# 修 士 論 文 の 和 文 要 旨

| | | | |
|---|---|---|---|
| 研究科・専攻 | 大学院　　情報理工学　研究科　　情報・通信工学　専攻　博士前期課程 | | |
| 氏　　名 | 小林　祐樹 | 学籍番号 | 1431045 |
| 論 文 題 目 | モンテカルロ木探索を用いた強い囲碁プログラムの設計と開発 | | |


要　旨

本論文では，モンテカルロ木探索を用いた囲碁プログラムのデータ構造を考え，学習手法やヒューリスティックの有無がプログラムの強さにどのように影響するかを考察した．

モンテカルロ木探索は木探索部とシミュレーション部に分けられ，さらにシミュレーション部は大きく 2 つに分けられる．1 つは盤全体の着手確率のテーブルを保持して，その中からランダムに着手を選ぶ非決定論的シミュレーションであり，もう 1 つは直前の着手に対応する良さそうな手があったら，即座にその手を選ぶ決定論的シミュレーションである．決定論的シミュレーションを行うオープンソースソフトウェアに Pachi や Fuego などがあるが，それらと同等の棋力を持つ非決定論的シミュレーションを行うオープンソースソフトウェアは存在しない．

本研究では，非決定論的シミュレーションを用いる囲碁プログラムを設計し，Pachi や Fuego よりも強い囲碁プログラムの開発を目指した．

プログラムに利用している学習手法の妥当性や，ヒューリスティックの有効性を示し，13 路盤と 19 路盤では Pachi と Fuego よりも強いプログラムを開発できた．

非決定論的シミュレーションを行うプログラムを作成し，オープンソースソフトウェアとすることで，様々な手法の有効性を 2 つのシミュレーション手法の違いを含めて検証できるようになった．

電気通信大学情報理工学研究科
情報・通信工学専攻情報数理工学コース修士論文

# モンテカルロ木探索を用いた強い囲碁プログラムの設計と開発

平成 28 年 2 月 26 日

情報数理工学コース

学籍番号 1431045

小林 祐樹

指導教員 村松 正和 教授

# 目 次

# 1　はじめに

人工知能の研究は「人間の知的活動をコンピュータで実現する」ことを目的としている. 適度に難しく, 目標の設定と結果の評価が簡単な思考型ゲームは人工知能研究の題材として適している.

思考型ゲームの研究は様々なゲームで行われており, チェスや将棋, 囲碁などに代表される二人零和有限確定完全情報ゲームにおいては, アマチュアのトップレベルからプロに匹敵するほどの強さのプログラムが生み出されている. チェスにおいては, 1997 年に International Business Machines Corporation(IBM 社) が開発したチェスプログラム Deep Blue が当時の世界チャンピオンである Garry Kimovich Kasparov に 2 勝 1 敗 3 引き分けと勝利を収めている [1]. 将棋においては, 2014 年に行われた将棋電王戦 FINAL [15] で現役のプロ棋士相手にコンピュータ側の 2 勝 3 敗と負け越したものの, 将棋電王戦通算でコンピュータ側の 10 勝 5 敗 1 引き分けとコンピュータ側が勝ち越しており, 2015 年に情報処理学会が「2015 年の時点でトッププロ棋士に追い付いている」としてコンピュータ将棋プロジェクトの終了宣言をしている [14].

一方で囲碁では, 2015 年に行われた第 3 回電聖戦において, 対局時に石を 4 つ置くハンデ戦で Lim Jaebum によって開発された囲碁プログラム DolBaram が二十五世本因坊治勲に勝利したものの, 対局時に石を 3 つ置くハンデ戦で Rémi Coulom によって開発された囲碁プログラム Crazy Stone が敗北している [16]. 囲碁においては, チェスや将棋と異なり, 依然としてプロの実力に匹敵していないことから, 現在でもプロを超える強さを実現する手法の研究が行われ続けている.

従来, 囲碁プログラムは知識ベースのアルゴリズムが主流であり, 囲碁プログラムの開発者たちは正確で高速な評価関数の作成に尽力してきたが, アマチュア段位者レベルには到底及ばなかった. しかし, 2006 年の Computer Olympiad の 9 路盤部門で Crazy Stone が優勝したことで, モンテカルロ木探索が注目を集めた.

モンテカルロ木探索は, 評価したい着手を選択し, その局面から終局までランダムに着手を行う対局のシミュレーションを行い, 最も勝率の高い手を選ぶという探索アルゴリズムである. 従来の知識ベースのアルゴリズムと異なり, 着手の評価にランダムシミュレーションの勝敗を用いるため評価関数が必要なく, また評価関数の作成のための高度な囲碁の知識も必要ない. このため, プログラムの開発に多大な労力と囲碁に関する知識を要しないにもかかわらず, 従来の知識ベースのアルゴリズムよりも強いという画期的で革新的なアルゴリズムであった. 現在ではモンテカルロ木探索が囲碁プログラムの主流となっており, 評価したい着手の選択の際に UCB(Upper Confidence Bound) 値を用いる UCT(Upper Confidence bound applied to Trees) アルゴリズムが用いられている [2].

モンテカルロ木探索には木探索部とシミュレーション部に分かれている. シミュレーション部に関しては, 大きく 2 つの方法に分けられる. 1 つは盤全体の着手確率のテーブルを保持して, その中からランダムに着手を選ぶ非決定論的シミュレーションであり, もう 1 つは直前の着手に対応する良さそうな手があったら即座にその手を選ぶ決定論的シミュレーションである. どちらのシミュレーションも評価したい着手がどの程度終局時の勝敗に影響するかを評価できるように, 前者は教師データから確率分布を機械学習した結

果を用いて, 後者は手作りのパターンを用いて, シミュレーションを行う. 決定論的シミュレーションを用いるプログラムとして世界最強クラスのプログラムである Zen や, オープンソースソフトウェアの中で最強クラスのプログラムである Fuego [18] や Pachi [19] の他に, oakfoam [20] がある. 一方で決定論的シミュレーションを用いるプログラムには世界最強クラスのプログラムである Crazy Stone や日本の強豪プログラム Aya があるものの, オープンソースソフトウェアは存在していない.

本研究では, 非決定論的シミュレーションを行うプログラムの設計と開発を考える. モンテカルロ木探索に重要な要素である着手評価の学習手法とシミュレーションの確率分布の学習手法, 探索をより効率的に行うであろうヒューリスティックについて述べ, 各ヒューリスティックの効果について考える.

本稿の構成は以下の通りである. 2 章ではモンテカルロ木探索を用いたコンピュータ囲碁の基礎知識を示す. 3 章では自作の囲碁プログラム Ray に用いている先行研究について説明する. 4 章では囲碁プログラム Ray に用いられている技術と設計について説明する. 5 章では各手法の違いによる棋力の変化を測定するための実験の結果と考察を記す. 最後に, 6 章にまとめと今後の課題を述べる.

# 2　基礎知識

## 2.1　初期のモンテカルロ碁

モンテカルロ法を用いたコンピュータ囲碁の研究は 1993 年 Brügmann によって行われた [3]. 初期のモンテカルロ碁では, 探索開始局面から合法手の着手を 1 つ選び, それ以降の着手を合法手の中から乱数を用いて選択することを終局するまで行い, 終局時の結果を記録する. この動作を繰り返し行い, シミュレーションの勝率が最も高い手が最も良い手であると見なし, その着手を選ぶ手法である. またシミュレーションの際には, 自分の眼には着手をしないという制約をつけている. この制約には, ゲームを終局させる働きがある. 図 1 は評価したい局面の例であり, 図 2 は図 1 から終局までランダムに着手した局面である.

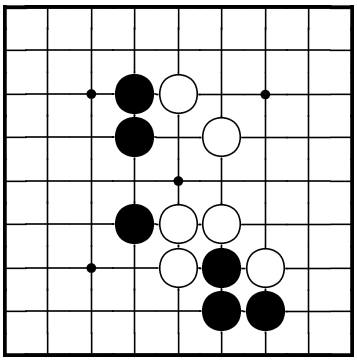

図 1: 探索開始局面

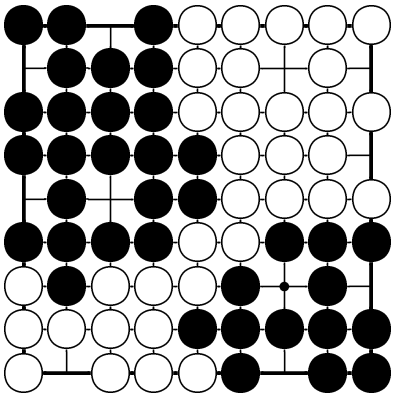

図 2: 終局時のプレイアウトの局面

図 2 を見ると, どちらの手番も自分の眼に着手する以外の合法手が存在しなくなっている. このような局面になった時に終局と見なし, 黒と白のどちらの勝ちであるかを判断する. この例では, コミ 6.5 目の場合, 白の勝ちになるので, 白の勝ちを記録しておく. このような記録を全合法手に保持しておき, 最も勝率の高い着手を選ぶ.

初期のモンテカルロ碁では, 評価したい着手を選んだ後は即座にランダムに着手するシミュレーションが行われるため, 2 手以上読むことができず, 相手のミスを期待する着手を選ぶ傾向がある.

## 2.2　モンテカルロ木探索

囲碁に対しては, チェスや将棋などで成功を収めてきた minimax 探索や alpha-beta 探索はうまく機能しなかった. これは囲碁がチェスや将棋よりも膨大な探索空間を持つことと, チェスや将棋のような高速で正確な評価関数の作成が困難であることに起因する.

モンテカルロ木探索では以下の手順を繰り返し行う.

1. 末端ノードに到達するまで, 評価したい着手を選択する.
2. 末端ノードの探索回数が閾値を超えていたら, 新たに末端ノードの先の局面のノードを展開して, 1 に戻る.
3. 終局するまでランダムに合法手を選び, 対局のシミュレーションを行う.
4. シミュレーションで得られた対局結果を選択してきたノードに反映させる.

この一連の処理をプレイアウトと呼ぶ. モンテカルロ木探索は対局のシミュレーションの結果のみを用いることで局面の評価を与える. 評価関数を必要としないので, 正確で高速な評価関数の作成が困難なゲームであっても有効であるとされている. また探索回数が閾値を超えていたら, ノードを展開するので, 相手の着手も含めて評価することができる. モンテカルロ木探索のアルゴリズムを Algorithm 1 に示す.

**Algorithm 1** モンテカルロ木探索 MCTS

**Require:** 現在の評価局面 $s$
**Ensure:** 対局の結果 $r$

**if** $s \notin$ leaf **then**
  $s' \leftarrow$ SelectChild($s$) // SelectChild 関数は局面 $s$ の次の局面を選ぶ関数
  $r \leftarrow$ MCTS($s'$)
**else**
  **if** $s$ 探索回数が閾値を超えている **then**
    ExpandNode($s$)
    $s' \leftarrow$ SelectChild($s$)
    $r \leftarrow$ MCTS($s'$)
  **else**
    $r \leftarrow$ Simulation($s$) // Simulation 関数は終局まで対局を行い, その結果を返す関数
  **end if**
**end if**
Update($s$, $r$) // Update 関数は局面 $s$ に探索結果 $r$ を反映させる関数
**return** $r$

## 2.3 UCT アルゴリズム

モンテカルロ木探索において, どの子ノードを選ぶかという問題は非常に重要である. 勝率のみを基準にすると, 1 回目のプレイアウトでたまたま負けてしまった場合に勝率が 0%になり, ずっと選択されなくなってしまう. 一方, プレイアウト回数が均等になるように, プレイアウト回数が少ない子ノードを選択すると, 探索が分散してしまい, 無駄な手の評価をする回数が多くなるので非効率である. 勝率の高いノードにプレイアウトを集中させつつ, 勝率が実際よりも低い可能性があるプレイアウト回数の少ないノードも探索す

る必要がある. この 2 つのジレンマの解決策として, ノードを選択する際の基準に UCB1 値を利用し, モンテカルロ木探索を行うものが UCT(Upper Confidence Bound applied for Tree) アルゴリズム [2] である.

UCT アルゴリズムは式 (1) で求められる UCB1 値が最大の子ノードを選択してモンテカルロ木探索を行う :

$$\mathrm{UCB}(i) = \bar{x}_i + \sqrt{\frac{2\log n}{n_i}}\ . \tag{1}$$

. ただし, 各パラメータは

$\bar{x}_i$ : ノード $i$ の勝率
$n_i$ : ノード $i$ のプレイアウト回数
$n$ : 全ての子ノードのプレイアウト回数の合計

を表す. 式 (1) の第 1 項により, 勝率の高い子ノードが選ばれやすくなる一方で, 第 2 項によって, プレイアウト回数の少ない子ノードが選ばれやすくなる. したがって, 勝率の高いノードとプレイアウト回数の少ないノードをバランスを取りながら探索することになる. 理論的には, 適切な条件下で UCB1 値を用いれば, 最終的には有望な子ノードに探索が集中することが知られている [2].

また, UCB1 値に報酬の分散を考慮したものとして, UCB1-TUNED 値 [11] がある. これは

$$V_i(s) = \left(\frac{1}{s}\sum_{\tau=1}^{s} X_{i,\tau}^2\right) - \bar{X}_{i,s}^2 + \sqrt{\frac{2\log n}{s}}$$

として,

$$\mathrm{UCB1\text{-}TUNED}(i) = \bar{x}_i + \sqrt{\frac{\log n}{n_i}\min\left\{\frac{1}{4}, V_i(n_i)\right\}}$$

で与えられる値である. ここで各パラメータは以下を表す :

$X_{i,\tau}$ : 候補手 $i$ の $\tau$ 回目の試行の報酬
$\bar{X}_{i,s}$ : 候補手 $i$ の報酬の平均 .

## 2.4 Progressive Widening

Progressive Widening [4] は初めに探索候補の数を絞り込み, 探索回数が増えるにしたがって, 探索候補を徐々に追加していく前向き枝刈りの手法である. ノードの探索候補の数を

$$t_0 = 0,$$
$$t_{n+1} = t_n + 40 \times 1.4^n$$

で制限する. すなわち $n$ 番目に有望な候補手はプレイアウト回数が $t_{n-1}$ に達したときに探索候補に追加される.

## 2.5 連

連とは上下左右に隣接している石の集合を表す. 縦横に繋がっている石は盤上から取り除かれるときは必ず全て一緒に取り除かれることから, 繋がりのある石をまとめて扱うことは有用である. 図 3 に例を示す. 図 3 における○, × , △の黒石と, □の白石はそれぞれ別の連である.

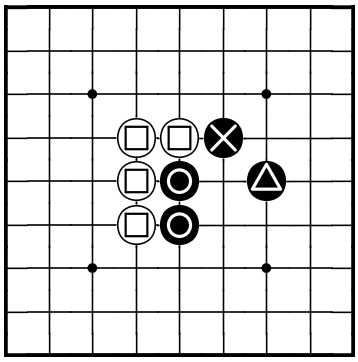

図 3: 連の一例

コンピュータ囲碁に連を導入するときには, 連を構成する石の座標, 個数, 色, 識別番号だけではなく, 呼吸点の座標と個数, 隣接している敵の連の識別番号を持たせておくと特徴の抽出に役立つ.

# 3 先行研究

## 3.1 Bradley-Terry モデルを用いた Minorization-Maximization アルゴリズム

ここでは Rémi Coulom による手法である Bradley-Terry モデルを用いた Minorization-Maximization アルゴリズム [4] を説明する.

### 3.1.1 Bradley-Terry モデル

Bradley-Terry モデルは, プレイヤの強さに基づいて, 試合結果を予測するモデルである. プレイヤ $i$ の強さを正の実数 $\gamma_i$ とし, プレイヤが強いほど $\gamma_i$ は大きな値となる. プレイヤ $i$ がプレイヤ $j$ に勝つ確率を

$$P(i \text{ が } j \text{ に勝つ}) = \frac{\gamma_i}{\gamma_i + \gamma_j}$$

で推定する. このモデルは 1 対 1 の試合だけではなく, $n$ 人の試合を扱うように一般化でき,

$$P(i \text{ が勝つ}) = \frac{\gamma_i}{\sum_{k=1}^{n} \gamma_k}$$

と表すことができる. さらに, 個人間だけでなくチーム間の試合を考慮するようにできる. この一般化において, チームの強さはチームを構成するメンバの強さ $\gamma$ の積で推定される. 例えば, プレイヤが 5 人いて, それぞれの強さを $\gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5$ としたときにプレイヤ 1 とプレイヤ 2 で構成されたチームが, プレイヤ 2,　プレイヤ 3, プレイヤ 5 で構成されたチームとプレイヤ 1, プレイヤ 3, プレイヤ 4, プレイヤ 5 で構成されたチームに勝つ確率は.

$$P(\text{1-2 が 2-3-5 と 1-3-4-5 に勝つ}) = \frac{\gamma_1\gamma_2}{\gamma_1\gamma_2 \ + \gamma_2\gamma_3\gamma_5 + \gamma_1\gamma_3\gamma_4\gamma_5} \tag{2}$$

で推定される. ここで同じプレイヤが複数のチームに現れることはあるが, 同じプレイヤが同じチームに 2 回以上現れることはないことに注意する.

### 3.1.2 Minorization-Maximization アルゴリズム

プレイヤ $i$ に着目して独立な $N$ 回の試合の結果が既知であるとすると, 試合 $j$ の結果が起こる確率は.

$$P(R_j) = \frac{A_{ij}\gamma_i + B_{ij}}{C_{ij}\gamma_i + D_{ij}} \tag{3}$$

と表すことができ, 各変数は

$$R_j : \text{試合 } j \text{ の結果}$$
$$\gamma_i : \text{プレイヤ } j \text{ の強さ}$$
$$A_{ij}, B_{ij}, C_{ij}, D_{ij} : \gamma_i \text{ について独立な係数}$$

である. 例えば, プレイヤ 1 に着目すると, 式 (2) は次のように書き表せる:

$$R_1 = \text{1-2 が 2-3-5 と 1-3-4-5 に勝つ}$$
$$P(R_1) = \frac{\gamma_2 \cdot \gamma_1}{(\gamma_2 + \gamma_3\gamma_4\gamma_5) \cdot \gamma_1 + \gamma_2\gamma_3\gamma_5} .$$

すなわち

$$\begin{aligned} A_{11} &= \gamma_2 \\ B_{11} &= 0 \\ C_{11} &= \gamma_2 + \gamma_3\gamma_4\gamma_5 \\ D_{11} &= \gamma_2\gamma_3\gamma_5 \end{aligned}$$

である. 同様に他のプレイヤを考えると $A_{21} = \gamma_1$, $A_{31} = A_{41} = A_{51} = 0$ となり, プレイヤ $i$ の勝ち数 $W_i$ は

$$W_i = |\{j | A_{ij} \neq 0\}|$$

で求められる. 目的は.

$$L = \prod_{j=1}^{N} P(R_j) \tag{4}$$

の最大化である. 式 (3) と式 (4) から, プレイヤ $i$ に着目した目的関数は

$$L = \prod_{j=1}^{N} \frac{A_{ij}\gamma_i + B_{ij}}{C_{ij}\gamma_i + D_{ij}} \tag{5}$$

と表すことができ, 式 (5) は $\gamma_i$ の関数と見なすことができる. 式 (5) の対数を取ると.

$$\log L(\gamma_i) = \sum_{j=1}^{N} \log(A_{ij}\gamma_i + B_{ij}) - \sum_{j=1}^{N} \log(C_{ij}\gamma_i + D_{ij}) \tag{6}$$

が得られる. ここで

$$\begin{aligned} &A_{ij} \neq 0, B_{ij} = 0 \Longleftrightarrow \text{プレイヤ } i \text{ が試合 } j \text{ に勝ったチームのメンバである} \\ &A_{ij} = 0, B_{ij} \neq 0 \Longleftrightarrow \text{プレイヤ } i \text{ が試合 } j \text{ に勝ったチームのメンバでない} \end{aligned}$$

であるから, 式 (6) は式 (7) に変形できる:

$$\log L(\gamma_i) = \sum_{j=1, B_{ij}=0}^{N} \{\log(A_{ij}) + \log(\gamma_i)\} + \sum_{j=1, A_{ij}=0}^{N} \log(B_{ij}) - \sum_{j=1}^{N} \log(C_{ij}\gamma_i + D_{ij}) . \tag{7}$$

式 (7) の $\gamma_i$ を含まない項は定数と見なせるので, 定数項を除くと最大化する関数は

$$f(x) = W_i \log x - \sum_{j=1}^{N} \log(C_{ij}x + D_{ij}) \tag{8}$$

となる. ここで,$a, b$ を定数とし,

$$g(x) = -\log(ax + b) \tag{9}$$

とすると, $x = \gamma_i$ における式 9 の接線は.

$$y = -\frac{a}{a\gamma_i + b}(x - \gamma_i) - \log(a\gamma_i + b) \tag{10}$$

と表せる. 式 (10) は式 (9) の接線であるので,

$$g(x) \geq -\frac{a}{a\gamma_i + b}(x - \gamma_i) - \log(a\gamma_i + b)$$

を満たす. したがって, 式 (8) の第 2 項を式 (10) に置き換え, $x$ を含まない項を取り除くと,

$$m(x) = W_i \log x - \sum_{j=1}^{N} \frac{C_{ij}x}{C_{ij}\gamma_i + D_{ij}}$$

が得られる. $m(x)$ の最大値は $m'(x) = 0$ を解くことによって見つけられるので, $E_j = C_{ij}\gamma_i + D_{ij}$ とすれば,

$$x = \frac{W_i}{\sum_{j=1}^{N} \frac{C_{ij}}{E_j}}$$

から $m(x)$ が最大となる $x$ が求められる. よって, Minorization-Maximization アルゴリズムは,

$$\gamma_i \leftarrow \frac{W_i}{\sum_{j=1}^{N} \frac{C_{ij}}{E_j}}$$

にしたがってパラメータ $\gamma_i$ を繰り返し更新する. ここでの各変数は

$C_{ij}$ : 試合 $j$ でプレイヤ $i$ が所属するチームの仲間の強さ
$E_j$ : 試合 $j$ に参加したチームの強さの総和

である.

### 3.1.3 囲碁への適用

学習する棋譜のある局面で打たれた手を勝者, それ以外の候補手を敗者とする試合と見なすことで Minorization-Maximization アルゴリズムを適用できる. Bradley-Terry モデルにおけるプレイヤとはそれぞれの特徴の打たれやすさを表し, それぞれのチームは 1 つの着手の打たれやすさを表すことになる.

## 3.2 Latent Factor Ranking アルゴリズム

Latent Factor Ranking(LFR) アルゴリズム [7] は 2 つの特徴の相互作用を考慮することができる着手評価アルゴリズムである. 2 つの特徴の相互作用を考慮に入れる際には, 一見 $O(m^2)$ 個の重みを用意して, それぞれのパラメータを調整すればよいように見えるが, コンピュータ囲碁のようにパラメータの個数が大きい場合, 学習するデータの個数が実際に調整すべきパラメータの個数より少なくなるために学習が十分に行われないばかりか, 過学習をしてしまう. 2 つの特徴の相互作用を用いながら, パラメータの個数の増加を抑えるためには工夫が必要であり, この要求をうまく満たしているものが Factorization Machines [8] である. LFR アルゴリズムは Factorization Machines のアイデアを用いて, 囲碁の着手評価を行う.

LFR アルゴリズムにおける着手評価を求める式を式 (11) に示す:

$$\hat{y}(\boldsymbol{x}) := w_0 + \sum_{i=1}^{m} w_i x_i + \sum_{i=1}^{m} \sum_{j=i+1}^{m} \boldsymbol{v_i^T v_j} x_i x_j \ . \tag{11}$$

ここで, 各パラメータは

$\boldsymbol{x} \in \{0,1\}^m$ : 特徴の有無を表すベクトル

$w_0$ : バイアス項

$w_i$ : 特徴 $i$ の重み

$\boldsymbol{v_i}$ : 特徴 $i$ の相互作用ベクトル ($k$ 次元)

$m$ : 特徴の個数

である. $k$ は非常に小さい正の整数であり, $k \ll m$ である. $m$ 個の特徴に対して, 2 つの特徴の組み合わせた特徴の重みを $m^2$ 個のパラメータで表現するのではなく, 相互作用ベクトル $\boldsymbol{v_i}$, $\boldsymbol{v_j}$ の内積で表現することによって, 必要なパラメータの個数を $km$ 個に削減している. 教師データから与える着手の評価値は,

$$y(\boldsymbol{x}) := \begin{cases} 1 & (\text{教師データの着手が特徴 } \boldsymbol{x} \text{ を持つとき}) \\ 0 & (\text{otherwise}) \end{cases}$$

で求める. これらを用いて, LFR アルゴリズムの目的関数は.

$$\frac{1}{2} \sum_{D_j \in D} \sum_{\boldsymbol{x_i} \in D_j} \left\{ \hat{y}(\boldsymbol{x_i}) - y(\boldsymbol{x_i}) \right\}^2 + \frac{\lambda_w}{2} \sum_{i=1}^{m} w_i^2 + \frac{\lambda_v}{2} \sum_{i=1}^{m} ||\boldsymbol{v_i}||^2 \tag{12}$$

と表せる. 各パラメータは

$D$ : 教師データの持つ局面の集合

$D_j$ : 教師データのの局面 $i$ における合法手が持つ特徴ベクトルの集合

$x_i$ : 教師データの局面 $i$ における合法手 $j$ が持つ特徴ベクトル

$\lambda_w$ : 特徴の重みに対する正則化パラメータ

$\lambda_v$ : 特徴の相互作用ベクトルに対する正則化パラメータ

である. 式 (12) が小さくなるように確率的勾配法を用いてパラメータ $w_i$, $\boldsymbol{v_i}$ を更新していく. ある局面 $D_i$ を固定して考えると, 式 (11) を $w_i$ について偏微分すると,

$$\frac{\partial \hat{y}(\boldsymbol{x})}{\partial w_i} = \begin{cases} 1 & (x_i = 1) \\ 0 & (\text{otherwise}) \end{cases}$$

が得られる. 同様に, 式 (11) を $\boldsymbol{v_i}$ の第 $f$ 番目の要素 $v_{i,f}$ について偏微分すると.

$$\frac{\partial \hat{y}(\boldsymbol{x})}{\partial v_{i,f}} = \begin{cases} \sum_{j=1, j\neq i}^{m} v_{j,f} x_j & (x_i = 1) \\ 0 & (\text{otherwise}) \end{cases}$$

が得られる. $w_0$ については, 簡単に求めることができて

$$\frac{\partial \hat{y}(\boldsymbol{x})}{\partial w_0} = 1$$

となる.

これらを用いることで, 各パラメータを更新する. また $w_0$, $w_i$ は 0 で初期化し, $\boldsymbol{v_i}$ は平均 0, 分散 0.01 の正規分布に従う乱数によって初期化する. LFR アルゴリズムを Algorithm 2 に示す.

$\alpha$, $\lambda_w$, $\lambda_v$ をあらかじめ決める必要があるが, [7] では $k = 5$ のとき, $\alpha = 0.001$, $\lambda_w = 0.001$, $\lambda_v = 0.002$ が最適としている. また学習の停止条件である “改善がない” とは最近 3 回の学習結果の着手予想精度が向上していないことを意味する.

### 3.3 行動評価関数を用いたモンテカルロ木探索の重点化

UCT アルゴリズムでは, UCB1 値が最大のノードを選択して木を下っていく. UCB1 値は子ノードの勝率, 子ノードの探索回数, 親ノードの探索回数の 3 つの要素にのみ影響され, それ以外の要素の影響は受けないので, Minorization-Maximization アルゴリズムや Latent Factor Ranking アルゴリズムで学習した着手評価は Progressive Widening のような前向き枝刈りに用いられるものの, UCB1 値には一切作用しない. シミュレーション部に囲碁の知識を導入して探索の効率化を図ることができるならば, 木探索部に囲碁の知識を導入して効率的に探索を行うことを考えるのは自然な要求である. Chaslot ら [9] は手動および一部自動で調整した行動評価関数を UCB 値の補正に用いることで囲碁プログラムの棋力が向上することを示しており, 池田ら [10] は UCB 値の補正する行動評価関数に BT モデルを用いた Minorization-Maximization アルゴリズムで得られた各着手の着手確率を用いることで囲碁プログラムの棋力が向上することを示している. UCB1 値に行動評価関数による補正項を加えたものを式 13 に示す:

$$\text{UCB1}(i) = \bar{x}_i + C\sqrt{\frac{2\log n}{n_i}} + C_H\sqrt{\frac{K}{n+K}} \cdot H(i)\ . \tag{13}$$

**Algorithm 2** Latent Factor Ranking アルゴリズム

**Require:** 教師データの集合 $D$
**Ensure:** 学習で得られたバイアス項 $w_0$
**Ensure:** 学習で得られた特徴の重み $\boldsymbol{w} = (w_1, ..., w_m)$
**Ensure:** 相互作用ベクトルの集合 $V = (\boldsymbol{v_1}, ..., \boldsymbol{v_m})$

InitializeParameter($w_0$, $\boldsymbol{w}$, $V$) // 各パラメータを初期化する関数
**while** 改善がない **do**
  **for all** $D_j \in D$ **do**
    $\bar{\boldsymbol{x}} \leftarrow$ GetProMove($D_i$) // 局面 $D_j$ から教師データの着手の持つ特徴を返す関数
    **for all** $\boldsymbol{x} \in D_j$ **do**
      **if** $\hat{y}(\boldsymbol{x}) \geq \hat{y}(\bar{\boldsymbol{x}})$ **then**
        $\Delta y \leftarrow \hat{y}(\boldsymbol{x}) - y(\boldsymbol{x})$
        $w_0 \leftarrow w_0 - \alpha \Delta y$
        **for all** $x_i \in \boldsymbol{x}$ **do**
          **if** $x_i = 1$ **then**
            $w_i \leftarrow w_i - \alpha(\Delta y + \lambda_w w_i)$
            **for** $f = 1$ to $k$ **do**
              $v_{i,f} \leftarrow v_{i,f} - \alpha \left( \Delta y \frac{\partial}{\partial v_{i,f}} \hat{y}(\boldsymbol{x}) + \lambda_v v_{i,f} \right)$
            **end for**
          **end if**
        **end for**
      **end if**
    **end for**
  **end for**
**end while**

ただし, 各パラメータは以下を表している：

$\bar{x}_i$ : 着手 $i$ の勝率
$n$ : 現局面の探索回数
$n_i$ : 着手 $i$ の探索回数
$C$ : UCB1 値の補正項の重み
$C_H$ : 行動評価関数による補正項の重み
$K$ : 行動評価関数による補正項の減衰パラメータ
$H(i)$ : 着手 $i$ の行動評価値 .

行動評価関数による補正項を追加する狙いは, 良いと思われる着手を優先的に探索することにある. UCB1 値のみで着手を選択すると, 着手評価の高い, すなわち良いと思われる手であっても, 連続して負けのシミュレーションが続いてしまうと, その手が選ばれにくくなってしまう. しかし, 行動評価関数による補正項によって, 最初に敗けのシミュレーションが連続しても, その手をある程度集中的に探索してくれるようになる.

## 3.4 Ownership と Criticality

Ownership [4], Criticality [5] ともにモンテカルロシミュレーションに基づく動的な指標である.

Ownership はモンテカルロシミュレーションによる終局時の各座標の占有率を表すもので, 式 (14) によって求められる.

$$\text{Ownership}(c, x) = \frac{t(c, x)}{N} \tag{14}$$

$c$ : 色
$x$ : 座標
$t(c, x)$ : 座標 $x$ が色 $c$ の陣地になった回数
$N$ : プレイアウト回数

色 $c$ を固定して考えると, Ownership の値が大きいほど, その手番の陣地になるシミュレーションの割合が多く, Ownership の値が小さいほど, その手番の陣地になるシミュレーションの割合が少ないということがわかる. すなわち Ownership の値が 0, または 1 に近い場合にはどちらかの安定した陣地になっており, 0.5 に近い値を取る箇所がどちらの陣地になるか未確定であると判別できる. よって, Ownership の値が 0.5 に近い箇所ほど重要な箇所であると判断できる.

Criticality は各座標がモンテカルロシミュレーションの勝敗にどの程度影響を与えているかを表す指標で, 式 (15) で表される：

$$\text{Criticality}(x) = \frac{v(x)}{N} - \left( \frac{w(x)}{N} \times \frac{W}{N} + \frac{b(x)}{N} \times \frac{B}{N} \right) . \tag{15}$$

ただし,

$x$ : 座標
$N$ : プレイアウト回数
$B$ : 黒が勝ったプレイアウト回数
$W$ : 白が勝ったプレイアウト回数
$v(x)$ : プレイアウトに勝った方が $x$ を陣地にしていたプレイアウト回数
$b(x)$ : 黒が $x$ を陣地にしていたプレイアウト回数
$w(x)$ : 白が $x$ を陣地にしていたプレイアウト回数

である. Criticality は Ownership と同様に各座標がどの程度重要であるかを表す指標になる. Criticality の値が大きいほど, 重要度が高く, Criticality の値が小さいほど, 重要度が低いと言える.

# 4　囲碁プログラム Ray

ここでは自作の囲碁プログラム Ray について説明する. Ray は次の特徴を持っている.

- Latent Factor Ranking アルゴリズムの学習結果を木探索部の着手評価に利用
- Minorization Maximization アルゴリズムを利用してシミュレーションでの着手確率を計算
- UCB 値と行動評価関数を組み合わせた指標を用いるモンテカルロ木探索
- Ownership と Criticality を用いた候補手の並べ替え
- Progressive Widening による前向き枝刈り

## 4.1　行動評価関数を組み込んだ UCB 値

囲碁プログラム Ray では, 行動評価関数を組み込んだ UCB 値を

$$u_{\mathrm{Ray}}(i) = \mathrm{UCB1\text{-}TUNED}(i) + C_H\sqrt{\frac{K}{n+K}} \cdot H(i)$$

とし, 各パラメータについては

$$C_H = 0.35$$
$$K = 1000$$

とし, $H(i)$ は Latent Factor Ranking アルゴリズムによって得られる着手のスコアをそのまま用いている. UCB-TUNED1 値の計算において, Ray では勝ちを 1, 敗けを 0 としているため,

$$V_i(s) = \left(\frac{1}{s}\sum_{\tau=1}^{s} X_{i,\tau}^2\right) - \bar{X}_{i,s}^2 + \sqrt{\frac{2\log n}{s}}$$

の第 1 項については,

$$X_{i,\tau}^2 = \begin{cases} 1 & (X_{i,\tau} = 1 \text{ のとき}) \\ 0 & (X_{i,\tau} = 1 \text{ のとき}) \end{cases}$$

であるから, 候補手 $i$ の勝率そのものを表し, 第 2 項については報酬の平均の 2 乗であるから, 候補手 $i$ の勝率の 2 乗を表している. よって, $V_i(s)$ は次のように変形できる :

$$V_i(s) = \bar{x}_i - \bar{x}_i^2 + \sqrt{\frac{2\log n}{s}}\ .$$

## 4.2 Ownership と Criticality

囲碁プログラム Ray では, ノードの探索回数が 128 の倍数になった時に, Ownership, Criticality, Latent Factor Ranking アルゴリズムによる着手評価の 3 つを合わせて探索候補の並び替えをしている.

Ownership は 0 から 1 の値を取りうるが, 図 4 に示すように 11 個の区間に分けて, それぞれに対応するスコアを与えている. Criticality も Onwership と同様に, 図 5 に示すように 7 個の区間に分けて, それぞれにスコアを与えている.

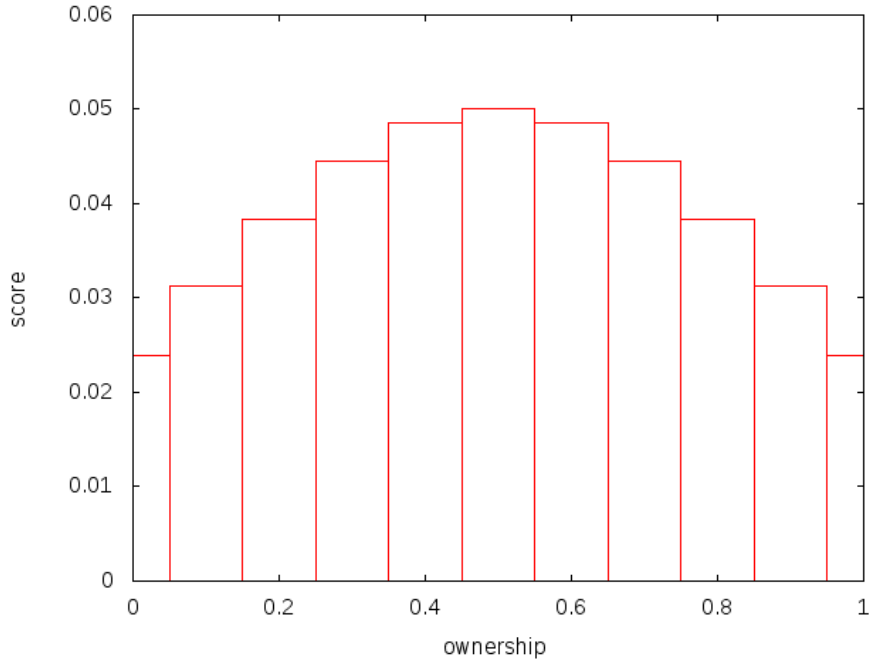


図 4: Ownership と対応するスコア (縦軸 : スコア, 横軸 : Ownership の値)

ノード $i$ の Ownership, Criticality, 着手評価を合わせたスコアを

$$\text{score}(i) = H(i) + S_o(\text{Ownership}(c, p_i)) + S_c(\text{Criticality}(p_i))$$

で求める. ただし, 各パラメータは

$p_i$ : ノード $i$ の着手の座標
$c$ : 現在の手番の色
$H(i)$ : Latent Factor Ranking アルゴリズムによる着手評価値
$S_o$ : 図 4 によって定められる Ownership によるスコア
$S_c$ : 図 5 によって定められる Criticality によるスコア

である. また Ownership と Criticality の値は候補手の順序の並び替えの際に逐次計算している.

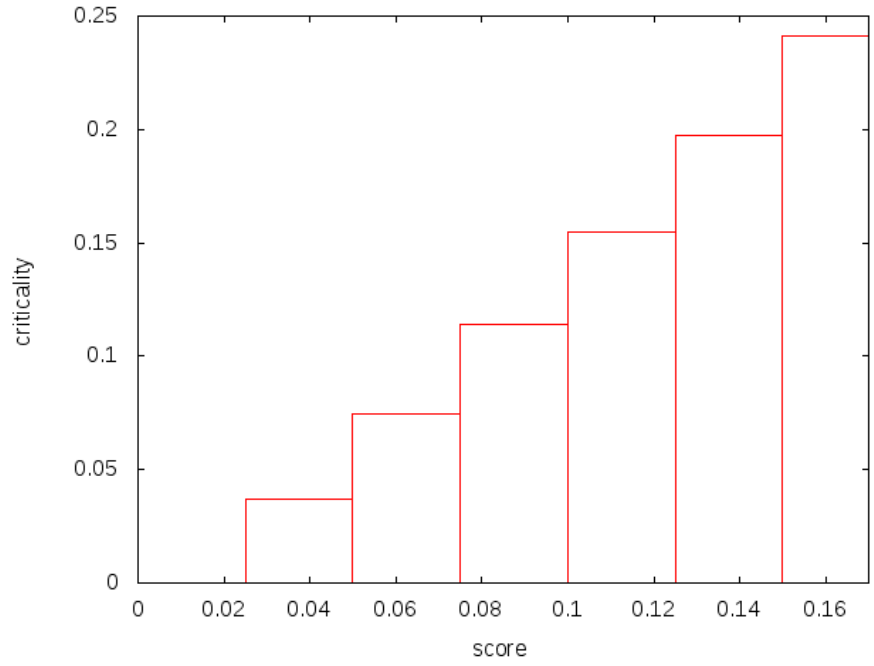


図 5: Criticality と対応するスコア (縦軸：スコア, 横軸：Criticality の値)

## 4.3 碁盤を表現するデータ構造

囲碁プログラム Ray では, 碁盤を表現するデータ構造は以下のもので構成されている.

```
struct game {
    int board[BOARD_MAX] ： 盤上にある石の色を記録する配列
    int moves ：  現在の着手数を記録する変数
    struct string_data strings ：  盤上に存在する連の情報を記録する構造体
}
```

board は盤上のそれぞれの座標を記録する配列であり, 各座標が { 空点, 黒, 白 } のどの状態かを表すものである. 例えば, 図 6 の局面を表すことを考える. 碁盤の座標は $(x, y)$ の 2 次元で表現できるが, ここでは図 7 のように, 左上より 1 から順番に番号を割り振り, 1 次元で表すことにする.

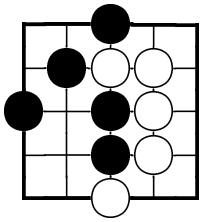

図 6: 局面の一例

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

図 7: 各箇所の座標

空点 = 0, 黒 = 1, 白 = 2 として, 図 6 の局面を表す board の配列のそれぞれの要素が図 8 である.

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 2 | 2 | 0 |
| 1 | 0 | 1 | 2 | 0 |
| 0 | 0 | 1 | 2 | 0 |
| 0 | 0 | 2 | 0 | 0 |

図 8: board の各要素の値

このように記録することで, 座標 $x$ の状態を調べたいときは, board[$x$] を調べることによって, 空点なのか, 黒石があるのか, 白石があるのかを判別することができる.

#### 4.3.1 盤上の連の集合

囲碁プログラムにとって, 碁盤を表現するときに最も重要になるものは連を表現するデータ構造の設計である. 囲碁プログラム Ray では, 連全体で以下の情報を共有している.

```
struct string_data {
    struct string string[STRING_MAX] : 各々の連のデータを保持する構造体の配
    列
    int string_id[BOARD_MAX] : 各座標の連の ID
    int string_next[BOARD_MAX] : 連の構成を記録する配列
}
```

同一の座標に 2 個以上連が存在することはないため, string_id と string_next は連全体で共有している. またそれぞれの連で保持する必要のあるデータについては string に記録している. ここで図 6 の例を用いて, string_id と string_next をどのように表現しているかを説明する.

この時の string_next は図 9 で表されるように, 配列をデータ構造のリストのように扱うことで実装する. ここでの x は, 連の終端を表現しており, この座標以降に, その連を構成する石がないことを表している.

| | | | | |
|---|---|---|---|---|
| 0 | 0 | x | 0 | 0 |
| 0 | x | 9 | 14 | 0 |
| x | 0 | 18 | 19 | 0 |
| 0 | 0 | x | x | 0 |
| 0 | 0 | x | 0 | 0 |

図 9: 各箇所の石の繋がり (string_next)

このように表現することによって, 連の始点の座標さえ分かれば, 石の繋がりをたどることで, 連を構成する石の座標を全て求めることができる. 例えば, 座標 8 を始点とする連を構成する石の座標を求めるときは,

```
string_next[8] = 9

string_next[9] = 14

string_next[14] = 19

string_next[19] = x
```

と順番にたどることで, この連を構成する石の座標は, 8, 9, 14, 19 であることがわかる.

それぞれの連が図 10 のように ID を割り振られていたとすると, 各座標に保持される連の ID は図 11 で表現される. 連の ID が 0 になっている箇所は, 石が存在しないため, 連の ID が割り振られていないことを表している.

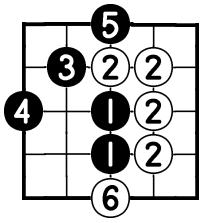


図 10: 連の ID

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 5 | 0 | 0 |
| 0 | 3 | 2 | 2 | 0 |
| 4 | 0 | 1 | 2 | 0 |
| 0 | 0 | 1 | 2 | 0 |
| 0 | 0 | 6 | 0 | 0 |

図 11: 各箇所の連 ID(string_id)

これを用いることによって, 新たに空点に石を置くときに, 上下左右の string_id から隣接する連の ID を求めることによって, 様々な情報を求めることができる. 例えば, 座標 22 の空点の右にある石が ID が 6 であり, そこから石の数が 1 個で, 呼吸点の数が 2 個という情報を取り出すことができるので, ここに黒石を置くとあと 1 手でその石を取れる状態できることがわかる.

それぞれの連が保持しているデータは以下のものである.

```
struct string {
    int origin :  連の始点
    int size :  連を構成する石の個数
    int color :  連を構成する石の色
    int liberty_num :  連が持つ呼吸点の個数
    int liberty_pos :  連が持つ呼吸点の座標
    int neighbor_num :  隣接する敵連の個数
    int neighbor_id :  隣接する敵連の ID

}
```

例 図 10 の連 ID が 2 番である白石の連

この連が持つ情報をまとめると以下のようになる.

1. 連の始点：8
2. 連を構成する石の個数：4 個
3. 連を構成する石の色：白
4. 連が持つ呼吸点の個数：5 個
5. 連が持つ呼吸点の座標：4, 10, 15, 20, 24
6. 隣接する敵の連の個数：3
7. 隣接する敵の連の ID：1, 3, 5

連が持つ呼吸点の座標と隣接する敵連の ID は string_next のように配列をリストに見立てて扱うことで実装する. 連 ID2 番の連が持つ呼吸点の座標は表 1 で表現される. ここで x は呼吸点の終端を表すものである.

表 1: 呼吸点の座標

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| - | 4 | 0 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 0 | 15 | 0 | 0 |
| index | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| - | 0 | 0 | 20 | 0 | 0 | 0 | 0 | 24 | 0 | 0 | 0 | x | 0 |

このように保持することによって, index = 0 から順番にたどることで, string_next と同様に, 全ての呼吸点の座標を求めることができる. さらにある座標 $x$ がその連の呼吸点に

含まれているかどうかは,

$$\texttt{liberty\_pos}[x] \texttt{ != } 0 \Longleftrightarrow x \text{ が呼吸点である}$$
$$\texttt{liberty\_pos}[x] \texttt{ == } 0 \Longleftrightarrow x \text{ が呼吸点ではない}$$

と判定できるので, 高速に判定できる.

隣接する敵連の ID は表 2 で表現される. ここでの x は隣接する敵連の ID の終端を表すものである.

表 2: 隣接する敵連の ID

| id | 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... |
|---|---|---|---|---|---|---|---|---|
| - | 1 | 3 | 0 | 5 | 0 | x | 0 | ... |

これも呼吸点の座標の配列と同様に index = 0 から順番にたどることで, 全ての隣接する敵連の ID を求めることができ, ID が $x$ である敵の連 $s$ がその連に隣接する敵連かどうかは,

$$\texttt{neighbor\_id}[x] \texttt{ != } 0 \Longleftrightarrow \text{隣接する敵連である}$$
$$\texttt{neighbor\_id}[x] \texttt{ == } 0 \Longleftrightarrow \text{隣接する敵連ではない}$$

と判定できるので, 高速に判定できる.

### 4.3.2 石を置く処理

碁盤に石を置くときに, その石が影響を与えるのは上下左右の 4 箇所のみである. 連のデータ構造に依存するものの, 置かれた石の上下左右を確認する処理はほとんどの囲碁プログラムに共通する部分である. 囲碁プログラム Ray の石を置く処理のアルゴリズムを Algorithm 3 に示す.

連を構成する石の座標, 連の持つ呼吸点の座標, 隣接する敵連の ID の 3 つのデータは配列をリストのようにたどる扱い方をする. これらのデータに対するデータの挿入, 削除, 結合, 重複の確認をそれぞれ Algorithm 4 から 7 に示す.

これらを用いることにより, RemoveLiberty 関数, RemoveString 関数, MakeNewString 関数, AddStone 関数, MergeStrings 関数を定義できる.

**Algorithm 3** 石を置く処理

**Require:** 盤上の石の色を記録している配列 board
**Require:** 連の配列 strings
**Require:** 盤上の連の ID string_id
**Require:** 石を置く座標 pos
**Require:** 置く石の色 play_color

```
neighbors ← GetNeighbors(pos) // GetNeighbors 関数は pos の上下左右の座標を求める関数
opponent_color ← FlipColor(play_color) // FlipColor 関数は play_color でない方の色を求める関数
S ← φ
for all neighbor ∈ neighbors do
  if neighbor に石が存在している then
    i ← string_id[neighbor]
    RemoveLiberty(strings[i], pos) // RemoveLiberty 関数は string[i] の呼吸点の集合から pos を取り除く関数
    if board[neighbor] = play_color then
      S ← S ∪ {i}
    else if board[neighbor] = opponent_color then
      l ← strings[i].liberty_num
      if l = 0 then
        RemoveString(strings, i) // RemoveString 関数は strings[i] を盤上から取り除く関数
      end if
    end if
  end if
end for
if | S |= 0 then
  MakeNewString(strings, pos, player_color) // MakeNewString 関数は座標 pos だけで構成される色 player_color の連を作成する関数
else if | S |= 1 then
  AddStone(strings, S, pos) // AddStone 関数は連に pos を加える関数
else if | S |≥ 2 then
  MergeStrings(strings, S, pos) // MergeStrings 関数は 2 つ以上の連と pos を結合させる関数
end if
```

**Algorithm 4** データの挿入処理：InsertListArrayElement 関数

**Require:** リスト状データ配列 list_array
**Require:** リスト上データ配列のデータの始点 origin
**Require:** 挿入するデータ insert_data
**Require:** リスト状データ配列の要素数 size

```
1: if list_array[insert_data] ≠ 0 then
2:     return
3: end if
4: i ← origin
5: while list_array[i] < insert_data do
6:     i ← list_array[i]
7: end while
8: list_array[insert_data] ← list_array[i]
9: list_array[i] ← insert_data
10: size ← size + 1
```

**Algorithm 5** データの削除処理：DeleteListArrayElement 関数

**Require:** リスト状データ配列 list_array
**Require:** リスト上データ配列のデータの始点 origin
**Require:** 削除するデータ delete_data
**Require:** リスト状データ配列の要素数 size

```
1: if list_array[delete_data] = 0 then
2:     return
3: end if
4: i ← origin
5: while list_array[i] ≠ delete_data do
6:     i ← list_array[i]
7: end while
8: n ← list_array[i]
9: list_array[i] ← n
10: list_array[delete_data] ← 0
11: size ← size − 1
```

**Algorithm 6** データの結合処理：MergeListArray 関数

**Require:** 結合先のリスト状データ配列 dst
**Require:** 結合元のリスト状データ配列 src
**Require:** 結合先のリスト状データ配列のデータの始点 dst_origin
**Require:** 結合元のリスト状データ配列のデータの始点 src_origin
**Require:** 結合先のリスト状データ配列の要素数 dst_size
**Require:** データの終端を表す値 end

```
1: s ← src_origin
2: d ← dst_origin
3: n ← 0
4: while s ≠ end do
5:    if dst[s] = 0 then
6:       while dst[d] < s ∧ dst[d] ≠ end do
7:          d ← dst[d]
8:       end while
9:       dst[s] ← dst[d]
10:      dst[d] ← s
11:      n ← n + 1
12:   end if
13:   s ← src[s]
14: end while
15: dst_size ← dst_size + n
```

**Algorithm 7** データの存在確認処理：IsListArrayExist

**Require:** リスト状データ配列 list_array
**Require:** 確認するデータ delete_data
**Ensure:** データの有無を表すフラグ flag

```
1: if list_array[delete_data] ≠ 0 then
2:    flag ← exist
3: else
4:    flag ← not_exist
5: end if
6: return flag
```

**Algorithm 8** RemoveLiberty 関数

**Require:** 呼吸点を取り除く連 string
**Require:** 取り除く呼吸点の座標 pos

```
1: DeleteListArrayElement(string.liberty_pos, 0, string.liberty_num)
```

**Algorithm 9** RemoveString 関数

**Require:** 連の配列 strings
**Require:** 連の ID の配列 string_id
**Require:** 連の座標の配列 string_next
**Require:** 取り除く連の ID id

```
1: p ← strings[id].origin
2: while p ≠ string_end do
3:    neighbors ← GetNeighbors(p)
4:    for all neighbor ∈ neighbors do
5:       i ← string_id[neighbor]
6:       if strings[i].flag ≠ 0 then
7:          InsertListArrayElement(string[i].liberty, 0, p, string[i].liberty_num)
8:       end if
9:    end for
10:   n ← string_next[p]
11:   string_id[p] ← 0
12:   string_next[p] ← 0
13:   p ← n
14: end while
15: i ← strings[id].neighbor_id[0]
16: while i ≠ neighbor_end do
17:   DeleteListArrayElement(strings[i].neighbor_id, id, strings[i].neighbor_num)
18:   i ← string[id].neighbor_id[i]
19: end while
```

**Algorithm 10** MakeNewString 関数

**Require:** 連の配列 strings
**Require:** 新たにできる連の座標 pos
**Require:** 新たにできる連の色 color
**Require:** 連の ID の配列 string_id
**Require:** 連の座標の配列 string_next

```
1: i ← 1
2: while strings[i].flag = true do
3:     i ← i + 1
4: end while
5: strings[i].libery[0] ← liberty_end
6: strings[i].neighbor[0] ← neighbor_end
7: strings[i].liberty_num ← 0
8: strings[i].neighbor_num ← 0
9: strings[i].color ← color
10: strings[i].origin ← pos
11: strings[i].size ← 1
12: strings[i].flag ← true
13: string_id[pos] ← i
14: string_next[pos] ← string_end
15: neighbors ← GetNeighbors(p)
16: for all neighbor ∈ neighbors do
17:     if board[neighbor] が空点 then
18:         InsertListArrayElement(strings[i].liberty, 0, neighbor, strings[i].liberty_num)
19:     else if board[neighbor] が敵の石 then
20:         j ← string_id[neighbor]
21:         InsertListArrayElement(strings[i].neighbor, 0, j, strings[i].neighbor_num)
22:         InsertListArrayElement(strings[j].neighbor, 0, i, strings[j].neighbor_num)
23:     end if
24: end for
```

**Algorithm 11** AddStone 関数

**Require:** 連の配列 strings
**Require:** 石を追加する連の ID id
**Require:** 新たにできる連の座標 pos
**Require:** 新たにできる連の色 color
**Require:** 連の ID の配列 string_id
**Require:** 連の座標の配列 string_next

```
1: if strings[id].origin > pos then
2:     string_next[pos] ← strings[id].origin
3:     strings[id].origin ← pos
4: else
5:     InsertListArrayElement(string_next, strings[id].origin, pos, strings[id].size)
6: end if
7: neighbors ← GetNeighbors(p)
8: for all neighbor ∈ neighbors do
9:     if board[neighbor] が空点 then
10:        InsertListArrayElement(strings[i].liberty, 0, neighbor, strings[i].liberty_num)
11:    else if board[neighbor] が敵の石 then
12:        j ← string_id[neighbor]
13:        InsertListArrayElement(strings[i].neighbor, 0, j, strings[i].neighbor_num)
14:        InsertListArrayElement(strings[j].neighbor, 0, i, strings[j].neighbor_num)
15:    end if
16: end for
```

**Algorithm 12** MergeStrings 関数

**Require:** 連の配列 strings
**Require:** 結合させる連の ID の配列 ids
**Require:** 新たにできる連の座標 pos
**Require:** 新たにできる連の色 color
**Require:** 連の ID の配列 string_id
**Require:** 連の座標の配列 string_next

```
id ← ids[1]
for i = 2 to |ids| do
    MergeListArray(strings[id].liberty, strings[i].liberty, 0, 0, strings[id].liberty_num, liberty_end) // 呼吸点の結合
    if strings[id].origin > strings[i].origin then
        string_next[strings[i].origin] ← strings[id].origin
        strings[id].origin ← strings[i].origin
    end if
    s ← strings[id].origin
    o ← strings[i].origin
    MergeListArray(string_next, string_next, s, o, strings[id].size, string_end) // 連の結合
    MergeListArray(strings[id].neighbor, strings[i].neighbor, 0, 0, strings[id].neighbor_num, neighbor_end) // 隣接する敵連の ID の結合
    n ← strings[i].neighbor[0]
    while n ≠ neighbor_end do
        DeleteListArrayElement(strings[n].neighbor, 0, i, strings[n].neighbor_num)
        n ← strings[i].neighbor[n]
    end while
end for
```

## 4.4 探索木の情報のデータ構造

Ray は探索のために, 探索木の各ノードに以下の情報を持たせている.

```
struct node {
    int move_count :  ノードの探索回数
    int width :  Progressive Widening によって枝刈りされた子ノードの個数
    int child_num :  子ノードの個数
    struct child child[CHILD_MAX] : 子ノードの情報

}
```

子ノードの情報は以下の構造体で表されている.

```
struct child {
    int pos :  着手する座標
    int move_count :  探索回数
    int win :  勝った探索の回数
    struct node *ptr :  子ノードの遷移先を示すポインタ
    double score :  着手の評価値
    bool flag :  Progressive Widening による枝刈りのフラグ

}
```

これらを用いることによって node[$x$] における子ノード $i$ の UCB 値を求められる. UCB 値最大の子ノードを求める SelectMaxUcbChild 関数のアルゴリズムを Algorithm13 に示す.

各ノードの情報をトランスポジションテーブルを利用して, 合流を検知することによって, 探索の効率化を図っている. 例えば, 図 12 の局面を考えると, この局面に至る応手列は図 13 から図 16 の 4 つである. このように囲碁の局面は容易に合流し得るものであるので, 石の配置が同じ局面を同一のものと見なすことは, 探索の効率化につながる.

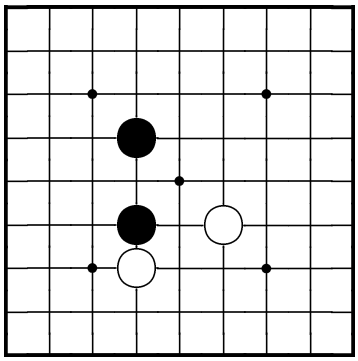

図 12: 初期局面から 4 手進めた局面

**Algorithm 13** SelectMaxUcbChild 関数

**Require:** 現局面のノード node

**Ensure:** UCB 値最大の子ノードのインデックス $n$

$n \leftarrow 0$
$r \leftarrow -\infty$
**for** $i = 1$ to node.child_num **do**
  **if** node.child[$i$].flag == true **then**
    $w \leftarrow \dfrac{\text{node.child}[i].\text{win}}{\text{node.child}[i].\text{move\_count}}$
    $v \leftarrow w - w^2 + \sqrt{\dfrac{2\log(\text{node.move\_count})}{\text{node.child}[i].\text{move\_count}}}$
    $u \leftarrow w + \sqrt{\dfrac{\log(\text{node.move\_count})}{\text{node.child}[i].\text{move\_count}}\min(\frac{1}{4}, v)}$
    $s \leftarrow u + C_H \cdot \text{node.child[i].score}$
    **if** $s > r$ **then**
      $n \leftarrow i$
      $r \leftarrow s$
    **end if**
  **end if**
**end for**
**return** $n$

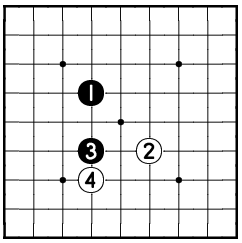


図 13: 応手列 1

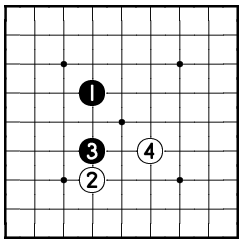


図 14: 応手列 2

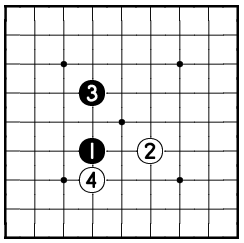


図 15: 応手列 3

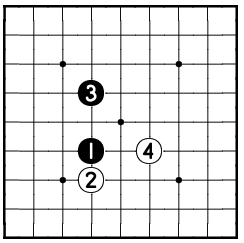


図 16: 応手列 4

局面が同一のものであるかを計算するために Zobrist Hash[6] を用いて, 各局面にハッシュ値を与えている. 局面のハッシュ値を求めるアルゴリズムを Algorithm14 に示す. ここでの random_bit はプログラム起動時に乱数生成器を用いて生成したビット列の配列であり, board は碁盤上の各座標の情報を保持している配列である. 探索木に新たにノードを追加する際に, このアルゴリズムで求めたハッシュ値と手番情報をキーとする. 同一のハッシュ値と手番情報を持つ局面情報が見つかれば, 新たに展開しようとしているノードをそのノードに合流させ, 見つからなければ, トランスポジションテーブルに新しくノードを登録する.

**Algorithm 14** 局面のハッシュ値の計算

**Require:** 碁盤の各座標の情報 board
**Require:** 各座標に割り当てられたビット列 random_bit
**Ensure:** 局面のハッシュ値 hash

```
hash ← 0
for all b ∈ board do
    c ← GetColor(b) // GetColor 関数は石の色を返す関数
    p ← GetPosition(b) // GetPosition 関数は座標を返す関数
    hash ← hash ⊕ random_bit[p][c]
end for
return hash
```

トランスポジションテーブルで実装することによって, 探索結果が木ではなくグラフで構成されることになる. ここで注意すべきことが 2 つある. 1 つ目は劫の取り扱い, 2 つ目はパスの取り扱いである.

まず劫の取り扱いについて説明する. 直前の手で劫が発生すると, その劫を取り返す手が非合法手になり, 着手することができなくなるが, その局面にある石の色だけでは, 劫が発生しているかどうかはわからない. 例えば, 図 17 を考えると, 図 18 の 5 手の応手の後に, 白が×の箇所に着手することはできるが , 図 19 の 5 手の応手の後に, 白が×の箇所に着手することはできない. これらの局面を合流させてしまうと, 読みの中に非合法手が含まれ, 探索が正常に行えないばかりか, 以前の探索結果を再利用した場合に非合法手に着手する可能性もある. この問題を避けるために, 劫が発生している箇所を識別できるよう

に, 局面を表すハッシュ値に劫が発生している座標を追加する. このことによって, 直前の着手で劫が発生したか否かを判別し, 劫の発生した局面とそうでない局面を別のものと認識することができる.

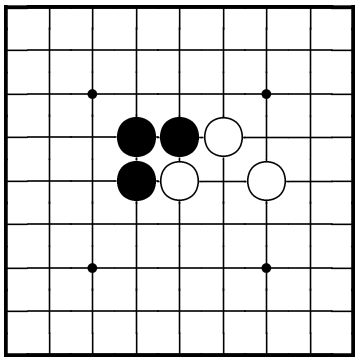

図 17: 探索局面 1

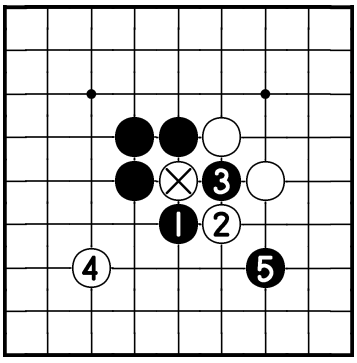


図 18: 白の着手

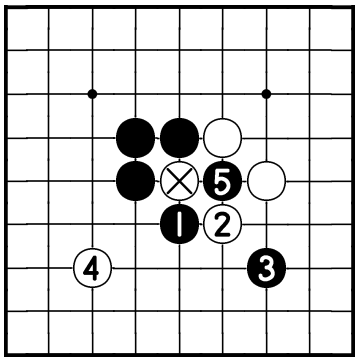


図 19: 黒の着手

次にパスの取り扱いについて説明する. 図 20 の局面に対する, 白番の応手を探索しようとする. 図 21 から図 24 の 4 手の応手の後の局面を考えると, これは図 20 に一致する. どちらの局面も出番とハッシュ値は同じなので, 図 23 の局面から図 24 の局面を展開すると, 図 20 の局面を表すノードに合流してしまう. このことにより, 木探索部の候補手を選んでいく時に末端ノードに到達することができずに, 無限ループに陥ってしまう. この現象を避けるために, 局面のハッシュ値と手番の情報とともに, 着手数もトランスポジションテーブルのキーとして導入する. 着手数もキーに含むことで, 図 24 は図 20 から 4 手進んだ局面であることから, これらの局面は異なるものと認識し, 無限ループを回避できる.

従って, ハッシュテーブルに登録する情報は以下のようになる.

```
struct hash_table {
    unsigned long long int hash :  局面のハッシュ値
```

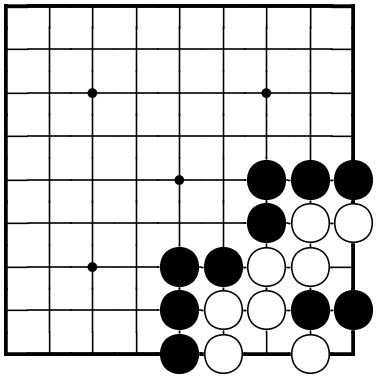

図 20: 探索局面 2

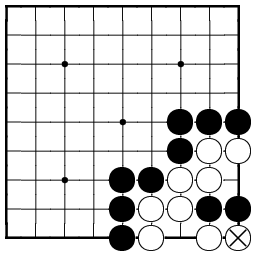

図 21: 白の着手

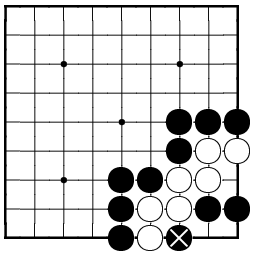

図 22: 黒の着手

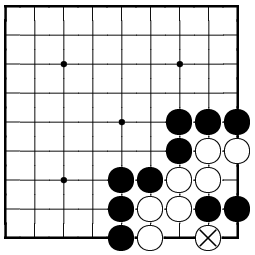

図 23: 白の着手

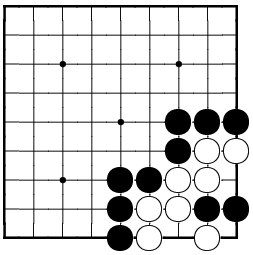

図 24: 黒のパス

```
    int color :  登録された局面での手番
    int moves :  登録された局面での着手数
    int index :  局面の登録先のインデックス
```

}

## 4.5 着手評価に用いる特徴

ここでは囲碁プログラム Ray に用いている着手評価の指標となる特徴を説明する. 囲碁プログラム Ray では木探索部とシミュレーション部で異なる特徴の使い方をしている. 木探索部の着手評価はノード展開の時にしか行われないので, 計算コストのかかる特徴を使っても探索速度があまり低下しないのに対し, シミュレーション部の着手評価はシミュレーションの 1 回の着手ごとに計算する必要があるので, 計算コストのかかる特徴を使うと探索速度が低下するからである.

### 4.5.1 木探索部に用いる特徴

木探索部の着手評価に用いている特徴は以下のものが挙げられる.

- 直前の着手からの距離
- 2 手前の着手からの距離
- 盤上の位置
- 局所的な配石パターン
- 戦術的特徴

着手からの距離は式 16 で定義する.

$$d(dx, dy) = |dx| + |dy| + max(|dx|, |dy|) \tag{16}$$

$dx : x$ 方向の距離
$dy : y$ 方向の距離

着手距離の例を図 25 に示す. 各座標の数字は点 x からの距離を表している. 特徴として用いる距離は $d = 0, 2, 3, ..., 14, 15$ 以上 の 15 個である.

盤上の位置の識別番号を表 3 に示す. 回転や対称を考慮した際に同じ位置になる箇所は同じ番号としている. 盤上の位置の特徴は表 3 より, 55 個である.

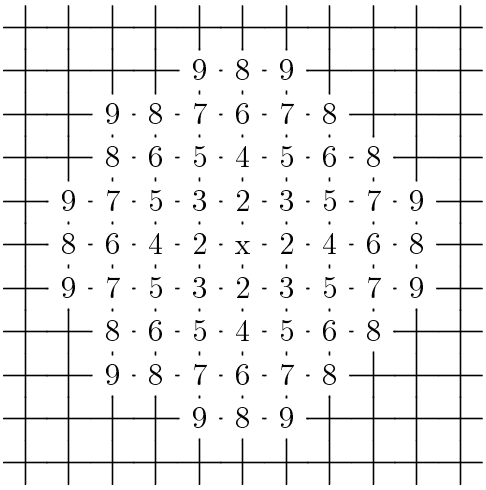


図 25: 着手距離の例

表 3: 盤上の位置の識別番号

| | A | B | C | D | E | F | G | H | J | K | L | M | N | O | P | Q | R | S | T |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 19 | 1 | 2 | 4 | 7 | 11 | 16 | 22 | 29 | 37 | 46 | 37 | 29 | 22 | 16 | 11 | 7 | 4 | 2 | 1 |
| 18 | 2 | 3 | 5 | 8 | 12 | 17 | 23 | 30 | 38 | 47 | 38 | 30 | 23 | 17 | 12 | 8 | 5 | 3 | 2 |
| 17 | 4 | 5 | 6 | 9 | 13 | 18 | 24 | 31 | 39 | 48 | 39 | 31 | 24 | 18 | 13 | 9 | 6 | 5 | 4 |
| 16 | 7 | 8 | 9 | 10 | 14 | 19 | 25 | 32 | 40 | 49 | 40 | 32 | 25 | 19 | 14 | 10 | 9 | 8 | 7 |
| 15 | 11 | 12 | 13 | 14 | 15 | 20 | 26 | 33 | 41 | 50 | 41 | 33 | 26 | 20 | 15 | 14 | 13 | 12 | 11 |
| 14 | 16 | 17 | 18 | 19 | 20 | 21 | 27 | 34 | 42 | 51 | 42 | 34 | 27 | 21 | 20 | 19 | 18 | 17 | 16 |
| 13 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 35 | 43 | 52 | 43 | 35 | 28 | 27 | 26 | 25 | 24 | 23 | 22 |
| 12 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 44 | 53 | 44 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 |
| 11 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 54 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 |
| 10 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 54 | 53 | 52 | 51 | 50 | 49 | 48 | 47 | 46 |
| 9 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 54 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 |
| 8 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 44 | 53 | 44 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 |
| 7 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 35 | 43 | 52 | 43 | 35 | 28 | 27 | 26 | 25 | 24 | 23 | 22 |
| 6 | 16 | 17 | 18 | 19 | 20 | 21 | 27 | 34 | 42 | 51 | 42 | 34 | 27 | 21 | 20 | 19 | 18 | 17 | 16 |
| 5 | 11 | 12 | 13 | 14 | 15 | 20 | 26 | 33 | 41 | 50 | 41 | 33 | 26 | 20 | 15 | 14 | 13 | 12 | 11 |
| 4 | 7 | 8 | 9 | 10 | 14 | 19 | 25 | 32 | 40 | 49 | 40 | 32 | 25 | 19 | 14 | 10 | 9 | 8 | 7 |
| 3 | 4 | 5 | 6 | 9 | 13 | 18 | 24 | 31 | 39 | 48 | 39 | 31 | 24 | 18 | 13 | 9 | 6 | 5 | 4 |
| 2 | 2 | 3 | 5 | 8 | 12 | 17 | 23 | 30 | 38 | 47 | 38 | 30 | 23 | 17 | 12 | 8 | 5 | 3 | 2 |
| 1 | 1 | 2 | 4 | 7 | 11 | 16 | 22 | 29 | 37 | 46 | 37 | 29 | 22 | 16 | 11 | 7 | 4 | 2 | 1 |

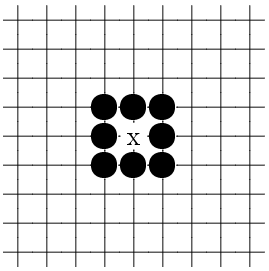

図 26: 3x3 パターン

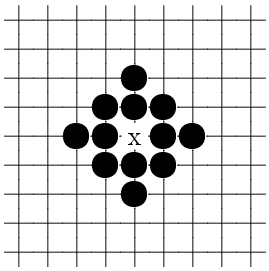

図 27: MD2 パターン

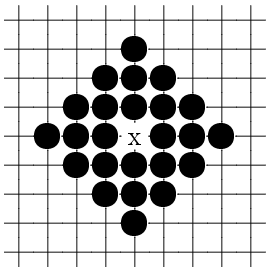

図 28: MD3 のパターン

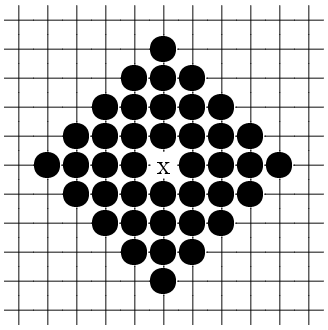

図 29: MD4 パターン

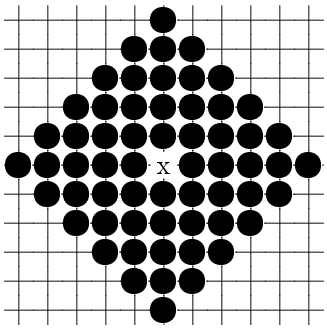

図 30: MD5 パターン

局所的な配石パターンは 3x3 の 8 近傍のパターンと, 式 17 で定義されるマンハッタン距離 2 から 5 までパターンの 5 種類の大きさのパターンからなり, それらを図 26 から図 5 に示す. MD$n$ はマンハッタン距離 $n$ のパターンを表す.

$$md(dx, dy) = |dx| + |dy| \tag{17}$$

パターンを着手評価に用いる際は, 大きいパターンから順番にマッチングを行い, 最初にマッチングしたものをその着手の持つ配石パターンの特徴としている. パターンの実装方法には以下の 2 つの方法がある.

1. ビット列による実装方法
   1 つの座標に着目すると, 取りうる状態は { 空点, 黒石, 白石, 盤外 } の 4 つであり, それぞれに {00, 01, 10, 11} と 2 ビットで表現することができる. したがって, 各パターンの考慮する点の数は 3x3 が 8 箇所, MD2 が 12 箇所, MD3 が 24 箇所, MD4 が 40 箇所, MD5 が 60 箇所となり, 3x3 は 16 ビット, MD2 は 24 ビット, MD3 は 48 ビット, MD4 は 80 ビット, MD5 は 120 ビットで表現できる. この方法はビット列をそのままインデックスとして扱うことで, パターンを 1 対 1 対応させることができるので高速に実行できるが, ビット列の長さを $l$ とすると, パターンのスコアを格納するのに必要な配列の個数は $2^l$ 個必要になるので, パターンのサイズが大きくなるとビット列の長さも長くなり, 配列のサイズも爆発的に増えてしまうという欠点がある. そのため, 囲碁プログラム Ray ではこの手法は 3x3 パターンと MD2 パターンにのみ用いている. 例えば, 図 31 は表 1 と表現できる.

図 31: 3x3 パターンの例

表 4: パターンのビット列

| | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|
| ビット列 | 10 | 01 | 00 | 00 | 00 | 01 | 00 | 00 |

表 5: 各点のビット表現

| 状態 | ビット表現 | 10 進表現 |
|---|---|---|
| 空点 | 00 | 0 |
| 黒石 | 01 | 1 |
| 白石 | 10 | 2 |
| 盤外 | 11 | 3 |

2. ハッシュ値による実装手法
   パターンを Zobrist Hash [6] を用いて表現する．各点の状態に乱数を割り当て, それらすべての排他的論理和を取ることで表現する．例えば, 図 31 の 3x3 パターンを Zobrist Hash で表現すると, あらかじめ計算して乱数を割り当てた配列

   $$\text{unsigned int random\_bit}[8][4]$$

   を用いて

   $$\begin{aligned}\text{hash} = {} & \text{random\_bit}[\text{点 1}][0] \ \hat{}\ \text{random\_bit}[\text{点 2}][0] \\ & \hat{}\ \text{random\_bit}[\text{点 3}][1] \ \hat{}\ \text{random\_bit}[\text{点 4}][0] \\ & \hat{}\ \text{random\_bit}[\text{点 5}][0] \ \hat{}\ \text{random\_bit}[\text{点 6}][0] \\ & \hat{}\ \text{random\_bit}[\text{点 7}][1] \ \hat{}\ \text{random\_bit}[\text{点 8}][2]\end{aligned}$$

   で得られる．この手法はビット列による手法と異なり, ビット長が固定されているという利点がある一方で, パターンのスコアを求める際にハッシュ値に対応するパターンを探す必要があるという欠点がある．囲碁プログラム Ray では MD3 パターン, MD4 パターン, MD5 パターンに対して, この手法を利用している．

戦術的特徴は以下のものを用いている．

1. 直前の着手で呼吸点が 1 つになった自分の連に隣接する敵連を取る手
   自分の連の石の個数, 相手の連の石の個数, 着手の後に自己アタリになるかどうかで以下の 10 種類の特徴に分けている．

   (a) 自分の石 1 つの連に隣接する相手の石 1 つの連を取る手

   (b) 自分の石 1 つの連に隣接する相手の石 2 つの連を取る手

(c) 自分の石 1 つの連に隣接する相手の石 3 つ以上の連を取る手

(d) 自分の石 2 つの連に隣接する相手の石 1 つの連を取る手

(e) 自分の石 2 つの連に隣接する相手の石 2 つの連を取る手

(f) 自分の石 2 つの連に隣接する相手の石 3 つ以上の連を取る手

(g) 自分の石 3 つ以上の連に隣接する相手の石 1 つの連を取る手

(h) 自分の石 3 つ以上の連に隣接する相手の石 2 つの連を取る手

(i) 自分の石 3 つ以上の連に隣接する相手の石 3 つ以上の連を取る手

(j) 自分の連に隣接する相手の連を取った結果, 自己アタリになる手

2. 直前の着手で呼吸点が 2 つになった自分の連に隣接する敵連を取る手
自分の石の連の個数, 相手の石の連の個数によって以下の 4 種類の特徴に分けている.

(a) 自分の石 2 個以下の連に隣接する相手の石 2 個以下の連を取る手

(b) 自分の石 2 個以下の連に隣接する相手の石 3 個以上の連を取る手

(c) 自分の石 3 個以上の連に隣接する相手の石 2 個以下の連を取る手

(d) 自分の石 3 個以上の連に隣接する相手の石 3 個以上の連を取る手

3. 直前の着手で呼吸点が 3 つになった自分の連に隣接する敵連を取る手
自分の石の連の個数, 相手の石の連の個数によって以下の 4 種類の特徴に分けている.

(a) 自分の石 2 個以下の連に隣接する相手の石 2 個以下の連を取る手

(b) 自分の石 2 個以下の連に隣接する相手の石 3 個以上の連を取る手

(c) 自分の石 3 個以上の連に隣接する相手の石 2 個以下の連を取る手

(d) 自分の石 3 個以上の連に隣接する相手の石 3 個以上の連を取る手

4. 敵の連を取って劫を解消する手

5. 呼吸点が 1 つの自分の連に隣接する敵連を取る手

6. その他の石を取る手
図 32 から 2 手進んだ局面である図 33 を考えると, ×印の石の呼吸点が 1 つ, ○印の連の呼吸点が 2 つ, □印の連の呼吸点が 3 つになっているので, 点 a に 1 と 2 の手, 点 b に 2 の手, 点 c に 3 の手, 点 d に 4 の手, 点 e に 5 の手, 点 f に 6 の手の特徴が表れている.

7. 自己アタリ
自己アタリは以下の 3 つの特徴に分けている.

(a) 取られる石が 2 個以下の自己アタリ

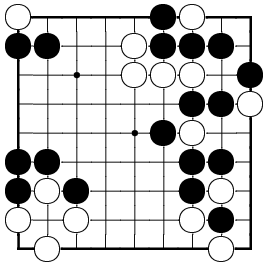

図 32: 局面の一例

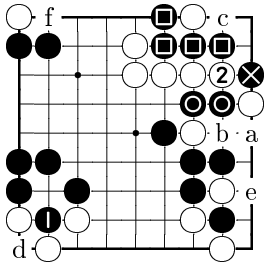


図 33: 図 32 から 2 手進めた局面

(b) 取られる連の形がナカデになる自己アタリ

(c) その他の自己アタリ

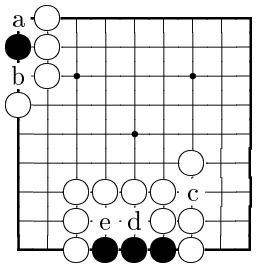


図 34: 自己アタリの例

図 34 の点 a, b, c が「取られる石が 2 個以下の自己アタリ」, 点 d が「取られる連の形がナカデになる自己アタリ」, 点 e が「その他の自己アタリ」に該当する.

8. 直前の着手で呼吸点が 1 つになった自分の連の呼吸点に打つ手自分の連の石の数とシチョウで取られるかどうかで以下の 4 種類の特徴に分けている.

   (a) シチョウで取られる自分の連の呼吸点に打つ手

   (b) 自分の石 1 つの連の呼吸点に打つ手

   (c) 自分の石 2 つの連の呼吸点に打つ手

   (d) 自分の石 3 つの連の呼吸点に打つ手

9. 直前の着手で呼吸点が 2 つになった自分の連に隣接する敵連をアタリにする手自分の連の石の数, 相手の連の石の数, 相手の石が 1 手で取れるかどうかで特徴を分けている.

   (a) 自分の石 2 個以下の連に隣接する相手の石 2 個以下の連をアタリにする手

   (b) 自分の石 2 個以下の連に隣接する相手の石 3 個以上の連をアタリにする手

(c) 自分の石 3 個以上の連に隣接する相手の石 2 個以下の連をアタリにする手

(d) 自分の石 3 個以上の連に隣接する相手の石 3 個以上の連をアタリにする手

この 4 種類に対して, 相手が逃げても 1 手で取れるかどうかで合計 8 種類の特徴に分けている. 例えば, 図 35 の白 1 の着手によって, ×の黒石の呼吸点が 2 つになった状況を考えると, ×の黒石の石の個数は 2 個, □の白石の石の個数が 2 個なので, 点 a と点 b に (a) の特徴が現れる. さらに, 点 a からアタリにすると相手が点 b に逃げても 1 手で取れる一方で, 点 b からアタリにすると 1 手で取ることができないので, 点 a は 1 手で取れるアタリ, 点 b はそうでないアタリと特徴を判別する.

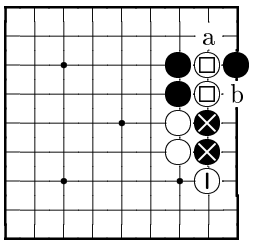


図 35: 局面の一例

10. 直前の着手で呼吸点が 3 つになった自分の連に隣接する敵連をアタリにする手自分の連の石の数, 相手の連の石の数, 相手の石が 1 手で取れるかどうかで分けている.

    (a) 自分の石 2 個以下の連に隣接する相手の石 2 個以下の連をアタリにする手

    (b) 自分の石 2 個以下の連に隣接する相手の石 3 個以上の連をアタリにする手

    (c) 自分の石 3 個以上の連に隣接する相手の石 2 個以下の連をアタリにする手

    (d) 自分の石 3 個以上の連に隣接する相手の石 3 個以上の連をアタリにする手

    この 4 種類に対して, 相手が逃げても 1 手で取れるかどうかで合計 8 種類の特徴に分けている.

11. オイオトシ

12. ウッテガエシ

13. 1 手で取れるアタリにする手

14. その他のアタリにする手

15. 直前の着手で呼吸点が 2 つになった自分の連の呼吸点に打つ手呼吸点に打った結果, 呼吸点の個数がどのように変化するかで以下の 3 つの特徴に分けている.

    (a) 呼吸点に打った結果, 呼吸点の個数が打つ前から増えない.

(b) 呼吸点に打った結果, 呼吸点の個数が打つ前から 1 つだけ増える.

(c) 呼吸点に打った結果, 呼吸点の個数が打つ前から 2 つ以上増える.

16. 直前の着手で呼吸点が 3 つになった自分の連に隣接する呼吸点が 3 つの敵連の呼吸点に打つ手自分の連の石の個数と相手の石の連の個数で以下の 4 つの特徴に分けている.

(a) 自分の石 2 個以下の連に隣接する相手の石 2 個以下の連の呼吸点に打つ手

(b) 自分の石 2 個以下の連に隣接する相手の石 3 個以上の連の呼吸点に打つ手

(c) 自分の石 3 個以上の連に隣接する相手の石 2 個以下の連の呼吸点に打つ手

(d) 自分の石 3 個以上の連に隣接する相手の石 3 個以上の連の呼吸点に打つ手

17. 直前の着手で呼吸点が 3 つになった自分の連の呼吸点に打つ手呼吸点に打った結果, 呼吸点の個数がどのように変化するかで以下の 3 つの特徴に分けている.

(a) 呼吸点に打った結果, 呼吸点の個数が打つ前から増えない.

(b) 呼吸点に打った結果, 呼吸点の個数が打つ前から 1 つだけ増える.

(c) 呼吸点に打った結果, 呼吸点の個数が打つ前から 2 つ以上増える.

18. 2 目の抜き跡を欠け眼にするホウリコミ
図 36 の×の黒石を取った後に , 図 37 の□の黒石を打つと, □の黒石は即座に取り返されるが, 白は 2 眼を確保できなくなる. このように相手の眼形を奪う手は重要な特徴である.

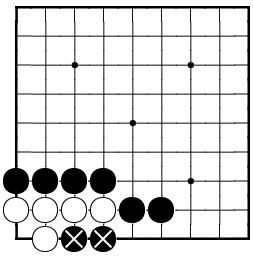

図 36: ホウリコミの一例

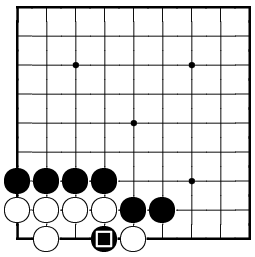

図 37: 欠け目にするホウリコミ

19. 劫を解消するツギ
図 38 の×の白石を取った後に , 図 39 の点 x に打つような特徴.

### 4.5.2 シミュレーション部に用いる特徴

シミュレーション部は 1 回の着手ごとに確率テーブルを更新する必要がある. 具体的には以下にあげるものに該当する箇所の確率テーブルを再計算している.

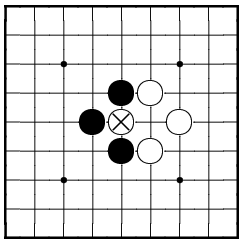

図 38: 劫が発生する前の局面

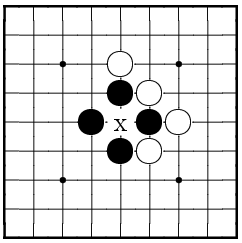


図 39: 劫が発生した 2 手後の局面

1. 直前の着手の周囲 (配石パターンの及ぶ範囲)
2. 直前の着手で戦術的特徴が変化した箇所
3. 直前の着手で石が取り除かれた箇所

シミュレーション部には以下の特徴を用いている.

- 直前からの着手距離
- MD2 パターン
- 戦術的特徴

局所的な配石パターンにはハッシュ値を計算して, 対応するパターンを導出するコストがかかるため, MD3 以上のパターンは使用せず, MD2 パターンを用いることにしている.

直前からの着手距離は $d = 2, 3, 4$ のみを使用している. これは図 40 に示すように, MD2 パターンの範囲内に収まる着手距離であるからである. また直前の着手で置かれた石は必ず盤上に存在するので $d = 0$ は考慮する必要がない.

戦術的特徴は以下のものを用いている.

1. 直前の着手で呼吸点が 1 つになった自分の連に隣接する敵連を取る手
   木探索部と同じ 10 種類の特徴に分けて使用.
2. 直前の着手で呼吸点が 2 つになった自分の連に隣接する敵連を取る手
   相手の石の数が 2 個以下か 3 個以上かの 2 つの特徴に分けて使用.
3. 直前の着手で呼吸点が 3 つになった自分の連に隣接する敵連を取る手
   相手の石の数が 2 個以下か 3 個以上かの 2 つの特徴に分けて使用.
4. 敵の連を取って劫を解消する手
   木探索部と同じ特徴.
5. その他の石を取る手

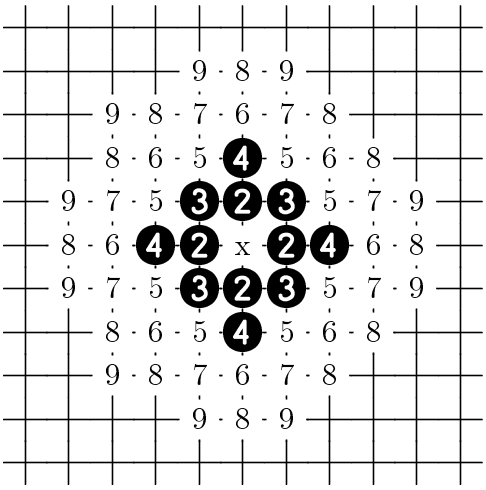


図 40: 着手距離と MD2 パターンの関係

6. 自己アタリ
   木探索部と同じ 3 種類の特徴に分けて使用.

7. 直前の着手で呼吸点が 1 つになった自分の連の呼吸点に打つ手
   自分の連の石の数が 1 個, 2 個, 3 個以上の 3 つに特徴を分けて使用.

8. 直前の着手で呼吸点が 2 つになった自分の連に隣接する敵連をアタリにする手
   相手の石の個数が 2 個以下か 3 個以上かの 2 つの特徴に分けて使用.

9. 直前の着手で呼吸点が 3 つになった自分の連に隣接する敵連をアタリにする手
   相手の石の個数が 2 個以下か 3 個以上かの 2 つの特徴に分けて使用.

10. その他のアタリにする手

11. 直前の着手で呼吸点が 2 つになった自分の連の呼吸点に打つ手

12. 直前の着手で呼吸点が 3 つになった自分の連に隣接する呼吸点が 3 つの敵連の呼吸点に打つ手
    相手の石の個数が 2 個以下か 3 個以上かの 2 つの特徴に分けて使用.

13. 直前の着手で呼吸点が 3 つになった自分の連の呼吸点に打つ手

14. 2 目の抜き跡を欠け眼にするホウリコミ
    木探索部と同一の特徴.

# 5　実験

囲碁プログラム Ray に用いた手法の有用性と棋力を測定するために, 対局実験を行った. 木探索部の学習, シミュレーション部の学習ともに, KGS 高段者の対局のうち, 置石のない棋譜を用いた. 特に断りがない限り, 木探索部の着手評価は Latent Factor Ranking アルゴリズムを使い, 70000 棋譜を学習したものを利用し, シミュレーション部の着手評価は Minorization Maximization アルゴリズムを使い, 10000 局を学習したものを利用した.

考察中で, 「有意に強くなった」や「有意な差は見られなかった」というのは付録 B に示す手順で検定した結果から言及している.

## 5.1　実験の環境

実験は全て以下の環境で行った.

- OS : Cent OS 6.5
- CPU : Intel Xeon E5-2687W v2 2 個
- Memory : 64GB

## 5.2　対局実験 1

木探索部に用いる着手評価として, Latent Factor Ranking アルゴリズムと Minorization Maxmization アルゴリズムのどちらがより棋力を高めるかを測定した. Latent Factor Ranking アルゴリズムで 10000 棋譜学習した結果を用いた Ray と Minorization Maximization アルゴリズムで 10000 棋譜学習した結果を用いた Ray を対局させる.

### 5.2.1　対局の設定

1 手あたりのプレイアウト回数を固定して対局を行った. 木探索部の学習には置石のない KGS 高段者の棋譜 10000 局を用い, MD2, MD3, MD4, MD5 は学習する棋譜の中で 10 回以上打たれたものを使用した.

- 中国ルール
- 9 路盤, 13 路盤, 19 路盤
- コミ 6.5 目
- 白黒交互に入れ替えて 1000 局
- 各盤の大きさに対して, 1 手 1000PO, 1 手 2000PO, 1 手 4000PO, 1 手 8000PO
- 1 スレッドのみ使用

### 5.2.2 対局結果

対局結果を表6に示す. 括弧内は対局結果が正規分布に従うと近似したときの両裾2.5%点を表す (＋が右裾 2.5%点, - が左裾 2.5%点).

表 6: 対局結果 (Latent Factor Ranking アルゴリズムを用いた Ray から見た勝率)

| 1 手あたりの PO 回数 | 1000PO | 2000PO | 4000PO | 8000PO |
|---|---|---|---|---|
| 9x9 | 59.4%(±3.0%) | 52.3%(±3.1%) | 50.3%(±3.1%) | 51.8%(±3.1%) |
| 13x13 | 51.0%(±3.1%) | 56.4%(±3.1%) | 54.5%(±3.1%) | 57.7%(±3.1%) |
| 19x19 | 48.5%(±3.1%) | 49.1%(±3.1%) | 53.6%(±3.1%) | 55.8%(±3.1%) |

### 5.2.3 考察

表6を見ると, 9路盤の1000PO, 13路盤の1000PO, 2000PO, 4000PO, 19路盤の4000PO, 8000PO において, Latent Factor Ranking アルゴリズムを用いた Ray が Minorization-Maximization アルゴリズムを用いた Ray に有意に勝ち越している. 一方, Minorization-Maximization アルゴリズムを用いた Ray が Latent Factor Ranking アルゴリズムを用いた Ray に有意に勝ち越している結果はない. このことから, 木探索部の着手評価評価関数としては, Latent Factor Ranking アルゴリズムが Minorization-Maximization アルゴリズムより優れていると言える.

## 5.3 対局実験 2

シミュレーション部に用いる配石パターンの大きさがどの程度棋力に影響を与えるかを測定した.

### 5.3.1 対局の設定 1

GNU Go を用いて, 1 手あたりのプレイアウト回数を固定して対局を行った. 対局の設定は以下の通りである.

- 対局相手 : GNU Go 3.8 Level 10
- 中国ルール
- 9 路, 13 路, 19 路
- コミ 6.5 目

- 黒白交互に入れ替えて 1000 局
- 9 路 1 手 800PO, 13 路 1 手 3000PO, 19 路 1 手 15000PO
- Ray は 1 スレッドのみ使用

学習する着手距離は 3x3 パターンは $d = 2, 3$ の 2 種類, MD2 パターンは $d = 2, 3, 4$ の 3 種類の特徴を学習した. これはパターンの範囲内に着手距離の特徴が収まるようにしたためである.

#### 5.3.2 対局結果 1

対局結果を表 7 に示す. 括弧内は対局結果が正規分布に従うと近似したときの両裾 2.5%点を表す (＋が右裾 2.5%点, - が左裾 2.5%点).

表 7: 対局結果 (対局相手 GNU Go ver 3.8 Level 10)

| | 3x3 | MD2 |
|---|---|---|
| 9x9 | 53.7%(±3.1%) | 52.7%(±3.1%) |
| 13x13 | 48.8%(±3.1%) | 56.4%(±3.1%) |
| 19x19 | 43.5%(±3.1%) | 65.0%(±3.0%) |

#### 5.3.3 対局の設定 2

パターンの大きさによって, シミュレーションの速度が変化するので, 探索速度を考慮した棋力を比較するために, 3x3 パターンを用いた Ray と MD2 パターンを用いた Ray の対局を行った. 対局の設定は以下の通りである.

- 中国ルール
- 9 路盤, 13 路盤, 19 路盤
- コミ 6.5 目
- 白黒交互に入れ替えて 1000 局
- 各盤の大きさに対して, 1 手 1 秒, 1 手 2 秒, 1 手 4 秒の思考時間
- 1 スレッドのみ使用

### 5.3.4 対局結果 2

対局結果を表 8 に示す. 勝率は MD2 パターンを用いた Ray から見たものである. 括弧内は対局結果が正規分布に従うと近似したときの両裾 2.5%点を表す (＋が右裾 2.5%点, -が左裾 2.5%点). また表 9 に各パターンの単位時間あたりのプレイアウト回数を示す.

表 8: 対局結果

| 1 手あたりの思考時間 | 1 秒 | 2 秒 | 4 秒 |
|---|---|---|---|
| 9x9 | 42.7%(±3.1%) | 41.9%(±3.1%) | 41.5%(±3.1%) |
| 13x13 | 63.5%(±3.0%) | 61.2%(±3.0%) | 57.4%(±3.1%) |
| 19x19 | 82.0%(±2.4%) | 82.2%(±2.4%) | 84.7%(±2.2%) |

表 9: 単位時間あたりのプレイアウト回数 (PO/sec)

| パターンの大きさ | 3x3 | MD2 |
|---|---|---|
| 9x9 | 8500 | 7000 |
| 13x13 | 4250 | 3500 |
| 19x19 | 2020 | 1670 |

### 5.3.5 考察

表 7 からわかる通り, プレイアウト回数を固定すると, 碁盤の大きさが大きいほど, 3x3 よりも MD2 の方が強くなることがわかる. これは碁盤がの大きさが大きいほど, 盤上の様々な場所に石が点在し, それらの死活を正確に判断できないと, なかなか深く読むことができないからだと考えられる. 9 路盤ではほとんどの場合, 1 箇所の死活がそのまま勝敗に直結するので, シミュレーションの質よりも読みの深さが棋力に影響すると言える, 一方で, 13 路盤や 19 路盤では読みに影響されない箇所での死活をきちんとシミュレーションしないと読みの評価が不正確になるので, シミュレーションの質が重要になると言えよう.

表 8 を見ると, プレイアウト回数が同じだと棋力の差が出なかった 9 路盤では, 探索速度の差で 3x3 パターンを用いた Ray が勝ち越しているが, 13 路盤と 19 路盤では MD2 パターンを使った Ray が勝ち越している. この結果から, 盤の大きさが大きいほど, シミュレーションに用いるパターンの大きさが棋力に影響していると言える. さらに 13 路盤よりも 19 路盤の勝率が高くなっていることからも, 盤の大きさに関係して, シミュレーションに用いるパターンの大きさが与える影響が大きくなっていることがわかる.

## 5.4 対局実験 3

以下の 3 つの手法の有無によってどの程度棋力が変化するかを測定した.

1. 着手評価関数を組み込んだ UCB 値
2. Criticality による着手の並び替え
3. Ownership による着手の並び替え

### 5.4.1 対局の設定

対局の設定は以下の通りである.

- 対局相手 : GNU Go 3.8 Level 10
- 中国ルール
- 19 路
- コミ 6.5 目
- 黒白交互に入れ替えて 1000 局
- 1 手 1000PO, 1 手 2000PO, 1 手 4000PO, 1 手 8000PO

### 5.4.2 対局結果

対局結果を図 41 に示す. 縦軸は勝率, 横軸は 1 手あたりのプレイアウト回数を表す. 図中のバーは対局結果が正規分布に従うと近似したときの両裾 2.5%点を表す (＋が右裾 2.5%点, - が左裾 2.5%点).

また, 図 41 のそれぞれの線は

**赤** 全てのヒューリスティックを未使用

**緑** Ownership のみ使用

**青** Criticality のみ使用

**灰色** Ownership と Criticality を使用

**紫** 着手評価関数を組み込んだ UCB 値のみ使用

**橙** Ownership と着手評価関数を組み込んだ UCB 値を使用

**水色** Criticality と着手評価関数を組み込んだ UCB 値を使用

**黒** 全てのヒューリスティックを使用

を表している.

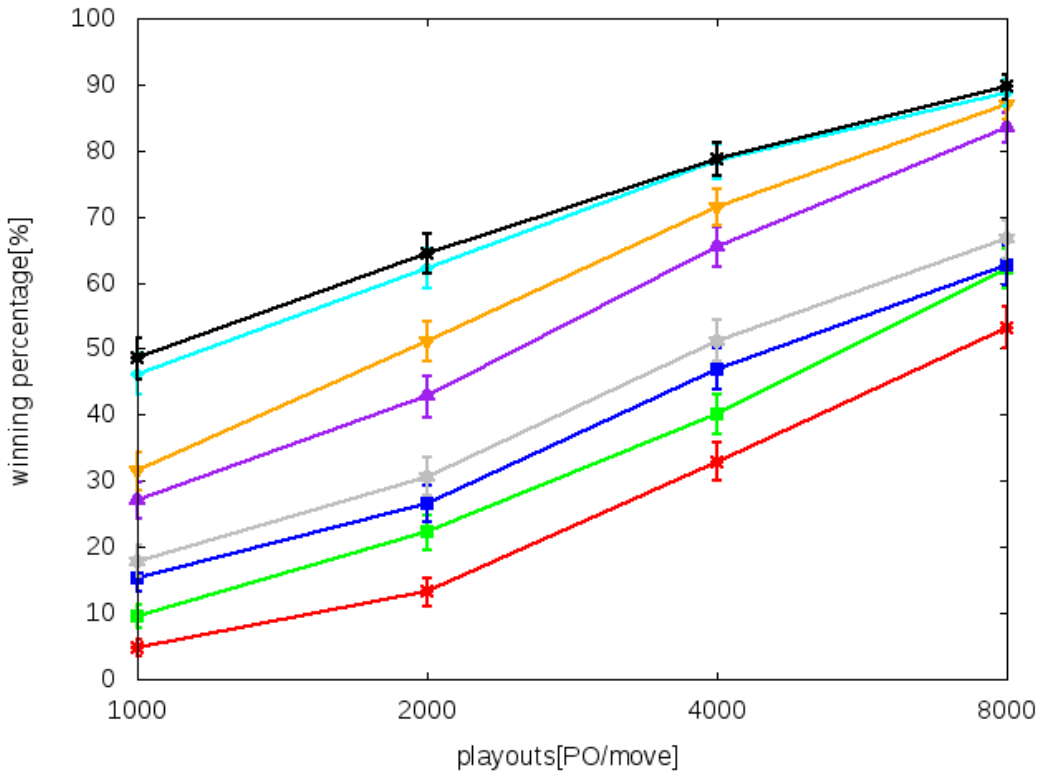


図 41: 各ヒューリスティックとプレイアウト回数による勝率の変化

### 5.4.3 考察

図 41 を見ると, 3 つの手法とも, 単体で使うだけでも棋力を向上させることがわかる. またどの手法でも, プレイアウト回数を増やすにしたがって棋力が向上していることがわかる. 単体での棋力向上での影響力は UCB が最も大きく, そのあとに Criticality と Ownership が続いているように見えるが, Ownership や Criticality のようなモンテカルロシミュレーションからのフィードバックよりも, UCB 値に補正をかけて, 良い手を優先的に探索した方が読みの精度が向上するからであると考えられる.

また UCB 値を補正する行動評価関数は事前の学習で求めている値のみを用いるので, Ownership や Criticality といった動的な評価と合わせることで, 大幅に棋力が向上していることが伺える. 一方で, 図 41 の黒線, 水色の線, 橙色の線に着目すると, 着手評価関数を組み込んだ UCB 値と Criticality を組み合わせた手法である水色の線と全ての手法を使った黒線に有意な差が見られない一方で, 着手評価関数を組み込んだ UCB 値と Ownership を組み合わせた手法である橙色の線と全ての手法を使った黒線に有意な差がみられる. このことからシミュレーション結果を用いた動的評価の性能としては Ownership よりも Criticality の方が優れていると言える.

## 5.5 対局実験 4

実験 3 で用いた手法を導入した囲碁プログラム Ray の棋力をモンテカルロ木探索を用いた最強クラスのオープンソースソフトウェアである Fuego [18] と Pachi [19] を用いて測定した.

### 5.5.1 対局の設定 1

対局の条件は以下の通りである.

- 対局相手：Fuego ver 1.1 と Pachi ver 11.00
- 中国ルール
- 9 路, 13 路, 19 路
- コミ 6.5 目
- 黒白交互に入れ替えて 1000 局
- 1 手 1 秒, 1 手 2 秒, 1 手 4 秒
- プログラムの使用するスレッド数を 16 個に固定
- 相手の思考時間中には思考しない
- 前の探索の結果を再利用する

### 5.5.2 対局結果 1

Fuego との対局結果を表 10, Pachi との対局結果を表 11 に示す. 括弧内は対局結果が正規分布に従うと近似したときの両裾 2.5%点を表す (＋が右裾 2.5%点, - が左裾 2.5%点).

表 10: 対局結果 (対局相手 Fuego ver 1.1)

| 1 手あたりの思考時間 | 9x9 | 13x13 | 19x19 |
|---|---|---|---|
| 1 秒 | 34.3%(±2.9%) | 75.7%(±2.7%) | 90.7%(±1.8%) |
| 2 秒 | 31.8%(±2.9%) | 73.3%(±2.7%) | 93.7%(±1.5%) |
| 4 秒 | 35.0%(±3.0%) | 78.2%(±2.6%) | 96.9%(±1.0%) |

表 11: 対局結果 (対局相手 Pachi ver 11.00)

| 1 手あたりの思考時間 | 9x9 | 13x13 | 19x19 |
|---|---|---|---|
| 1 秒 | 53.5%(±3.1%) | 70.6%(±2.8%) | 88.7%(±2.0%) |
| 2 秒 | 52.7%(±3.1%) | 63.4%(±3.0%) | 88.3%(±2.0%) |
| 4 秒 | 49.0%(±3.1%) | 62.8%(±3.0%) | 87.1%(±2.0%) |

### 5.5.3 対局の設定 2

大会のレギュレーションを想定して, 対局の条件を次のように設定した.

- 対局相手 : Fuego ver 1.1 と Pachi ver 11.00
- 中国ルール
- 19 路盤
- コミ 6.5 目
- 黒白交互に入れ替えて 1000 局
- 持ち時間 10 分
- プログラムの使用するスレッド数を 8 個に固定
- 相手の思考時間中にも思考する
- 前の探索結果を再利用する

### 5.5.4 対局結果 2

対局結果を表 12 に示す．括弧内は対局結果が正規分布に従うと近似したときの両裾 2.5%点を表す (＋が右裾 2.5%点, - が左裾 2.5%点).

表 12: 対局結果

| 対局相手 | Fuego | Pachi |
|---|---|---|
| 勝率 | 93.2%(±1.6%) | 74.3%(±2.7%) |

### 5.5.5 考察

表 10 と表 11 を見ると, 9 路盤においては, Fuego に有意に負け越しており, Pachi とは有意な差が見られないため, 同じ程度の強さであるので, 9 路盤ではオープンソースソフトウェアより強くすることはできていないと言える．これは 9 路盤は盤の大きさが小さく, 対局実験 1 からもわかる通り, シミュレーションの実行速度の速さが強さに大きく影響するからだと考えられる.

一方で, 13 路盤や 19 路盤では Fuego, Pachi の両者に有意に勝ち越している．13 路盤や 19 路盤では布石の段階があり, よりグローバルに着手を評価できる非決定論的シミュレーションの利点が出やすいからだと考えられる.

9 路盤では, シミュレーションの質よりも探索の速度が重要であり, 13 路盤と 19 路盤では探索の速度よりもシミュレーションの質が重要であると言える.

また表 12 を見ると, Fuego にも Pachi にも有意に勝ち越しているので, 持ち時間を固定して, 同じ環境で対局する場合には両者よりも強いと言える.

# 6　おわりに

## 6.1　全体のまとめ

本研究では, 非決定論的シミュレーションを行う強い囲碁プログラムの設計と開発を行った. また, 設計にあたって, 木探索部に用いる学習手法の違い, シミュレーションに用いるパターンの大きさの違い, 各種ヒューリスティックの有無による棋力の変化を測定した.

木探索部に用いる学習手法については, 著しい変化は見られなかったが, Minorization-Maximization アルゴリズムよりも 2 つの特徴の組合せたときの特徴を考慮できる Latent Factor Ranking アルゴリズムの方が優れていることがわかった.

シミュレーションに用いるパターンのサイズについては, 対局する盤の大きさが大きくなるほど, 大きなパターンを用いる棋力向上が明確になることがわかった.

各種ヒューリスティックについては, 全てのヒューリスティックについて, ヒューリスティック無しよりも有意に強く, またヒューリスティックを組み合わせて使うことにより, 更なる棋力向上が得られることを確かめられた.

## 6.2　今後の課題

本研究で用いたヒューリスティックについて, Ownership と Criticality はモンテカルロシミュレーションからのフィードバックであり, 着手評価関数を組み込んだ UCB 値はあらかじめ学習した結果を用いただけのものであるので, シミュレーション部から木探索部へのフィードバックだけでなく, 木探索部からシミュレーション部へのフィードバックの手法も考えていきたい. 特に木探索部では明確にわかる死活も, シミュレーション部においては確率によって間違え得るので, 木探索部とシミュレーション部の相互フィードバックは更なる棋力向上に不可欠だろう.

# 謝辞

本研究を行うにあたり, 日頃より丁寧に指導していただいた村松正和教授に心より御礼申し上げます. また, 計算資源を貸してくださった保木邦仁准教授と高橋里司助教, コンピュータ囲碁に関する様々な知識や助言をいただいた村松研究室博士後期課程の荒木伸夫氏, 本研究や論文執筆にあたり, 助言をしたいただいた研究室の皆様に深く感謝致します.

# 参考文献

[1] Murray Campbell, Feng-hsiung Hsu, A. Joseph Hoane Jr. “Deep Blue”, *Artifical Intelligence*, 134, 53-83, 2002.

[2] Levente Kocsis, Csaba Szepesvári, “Bandit Based Monte-Carlo Planning”, *Machine Learning: ECML 2006*, pp.282-293, 2006.

[3] Bernd Brügmann, “Monte Carlo Go”, *Technical Report*, Physics Department, Syracuse University, 1993.

[4] Rémi Coulom, “Computing Elo Ratings of Move Patterns in the Game of Go”, *ICGA journal*, Vol.30, No.4, pp.198-208, 2007.

[5] Rémi Coulom, “Criticality: a Monte-Carlo Heuristic for Go Programs”, Invited talk at the University of Electro-Communications, Tokyo, Japan, 2009. http://www.remi-coulom.fr/Criticality/Criticality.pdf , (2016 年 1 月 25 日閲覧)

[6] Albert Lindsey Zobrist, “A New Hashing Method with Application for Game Playing”, *Technical Report 88*, Computer Sciences Department, University of Wisconsin, 1970.

[7] Martin Wistuba, Lars Schmidt-Thieme, “Move Prediction int Go - Modelling Feature Interactions Using Latent Factors”, *KI 2013 : Advances in Artificial Inteligence*, Vol. 8077, pp 260-271, 2013.

[8] Steffen Rendle, “Factorization Machines”, *Proceedings of the 10th IEEE International Conference on Data Mining*, IEEE Computer Society, 2010.

[9] Cuillaume Chaslot, Christophe Fiter, Jean-Baptiste Hoock, Arpad Rimmel, Olivier Teytaud, “Adding expert knowledge and exploration in Monte-Carlo Tree Search” *LNCS*, Advances in Computer Games, Vol. 6048, pp 1-13, 2010

[10] Kokolo Ikeda, Simon Viennot, “Efficiency of Static Knowledge Bias in MonteCarlo Tree Search” *LNCS*, Computer and Games, Vol. 8427, pp 26-38, 2013

[11] Peter Auer, Nicolò Cesa-Bianchi, Paul Fischer, “Finite-time Analysis of the Multi-armed Bandit Problem” *Machine Learning*, Volume 47, Issue 2, pp 235-256, 2002

[12] 永田　靖　著入門 統計解析法, 日科技連, 1992 年出版

[13] 森棟 公夫, 照井 伸彦, 中川 満, 西埜晴久, 黒住英司 著統計学 Statistics : Data Science for Soucial Studies, 有斐閣, 2008 年出版

[14] 情報処理学会-コンピュータ将棋プロジェクトの終了宣言 http://www.ipsj.or.jp/50anv/shogi/20151011.html , (2016 年 1 月 2 日閲覧)

[15] 将棋電王戦 FINAL — ニコニコ動画, http://ex.nicovideo.jp/denou/final/ , (2016 年 1 月 2 日閲覧)

[16] 第 3 回電聖戦, http://entcog.c.ooco.jp/entcog/densei/densei3/, (2016 年 1 月 2 日閲覧)

[17] 囲碁用語辞典 http://www.godictionary.net/ , (2016 年 1 月 4 日閲覧)

[18] Fuego
http://fuego.sourceforge.net , (2016 年 1 月 2 日閲覧)

[19] Pachi - Board Game of Go / Weiqi / Baduk
http://pachi.or.cz/ , (2016 年 1 月 2 日閲覧)

[20] Oakfoam
http://oakfoam.com/ , ‘(2016 年 1 月 20 日閲覧)

[21] GNU Go
http://www.gnu.org/software/gnugo/ , (2016 年 1 月 2 日閲覧)

# A　付録

## A.1　囲碁の用語

本節では, 囲碁に用いられている用語について説明する [17].

- 対局
  囲碁で対戦すること.
- 着手
  盤上に石を置くこと. 囲碁ではこの動作を「打つ」と言う.
- 局面
  盤上の状態のこと.
- 終局
  対局が終了すること.
- 地
  どちらかの生き石にのみ囲まれた場所のこと.
- 空点
  石が置かれていない点
- 呼吸点
  石に隣接している空点. 呼吸点の数が 0 になると, その石は盤上から取り除かれる.
- コミ
  黒が白に与えるハンディキャップのこと.
- アタリ
  あと 1 手で相手の石を取ることができる状態, もしくはそのような状態にする着手. 図 42 の点 a に黒石を打てば, ×の白石を , 点 b に黒石を打てば, ○の白石を, 点 c に黒石を打てば, □の白石をアタリにできる.

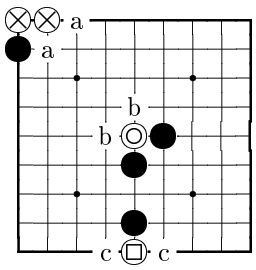


図 42: アタリ

- トリ
  石を取る手のこと. 図 43 の点 a に黒石を打てば, ×の白石を , 点 b に黒石を打てば, ○の白石を, 点 c に黒石を打てば, □の白石を取れる.

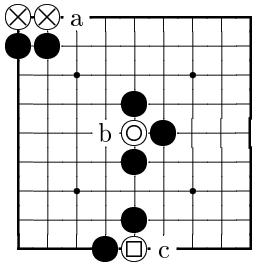


図 43: トリ

- シチョウ
  石を階段状につなげて逃げたり, 追いかけたりする形

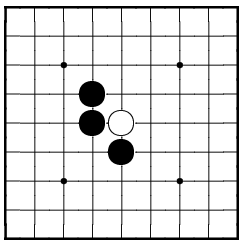

図 44: シチョウの例

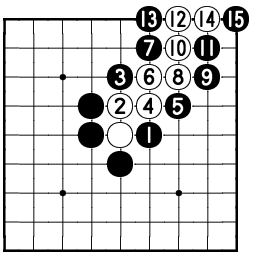

図 45: シチョウの応手

- ナカデ
  相手が手を入れると 2 眼できるひとかたまりの相手の地の中に打って 1 眼にする手. 図 46 の局面で白が x に着手すると黒は 2 眼を作ることができず, 死んでしまい, 黒が打てば 2 眼を確保できる. ナカデには 3 目ナカデ, 4 目ナカデ, 5 目ナカデ, 6 目ナカデがあり, それらを図 47 から図 50 に示す.

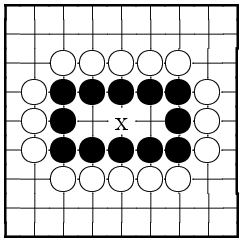


図 46: 3 目のナカデの例

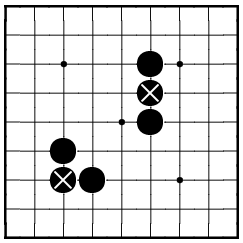

図 47: 3 目のナカデの形

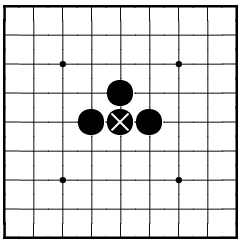

図 48: 4 目のナカデの形

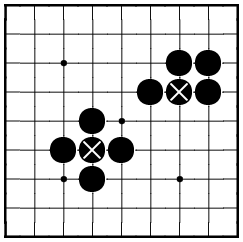

図 49: 5 目のナカデの形

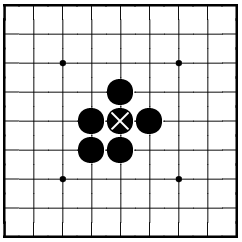

図 50: 6 目のナカデの形

- ウッテガエシ
  一度打った石を取られるが, 再度同じところに打って相手の石を取る手順のこと. 図 51 における黒の着手で, ×の白石を取れる . 白が石を助けようとしても, 図 52 の黒 3 で取られてしまう.

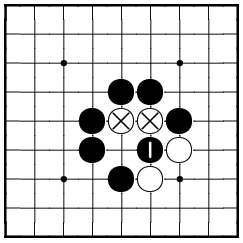


図 51: ウッテガエシの例

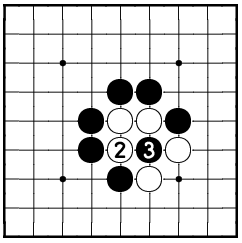


図 52: ウッテガエシの応手

- オイオトシ
  連続でアタリにして相手の石を取る手順のこと. 図 53 の黒 1 の手に対して, 白が点 a に打つと, 黒が点 b に打って, 全体を取ることができる.

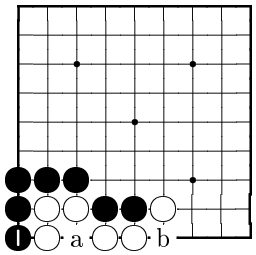


図 53: オイオトシの例

# B　仮説検定

本節では, 対局実験の結果から, プログラムが有意に強くなったかを検証する検定の方法を説明する. 検定の手順や数値は [12] と [13] を参考にした.

## B.1　二項分布

本研究でプログラムの棋力を測定する際に行った対局実験の 1 局ごとの結果は独立であり, 「勝ち」か「負けか」の 2 種類の結果しかないので, ベルヌーイ試行である. したがって, 対局実験における勝ち数は二項分布に従う. よって, 確率質量関数は

$$P(x=k) = \left( \begin{array}{c} n \\ k \end{array} \right) p^k (1-p)^{(n-k)} \tag{18}$$

で求められ, 対局実験に合わせると各変数は

$p$ : 真の勝率
$n$ : 対局回数
$k$ : 対局で勝った回数

となる. 二項分布は $n$ と $p$ が決まれば, 一意に定まるので $B(n,p)$ と表記する. 確率変数 $x$ が二項分布に従うとき, 平均 $E(x)$ と分散 $V(x)$ は

$$E(x) = np$$
$$V(x) = np(1-p)$$

となる. さらに $\hat{p} = x/n$ とおけば,

$$E(\hat{p}) = p$$
$$V(\hat{p}) = \frac{p(1-p)}{n}$$

となる.

## B.2　二項分布の正規近似

De Moivre-Laplace の定理より, 試行回数が十分に大きければ, 二項分布は正規分布に近似できる.

$$B(n,p) \sim N(np, np(1-p))$$

二項分布は $n$ が大きくなると計算が難しくなるが, 正規分布に近似すると扱いやすい. 本研究で行った仮説検定では, 対局結果を全て正規分布に近似して扱っている.

## B.3　検定の手順

1 つの対局実験の結果に基づいて, 有意に強いかを検定する手順を以下に示す.

1. 帰無仮説 $H_0 : P = P_0 (= 0.5)$,
   対立仮説 $H_1 : P > P_0$ を設定する.
2. 有意水準 $\alpha$ を定める (本研究では $\alpha = 0.05$).
3. 対立仮説と有意水準に応じて棄却域 $R$ を定める.
   対立仮説は $P > 0.50$ なので, $u_0 \geq 1.645$ とする
4. $$u_0 = \frac{r - P_0}{\sqrt{P_0(1-P_0)/n}}$$
   を計算する. ただし $r$ は対局結果の勝率で, $n$ は対局回数.
5. $u_0$ が棄却域にあれば有意と判定し, $H_0$ を棄却する.

実際に表 11 の 1 手 4 秒の 13 路盤の対局結果を用いて計算してみると, 勝率は 0.628, 対局回数は 1000 回なので,

$$\begin{aligned} u_0 &= \frac{0.628 - 0.5}{\sqrt{0.5 \cdot 0.5/1000}} \\ &= \frac{0.128}{0.0158\cdots} \\ &\simeq 8.10 \end{aligned}$$

となり, $u_0 \geq 1.645$ が成り立ち, 囲碁プログラム Ray は囲碁プログラム Pachi との 1 手 4 秒の 13 路盤において, Pachi より有意に強いと言える.

同じプログラム相手の 2 つの対局結果に基づいて, どちらが有意に強いかを検定する手順を以下に示す.

1. 帰無仮説 $H_0 : P_1 = P_2$,
   対立仮説 $H_1 : P_1 > P_2$ を設定する.

2. 有意水準 $\alpha$ を定める (本研究では $\alpha = 0.05$).

3. 対立仮説と有意水準に応じて棄却域 $R$ を定める.
   対立仮説は $P > 0.50$ なので, $u_0 \geq 1.645$ とする

4. $$u_0 = \frac{r_1 - r_2}{\sqrt{r^*(1-r^*)\left(\frac{1}{n_1} + \frac{1}{n_2}\right)}} \tag{19}$$
   を計算する.
   ただし, $n_1$ は $P_1$ が得られた対局実験の対局回数, $n_2$ は $P_2$ が得られた対局実験の対局回数を表す. また, $r^*$ は
   $$r^* = \frac{x_1 + x_2}{n_1 + n_2}$$
   であり, $x_1$ は $P_1$ が得られた対局実験の勝ち数, $x_2$ は $P_2$ が得られた対局実験の勝ち数である.

5. $u_0$ が棄却域にあれば有意と判定し, $H_0$ を棄却する.

実際に, 表 7 の 19 路盤の実験結果から, MD2 パターンを用いた Ray(勝率 65.0%) が 3x3 パターンを用いた Ray(勝率 43.5%) より有意に強いかを確かめてみると,

$$\begin{aligned} r^* &= \frac{650 + 435}{1000 + 1000} \\ &= 0.5425 \end{aligned}$$

なので,

$$\begin{aligned} u_0 &= \frac{0.650 - 0.435}{\sqrt{0.5425 \cdot (1 - 0.5425) \cdot \frac{1}{500}}} \\ &= \frac{0.215}{0.022\cdots} \\ &\simeq 9.65 \end{aligned}$$

となり, $u_0 \geq 1.645$ なので, MD2 パターンを用いた Ray は 3x3 パターンを用いた Ray よりも有意に強いと言える.

対局結果をまとめた表や図に「対局結果が正規分布に従うと近似したときの両裾 2.5%点を表す」値を記したが, 有意水準 5%での片側検定の時は, 右裾 5.0%点よりも大きい, または左裾 5.0%点よりも小さい領域に, 有意水準 5%での両側検定の時は, 両裾 2.5%点より外側に帰無仮説の棄却域ができるので, 簡易的な目安として示したものである. 考察において, 有意に強いかどうかを議論する際には, 全て有意水準 5%での片側検定の結果を用いている.