIN=355、LOSE=229、DRAW=16 となった。

表 9：先読み設定数＋X における勝敗

Thread	WLD	X= 1	X=2	X=3	X=4	X=5	X=6	X=7	X=8	X=9
SSAI	WIN	62	61	54	55	57	66	-	-	-
	LOSE	34	38	45	42	38	32	-	-	-
	DRAW	4	1	1	3	5	2	-	-	-
MSAI	WIN	69	72	67	67	75	73	65	60	70
	LOSE	30	23	30	29	24	26	32	37	26
	DRAW	1	5	3	4	1	1	3	3	4

表 10：ある先読み設定数で 100 回実行した時の合計所要時間（秒）

Thread	X=1	X=2	X=3	X=4	X=5	X=6	X=7	X=8	X=9
SSAI	261.08	260.03	271.35	276.28	315.63	615.95	-	-	-
MSAI	21.1	19.63	19.08	19.42	20.05	19.53	19.21	19.42	19.65

4.5 VT64 と Let'sNote 上での比較

本実験では前節までの実験結果から Let'sNote と VT64 の MSAI における処理時間に変化が求められなかった。これについて eclipse とターミナルという環境の差が考えられたので本節では eclipse を使用した Let'sNote_eclipse、ターミナルを使用した Let'sNote_CP、ターミナルを使用した VT64 と定義して実験を行う。前節では後攻の時と先攻の時の処理時間の差がなかったことから後攻だけで実験する。また、本実験では両者とも MSAI を使用するため、勝敗比較はしない。同様の理由から先読み初期設定数を表 11 の様に負荷を大きくする。

表 11：先読み初期設定数

	preseach_depth	normal_depth	wld_depth	perfect_depth
Let'sNote（MSAI)	7	20+X	25+X	23+X
VT64(MSAI)	7	25+X	25+X	23+X

表 12：ある先読み設定数で 100 回実行した時の合計所要時間（秒）

Thread	X=1	X=2	X=3	X=4	X=5	X=6	X=7	X=8	X=9	X=10
Let'sNote_Eclipse (MSAI)	167.04	171.59	177.45	189.21	182.53	175.85	184.98	170.2	188.31	174.96
Let'sNote_CP (MSAI)	334.19	369.91	384.36	418.18	424.9	428.67	374.83	390.03	377.81	356
VT64 (MSAI)	277.35	260.91	282.53	264.81	278.42	273.04	273.93	271.16	276.83	295.07

表 12 から読み取れる様に、コンソール出力である Let'sNote_CP は eclipse を使わない事により処理時間が伸びている。このことから、前節までの実験においても VT64 に eclipse を用いて実験していればより高速化できた可能性がある。

5　結論・今後の課題

本研究ではリバーシゲームにおけるシングルスレッドプログラムとマルチスレッドプログラムの動作時間の比較、対戦結果を検証した。先読み数の増加に伴い、動作時間が大幅に遅延することはあったものの、マルチスレッド化することによりシングルスレッドよりも軽快な動作を実現した。また、実行速度において VT64 と Let'sNote の両者共ターミナルで実行すると、Let'sNote よりも VT64 は高速化できていたため、今後の課題として VT64 を eclipse 環境で行うと更なる高速化が望めそうである。

また、評価関数の精度が良くなかったのか、先読み数増加を対戦結果に結びつけることはできなかった。このことを踏まえ、本研究で使用していないより複雑な評価関数を用いる事で勝率を上げることに繋げることが今後の課題である。

謝辞

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

参考文献

[1] Seal Software ,リバーシのアルゴリズム,工学社,2003

[2] 結城浩,Java言語で学ぶデザインパターン入門[マルチスレッド編],SoftBankCreative,2006

[3] 日本オセロ連盟,http://www.othello.gr.jp/beginner/s_01.html,オセロ講座,初心者,2002

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

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

[6] 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

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

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

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

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

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

[12] 橋本剛, 上田徹, 橋本隼一, オセロ求解へ向けた取り組み, 組合せゲーム・パズル ミニプロジェクト, 第3回ミニ研究会, 2008, http://www.lab2.kuis.kyoto-u.ac.jp/~itohiro/Games/Game080307.html#anchor3

[13] T.Ishii,MasterReversi,他アプリとの対局結果 http://homepage2.nifty.com/t_ishii/mr/gameresult.html,2007,

[14] 森田悠樹, 橋本剛 , 小林康幸, オセロ求解に向けた単純な縦型探索をベースにする探索方法の研究, 情報処理学会 ゲームプログラミングワークショップ2010論文集, No.12, pp.36-41, (2010), http://id.nii.ac.jp/1001/00071311/

付録

以下に本研究で作成したプログラムのソースを示す。

SSAI におけるクラス AI

```
  import java.util.*;

//シングルスレッド

abstract class AI {
          abstract public void move(Board board);
          public int kazu=0;
          public int presearch_depth =1;//3;
                              //α－β法やNegaScout法において、事前に手を調べて探索順序を決めるための先読み数
          public int normal_depth=2;//15;//序盤・中盤の探索における先読み数
          public int wld_depth=5;//15;//終盤において必勝読みを始める残り手数。W=winLlose=D=draw
          public int perfect_depth=3;//13;//終盤の完全読み

          public void setDepth(int kazu){
                    this.kazu = kazu;
                    //System.out.println("aaaaaaaaaaaaaaaaaaaaaaaa"+this.kazu);
                    presearch_depth = kazu+1;//3;
                              //α－β法やNegaScout法において、事前に手を調べて探索順序を決めるための先読み数
                    normal_depth = kazu+3;//15;//序盤・中盤の探索における先読み数
                    wld_depth = kazu+5;//15;//終盤において必勝読みを始める残り手数。W=winLlose=D=draw
                    perfect_depth = kazu+3;//13;//終盤の完全読み
          }
}

class AlphaBetaAI extends AI {
          class Move extends Point {
                    public int eval = 0;
                    public Move() {
                              super(0, 0);
                    }

                    public Move(int x, int y, int e) {
                              super(x, y);

                              eval = e;
                    }
          };

          private Evaluator Eval = null;

          public void move(Board board) {
                    BookManager book = new BookManager();
```

```
		Vector movables = book.find(board);

		if(movables.isEmpty()) {
			// 打てる箇所がなければパスする
			board.pass();
			return;
		}

		if(movables.size() == 1) {
			// 打てる箇所が一カ所だけなら探索は行わず、即座に打って返る
			board.move((Point) movables.get(0));
			return;
		}

		int limit;
		Eval = new MidEvaluator();
		sort(board, movables, presearch_depth); // 事前に手を良さそうな順にソート

		if(Board.MAX_TURNS - board.getTurns() <= wld_depth){
			limit = Integer.MAX_VALUE;
			if(Board.MAX_TURNS - board.getTurns() <= perfect_depth)
				Eval = new PerfectEvaluator();
			else Eval = new WLDEvaluator();
		} else {
			limit = normal_depth;
		}

		int eval, eval_max = Integer.MIN_VALUE;
		Point p = null;
		for(int i=0; i<movables.size(); i++) {
			board.move((Point) movables.get(i));
			eval = -alphabeta(board, limit-1, -Integer.MAX_VALUE, -Integer.MIN_VALUE);
			board.undo();

			if(eval > eval_max) p = (Point) movables.get(i);
		}

		board.move(p);
	}

	private int alphabeta(Board board, int limit, int alpha, int beta) {
		// 深さ制限に達したら評価値を返す
		if(board.isGameOver() || limit == 0) return evaluate(board);

		Vector pos = board.getMovablePos();
		int eval;

		if(pos.size() == 0) {
```

```
                    // パス
                    board.pass();
                    eval = -alphabeta(board, limit, -beta, -alpha);
                    board.undo();
                    return eval;
          }

          for(int i=0; i< pos.size(); i++) {
                    board.move((Point) pos.get(i));
                    eval = -alphabeta(board, limit-1, -beta, -alpha);
                    board.undo();

                    alpha = Math.max(alpha, eval);

                    if(alpha >= beta) {
                              // β刈り
                              return alpha;
                    }
          }
          return alpha;

}

private void sort(Board board, Vector movables, int limit) {
          Vector moves = new Vector();

          for(int i=0; i<movables.size(); i++) {
                    int eval;
                    Point p = (Point) movables.get(i);

                    board.move(p);
                    eval = -alphabeta(board, limit-1, -Integer.MAX_VALUE, Integer.MAX_VALUE);
                    board.undo();

                    Move move = new Move(p.x, p.y, eval);
                    moves.add(move);
          }

          // 評価値の大きい順にソート(選択ソート)

          int begin, current;
          for(begin = 0; begin < moves.size() - 1; begin++) {
                    for(current = 1; current < moves.size(); current++) {
                              Move b = (Move) moves.get(begin);
                              Move c = (Move) moves.get(current);
                              if(b.eval < c.eval) {
                                        // 交換
                                        moves.set(begin, c);
```

```
                                        moves.set(current, b);
                                }
                        }
                }
                // 結果の書き戻し

                movables.clear();
                for(int i=0; i<moves.size(); i++) {
                        movables.add(moves.get(i));
                }

                return;
        }

        private int evaluate(Board board) {
                return Eval.evaluate(board);
        }
}
```

MSAI におけるクラス AI

```
  import java.util.*;

//multithread
abstract class AI extends Thread {
        abstract public void move(Board board);

        private ReversiGame rga = new ReversiGame();
        public int kazu = rga.getKazu();
        public int presearch_depth =1+kazu;//3;
                //α－β法やNegaScout法において、事前に手を調べて探索順序を決めるための先読み数
        public int normal_depth=10+kazu;//15;//序盤・中盤の探索における先読み数
        public int wld_depth=15+kazu;//15;//終盤において必勝読みを始める残り手数。W=winLlose=D=draw
        public int perfect_depth=13+kazu;//13;//終盤の完全読み
}

class AlphaBetaAI extends AI  {

        class Move extends Point {
                public int eval = 0;
                public Move() {
                        super(0, 0);
                }

                public Move(int x, int y, int e) {
                        super(x, y);
```

```
                    eval = e;
            }
};

//Thread作る
// 1. Runnable型のオブジェクトを必要数突っ込む（必要数のスレッド作成
// 2. スレッド開始
// 3. スレッド.join()で同期

private Evaluator Eval = null;
private Board board;
private Vector movables;
private int eval;
private int eval_max;
private int limit;
//public int i=0;
private Thread[] thread;
private Point p = null;
private MultiAI[] multi;

public void move(Board board) {

            BookManager book = new BookManager();
            Vector movables = book.find(board);
            //ハブる作業
            if(movables.size() == 1) {
                    // 打てる箇所が一カ所だけなら探索は行わず、即座に打って返る
                    board.move((Point) movables.get(0));
                    return;
            }
            if(movables.isEmpty()) {
                    // 打てる箇所がなければパスする
                    board.pass();
                    return;
            }

            //ソート、先読み選択
            int limit;
            Eval = new MidEvaluator();
            setEval(Eval);
            sort(board, movables, presearch_depth); // 事前に手を良さそうな順にソート

            //limit変更
            if(Board.MAX_TURNS - board.getTurns() <= wld_depth) {
                    limit = Integer.MAX_VALUE;
                    if(Board.MAX_TURNS - board.getTurns() <= perfect_depth){
                            Eval = new PerfectEvaluator();
                            setEval(Eval);
```

```
                } else {
                        Eval = new WLDEvaluator();
                        setEval(Eval);
                }
        } else {
                limit = normal_depth;
        }

        int eval=0, eval_max = Integer.MIN_VALUE;

        thread = new Thread[movables.size()];
        multi = new MultiAI[movables.size()];
        for(int i=0; i<movables.size(); i++) {//ここから
                board.move((Point) movables.get(i));
                //eval = -alphabeta(board, limit-1, -Integer.MAX_VALUE, -Integer.MIN_VALUE);
                multi[i] = new MultiAI(board, limit-1, -Integer.MAX_VALUE, -Integer.MIN_VALUE);
                multi[i].setEval(Eval);
                thread[i] = new Thread(multi[i]);
                board.undo();
                //eval = multi[i].getAlpha();
                //if(eval > eval_max) p = (Point) movables.get(i);
                //System.out.printf("\n"+i+"\n");
        }
        for(int i=0;i<movables.size();i++){
                try{thread[i].join();}
                catch(InterruptedException e){System.out.println("ぬるぽ");}
        }
        int maximumEval=-Integer.MIN_VALUE;

        for(int i=0;i<movables.size();i++){
                eval = -multi[i].getAlpha();
                if(eval > maximumEval) {
                        maximumEval = multi[i].getAlpha();
                        p = (Point) movables.get(i);
                }
        }
        System.out.println("presearhdepth = "+ presearch_depth + "\n"
                         + "normal_depth = " + normal_depth + "\n"
         + "wld_depth = " + wld_depth + "\n"
         +"perfect_depth = " + perfect_depth + "\n");
//      System.out.println("\nAIの数受け取り確認"+kazu);
        board.move(p);
}

/**
int alphabeta(Board board, int limit, int alpha, int beta) {
        // 深さ制限に達したら評価値を返す
        if(board.isGameOver() || limit == 0) return evaluate(board);
```

```
		Vector pos = board.getMovablePos();
		int eval;

		if(pos.size() == 0) {
			// パス
			board.pass();
			eval = -alphabeta(board, limit, -beta, -alpha);
			board.undo();
			return eval;
		}

		for(int i=0; i< pos.size(); i++) {
			board.move((Point) pos.get(i));
			eval = -alphabeta(board, limit-1, -beta, -alpha);
			board.undo();

			alpha = Math.max(alpha, eval);

			if(alpha >= beta) {
				// β刈り
				return alpha;
			}
		}

		return alpha;
	}
	*/
	private void sort(Board board, Vector movables, int limit) {
		Vector moves = new Vector();
		thread = new Thread[movables.size()];
		multi = new MultiAI[movables.size()];
		for(int i=0; i<movables.size(); i++) {
			int eval;
			Point p = (Point) movables.get(i);

			board.move(p);
			//eval = -alphabeta(board, limit-1, -Integer.MAX_VALUE, Integer.MAX_VALUE);
			multi[i] = new MultiAI(board, limit-1, -Integer.MAX_VALUE, -Integer.MIN_VALUE);
			multi[i].setEval(Eval);
			thread[i] = new Thread(multi[i]);

			board.undo();

			//Move move = new Move(p.x, p.y, eval);
			//moves.add(move);

		}
```

```
                for(int i=0;i<movables.size();i++){
                        try{thread[i].join();}
                        catch(InterruptedException e){System.out.println("ぬるぽ");}
                }
                for(int i=0; i<movables.size(); i++) {
                        Point p = (Point) movables.get(i);
                        int eval = multi[i].getAlpha();
                        Move move = new Move(p.x, p.y, eval);
                        moves.add(move);
                }

                // 評価値の大きい順にソート(選択ソート)

                int begin, current;
                for(begin = 0; begin < moves.size() - 1; begin++) {
                        for(current = 1; current < moves.size(); current++) {
                                Move b = (Move) moves.get(begin);
                                Move c = (Move) moves.get(current);
                                if(b.eval < c.eval) {
                                        // 交換
                                        moves.set(begin, c);
                                        moves.set(current, b);
                                }
                        }
                }
                // 結果の書き戻し

                movables.clear();
                for(int i=0; i<moves.size(); i++) {
                        movables.add(moves.get(i));
                }

                return;
        }

        public void setEval(Evaluator Eval){
                this.Eval=Eval;
        }
        /**
        private int evaluate(Board board) {
                return Eval.evaluate(board);
        }
        */
}
```

MSAI におけるクラス MultiAI.java

```
import java.util.Vector;
```

```
public class MultiAI implements Runnable {

	private Board board;
	private int limit;
	private int alpha;
	private int beta;
	private Evaluator Eval;

	public MultiAI(Board board,int limit,int alpha,int beta){
		this.board = board;
		this.limit = limit;
		this.alpha = alpha;
		this.beta = beta;

	}

	public void setEval(Evaluator Eval){
		this.Eval = Eval;
	}
	private int evaluate(Board board)
	{

		int e = Eval.evaluate(board);
		return e;
	}

	public void setAlpha(int alpha){
		this.alpha = alpha;
	}
	public int getAlpha(){
		return alpha;
	}

	public void run(){
		// 深さ制限に達したら評価値を返す
				if(board.isGameOver() || limit == 0){
					int alpha = evaluate(board);
					setAlpha(alpha);
					//System.out.printf("\n a");
					return;
				}

				Vector pos = board.getMovablePos();
				int eval;

				if(pos.size() == 0) {
					// パス
```

```
            board.pass();
            //eval = -alphabeta(board, limit, -beta, -alpha);
            MultiAI multi = new MultiAI(board,limit,-beta,-alpha);
            Thread th = new Thread(multi);
            th.start();
            //System.out.println("ぬるぽMultiAI");
            try{th.join();}
            catch(InterruptedException e){System.out.println("ぬるぽ");};
            //System.out.println("ぬるぽMultiAI");
            eval = -getAlpha();
            setAlpha(eval);
            //System.out.println("ぬるぽMultiAI");
            board.undo();
            //System.out.printf("\n b");
            return;
        }
        //
        MultiAI[] multi = new MultiAI[pos.size()];
        Thread[] th = new Thread[pos.size()];
        int evalu[] = new int[pos.size()];
        for(int i=0; i< pos.size(); i++) {
            board.move((Point) pos.get(i));
            //eval = -alphabeta(board, limit-1, -beta, -alpha);
            multi[i] = new MultiAI(board,limit,-beta,-alpha);
            th[i] = new Thread(multi[i]);
            th[i].start();
            board.undo();
        }
        for(int i=0; i< pos.size(); i++) {
            try{th[i].join();}
            catch(InterruptedException e){System.out.println("ぬるぽ");};
        }
        for(int i=0; i< pos.size(); i++) {
            evalu[i] = -multi[i].getAlpha();
            alpha = Math.max(alpha, evalu[i]);

            if(alpha >= beta) {
                // β刈り
                //return alpha;
                setAlpha(alpha);
                return;
            }

            //System.out.printf("\n"+i+"\n");
        }
        /**
        for(int i=0;i<pos.size();i++){
            try{th[i].join();}
```

```
                    catch(InterruptedException e){System.out.println("ぬるぽ");}
                }
                */
                //return alpha;
                setAlpha(alpha);
                return;
            }
        }
```