修　　士　　論　　文

木探索の部分結果を利用した モンテカルロ囲碁の シミュレーション方策の動的調整

Dynamic Tuning of Simulation Policies Utilizing Partial Results of Tree Search in Monte-Carlo Go

指導教員　　鶴岡 慶雅 准教授

東京大学工学系研究科
電気系工学専攻

氏 名　　**37-126423 板持 貴之**

提 出 日　　平成 26 年 2 月 6 日

概 要

コンピュータゲームの研究は人工知能のテストベッドとして発展してきており，その一つであるコンピュータ囲碁プレイヤの研究も長く行われてきた．モンテカルロ木探索は 2006 年に登場した探索手法であり，コンピュータ囲碁プレイヤの性能を飛躍的に上昇させ，今ではゲーム以外の分野でも様々な応用が提案されている．モンテカルロ木探索が囲碁において成功を収めたのは，チェスや将棋に比べて探索空間が大きく，信頼できる盤面の評価関数を作るのが難しいという，コンピュータ囲碁プレイヤが苦手としていた問題に有効にはたらいた点にある．

モンテカルロ木探索は，従来使われていたゲーム木探索と，ランダムシミュレーションによって盤面評価を行なうモンテカルロ法を組み合わせた手法である．ゲーム木探索の改善やシミュレーション方策の改善により，今日でもその性能を改善する手法が提案されている．シミュレーション方策の改善は，事前に棋譜等から学習して得られた方策を用いることが多い．一方，対局中には方策の学習は行われず，常に同じ方策を固定して用いるのが一般的である．本研究ではこの点に注目し，モンテカルロ木探索を用いたコンピュータ囲碁プレイヤの性能改善を目的とし，木探索の部分結果を用い，モンテカルロ木探索におけるシミュレーション方策を対局中に動的に調整する手法を提案する．木探索から得られる情報は，複数のシミュレーションの結果を合わせた情報となるので，信頼性が高い情報であると考えられる．この情報を用いることで，より信頼性が高いシミュレーション方策を動的に調整する手法を提案する．

対戦実験の結果，従来手法を統計的に有意に上回る性能を達成することは出来なかったが，パラメータによっては従来手法を上回る傾向を示した．また，提案手法を検証するために，より理想的な設定で補足実験を行った．その結果，対象局面で打つべき着手が事前に分かっているような場合でも，その情報をシミュレーション方策に反映させるには，着手確率をどの程度上げるか，と言ったパラメータの調整や，着手位置の周辺状況が変化していないか確認を行うこと等が重要であると明らかになった．

目 次

第 1 章 序論 **1**
1.1 背景と目的 1
1.2 本研究の貢献と成果 2
1.3 本論文の構成 3

第 2 章 関連研究 **4**
2.1 コンピュータゲームプレイヤにおける探索手法 4
2.1.1 ミニマックス探索 4
2.1.2 モンテカルロ法 5
2.1.3 モンテカルロ木探索 7
2.1.4 Upper Confidence bound applied to Trees (UCT) 9
2.2 シミュレーション方策の改善 11
2.2.1 方策の改善 11
2.2.2 ルールベースによる改善 12
2.2.3 棋譜の着手を真似る手法による改善 13
2.2.4 その他の改善手法 15
2.3 シミュレーション方策の動的調整 16

第 3 章 モンテカルロ木探索を用いたコンピュータ囲碁プレイヤ **18**
3.1 各種ルール 18
3.1.1 地の数え方，終局判定 18
3.1.2 眼について 19
3.1.3 その他ルール 20
3.2 コンピュータ囲碁での UCT 21
3.2.1 木探索 21
3.2.2 シミュレーション 21

第 4 章 局所性を利用したシミュレーション方策の動的調整 **25**
4.1 手法の概要 25
4.2 方策の調整アルゴリズム 27
4.3 評価 30
4.3.1 実験設定 30

4.3.2 自己対戦実験 32
4.3.3 GNU Go との対戦実験 32
4.4 考察 33

第 5 章 最善手情報が既知である設定での検証 35
5.1 特定着手の着手確率変更のみに注目した検証実験 35
5.2 対戦実験による検証 36
5.2.1 自作プレイヤによる対戦実験 38
5.2.2 Fuego による対戦実験 39
5.2.3 対戦実験結果からの考察 45
5.3 周辺パターンにも注目した検証実験 45
5.3.1 自作プレイヤによる対戦実験 47
5.3.2 検証実験からの考察 48

第 6 章 結論 51
6.1 まとめ 51
6.2 課題 51

図 目 次

2.1 ミニマックス探索の例 5
2.2 モンテカルロ法による円周率の近似 5
2.3 原始的なモンテカルロ法による着手の決定 6
2.4 原始的なモンテカルロ法の問題点 7
2.5 モンテカルロ木探索 8
2.6 シミュレーション結果の反映 9
2.7 UCB の例 ($C = 1$) 10
2.8 方策の違いによるシミュレーションの違い 12
2.9 MoGo における「ハネ」パターン 12
2.10 MoGo のシミュレーション方策の比較 (図は [13] より引用) 13
2.11 特徴量の例 14

3.1 ルールの違いによる結果の違い 19
3.2 欠け眼 19
3.3 眼への着手による終局までの手数の増加 20
3.4 特徴量の例 22

4.1 提案手法の流れ 26
4.2 方策の動的調整を組み込んだ UCT 27
4.3 Static との対戦結果 (九路盤) 31
4.4 Static との対戦結果 (十三路盤) 31
4.5 GNU Go との対戦結果 32
4.6 動的調整が悪影響を及ぼしてしまう例 33

5.1 特定の着手確率を上げるシミュレーション方策 35
5.2 検証実験用プレイヤ 36
5.3 正規化されていない重みテーブルから効率的に着手を選択する手法 38
5.4 検証実験: Dynamic - Plain との対戦結果 (九路盤) 40
5.5 検証実験: Dynamic - Plain との対戦結果 (十三路盤) 40
5.6 検証実験: Dynamic - GNU Go との対戦 (九路盤) 41
5.7 Weak Dynamic Fuego プレイヤ 42
5.8 Medium Dynamic Fuego プレイヤ 42

5.9 検証実験: Weak Dynamic Fuego - Plain Fuego との対戦結果 (九路盤) 43
5.10 検証実験: Weak Dynamic Fuego - Plain Fuego との対戦結果 (十三路盤) 43
5.11 検証実験: Medium Dynamic Fuego - Plain Fuego との対戦結果 (九路盤) 44
5.12 検証実験: Medium Dynamic Fuego - Plain Fuego との対戦結果 (十三路盤) 44
5.13 検証実験: Weak Dynamic Fuego - GNU Go との対戦結果 46
5.14 検証実験: Medium Dynamic Fuego - GNU Go との対戦結果 46
5.15 検証実験: Pattern Dynamic - Plain との対戦結果 (九路盤) 49
5.16 検証実験: Pattern Dynamic - Plain との対戦結果 (十三路盤) 49
5.17 検証実験: Pattern Dynamic - GNU Go との対戦結果 50

アルゴリズム，擬似コード

2.1 対数尤度最大化によるソフトマックス方策の最適化 15
4.1 局所性を利用したシミュレーション方策の動的調整 29
5.1 検証用プレイヤのアルゴリズム 37
5.2 周辺パターンのチェックを加えた検証用アルゴリズム 47

第1章　序論

1.1　背景と目的

人工知能は，人間の判断を代替・支援するものとして発展し，研究されてきた．また，コンピュータゲームプレイヤの研究は，ルールが明確であること，ゲームそのものが人々から親しまれていることから，人工知能のテストベッドとして用いられてきた．特に，チェスや将棋に代表される二人零和有限確定完全情報ゲームは古くから研究されており，局面の静的評価値を求める静的評価関数とゲーム木探索を組み合わせた，ミニマックス探索を始めとする手法が大きく成功を収めてきた．これらの手法では，静的評価関数の精度が全体の性能に大きな影響を与えるが，一般に精度の高い評価関数を作るためには，ゲームに特化した知識が必要となる．

しかしながら，二人零和有限確定完全情報ゲームの代表例の 1 つである囲碁においては，ミニマックス探索と静的評価関数をベースとした手法では，その性能が頭打ちになっていた [11]．その理由には以下の 2 つがある．

- 探索空間が極端に大きい
  チェスは状態空間の大きさが約 10^{50}，将棋は約 10^{71} であるのに対し，囲碁は十九路盤で約 10^{171}
- 信頼性のある静的評価関数を作るのが難しい
  チェスや将棋は，駒の価値，玉の自由度，駒の効き具合等から先手後手がそれぞれどのくらい有利かを比較的数値化しやすい (静的評価関数を作りやすい) が，囲碁はそれが難しい (「石が厚い」「形が良い」等の表現が用いられ，具体的な数値として表現することが難しい)

特に後者の問題が大きく，将棋より状態数が少ないはずである九路盤の場合でも，ミニマックス法を利用した手法ではアマチュア初段程度の実力止まりであった．そこで，局面評価に静的評価関数を用いず，ランダムなシミュレーションを行なうことで評価値を近似するモンテカルロ法を組み込む方向で研究が進んできた．その 1 つであるモンテカルロ木探索 [6] (Monte-Carlo Tree Search: MCTS) は，決定的な探索である木探索と，確率的なモンテカルロシミュレーションを組み合わせた手法であり，近年のコンピュータ囲碁プレイヤのほぼ全てに採用されている．この MCTS を基礎として様々な応用が研究され，九路盤では互先でプロ同等の棋力，十九路盤でも強豪ソフトの 1 つである Crazy Stone がプロ相手に四子局で勝利するなど，コンピュータ囲碁プレイヤは着実にプロの実力に近づいている[1]．また，MCTS はコンピュータ囲碁以外の二人完全情報ゲーム [17,31] や不完全情報ゲーム [26]，一人用ゲーム [23]，多人数ゲーム [24]，リアルタイムストラテジーゲーム [3] 等にも用いら

[1]第 1 回 電聖戦: http://entcog.c.ooco.jp/entcog/densei/denseisen-1st.html

れている．さらに，General Game Playing (GGP) と呼ばれる，どのようなゲームに対しても動くコンピュータゲームプレイヤの研究にも利用されており [9,10,19]，更にはゲーム以外の分野にも利用されている [18,21,29].

モンテカルロ木探索の性能を改善するには，シミュレーションをどのように行うか，という点を改善するシミュレーション方策の改善と，その結果を利用した木探索の改善の 2 つに大きく分けられる．前者においては，様々なヒューリスティックスを組み込む手法，事前に棋譜から学習しておいた方策を用いる手法等が提案されている．後者については，Upper Confidence bound applied to Trees (UCT) [15] という手法をはじめ，様々な手法が提案されている．また，通常 MCTS においては，シミュレーションの結果得られた報酬だけを木探索で利用するのに対し，木探索とシミュレーションの間でより多くの情報をやり取りする手法も提案されている．All Moves As First (AMAF)，または Rapid Action Value Estimation (RAVE) [11] と呼ばれる手法が代表的である．これは，シミュレーションによって得られた報酬だけでなく，シミュレーション中で発生した状態遷移の情報も木探索にフィードバックする手法である．

上記のように，シミュレーション，木探索それぞれの改善に加え，シミュレーション結果をより効率的に木へ反映する手法等がモンテカルロ木探索では用いられるが，木探索の結果からシミュレーション方策を改善する手法で成功しているものはあまり報告されていない．対局中，モンテカルロ木探索によってゲーム木が成長すれば，そこから信頼性の高い情報 (どの着手がどのくらいの推定勝率を持つのか，等) が推定できる．このような高い信頼性を持つ情報は，シミュレーションを行う際にも有効だと考えられる．また，一般的なシミュレーション方策の改善法である「事前に組んでおいたヒューリスティックなルールベースを組み込む」「方策を事前に大量の棋譜から学習」では,「特定の対局・状態に依存しない一般的な方策」しかシミュレーションに組み込むことができない．そこで本研究では，対局中の状態に合わせたシミュレーション方策の改善を目的として，木探索の結果を用いて対局中に動的にシミュレーション方策を調整する手法を提案する．

対象のゲームとしては囲碁を用いる．囲碁は，モンテカルロ木探索が有効にはたらくゲームの代表例であり，また，シミュレーション方策の動的な調整が有効になりうると期待されているゲームでもあるため [33]，本研究での提案手法の評価を行うのに適した題材と言える．

1.2 本研究の貢献と成果

本研究の貢献と成果は以下のようになる．

- コンピュータ囲碁プレイヤであまり用いられていない，モンテカルロ木探索のシミュレーション方策の動的調整を扱った．シミュレーション方策の動的調整という方向性はコンピュータ囲碁コミュニティでも注目されており，本研究の成果が役立つと考えられる．
- 囲碁の場合は，ヒューリスティックスをシミュレーション方策に組み込んだり，過去に打たれた棋譜から事前に学習したりできるが，GGP のように，事前に方策の改善が出来ないタスクもある．本研究で提案する手法は，これらのタスクにも応用が可能になっている．

- 評価実験より，統計的には有意ではないものの，パラメータや対戦相手によっては従来手法によるプレイヤより強くなる傾向を示したことを確認した．さらに，より理想的な設定で検証実験を行った結果，調整を行う際には，良いと判断された着手の着手確率をどの程度上昇させるかと言ったパラメータ，着手位置の周辺状況が変化していないかチェック，等の調整が重要であることを確認した．このため，パラメータ調整や組み合わせる特徴量，対象とするゲーム次第では有用な学習方法になり得ると言える結果となった．

1.3 本論文の構成

第 1 章にて，本研究の背景，目的を述べる．続いて，第 2 章にて，関連研究として二人有限確定ゼロ和ゲームにおける探索手法，その中でも特にモンテカルロ木探索に関連した研究について述べる．第 3 章において本研究で用いるコンピュータ囲碁プレイヤの共通事項について述べ，第 4 章で局所性を利用したシミュレーション方策の動的調整の手法について述べ，評価実験を行った結果を示す．次に第 5 章にて，与えられた局面の最善手が既知な場合，という設定で手法の検証実験を行った結果を述べる．最後に，第 6 章にてまとめを行う．

第2章　関連研究

2.1　コンピュータゲームプレイヤにおける探索手法

本節では，コンピュータゲームプレイヤによく用いられる探索手法について説明する．また，簡単のため，終局で勝敗が決する二人有限確定ゼロ和ゲームを仮定して説明を行う．

2.1.1　ミニマックス探索

コンピュータゲームプレイヤの研究においては，対象のゲームとして，状態空間が大きく，全ての状態を解析することが計算資源の面から事実上不可能なものを選ぶことが一般的である．このようなゲームに対しては，静的評価関数と木探索を組み合わせたミニマックス探索が広く用いられてきた．静的評価関数は，与えられた状態に対して，その状態の良し悪しを表す評価値を出力する関数である．一般に，静的評価関数の精度が十分に高かったとしても，その値が最も高い次着手を選ぶだけで十分に強いプレイヤを作ることは困難である．そこで，数手先の評価値を高くするよう，木探索による先読みを行うことでその精度を高くできる．図 2.1 は 2 手先まで探索を行う場合の例である．末端ノードの値が静的評価関数によって出力された値であり，先手番 (青いノード) にとっての評価値となる．その 1 つ上の後手番 (赤いノード) は,「自分にとって一番有利 = 先手にとって一番不利」な状態を選ぶので，最小の評価値を選択する．その上のルートノードである先手番ノードは，自分にとって一番有利になる最大の評価値を選択する，といったように再帰的な探索を行う．このような探索はミニマックス探索と呼ばれ，チェスや将棋といったコンピュータプレイヤにおいて大きな成果を上げている．

一方で，このミニマックス探索があまり有効でなかったゲームもある．囲碁はその代表である．その理由として，1 つに探索空間の大きさがある．チェスは探索空間の大きさが 10^{50}，将棋は 10^{71} であるのに対し，囲碁は 19 路盤の場合 10^{171} と膨大な大きさとなり，ミニマックス探索では非常に探索に時間がかかってしまう．また，囲碁はチェスや将棋等に比べ，精度の高い静的評価関数が作りづらいという問題点がある．チェスや将棋の場合，駒の価値やキング (玉) の安全度等から精度の高い評価関数を作ることが可能だが，囲碁の場合，石の価値は平等であり，終局するまで領域を確定できない，局所的な最善手が必ずしも全局的な最善手になりにくい，といった特徴があり，静的評価関数を作ることが非常に難しい．以上のような問題点により，ミニマックス法によるコンピュータプレイヤは，十九路盤に置いてはアマチュア五級程度で棋力が頭打ちであった [11].

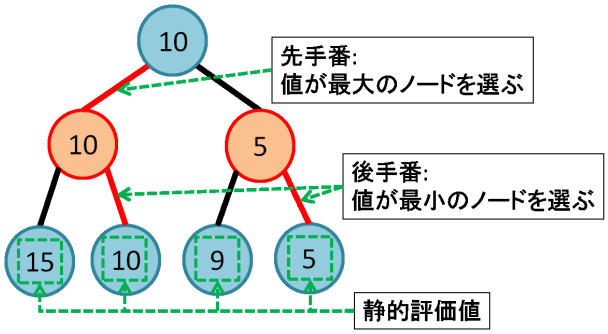


図 2.1 ミニマックス探索の例

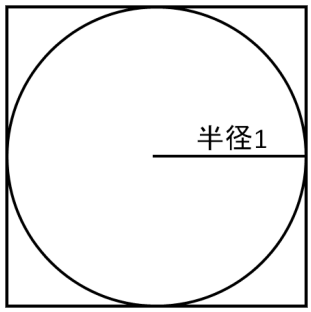


図 2.2 モンテカルロ法による円周率の近似

2.1.2　モンテカルロ法

チェスや将棋等では，状態評価に静的評価関数を用いていたが，これにモンテカルロ法を用いることも可能である．モンテカルロ法はコンピュータゲームプレイヤのための手法ではなく，ランダムなシミュレーションの繰り返しによって近似値を得る手法全般を指す．一般的な例として，モンテカルロ法を用いた円周率の近似がある．図 2.2 のように，半径 1 の円と，それに外接する正方形を考える．この正方形内にランダムに点を生成し，円の内側に入った点を数える (生成された点の座標を (x, y) とすると，$x^2 + y^2 \leq 1$ で判定できる)．このランダムな点生成を N 回行い，n 個の点が円の内側に入ったとすると，円周率は以下のように近似できる．

$$\pi \approx \frac{4n}{N} \tag{2.1}$$

大数の法則により，試行回数 N を増やしていくと真の円周率 π に収束する．

ゲームの場合，ある状態が与えられた時に，その状態から終端 (ゲームが終了，もしくは勝敗が明示的になった状態) までランダムに自己対戦を行うことで勝敗を決する．これを何回も繰り返し，その勝敗を平均した平均勝率を局面評価値の近似値 (推定値) として扱う．これを利用して，与えられた状態に対して行動を決定する手法は，図 2.3 のようになる [11]．ジグザグの矢印が終端までのシミュレーションを表し，ノード内の数字がシミュレーションの結果得られた平均勝率である．図のよ

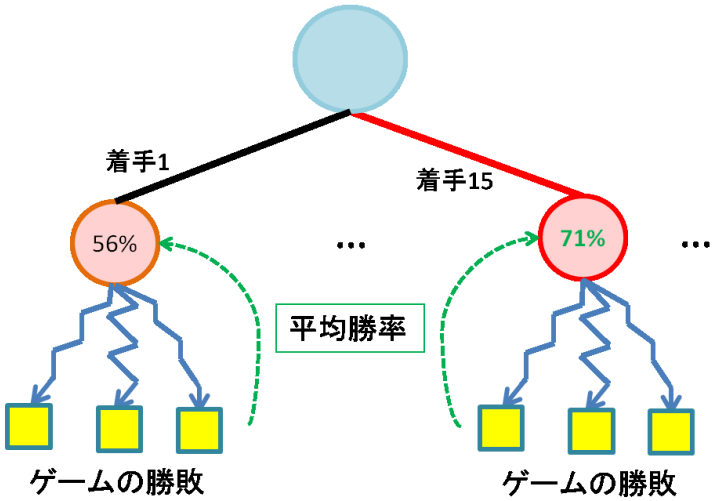


図 2.3 原始的なモンテカルロ法による着手の決定

うに，合法手を指した 1 手先の状態からそれぞれシミュレーションを行い，その平均勝率が一番高い着手を選択する，というようになる．

以上が原始的なモンテカルロ法を用いた行動決定の方法であるが，これには問題点がある．例えば図 2.4 のような場合である．これは，先手番から着手 A を指し (中央の赤いノード)，そこからミュレーションを開始し，A’，B’，C’ の 3 つの合法手があった，という状況を示している．この例では，着手 A の平均勝率は 56% となっている (勝率は先手番から見た勝率)．しかし，仮にこの着手 A を選択した場合，後手は先手の勝率が最小になる着手を選択するので，勝率 10% の A’ が選択される．つまり，モンテカルロ法によって求められた平均勝率 56%と，2 手進めたときの実際の平均勝率 (10%) に大きな差ができてしまうことになる．これは，後手番が正しい選択をすれば先手番にとっては勝率が 10% しかないが，後手番が仮に悪手を選択した場合のシミュレーション結果 (中央と右側の青いノード) も平均勝率に含めてしまっているので，勝率が高いように見えてしまう (相手の悪手を期待してしまう)．純粋なランダムシミュレーションでは全ての着手を等確率で選択するため，このような問題がよく発生する．このように，原始的なモンテカルロ法では,「相手の悪手に期待した着手を行なってしまう」という問題点がある．これに対応するため，次の 2 つの手法が主に用いられる．

- モンテカルロ法と木探索を組み合わせたモンテカルロ木探索 [6]
- シミュレーション方策 (シミュレーションをどのように行うか) の改善

モンテカルロ木探索はモンテカルロ法を探索の一部に用いるため，上記の 2 つの手法は併用されることが多い．

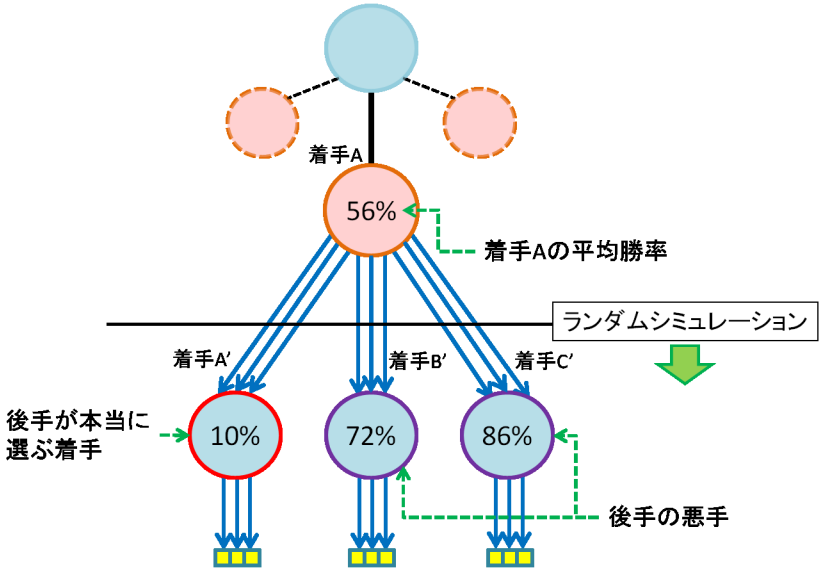


図 2.4 原始的なモンテカルロ法の問題点

2.1.3　モンテカルロ木探索

2.1.2 節で述べたように，原始的なモンテカルロ法では相手の悪手に期待してしまうという問題点があった．その問題を改善するため，木探索とモンテカルロ法を組み合わせたモンテカルロ木探索 (Monte Carlo Tree Search: MCTS) [6] が提案された．

MCTS による探索の流れを示したのが図 2.5 となる．図にあるように，探索は 4 つのステージに分かれている．

(1) 選択: 構築された木に対して最良優先探索を行い，対象となる葉ノードを選ぶ

(2) 拡張: 選択したノードが条件を満たしていれば，子ノードを展開し，そのノードを選択する

(3) シミュレーション: 選択されたノードを始点とし，モンテカルロ法によって状態の評価を行う

(4) 更新: シミュレーション結果を木を遡りながら各ノードに反映する

一定回数，もしくは時間の許す限りこの繰り返しを行い，その後ルートノードから見て最も良い子ノードを，それぞれのノードに格納された探索結果から選択することで着手を選択する．この探索は，1 回のイテレーション単位で探索を打ち切ることができるので，ミニマックス探索と異なり，Anytime な探索 (いつでも打ち切りが可能な探索) となるのが特徴である．1 回の探索に使用できる時間が限られている場合の探索では，この特徴は非常に有効にはたらく．基本的に 1 回あたりのイテレーションにかかる時間は，ゲーム上定められる制限時間よりはるかに短い場合が多い．このため，Anytime な探索であれば，制限時間ギリギリまで探索を行い，制限時間直前で探索を打ち切る，といったこと

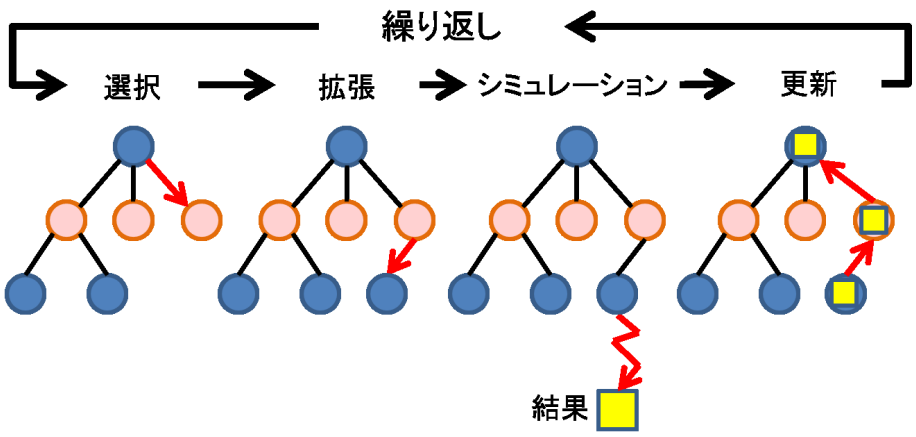


図 2.5 モンテカルロ木探索

も可能になる．ミニマックス探索にも反復深化法等の時間制御を考慮した手法が存在するが，モンテカルロ木探索ほど細かく制御はできないのが一般的である．

続いて，それぞれのステージについて説明を行うが，それぞれの具体的な手法は対象の問題によって変わってくる．どのようにルートノードから見て最善の子ノードを選択するか，という点も同様である．ここでは一般的な話題に留めて説明を行うが，コンピュータ囲碁の場合，後述する Upper Confidence bound applied to Trees (UCT) [15] という手法が具体的な手法として用いられることが多い．

選択・拡張　探索対象となるノードの選択・拡張を行い，シミュレーションの対象となるリーフノードを決定する．各ノードの平均報酬 (ゲームの場合は平均勝率)，訪問回数等を基準に，リーフノードまで再帰的に子ノードを選択する．さらに，選択されたリーフノードが，ノードが拡張されるべき条件 (例えば，訪問回数が一定以上か，等) を満たしていれば，そのノードの子ノードを展開し，1 つを選択する．ノードの選択・拡張の方針は,「ツリー方策 (Tree Policy)」と呼ばれる．ここで選択されたノードはシミュレーションステージに利用される．

シミュレーション　対象となったリーフノードに対し，そのノードを評価するためのモンテカルロシミュレーションを行う．一般には，それ以上状態を変化させられない終端状態までシミュレーションを行い，その時点での状態の評価値 (ゲームの場合，勝敗) を求める．例えば囲碁の場合，リーフノードの局面から，自己対戦によってパス以外の合法手が存在しなくなる状態までランダムにゲームを進め，その時点での勝敗を返す．

詳しくは後述するが，このシミュレーションステージにおいて，次状態を完全にランダムに決めるのではなく，ゲームに関する知識を導入することで探索全体の性能を上げることが可能になる．シミュレーションに知識を導入することで，1 回 1 回のプレイアウト (与えられた状態から終端までの状態遷移) の精度が高まり，少ないプレイアウト数でより高い精度の平均報酬を得ることができるか

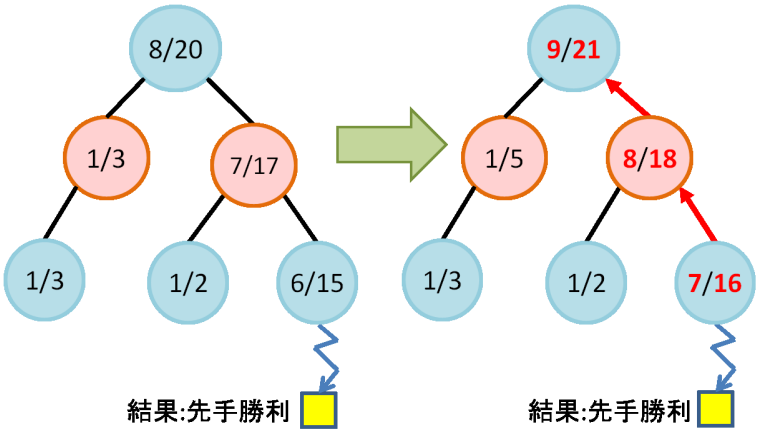


図 2.6 シミュレーション結果の反映

らである．

更新 シミュレーションで得た結果を木に反映させるのがこの更新ステージである．シミュレーションの結果は，選択・拡張ステージでリーフノードに達するまでに経由した全てのノードに対して反映される．どのように反映されるか，二人ゲームの場合の例を図 2.6 に挙げる．青色のノードを先手番，赤色のノードを後手番とし，ノード内にある数字は,「合計報酬/訪問回数」である．この例での「報酬」は，先手が勝った場合 +1，後手が勝った場合 −1 となる．図ではシミュレーションの結果先手が勝利しているので，シミュレーションステージから +1 の報酬が返ってくる．これを対象ノードに反映させた結果が図中の右側の状態である．対象ノードの訪問回数が全て 1 増え，合計報酬も +1 されている．ここで各ノードの値が更新されることにより，選択・拡張ノードで選択されるノードに変化が生じる．

2.1.4 Upper Confidence bound applied to Trees (UCT)

Upper Confidence bound applied to Trees (UCT) [15] は，MCTS の具体的な手法の 1 つとして広く用いられている手法であり，囲碁に加え，種々の分野に広く応用されている手法である．

UCT はその名前が示すとおり， Upper Confidence Bound (UCB) [2] と呼ばれる値をツリー方策に用いた手法である．式 (2.2) がその UCB 値である．

$$UCB_i = \overline{X_i} + C\sqrt{\frac{\log N}{N_i}} \tag{2.2}$$

UCT では，この UCB 値が最も高い子ノードを再帰的に選択していく．UCB_i は子ノード i の UCB 値を示す．$\overline{X_i}$ はノード i の平均報酬であり，N_i はノード i の訪問回数，N はノード i を含む全ての子ノードの訪問回数の総計である．また，C は事前に設定しておくパラメータである．図 2.7 はこれを適用した具体例である．ノード中の数字は,「合計報酬/訪問回数」を示す．図 2.7 で示している

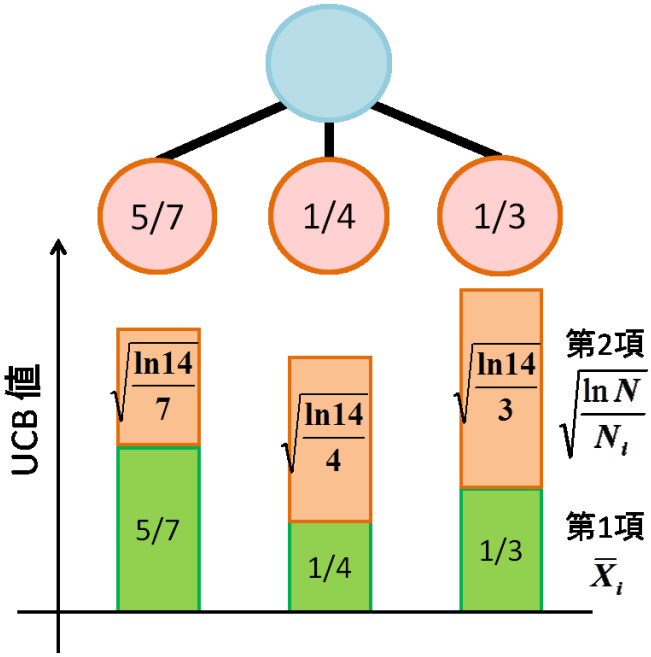


図 2.7 UCB の例 ($C = 1$)

ように，この式は，第 1 項が平均報酬である一方，第 2 項は「訪問されていない度合い」を示す．つまり，ノードが比較的あまり訪問されていない場合，$\sqrt{\log N/N_i}$ の項が大きくなるため，その段階での平均報酬が小さくても選択されやすくなる (一番右のノード)．逆に，訪問回数が比較的多いノードの場合，$\overline{X_i}$ が UCB 値の中で支配的になるため，第 2 項の影響を受けにくくなる (一番左のノード)．C はそのバランスを調整するパラメータであり，これが小さいほど平均報酬を重視したノード選択になり，大きい場合は逆に広くノード選択をするようになる．

以上の説明でも述べたように，「平均報酬が高いノードを選びたい (搾取: exploitation)」要望と，「まだ調べていないところは平均報酬に関わらず調べたい (探索: exploration)」要望のジレンマを解決する手法の 1 つが UCB である．この問題は一般に「多腕バンディット問題」と呼ばれる問題である．これは，K 台のスロットマシンが，それぞれ独立で固有の確率分布に従って「あたり」「はずれ」を出すとき，リグレット (期待損失: regret) が最小になるようにはどのように K 台のスロットマシンにコインを入れていくべきか，という問題である．リグレットを定式化すると式 (2.3) のようになる．

$$\text{minimize:} \quad \mu^* N - \mu_j \sum_i^K E[T_i(N)] \quad where \quad \mu^* \stackrel{\text{def}}{=} \max_{1 \le i \le K} \mu_i \tag{2.3}$$

N はコインを投入した回数，μ_i はマシン i の当たりの確率 (期待報酬)，$T_i(N)$ はマシン i を選んだ回数を示す．このリグレットは，少なくとも $O(\log N)$ で増加することが分かっている [16]．この問題を解決する手法の 1 つが，式 (2.4) で表される UCB1 (Upper Confidence Bound) 値が最大にな

るマシンを選ぶ手法である．

$$UCB1(i) = \overline{X}_i + \sqrt{\frac{2\log N}{n_i}} \tag{2.4}$$

$\overline{X}_i$ はマシン i から得られた平均報酬，n_i はマシン i を選択した回数である．これを，各スロットマシンを木探索での子ノードとみなし，次状態を子ノードから選択 (コインを投入するスロットを選択) するのが UCT となる．なお，木の成長に伴い各子ノードの平均報酬に変化が生じるため，UCB 値 (式 (2.2)) では，第 2 項にバランス調整のための定数 C が掛けられている．

以上が UCT の原理となる．この UCT を用いてゲーム木探索を行う場合，MCTS 全体の特徴である「Anytime なアルゴリズム」という点に加え，さらに以下のような特徴を持つことになる．

- ミニマックス値への収束: 探索対象の局面の真の評価値であるミニマックス値と，局面の平均報酬の期待誤差が $O(\log N/N)$ で減少し，それ以外の最適ではないノードの選択確率が 0 に収束することが証明されている [15].
- 非対称な探索木の成長: 探索途中での UCB 値からシミュレーションを行うノードを選択するため，ノードの成長や展開が非対称になる．一方，ミニマックス探索では指定した深さまで木を全て展開するため，無駄な探索が増える可能性がある．

イテレーション回数 N を増やすとミニマックス値に収束することが特徴としてあるが，実際にコンピュータプレイヤを作る際には，誤差が十分に小さくなるまで N を増やすことは難しい．そこで，ミニマックス値への収束性はなくなるが，UCB 値に対象のゲームのヒューリスティック値を加えた値をノード選択時の評価値とする Progressive Bias [20] と呼ばれる手法や，ゲーム開始直後 (序盤) では展開するノードをヒューリスティック値が高いノードだけに制限する Progressive Widening [7], Progressive Unpruning [20] 等が利用されている．これらの手法を用い，現実的な時間内で，より精度の高い探索が行われる．また，シミュレーションの結果だけでなく，シミュレーションの途中情報も探索木にフィードバックする手法である RAVE も，ミニマックス値への収束性が失われるが，その精度を高めるための手法である．

2.2 シミュレーション方策の改善

2.2.1 方策の改善

2.1.3 節でも少し触れたが，基本的なモンテカルロ木探索では，シミュレーション中に全ての着手を等確率で選択する方策を取る (以下，一様方策と呼ぶ)．この方策ではシミュレーションの精度が低く，全体としてプレイアウトの精度も悪くなる．シミュレーションは与えられた局面の平均報酬を求めるために行うものであるので，その内容もよりもっともらしい方が望ましい．囲碁を例に挙げると，例えば図 2.8 のようになる．与えられた局面に対し，一様方策でのシミュレーションが左側で，ヒューリスティックスに基づいて着手を生成した場合は右側のようになる．左側の一様方策でのシミュレーションの場合，たとえ大量の石が取られそうになっていようと，それを気にすることなく全ての合法手から等確率で着手を選択するため，左側のようになることは珍しくない．故に，右側の

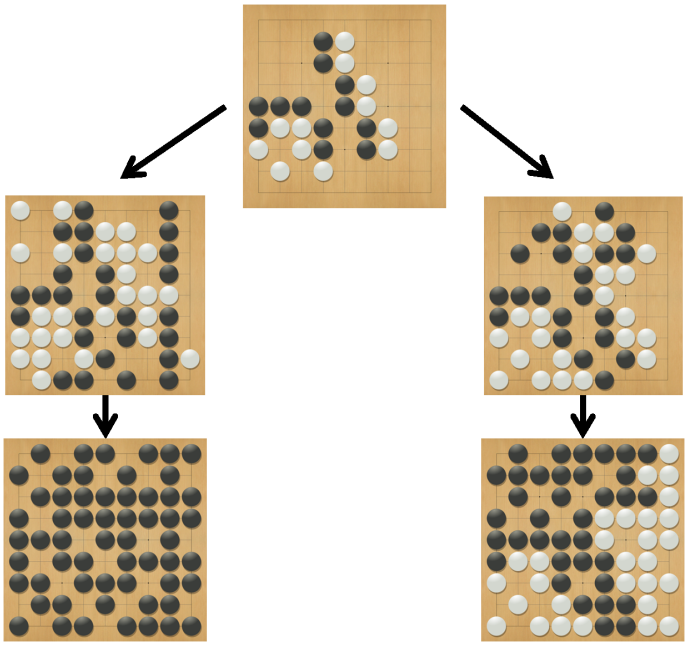


図 2.8 方策の違いによるシミュレーションの違い

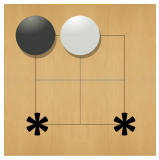
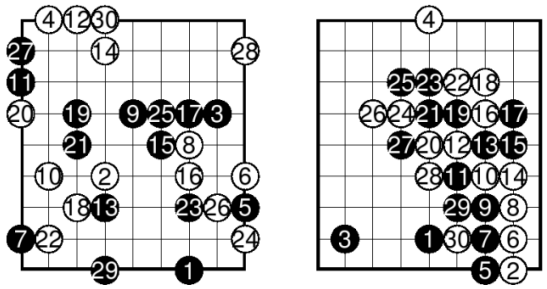


図 2.9 MoGo における「ハネ」パターン

改善されたシミュレーションの方が「よりもっともらしい勝率」を推定できるのは明らかである．実際に，このようにシミュレーション方策を改善することが全体としての性能向上に貢献することが報告されている [1, 7, 12, 28, 32]．以下，具体的に用いられる手法について説明を行っていく．また，本節における具体例は，特に断らない限り囲碁を例にする．

2.2.2 ルールベースによる改善

具体的な改善手法として，ルールベースによる方策の改善がその 1 つにある．コンピュータ囲碁プレイヤにおいて広く用いられているルールベースとして，MoGo が用いている手作りパターンによるルールベースがある [13]．図 2.9 がその一例である．図はパターンのテンプレートを示す (* は don't care を意味し，ここの状態を問わない)．「局所的な 3×3 の領域をチェックし，これに一致した場合はその中央に打つ」，というルールがこれによって構築される．また，「ここには打つべきではない」というパターンも同時に MoGo は組み込んでいる．図 2.10 が，一様方策と MoGo のルール



(a) 一様方策 (b) MoGo のルールによる方策

図 2.10 MoGo のシミュレーション方策の比較 (図は [13] より引用)

を用いたシミュレーションの比較である．図 2.10(b) の方がよりもっともらしいシミュレーションを行えていることが分かる．

以上のようなパターンによるルールに加え,「一手前の着手から近い位置に打つ」「石が取られそうな場合は逃げるように打つ」等の一般的なルールも利用している．さらに囲碁の場合，ナカデ，シチョウ，といった，特殊な場合を除いて応手が決まっている状況もあるが，シミュレーションは原則として高速でなければいけないので，計算に時間がかかるルールはシミュレーション方策に導入されないことが多い．

2.2.3 棋譜の着手を真似る手法による改善

MoGo の手作りのパターンによるシミュレーション方策の改善は強力であるが,「大量のパターン/ルールを作るのにコストがかかる」「パターンを更新するのに手間がかかる」といったデメリットがある．そのため，機械学習を用いた自動的なパターン・ルールの構築を行う手法が提案されている [1,7,28].

その中でも，Rémi が提案した「過去の棋譜を教師データとし，その棋譜中の着手を再現するように方策を学習する」手法は，Crazy Stone というコンピュータ囲碁プレイヤに実装され，大きくその棋力を向上した [7]．この手法が導入されたことにより，Crazy Stone は GNU Go[1] ver.3.6 の level 10 に対し，9 路盤での対局で一様方策に比べて勝率 30%の上昇を得られたという報告をしている．さらにシミュレーション方策だけではなく，ツリー方策の Progressive Widening にもこの手法を応用しており，それによって更に 20%の勝率上昇という結果を残している．この手法は，深い囲碁の知識を必要とせず，論文として手法が広く公開されたことも合わせ，現在様々なコンピュータ囲碁プレイヤに広く実装されている．本研究でも，この手法をベースにした手法を提案する．

この手法は，棋譜中に出てくる局面 s_i に対し，実際に行われた着手 a_i を行う確率の積 (尤度) の最大化を行うことで,「過去の棋譜に出てくる着手を真似る」方策を目指している．具体的には式 (2.5)

[1]Alpha-Beta 法によるコンピュータ囲碁プレイヤ `http://www.gnu.org/software/gnugo/`

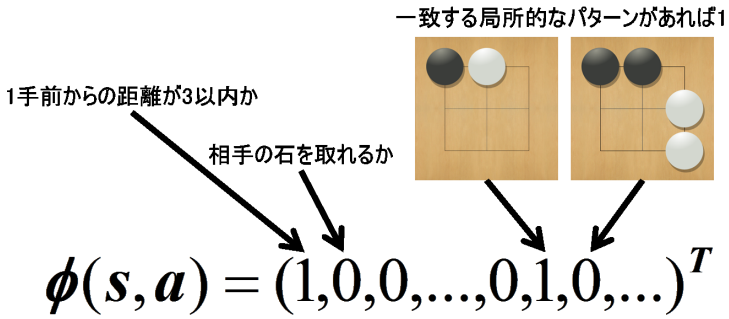


図 2.11 特徴量の例

のようになる．

$$\max \prod_{i=1}^{N} \pi_{\boldsymbol{\theta}}(s_i, a_i) \tag{2.5}$$

$\pi_{\boldsymbol{\theta}}(s_i, a_i)$ は，局面 s_i において着手 a_i を行う確率 (この確率分布関数を方策と呼ぶ) であり，$\boldsymbol{\theta}$ は方策を制御するパラメータである．N は棋譜中の局面と着手のペアの数である．ここで現れるパラメータ $\boldsymbol{\theta}$ を棋譜のデータ (s_i, a_i) から最適化することで，棋譜中に出てきた局面と同様の局面が現れた場合，棋譜で打たれた着手と同様の着手を行う確率が最も高くなる方策を得ることができる．

具体的な最適化は，式 (2.5) の対数を取った対数尤度 (式 (2.6)) を用い，θ について 1 階微分を行った値を用いて，確率的勾配降下法で最適化を行う (式 (2.7))．

$$\text{目的関数:} \qquad \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log \pi_{\boldsymbol{\theta}}(s_i, a_i) \tag{2.6}$$

$$\text{更新式:} \qquad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s_i, a_i) \tag{2.7}$$

α は学習率を示す．通常この値には，更新を繰り返すたびに小さくなり，最終的に 0 に収束するものを用いる．また，π_θ は確率分布を表すので，式 (2.8) を満たす形であれば良い．

$$\forall s, \sum_{a_i \in A_s} \pi_{\boldsymbol{\theta}}(s, a_i) = 1 \tag{2.8}$$

A_s は，局面 s でのすべての合法手を示す．この π_θ に使える形は複数あるが，次の式で表されるソフトマックス方策がよく用いられる．

$$\pi_{\boldsymbol{\theta}}(s, a) = \frac{e^{\boldsymbol{\phi}(s,a)^T \boldsymbol{\theta}}}{\sum_{b \in A_s} e^{\boldsymbol{\phi}(s,b)^T \boldsymbol{\theta}}} \tag{2.9}$$

$\boldsymbol{\phi}(s, a)$ は局面 s において a と打つ場合を，数値的なベクトルで表現した特徴ベクトルとなる．Rémi の手法では，MoGo と同様に，この特徴ベクトルに局所的なパターンの情報を組み込んでいる．ある局面 s で a と打つ手の特徴ベクトルを考えるとき，「a 周辺の 3x3 マスのパターン」を抽出する．

Algorithm 2.1 対数尤度最大化によるソフトマックス方策の最適化

```
S, A ← 棋譜中の局面と着手
θ ← 0
while 収束するまで do
    for (s_i, a_i) ∈ (S, A) do
        φ(s_i, a_i) ← (s_i, a_i) から特徴量抽出
        θ = θ + α∇_θ log π_θ(s_i, a_i)
        α を更新する
return θ
```

そして，そのパターンに対応する特徴ベクトルの次元を 1 とし，それ以外のパターンの次元を 0 とすることでパターンを特徴ベクトルで表現している．また，パターンの情報だけでなく,「1 手前からの距離」「相手の石を取れるか」等の，MoGo でも使われているルールと同種のルールも特徴ベクトルに組み込んでいる．例えば図 2.11 のようになる．1 になっている次元は，その次元のルールを満たしているか，その次元に対応するパターンが a を中心とした 3×3 パターンと一致していることを示す．また，式 (2.9) の分母は，式 (2.8) の制約を満たすための正規化項である．

ソフトマックス方策を用いると，式 (2.7) で示される更新式の一部は次のようになる．

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s_i, a_i) &= \nabla_{\boldsymbol{\theta}} (\boldsymbol{\phi}(s,a)^T \boldsymbol{\theta} - \log \sum_{b \in A_s} e^{\boldsymbol{\phi}(s,b)^T \boldsymbol{\theta}}) \\
&= \boldsymbol{\phi}(s,a) - \frac{\sum_{b \in A_s} \boldsymbol{\phi}(s,b) e^{\boldsymbol{\phi}(s,b)^T \boldsymbol{\theta}}}{\sum_{b \in A_s} e^{\boldsymbol{\phi}(s,b)^T \boldsymbol{\theta}}} \\
&= \boldsymbol{\phi}(s,a) - \sum_{b \in A_s} \boldsymbol{\phi}(s,b) \pi_{\boldsymbol{\theta}}(s,b) \qquad (2.10)
\end{aligned}$$

これを用い，棋譜中のデータに基づいて $\boldsymbol{\theta}$ を最適化する．この流れを疑似コードで表したものが Algorithm 2.1 となる．また，今後この手法によって学習された方策のことを「遷移方策」と呼ぶ．

2.2.4 その他の改善手法

2.2.3 節で述べた「棋譜に出てきた着手の真似をする」学習方法として，Maximum Entrypy Model (MEM) を用いた Araki らの手法も存在する [1]．この手法では，棋譜中で出現した局面に対し,「実際に行われた着手を正例」「それ以外の合法手を負例」として学習データを作り，そのデータを利用して MEM によって最適化を行っている．最適化の手法等は Rémi のものと異なるが，その目的は同じようになる．

Silver らは，方策による差し手を，遷移方策のように棋譜中の着手との誤差を小さくするのではなく，真のミニマックス値との誤差を小さくする手法を提案している [28]．この手法はシミュレーションバランシングと呼ばれる．ある局面 s_t での期待報酬を $\boldsymbol{E}_{\pi_\theta}[z|s_t]$，真のミニマックス値を $V^*(s_t)$

とすると，その二乗誤差 $(\boldsymbol{E}_{\pi_\theta}[z|s_t] - V^*(s_t))^2$ を最小化するように方策を学習する．シミュレーションの多様性を保ちながらその平均報酬を真のミニマックス値に近づけるという手法になるが，実際には真のミニマックス値 $V^*(s_t)$ は未知である．そのため，学習する際には，通常より長時間かけて対象局面のシミュレーションを行い，その平均報酬をミニマックス値として近似することになる．これにより，実用上シミュレーションバランシングは「長時間かけたシミュレーションの真似をする方策」を作ることになる (遷移方策は「棋譜の真似をする方策」)．シミュレーションバランシングは，特徴量に局所パターンや各種ヒューリスティックを用いたところ，5 路盤，9 路盤においては遷移方策より強いプレイヤを得られたという結果が報告されている [14, 28]．しかし，学習の際にミニマックス値を近似するために長時間シミュレーションを行う必要があるため，遷移方策に比べて学習コストが非常に高い．また，ミニマックス値の近似を行うので，その際のシミュレーションにも高い精度が求められる．これらの点から，この手法を実際に採用しているコンピュータプレイヤは見受けられない．

シミュレーションバランシングの比較手法として，Silver らは同時に強化学習を用いた方策の学習手法も提案している [28]．しかし，この手法はシミュレーションバランシングに比べ，収束速度も性能も劣る結果となっている．また，モンテカルロシミュレーションの結果を利用して強化学習を行い，そこから自動的に盤面評価関数を学習する手法もある [22]．この手法はシミュレーション方策そのものの改善ではないが，シミュレーションをうまく利用した学習を行い，プレイヤの強化を行っている (対象のゲームはリバーシ)．

特徴量として用いる局所的なパターンを，3×3 では小さすぎて特徴が捉えづらく，しかし 5×5 ではパターンの数が膨大になるのでメモリに乗り切らない，という問題に対応するために，4×4 のパターンを取り扱う手法も提案されている [30]．4×4 ではパターンの中央座標が自明ではなく，扱いづらくなる点に対処し，シミュレーション方策等に組み込んでいる．

2.3 シミュレーション方策の動的調整

2.2 節にて，シミュレーション方策を改善することによって MCTS 全体の性能を向上させる手法を説明した．これらの手法は，シミュレーションにヒューリスティックスを組み込んだり，事前に学習した結果からその方策を改善していた．それらの方策は対局中は固定である一方，事前の学習ではなく，対局中に動的にシミュレーション方策を調整する手法も提案されている．これらの手法は，対戦相手や対局固有の情報等，事前の学習やヒューリスティックスからでは導入が難しい情報をシミュレーション方策に取り入れようとしている．また，事前学習のための棋譜が殆ど存在しなかったり，ヒューリスティックスとして導入できるほどゲームに対する知識が固まっていないゲームに対しても有効な手法となる．

Rosin は，Morpion Solitaire という一人用ゲームにおいて，Nested Monte-Carlo Search (NMCS) [5] という MCTS の一種である手法に，動的にシミュレーション方策を調整する手法を組み合わせたプレイヤを提案している [25]．このプレイヤは，NMCS とモンテカルロシミュレーションによる探索の最中に，今までより良い着手が探索中に発見された場合,「よりその着手を行いやすい」ように方策を調整する．方策にはソフトマックス方策を用いており，2.2.3 節で説明した学習手法におい

て,「棋譜中の着手」と扱っている学習データを「良い着手」で置き換えた以外は同様の手法で調整を行っている．本研究とは対象が異なるが，事前の知識等を導入せずとも，動的な調整のみでプレイヤの改善を行うことが可能な手法である．

また，MCTS によるコンピュータ囲碁プレイヤで，Silver は強化学習によってゲーム中に動的にシミュレーション方策を調整する手法を提案している [27]．この手法では，1 回のプレイアウトについて，選択された局面と着手から特徴量を取り出し，学習に利用している．その際，特徴量を「永続的」な特徴量と「一時的」な特徴量とで区別しており，木探索で選択された局面と着手から行われる学習は「永続的」な方策に反映し，シミュレーションから得られたものでは「一時的」な方策に反映している．ただし，それらを利用する際は式 (2.11) のように単純に線形結合した値を着手決定に用いており，木探索かシミュレーションか，という区別は学習の際だけ行われる ($\overline{Q}(s,a)$ が着手決定に用いられる値，$\phi(s,a), \theta$ は永続的な特徴量とその重み，$\overline{\phi}(s,a), \overline{\theta}$ は一時的な特徴量とその重みを示す)．

$$\overline{Q}(s,a) = \phi(s,a)^T\theta + \overline{\phi}(s,a)^T\overline{\theta} \tag{2.11}$$

この手法では木探索とシミュレーションの結果両方を用いて方策を調整しているが，シミュレーション 1 回 1 回は，方策にそのまま反映するには信頼性が低い．一方，本研究では，同様に動的にシミュレーション方策を調整する手法を提案するが，複数のシミュレーションを合わせた結果が反映されている，探索木から取得できる情報のみを用いてシミュレーション方策を調整する手法を提案する．これにより，より信頼性の高いシミュレーション方策の動的調整を行う．

第3章　モンテカルロ木探索を用いたコンピュータ囲碁プレイヤ

本節では，モンテカルロ木探索，特に UCT を用いてコンピュータ囲碁プレイヤを作る際の基本的な事項について詳細に説明する．また，本研究では，自作のコンピュータ囲碁プレイヤと，Fuego [8] と呼ばれるオープンソースのコンピュータ囲碁プレイヤの 2 つを用いている．これらについて，共通する一般的な事項に加え，それぞれのプレイヤではどのようになっているか，といったことも説明を行う．

3.1　各種ルール

コンピュータ囲碁プレイヤを取り扱う際，いくつか囲碁のルールで気を付けなければならないものがある．特に,「モンテカルロ法を利用している」ことから注意しなければならないこともある．それらのルールについて，詳しく説明をしていく．

3.1.1　地の数え方，終局判定

地，つまり黒白それぞれの領域の広さをどのように数えるか，また，対局が終了したかどうかの判定は，人間のプレイヤが囲碁を学び始めてつまづきやすいポイントとされている．

コミを除くと，日本式ルールでは以下の式を使って地を計算する．

$$\text{地} = (\text{自陣の空白領域} - \text{死んだ石の数}) \tag{3.1}$$

相手が取った石 (アゲハマ) は，死んだ石の数，に入っている．一方，中国式ルールの場合は以下のようになる．

$$\text{地} = (\text{自陣の空白領域} + \text{生きている石の数}) \tag{3.2}$$

どちらのルールを用いても基本的に結果に影響が出ることは無いが，コンピュータプレイヤでは中国式ルールを用いることが多い．これは,「終局状態から領域を埋めても結果に変化がない」という点から，終局判定が容易になるのが理由である．例えば，図 3.1 を見ると分かるように，中国式ルールの場合は「結果が変化しない状態」と,「そこからお互い自陣を埋めていき，パス以外の合法手がなくなった状態」どちらでも結果に差が出なくなる．一方，コンピュータで「これ以上お互いの地が変化するかしないか」によって終局を判定するのは難しいが,「合法手がパス以外にあるかどうか」

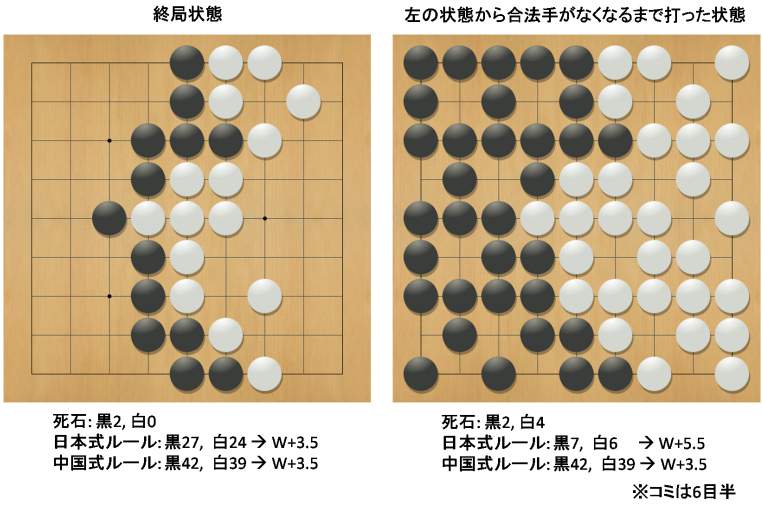


図 3.1 ルールの違いによる結果の違い

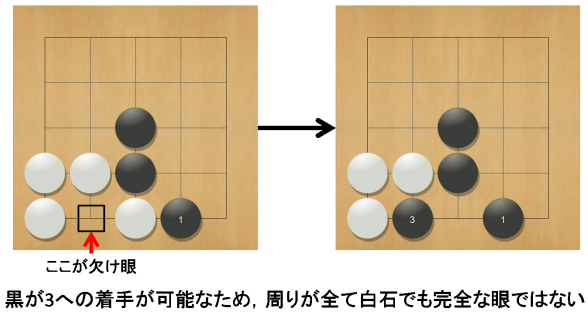


図 3.2 欠け眼

によって終局を判定するのは容易である．この,「最後まで打っても結果が変化しない」「最後まで打つと終局判定が容易」という 2 つの点から，MCTS を用いたコンピュータ囲碁プレイヤでは，お互い合法手がなくなるまでシミュレーションを行い，その後中国式ルールで勝敗を計算する，という手法を利用しているプレイヤが多い．特に，シミュレーションは高速である必要があるため，終局判定にコストをかけるのは望ましくない．その点からも理にかなっている方法である．本研究で用いるプレイヤも同様の手法を採用している．

3.1.2　眼について

周りを全て自分の石で囲われた場所を「眼」と呼ぶ．この眼は，欠け眼 (図 3.2) である場合を除き，自分の眼に打ち込むことにメリットは何もない．しかし，囲碁のルール上，自殺手 (打った瞬間相手に石が取られてしまう着手) を除き，眼に着手を行うことは禁じられていない．つまり，正しく

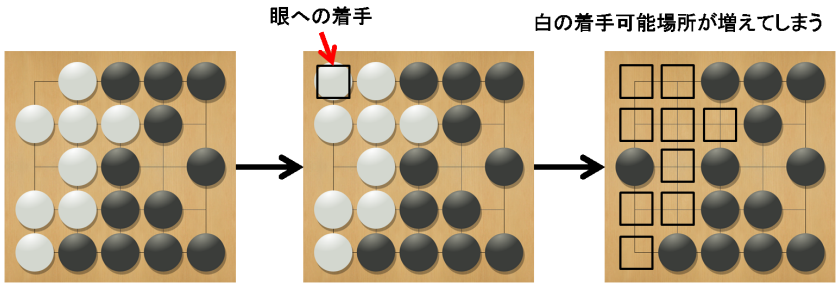


図 3.3 眼への着手による終局までの手数の増加

囲碁のルールに追従するのであれば，木探索やシミュレーション内でも眼への着手を考慮することになるのであるが，通常コンピュータプレイヤでは眼への着手は禁じ手として取り扱う．前述したように，自分が不利になるだけだからという理由もあるが，シミュレーションではお互いパス以外に合法手がなくなるまで打ち続けるので，眼への着手を認めると「シミュレーションの終局率が極端に下がる」もしくは「終局までの手数が極端に増える」からである．例えば，図 3.3 のように，シミュレーションの最中に，二眼ある連 (石の固まり) の眼以外への着手が存在しなくなったとする．この時，眼への着手を認めると，その後着手可能な場所が増えてしまう (眼への着手を認めなければ合法手が存在しないのでシミュレーションが終了する)．その上，黒 15 目，白 10 目，という結果であったのが，黒 25 目，白 0 目となってしまい，正しいとは言えないシミュレーション結果になってしまう．このようなことが発生するため，欠け眼を除く眼への着手は禁止とするのが一般的であり，本研究で用いるプレイヤでもそのようにしている．

3.1.3　その他ルール

これまで説明したルールが，MCTS を用いたコンピュータ囲碁プレイヤを作る上で特に注意が必要なルールとなる．その他にも，いくつか注意すべき細かいルールがある．以下，本研究でそれらのルールをどのように取り扱うかを示す．

- コミ
  コミは 2013，2014 年段階で通常用いられている 6 目半を使用している．
- 同一局面の出現
  通常のコウ以外でも，稀にだが三コウ[1]や長生[2]といった，どちらかが引かない限り無限に同一局面を繰り返す場合がある．対局中に同一局面が一定回数以上発生した場合，その対局は無勝負としている．
- 置き碁
  本研究では置き碁は対象外とする．

[1] http://ja.wikipedia.org/wiki/コウ#.E4.B8.89.E3.82.B3.E3.82.A6
[2] http://ja.wikipedia.org/wiki/長生

3.2 コンピュータ囲碁での UCT

本節では，本研究で使用したプレイヤについて，UCT のそれぞれのステージの詳細について説明を行う．

3.2.1 木探索

MCTS の木探索部分の方策であるツリー方策には，2.1.4 節で説明した UCB 値が使われる．UCB を用いる際，パラメータ C の設定が必要となる．これは，プレイヤ全体の性能を左右するパラメータであるので，その都度調整を行う．

また，子ノードを展開する条件，未探索ノードの扱いをどうするか，というのも定める必要がある．子ノードを展開する条件は，リーフノードであり，かつ木探索による訪問回数が一定以上になった場合，としている．「何回訪問したら」という部分はプレイヤの性質によって変わる部分ではあるが，一定回数以上で子ノードを展開するというのはよく用いられる方策である．また，子ノードを展開した直後は，すべての子ノードは未訪問状態であり，式 (2.2) の値を計算できない (分母が 0 になる)．この問題に対処する方法としては以下のような方法がある．

- 子ノードの中で未訪問ノードは優先的に訪問する
- 式 (2.2) の $\overline{X_i}, N_i, N$ に初期値を与える [12]
- バイアスを加える，第 2 項を消す，等 UCB の定式を変える

1 つ目と 2 つ目の方式がよく用いられるが，3 つ目の「評価に用いる値そのものを変える」という手法も提案されている [4]．本研究では，自作のプレイヤも Fuego でも 1 つ目の手法を用いている．

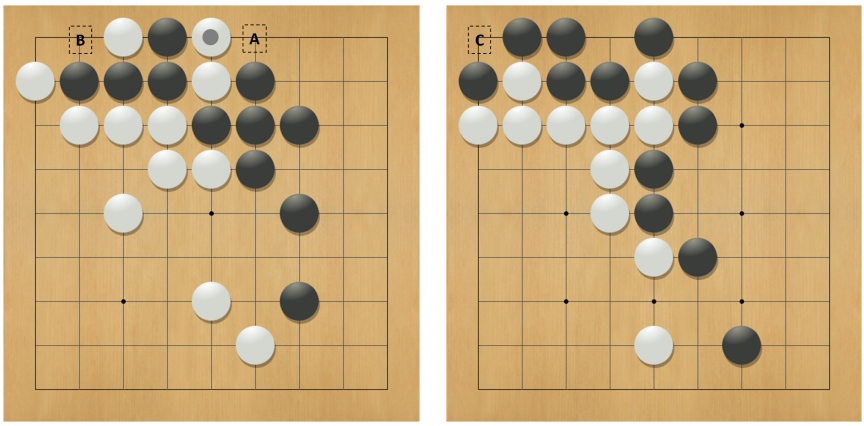
3.2.2 シミュレーション

シミュレーションでは，ツリー方策により選択されたノードをルートとし，ゲームが終局に達するまで自己対戦を行う．

本研究で用いる自作のプレイヤには，2.2.3 節で説明した「棋譜中の着手を真似る」ように学習を行った方策を用いる．学習方法に関しては若干の改善を行っている．この改善は，「特徴量の重み $\boldsymbol{\theta}$ の絶対値が大きくなり過ぎないようにする」という改善である．例えば，とある 3×3 パターンが学習データ中で 1 回だけ出現し，かつそこへの着手が行われていたとする．すると，特徴ベクトルの重みの更新量を示す，式 (2.10) のそのパターンの次元は常に 0 より大きい値になる．つまり，学習を続けるごとに重み $\boldsymbol{\theta}$ のその次元の値がどんどん大きくなってしまう (それこそ無限大に)．しかし，実際には，「たまたま学習データがそのようになっていた」だけであり，そのパターンが出現した時に必ずそこに打つのが正しいわけではない．この現象は「過学習」と呼ばれる現象であり，学習

表 3.1 用いた特徴量

1	着手周辺の 3×3 パターン (反転，回転は同一視する)
2	新たにアタリ状態になった連を，敵の石を取ることでアタリ解消
3	直前に相手が自分の石を取り，自分がその石を取りかえす
4	2, 3 以外の石取得
5	新たにアタリ状態になった連を，ノビによってアタリ解消
6	自ら自分の連をアタリ状態にする (self-atari)
7	コウが存在するときにアタリを作る
8	6, 7 以外のアタリ
9	1 手前からの距離が 3 以下かどうか (距離 D は以下で定義) $D := \|dx\| + \|dy\| + \max(\|dx\|, \|dy\|)$
10	2 手前からの距離が 3 以下かどうか



(a) 特徴量 2 の例　　(b) 特徴量 6 の例

図 3.4 特徴量の例

データに必要以上にフィッティングしてしまうことを指す．これを改善するために，式 (2.6, 2.7) で示される目的関数と更新式を，以下の式 (3.3, 3.4) のように変更する．

$$\text{目的関数：} \quad \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log \pi_{\boldsymbol{\theta}}(s_i, a_i) - \frac{\lambda}{2}|\boldsymbol{\theta}|^2 \tag{3.3}$$

$$\text{更新式：} \quad \boldsymbol{\theta} = \boldsymbol{\theta} + \alpha \left\{ \Delta_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s_i, a_i) - \lambda \boldsymbol{\theta} \right\} \tag{3.4}$$

式 (3.3) の第 2 項の $-\lambda/2|\boldsymbol{\theta}|^2$ が，新たに追加された項である．式が表す通り，$\boldsymbol{\theta}$ の絶対値が大きくなり過ぎると，最大化したい式全体の値が小さくなってしまうため，$\boldsymbol{\theta}$ が大きくなり過ぎないように学習が行われる．これによって過学習を抑制できる．これは「正則化」と一般に呼ばれるテクニックであり，追加された第 2 項は「正則化項」と呼ばれる．λ はどのくらいそれを抑制するかを制御する

パラメータである．λ が大きいほど $\boldsymbol{\theta}$ の絶対値が大きくならないように学習を行い，逆に 0 に近づくと正則化無しの学習と同じようになる．これを確率的勾配法によって最適化すると，その更新式が式 (3.4) となる．この更新式は，データ 1 つ毎の更新式を示し，α は更新量を制御する学習率を表している．

また，学習データには Kombilo[3] の 2013 年 4 月までの棋譜のうち，両対局者の棋力が KGS Go Server (KGS) 基準で 6d 以上であり，かつ置き碁でない棋譜のみを利用して行った．局面と着手のペアから抽出される特徴量は，表 3.1 のようになる．これらの特徴量は，Rémi が提案した手法で用いている特徴量とほぼ同じであり，シミュレーション方策に一般によく用いられる特徴量である．2 の特徴量は，図 3.4(a) のような状況 (灰色の丸が乗っている白石が直前の着手) で，A か B に黒が打って白石を取り，アタリになっている連を助けるような状況を指す (着手 A，B についてこの特徴量が有効になる)．また，6 の特徴量は，図 3.4(b) で黒が C に着手を行うような場合である (着手 C についてこの特徴量が有効になる)．これは，C へ着手してしまったがために，左上の連がアタリになり，次の白の着手で取られてしまうという状況である．

一方，Fuego はルールベースによるシミュレーションになる．以下のようなルールを上から順に適用していくことでシミュレーションでの着手を生成する．

(1) ナカデが存在すればナカデを打つ

(2) 直前の相手の打った石がアタリ状態ならそれを取る着手を打つ

(3) 直前の相手の着手によって自分の連がアタリになったのなら，その連を助ける (逃げるか相手の石を取る) 着手を打つ

(4) 直前の相手の着手の近隣の自分の連のうち，呼吸点 (連に隣接している石が無い点) が 2 個の連の呼吸点に打つ (ただし self-atari は避ける)

(5) 直前の着手点周辺，2 手前の着手点周辺を見て，パターンデータベースに一致するものがあればそこに打つ

(6) 石を取る着手があれば打つ

(7) 完全にランダムに打つ

(8) パスをする

実際には，これらに加え，生成された着手の修正 (self-atari 等を避ける) も行っている．

また，UCT ベースのコンピュータ囲碁プレイヤで問題になる点として，シミュレーションの報酬をどのようにするか，という点がある．今までの説明では,「黒が勝てば 1，負ければ 0 もしくは-1」を報酬としていた．一方で，囲碁では結果として「何目差か」という情報を使うことができ (「平均勝率」が「平均目数差」になる)，実際にモンテカルロ法を用いた初期の頃のコンピュータ囲碁プレイヤでは報酬をそのように扱っていた．しかし，それでは中々棋力が向上せず，報酬を「勝てば 1，

[3] http://www.u-go.net/gamerecords/

負ければ 0 か-1」に変えたところ，棋力が向上したと報告されており [6]，現在ではその形が一般的となっている．本研究では，自作プレイヤでは「勝てば 1，負ければ 0」という報酬にし，Fuego では「勝てば 1，負ければ-1」としている．両者の間に差はほぼ存在せず，引き分けの取り扱い方程度の差となる (引き分けを扱う場合は Fuego の形が良い).

第4章　局所性を利用したシミュレーション方策の動的調整

4.1　手法の概要

本章では，対局中に動的にシミュレーション方策を調整することでシミュレーション方策の改善を行う手法を提案する．ここで提案する手法は,「囲碁の局所性」を利用し,「木探索中，ある局面に対してよく選択される着手があった時，後のシミュレーションで局所的に見て同様な局面が現れた場合，同様な着手をより打ちやすくするように方策を調整する」という手法になる．

具体的に例を交えて説明する．例えば，図 4.1 のルートノードの局面について探索を行う場合を考える．まず，図 4.1 の 1 のように，UCT の M 回目のプレイアウトにおいて，局面の左上の領域 (青く囲われている領域) に対して，赤い領域に移るように着手が行われ，その段階から 2 のようにシミュレーションが行われたとする．そのシミュレーションが終わった後，3 で示しているように，木探索によって選択された局面と着手 (青い領域が赤い領域へとなる一連の局面と着手) について，これらを表した特徴量を用いて方策 ($\boldsymbol{\theta}$) が調整される．この時,「その特徴量と同じような特徴量が出た場合，より着手確率が高くなる」ように方策を調整する．続いて，後の $N(> M)$ 回目のプレイアウトの木探索で 4 のように局面の右側への着手が選択され，5 のようにシミュレーションが開始されたとする．この 5 のシミュレーション開始時，左上の領域を見ると，局所的には 1 の木探索の開始時と同じような状況になっている (青く囲われた領域)．さらに，5 のシミュレーションを行う時は,「3 で調整された方策 $\boldsymbol{\theta}$」を使ってシミュレーションを行うことになる．前述したとおり，この方策は「青い領域 (それに似た特徴量を持つ場所) に対しては，赤い領域になるような着手を生成する確率が以前より高い」ようになっている．つまり，図中の 5 で表しているように，青い領域に関しては M 回目の木探索で行われた着手と同様の着手が生成されやすくなる．このような流れで,「木探索でよく選択されていた着手を，後のシミュレーションでより打ちやすくする」ようにする．

以上が手法の概略である．UCT は探索が進むにつれて，より有望な着手についてより重点的にプレイアウトを行うようになる．つまり，有望な着手は，上記の手法によってシミュレーションに反映される回数が多くなる．これによって,「木探索上で有望とされている着手をシミュレーションに伝える」ことができる．また，囲碁においては，大局的な戦いばかりではなく，局所的な戦いが盤面上で何か所か発生することが珍しくない．そのため，図 4.1 のように，左側の領域のやり取りと右側の領域のやり取りをそれぞれ独立して考えてもあまり問題がない場面は珍しくないと考えられる．そこで，一度有望とされた着手は多少手順が違っても変わることは無い，という仮定で，上記のような手法が有効であると考え，これを提案する．この考えは，シミュレーションの状態遷移情報を木探索

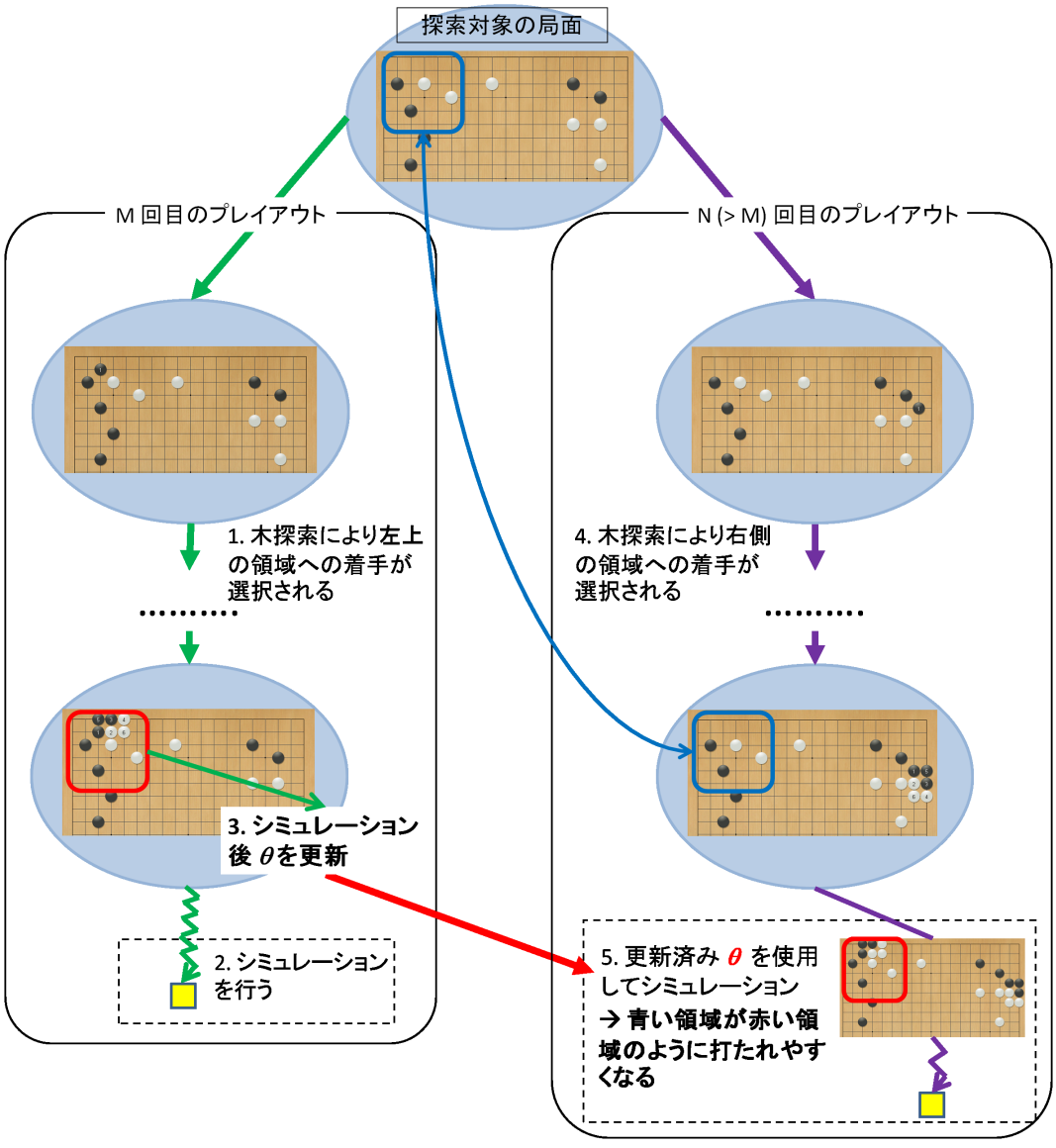

探索対象の局面
M 回目のプレイアウト
N (> M) 回目のプレイアウト
1. 木探索により左上の領域への着手が選択される
4. 木探索により右側の領域への着手が選択される
3. シミュレーション後 θを更新
2. シミュレーションを行う
5. 更新済み θ を使用してシミュレーション
→ 青い領域が赤い領域のように打たれやすくなる

図 4.1 提案手法の流れ

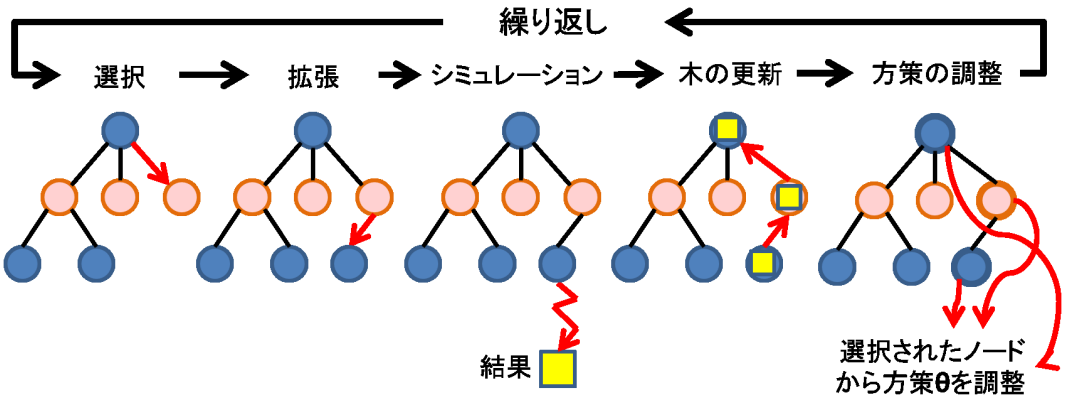


図 4.2 方策の動的調整を組み込んだ UCT

に用いる RAVE と同様の考えであり，RAVE と逆方向 (RAVE は「シミュレーション → 木探索」. 本手法は「木探索 → シミュレーション」) の考えである本手法も有効にはたらくと期待できる.

しかし，UCT は未探索のノードはある程度まで優先的に探索を行うため，常に木探索によって選択された着手をシミュレーション方策に反映していては，大量のノイズをシミュレーション方策に加えてしまう結果となる．そのため，今回は「一定回数以上木探索で選択された着手 (探索木のノード)」のみを調整対象とすることで，このノイズの問題に対応する.

4.2　方策の調整アルゴリズム

4.1 節で説明した概略について，より詳細にそのアルゴリズムを説明する.

方策の調整は，UCT のイテレーションの最後に組み込む．通常は，選択 → 展開 → シミュレーション → 木の更新，が 1 回のイテレーションとなる．ここに方策の調整を組み込むと，図 4.2 のように，木の更新の後に「方策の調整」が追加される.

続いて，方策のパラメータである $\boldsymbol{\theta}$ の具体的な調整方法を説明する．前述したように，本手法では「UCT の探索中に選択された手をより打ちやすくするように方策を調整する」という方針をとる. その目的関数は式 (4.2) となる.

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \left\{ \left(\prod_{m=1}^{M} \pi_{\boldsymbol{\theta}}(s_m, a_m) \right) \cdot e^{-\frac{\lambda}{2}|\boldsymbol{\theta}-\boldsymbol{\theta}_0|^2} \right\} \tag{4.1}$$

ここで，(s_m, a_m) はそのプレイアウト中の木探索で選択され，調整に用いられることになった局面と着手のペアであり，M は対象になった数である．$\pi_{\boldsymbol{\theta}}$ がそれらの局面と着手に対する着手確率であり，$\prod \pi_{\boldsymbol{\theta}}(s_m, a_m)$ でそれらの確率の積 (尤度) の最大化を行っている．これを最大化するような $\boldsymbol{\theta}$ を計算することで，対象となった局面と着手のペアについての着手確率を上げる．また，$\boldsymbol{\theta}_0$ は $\boldsymbol{\theta}$ の初期値を示し，後ろの $\exp(-\frac{\lambda}{2}|\boldsymbol{\theta} - \boldsymbol{\theta}_0|^2)$ の項は，「$\boldsymbol{\theta}$ の初期値 $\boldsymbol{\theta}_0$ からあまり離れすぎないように

調整する」ための項である．これは，棋譜から学習を行う際に用いた「正則化項」と同様の考え方である．これについては，実際に計算を行う際に用いる，目的関数の対数を取った形 (式 (4.2)) で説明を行う．

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \left\{ \sum_{m=1}^{M} \log \pi_{\boldsymbol{\theta}}(s_m, a_m) - \frac{\lambda}{2} |\boldsymbol{\theta} - \boldsymbol{\theta}_0|^2 \right\} \tag{4.2}$$

この式を見ると，第一項で確率の対数の和 (対数尤度) の最大化を行うと同時に，第二稿で「$\boldsymbol{\theta}$ と $\boldsymbol{\theta}_0$ の差の最小化」を行っている．これにより，$\boldsymbol{\theta}$ が初期値 $\boldsymbol{\theta}_0$ からあまり離れすぎないように方策の調整が行われる (正則化の際は,「$\boldsymbol{\theta}$ の絶対値が大きくなり過ぎないように」この項を用いていた)．初期値として $\boldsymbol{\theta}$ に棋譜から学習したデータを与えた時，このような「初期値から離れすぎないように」という制限を加えずに動的な調整を行うと，方策が初期値からかけ離れた方向へと変わる可能性がある．既存の棋譜から学習した結果と，対局中に学習した結果をうまく組み合わせるために重要な項となる．

実際に調整を行う際は，最急降下法により，式 (4.2) の微分をとった式 (4.3) を用いる．微分した値を用いてパラメータを更新することで，目的関数の値を上昇させる．

$$\Delta_{\boldsymbol{\theta}} = \frac{d}{d\boldsymbol{\theta}} \sum_{m=1}^{M} \log \pi_{\boldsymbol{\theta}}(s_m, a_m) - \lambda(\boldsymbol{\theta} - \boldsymbol{\theta}_0) \tag{4.3}$$

今回は方策にソフトマックス方策を用いるため，式 (2.10) を利用してさらに式を展開することによって次のようになる．

$$\Delta_{\boldsymbol{\theta}} = \sum_{m=1}^{M} \left(\boldsymbol{\phi}(s_m, a_m) - \sum_{b \in A_s} \boldsymbol{\phi}(s_m, b) \pi_{\boldsymbol{\theta}}(s_m, b) \right) - \lambda(\boldsymbol{\theta} - \boldsymbol{\theta}_0) \tag{4.4}$$

$$\pi_{\boldsymbol{\theta}}(s, a) = \frac{e^{\boldsymbol{\phi}(s,a)^T \boldsymbol{\theta}}}{\sum_{b \in A_s} e^{\boldsymbol{\phi}(s,b)^T \boldsymbol{\theta}}}$$

A_s は局面 s_m での可能な着手の集合を表す．この値を方策の調整フェーズで計算し，式 (4.5) で更新する．

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha \Delta_{\boldsymbol{\theta}} \tag{4.5}$$

α は調整量を制御するためのパラメータである．以上のような流れで，UCT の各イテレーションごとに方策を調整していく．

Algorithm 4.1 に全体をまとめた流れを疑似コードで示す (報酬の反映を示す BackUp 関数等，一部の共通処理については省略している)．SearchUCT が最初に呼ばれる関数であり，探索対象の局面 s_0 と現在の方策の重み $\boldsymbol{\theta}$ を引数で受け取り，その局面の最善手と新しく調整された方策を返している．22 行目から始まる SimulationWithPolicy 関数が，実際にソフトマックス方策に基づいてシミュレーションを行う部分である．ここでは，簡単のために 25 行目のように「全ての着手について式 (2.9) を計算」としているが，実際にこれをそのまま実装するとシミュレーションの実行速度が著しく低下する．特徴量の抽出処理と，指数関数の計算コストが高いためである．高速なシミュレーションを実現するためには,「特徴量が変化し得る場所」を過去の着手から特定し，変わり得る特徴量

Algorithm 4.1 局所性を利用したシミュレーション方策の動的調整

```
1: function SearchUCT(s_0, θ)
2:     while until no left time do
3:         s_l, seq ← TreeSearch(s_0)
4:         reward ← SimulationWithPolicy(s_l, θ)
5:         BackUp(s_l, reward)
6:         θ ← TunePolicy(seq, θ)
7:     return BestMove(s_0), θ
8: function TreeSearch(s)
9:     seq ← []
10:    while s is nonterminal do
11:        if visited(s) ≤ threshold then
12:            return child, seq
13:        else if v not fully expanded then
14:            child, action ← Expand(s)
15:            seq.append(s, action)
16:            return child, seq
17:        else
18:            child, action ← MaxUcbChild(s)
19:            seq.append(s, action)
20:            s ← child
21:    return s, seq
22: function SimulationWithPolicy(s, θ)
23:    prev ← NULLMOVE
24:    while s is nonterminal do
25:        probs ← Calculate (2.9) for all legal moves in s
26:        move ← Select a move according to probs
27:        if prev == PASS and move == PASS then
28:            break
29:        s = Move(s, move)
30:    return EvaluatePosition(s)
31: function TunePolicy(seq, θ)
32:    target_seq ← []
33:    for (s_m, a_m) ∈ seq do
34:        if visited(s_m) > threshold then
35:            Add (s_m, a_m) to target_seq
36:    Calculate Δ_θ from target_seq
37:    θ ← θ + αΔ_θ
38:    return θ
```

だけ計算して値を更新する方法 (差分更新) が用いられる．本手法を実装したプレイヤも差分更新によって計算しているが，その詳細については割愛する．次に，31 行目からの TunePolicy が提案手法の部分であり，ここで「UCT で選択された (報酬を得た) 手順に関して $\Delta_{\boldsymbol{\theta}}$ で $\boldsymbol{\theta}$ を調整」している．ただし，34 行目で示しているように，訪問回数の多いノードの方がより信頼できると考え，訪問回数が一定回数を超えたノードのみを調整に用いている．

4.3 評価

本節にて，手法の評価を行った結果を示す．評価は，ベースラインプレイヤとの自己対戦，GNU Go ver. 3.8 との対戦にて行った．

4.3.1 実験設定

今回の評価では，9 路盤上での対戦実験を行った．用いたプレイヤは以下の 3 つである．

- GNU Go: GNU Go ver. 3.8 level 10
- Static: 3 章にて説明した，ソフトマックス方策を用いた自作の UCT ベースの囲碁プレイヤ
- Dynamic: 評価する手法を実装したプレイヤ．ベースは Static と同じであり，方策調整の部分だけ異なる．方策の初期値には Static と同じ重みベクトルを用いる

Static が性能評価のためのベースラインプレイヤとなる．

これらのプレイヤで，Static と Dynamic の各種パラメータを表 4.1 のように設定をした (両プレイヤの共通パラメタ)．これらのパラメータは，Static と GNU Go との対戦によって事前に決めたものを用いている．本章で提案している動的なシミュレーション方策の調整は，シミュレーションを実行する時間に比べてかかる時間がわずかであったため，両プレイヤとも同じシミュレーション回数とした．

表 4.1 Static と Dynamic の各種パラメタ

パラメータ名	設定値
1 手当たりプレイアウト回数	九路盤: 5,000　十三路盤: 2,000
C (UCT のパラメータ)	0.3
ノード展開のための訪問回数	50
定石	Fuego が用いている定石と同様のもの
動的調整の対象とするための訪問回数 (Dynamic のみ)	50

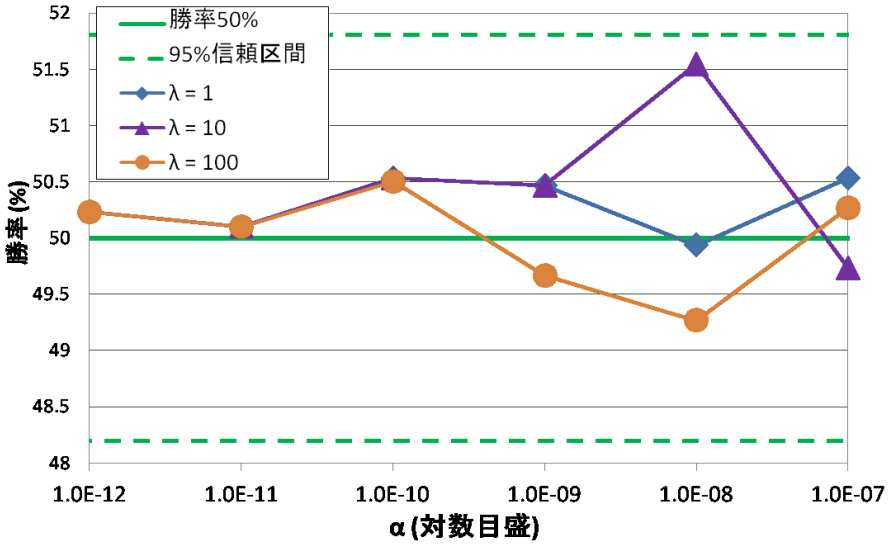


図 4.3 Static との対戦結果 (九路盤)

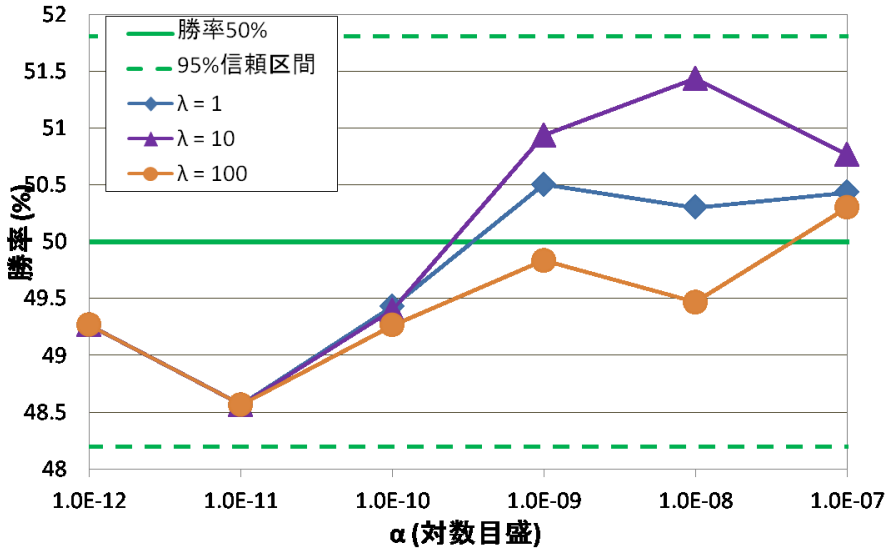


図 4.4 Static との対戦結果 (十三路盤)

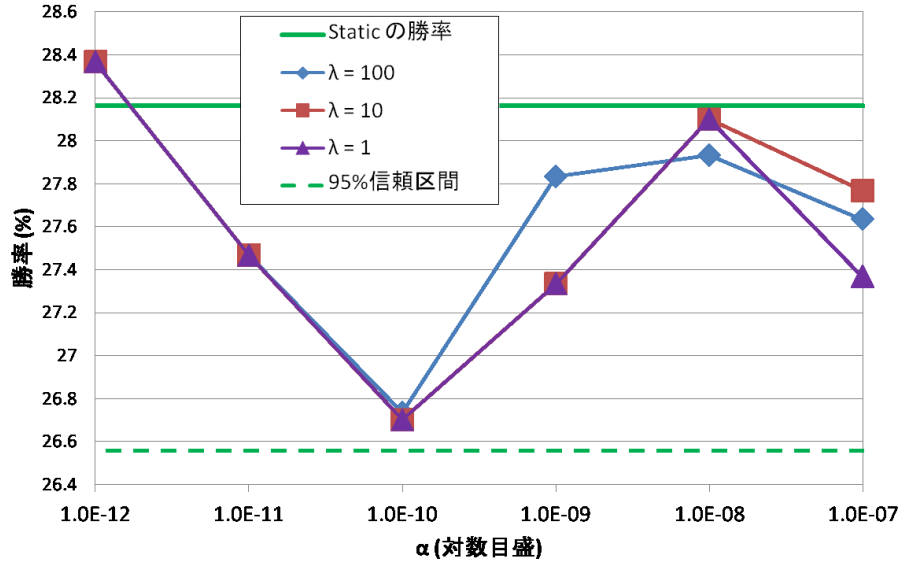


図 4.5 GNU Go との対戦結果

4.3.2　自己対戦実験

Static と Dynamic の直接対戦による実験の結果を図 4.3 と図 4.4 に示す．前者が九路盤での結果であり，後者が十三路盤での結果となる．横軸は Dynamic のパラメータである α を対数目盛りで示しており，縦軸は Dynamic の勝率となっている．勝率は，先手後手を入れ替えて合計 3,000 局の対戦を行った結果となっている．また,「95%信頼区間」としているのは，勝率を二項分布とみなした場合の有意水準 95%の境界を示す値であり，勝率の標準偏差を σ とした時の $\pm 2\sigma$ と近い値になる．

この結果を見ると，九路盤での対戦においては，$\lambda = 10$ の時に $\alpha = 10^{-8}$ あたりで Dynamic が勝ち越す傾向を示している．しかし，統計的に有意な勝率差では無く，ベースラインより強くなっているとは言えない結果になった．また，$\lambda = 1$ と $\lambda = 100$ の時は，勝ち越す傾向を示しておらず，ベースラインと同等程度の性能となる傾向を示している．

十三路盤での対戦も同様の傾向を示しており，$\alpha = 10^{-9}$〜10^{-8} あたりで $\lambda = 10$ については勝ち越す傾向となっているが，統計的に有意な差とはなっていない．また，$\lambda = 10$ の時が勝率が高めになり，次いで $\lambda = 1, 100$ という順番になっているのも九路盤と同様の傾向となっている．

4.3.3　GNU Go との対戦実験

Static，Dynamic それぞれと，GNU Go ver. 3.8 の level10 を対戦させた結果を図 4.5 に示す．GNU Go との対戦結果は九路盤の結果だけとなる．また，図 4.3 と同じく，横軸が Dynamic の α を対数軸に並べたもの，縦軸が勝率となる．対戦数は，Dynamic は先手後手入れ替えて 3,000 局対戦した結果，Static は先手後手入れ替えて 10 万局対戦した結果である．

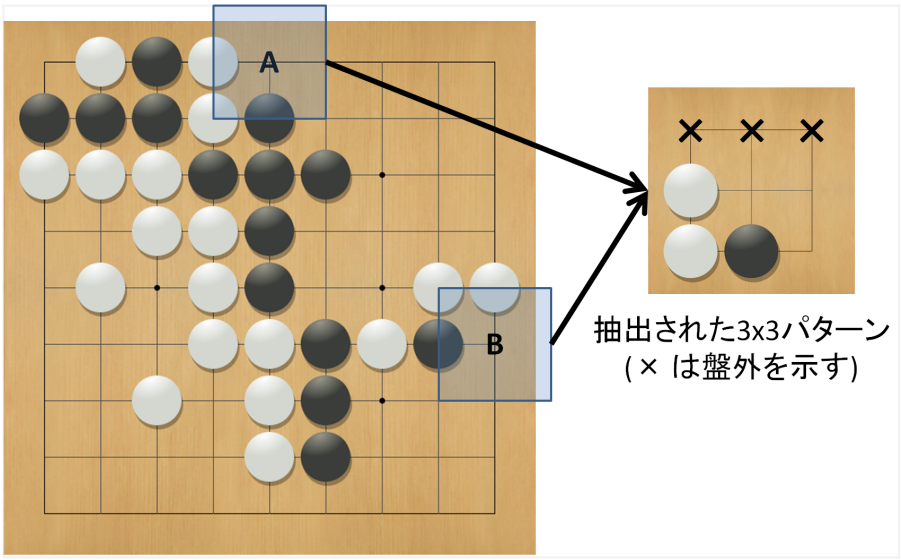


図 4.6 動的調整が悪影響を及ぼしてしまう例

自己対戦結果と同じく，$\alpha = 10^{-8}$ 近辺で他の部分に比べて勝率が高くなる傾向を示してはいるが，全体的にパラメータに関わらずベースラインより勝率が下回る傾向となっており，統計的に有意な差はついていないが，$\alpha = 10^{-10}$ で勝率が最も低くなっている．

4.4 考察

4.3.2 節で行った自己対戦の結果では，ベースラインを上回る傾向を示したものの，統計的には有意な差が見られなかった．また，4.3.3 節で行った GNU Go との対戦結果からは，全体的にベースラインを下回る傾向を示した．本手法では，独立して考えられる局所的な戦いが複数起こることを期待している．そのため，一か所で発生した戦い等が全体に波及しやすい九路盤ではこの結果は妥当なものであるが，九路盤より局所的な戦いが発生しやすいと思われる，十三路盤における自己対戦でも有意な差を得ることができなかった．

以上のようにうまくいかなかった理由として，ソフトマックス方策に用いていた特徴量が，本提案手法の調整方法に適していなかった，という理由が考えられる．局所的に見て状況が似ているような場合 (同様の特徴量になるような状況) が後に現れた場合同様に打つ，という目的では，「同様の特徴量になる」範囲が大きい場合，対象としたい局所範囲とは異なる場所まで打ちやすく調整を行っている可能性がある．これは，例えば図 4.6 のような場合に悪影響となってしまう．ここでは 3×3 のパターンだけを特徴量として考える．木探索上で着手 A の選択が多く行われたとすると，本手法では，後のシミュレーションで対象のパターン (図で示されている，抽出された 3×3 パターン) が出てきた時はその着手確率が以前より高くなる．手法の意図としては，後のシミュレーションで A への着手確率を高めることであるが，このパターンが存在するのは A だけではなく，B にも存在す

る (パターンの回転，反転は同一視される)．本手法の調整調整方法では，この A と B どちらへの着手確率も高くなってしまう．しかし，この局面において，A への着手は正しいが，B への着手は悪手となる．つまり，本来着手確率を上げたい場所だけでなく，着手確率を上げるべきではない場所まで上げてしまうことになる．実際には 3×3 パターン以外にも特徴量を用いている (表 3.1 参照) ので，完全に A への着手と B への着手の特徴量が同一にはならないが，それでもパターン部分の特徴量に関しては一致するので，やはり B への着手確率が上がってしまう．このような現象も含んでいる手法のため，本手法の性能がベースラインに比べて上がらなかったと考えられる．

次章では，この結果をさらに検証するために，着手位置をそのまま特徴量とするプレイヤで検証実験を行った．これは,「局所的に似ている状況が現れたら着手確率を上げる」という抽象的な方策ではなく,「良いとされた着手位置そのものへの着手確率を上げる」というシミュレーション方策となる．また，本手法では，木探索でより選択された着手を調整に用いるという「調整に用いる着手の選択」と,「選択された着手に基づいた方策の調整」を同時に行っていた．これでは，性能改善につながらなかった理由がどちらにあるか分からないため，検証実験では「良い着手が事前に分かっている」という設定で「選択された着手に基づいた方策の調整」を行うプレイヤを扱った．

第５章　最善手情報が既知である設定での検証

5.1　特定着手の着手確率変更のみに注目した検証実験

4 章で提案した手法を対戦実験で評価した結果，ベースラインプレイヤを勝率で上回ることはできなかった．その結果について考察を行ったが,「調整に用いるノードの選択」と「方策の調整」を同時に行っていることにより，それぞれがどのように影響しているか，詳細に検証することができなかった．そこで，それらを切り離して検証するための実験を行った．

本章では,「現局面での最善手が既知」であると仮定して，その情報を利用してシミュレーション方策を調整した場合，性能改善につながるかを検証する．また，4.4 節で述べたように，方策に用いていた特徴量によって望まない方策の調整が発生し，性能が改善しなかったと考えられる．そこで，方策にソフトマックス方策を用いず,「指定した位置の着手確率を上げる」方策を導入したプレイヤによって検証を行った．このプレイヤの性能を検証することで,「良い手を後のシミュレーションでより打ちやすくなるように方策を調整」すること自体が正しい方向性であるかを検証することが可能になる．

具体的にどのようにして「良い着手が既知」な状態を得るかであるが，今回は，実際に探索を行う時の 10 倍のプレイアウト回数で事前に探索を行い，そこで得られた最善手を「既知である最善手」として扱う．また,「指定した位置の着手確率を少し上げる」方策は，図 5.1 のように，碁盤と対応付けた二次元の重みテーブルによって実現する．長時間かけて UCT を行った結果得られた最善手の位置だけこのテーブルの値を少しだけ上げ，実際にシミュレーションを行う際は，このテーブルの値を

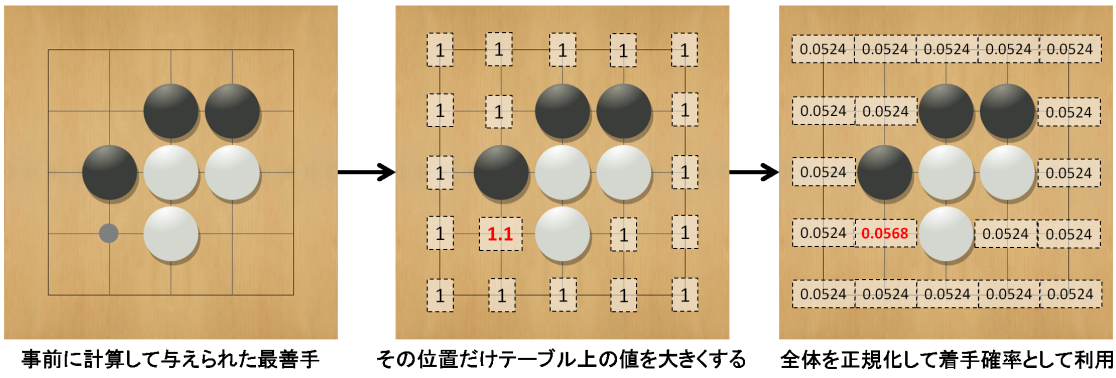


図 5.1 特定の着手確率を上げるシミュレーション方策

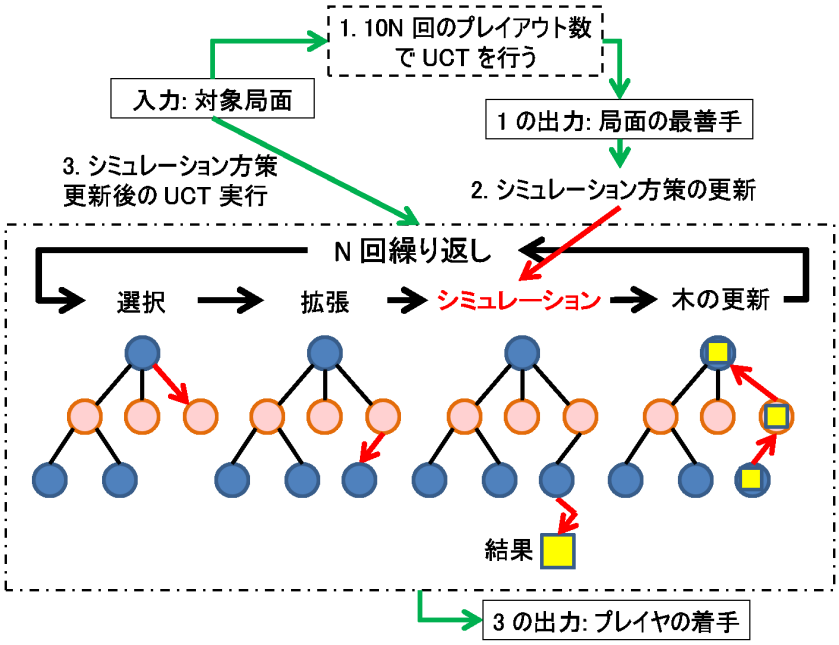


図 5.2 検証実験用プレイヤ

正規化することで着手確率とする．

図 5.2 が，以上のプレイヤの概要を図で示したものである．また，Algorithm 5.1 にこのアルゴリズムを疑似コードで示す (一部の共通処理については記述を省略している)．3 行目で通常の探索に比べて長い時間をかけて「既知の最善手」を決める．その次の 4 行目で，その手の重みを α だけ増やし，通常の探索で行われるシミュレーション中でこの手を打つ確率を高くしている．16 行目からが，与えられたテーブルに基づいて着手を決定するアルゴリズムである．単純には図 5.1 のように「全てのテーブルの値を正規化して，その値を確率として用いる」ことにより実現できるが，これを行うためには「全ての合法手についてテーブルを参照してその和を取る」「全ての合法手についてテーブルの値を求めておいた和で割る」というように，二回テーブル全体を参照する必要がある．このコストを抑えるため，本アルゴリズムでは，図 5.3 のように，盤面の縦方向もしくは横方向について値の和をとっておき，それを利用して盤面座標の片方だけをランダムに選び，その後決定した座標についてのみテーブルを参照し，もう一方の座標値を決定する．これにより着手の決定にかかる時間を抑えることができる．

5.2　対戦実験による検証

対戦実験によって検証を行った結果を示す．対戦実験は，ベースラインプレイヤとの対戦を九路盤と十三路盤で，GNU Go ver. 3.8 との対戦を九路盤で行った．

Algorithm 5.1 検証用プレイヤのアルゴリズム

```
1: function SelectBestMove(s_0, N)
2:     table ← [], table[all moves] ← 1.0
3:     best ← SearchUCT(s_0, null, 10N)
4:     table[best] += α
5:     return SearchUCT(s_0, table, N)
6: function SearchUCT(s_0, table, n)
7:     while repeat n times do
8:         s ← TreeSearch(s_0)
9:         reward ← SimulationWithProbabilityTable(s, table)
10:        BackUp(s, reward)
11: function SimulationWithProbabilityTable(s, table)
12:    while s is nonterminal do
13:        move ← SelectMoveByTable(table)
14:        s ← Move(s, move)
15:    return EvaluatePosition(s)
16: function SelectMoveByTable(table)
17:    if table == null then
18:        table ← [], table[all_moves] = 1
19:    sum = 0, sums ← []
20:    for x, y in all_legal_moves do
21:        sums[y] = sums[y] + table[(x, y)]
22:        sum = sum + table[(x, y)]
23:    rnd = Random(0,1)
24:    for y in 1 to boardsize do
25:        acc = acc + sums[y]/sum
26:        if rnd <= acc then
27:            selected_y = y
28:            break
29:    rnd = Random(0,1)
30:    acc = 0
31:    for x in 1 to boardsize do
32:        acc = acc + table[(x, selected_y)]/sums[selected_y]
33:        if rnd <= acc then
34:            selected_x = x
35:            break
36:    return selected_x, selected_y
```

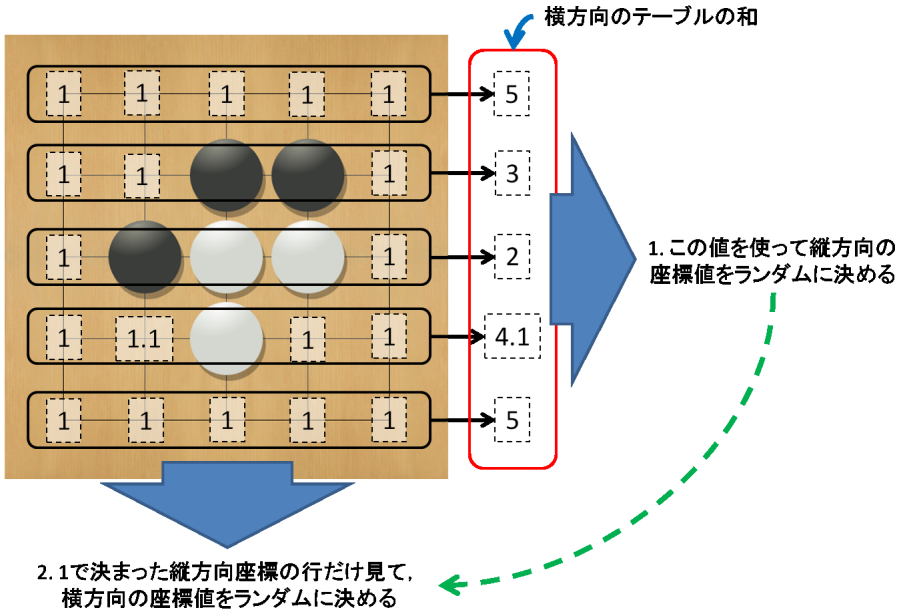


図 5.3 正規化されていない重みテーブルから効率的に着手を選択する手法

5.2.1　自作プレイヤによる対戦実験

3 章で説明した自作プレイヤをベースとしたプレイヤで検証実験を行った結果を示す.

5.2.1.1　実験設定

この実験で用いたプレイヤは次の 3 つとなる．Plain と Dynamic は，自作プレイヤのシミュレーション方策をソフトマックス方策から変更したプレイヤである.

- GNU Go: GNU Go ver. 3.8 level 10
- Plain: シミュレーション方策に一様方策を用いたプレイヤ
- Dynamic: シミュレーション方策を検証用アルゴリズムにしたプレイヤ

Plain は性能評価のためのベースラインプレイヤとなる．また，Plain と Dynamic のその他のパラメータは表 5.1 のようにして実験を行った．これらのパラメータは，Plain と GNU Go との対戦によって事前に決めたものを用いている.

5.2.1.2　自己対戦実験

Dynamic と Plain で自己対戦を行った結果を図 5.4，図 5.5 に示す．それぞれ，九路盤，十三路盤での実験結果である．それぞれ先手後手を入れ替えて合計 2,000 局の対戦を行った．また，図中の

エラーバーは，その勝率で二項分布を仮定した時の標準偏差を二倍した値 ($\pm 2\sigma$) を示している (以降，グラフにあるエラーバーは，断らない限り全て同様に $\pm 2\sigma$ を示す)．このエラーバーがベースラインに重なっていなければ，95% の有意水準で有意差が存在することになる．

結果の図を見ると分かるように，九路盤でも十三路盤でも，Plain に対して有意に勝ち越すことができない結果となった．また，α が大きくなると極端に勝率が下がる傾向にあり，$\alpha = 0.1$ の段階で有意に負け越す結果となっている．この図には載せていないが，$\alpha = 1$ ではさらに勝率が極端に下がる結果となった (30% 前後)．α が大きくなると，シミュレーションでの着手がその位置に極端に偏りやすくなり，もっともらしくないシミュレーションになってしまう上，シミュレーションの多様性が失われてしまうので，極端に勝率が下がってしまうと考えられる．

5.2.1.3 GNU Go との対戦実験

Plain と GNU Go の対戦を先手後手入れ替えて合計 10 万局行った結果，Dynamic と GNU Go の対戦を先手後手入れ替えて合計 2,000 局行った結果を図 5.6 に示す．

結果としては，全体的に Plain の勝率と同等，もしくは下回る結果となった．また，α の推移に対する勝率の変化も，自己対戦の結果と同様の傾向を示すことになった．

5.2.2 Fuego による対戦実験

Fuego をベースにしたプレイヤで検証実験を行った結果を示す．

5.2.2.1 実験設定

本実験で用いたプレイヤは次のようになる．GNU Go を除き，すべてのプレイヤは Fuego をベースにしており，そのパラメータやシミュレーション方策を変更して複数のプレイヤとしている．今回はシミュレーション方策の部分に注目した実験を行うため，ツリー方策で通常用いられる各種ヒューリスティックスをオフにして実験を行った．

表 5.1 Plain と Dynamic の各種パラメタ

パラメータ名	設定値
1 手当たりプレイアウト回数	九路盤: 10,000　十三路盤: 5,000
C (UCT のパラメータ)	0.3
ノード展開のための訪問回数	50
定石	Fuego が用いている定石と同様のもの

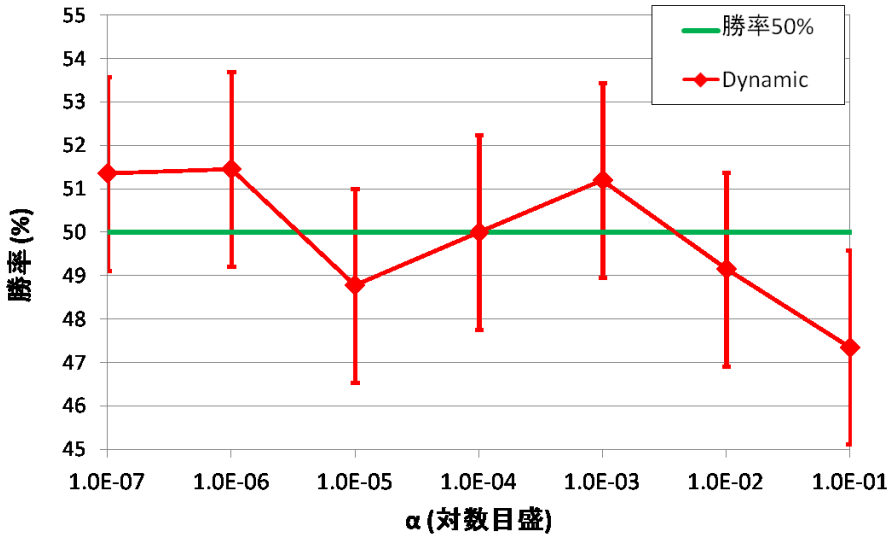


図 5.4 検証実験: Dynamic - Plain との対戦結果 (九路盤)

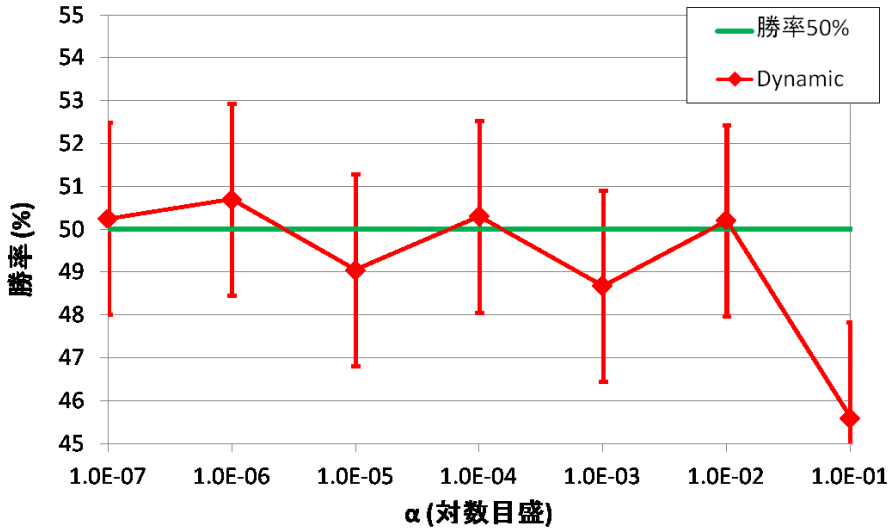


図 5.5 検証実験: Dynamic - Plain との対戦結果 (十三路盤)

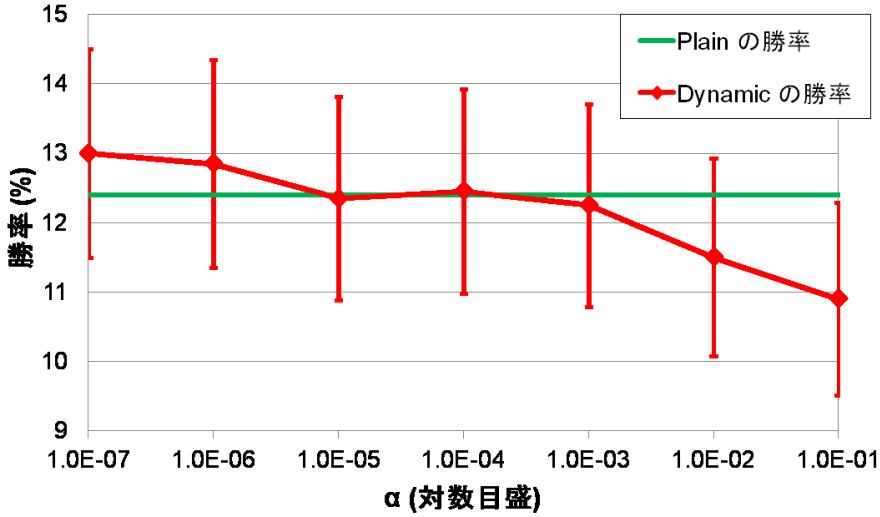


図 5.6 検証実験: Dynamic - GNU Go との対戦 (九路盤)

- GNU Go: GNU Go ver. 3.8 level 10
- Plain Fuego: 検証用プレイヤで $\alpha = 0$ としたプレイヤ
- Weak Dynamic Fuego: 検証用プレイヤで，事前探索時にも Plain と同じプレイヤを使用
- Medium Dynamic Fuego: 検証用プレイヤで，事前探索時にシミュレーションだけデフォルトの Fuego と同じ方策を利用

Plain Fuego は，シミュレーションで完全にランダムに着手を選択する一様方策と同じであり，性能評価のためのベースラインプレイヤとなる．

Weak Dynamic Fuego と Medium Dynamic Fuego の違いは，図 5.7 と図 5.8 に示すようになる．この図の通り，Medium Dynamic Fuego の方がより高精度なシミュレーションを事前探索に用いるため，より良い事前着手を得ることができる．

上記で説明した事項に加え，今回用いた各種パラメータは表 5.2 のようになる (F. は Fuego の略を示す)．これらのパラメータは，Plain Fuego と GNU Go との対戦によって事前に決めたものを用いている．

5.2.2.2　自己対戦実験

Weak Dynamic Fuego を Plain Fuego と直接対戦させた結果を図 5.9 (九路盤)，図 5.10 (十三路盤) に，Medium Dynamic Fuego と Plain Fuego の直接対戦結果を図 5.11 (九路盤)，図 5.12 (十三路盤) に示す．それぞれ先手後手を入れ替えて合計 2,000 局対戦した結果である．　これらの結果を

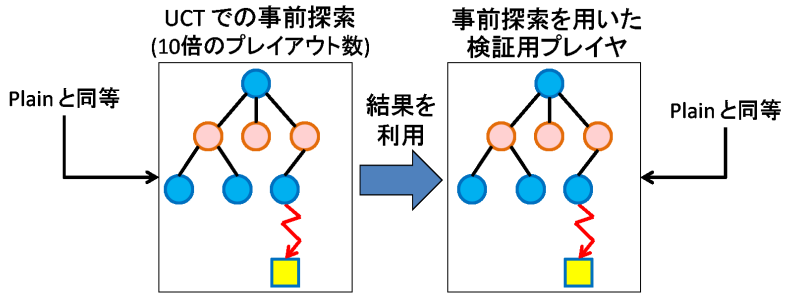


図 5.7 Weak Dynamic Fuego プレイヤ

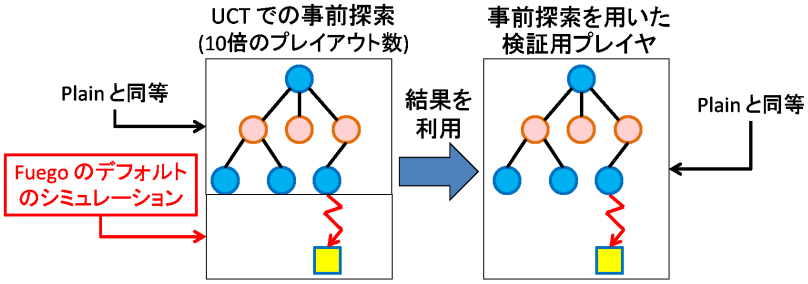


図 5.8 Medium Dynamic Fuego プレイヤ

見ると，Weak Dynamic，Medium Dynamic の勝率は α の増加に従い低下するか，もしくは Plain と同等の性能の範囲内に収まっていることが分かる．Medium Dynamic Fuego に関しては，十三路盤での対戦で，α の増加に伴う性能の低下が見られず，事前探索に良い着手を用いたことによる影響が出ていると考えられる．

表 5.2 Fuego ベースのプレイヤの各種パラメタ

パラメータ名	Plain F.	Weak Dynamic F.	Medium Dynamic F.
1 手当たりプレイアウト回数	九路盤: 10,000　十三路盤: 5,000		
C (UCT のパラメータ)	0.3		
ノード展開のための訪問回数	50		
事前探索時シミュレーション	無し	一様方策	Fuego デフォルトの方策
本探索時のシミュレーション	一様方策	Algorithm 5.1 のシミュレーション	
定石	デフォルトの定石を使用		

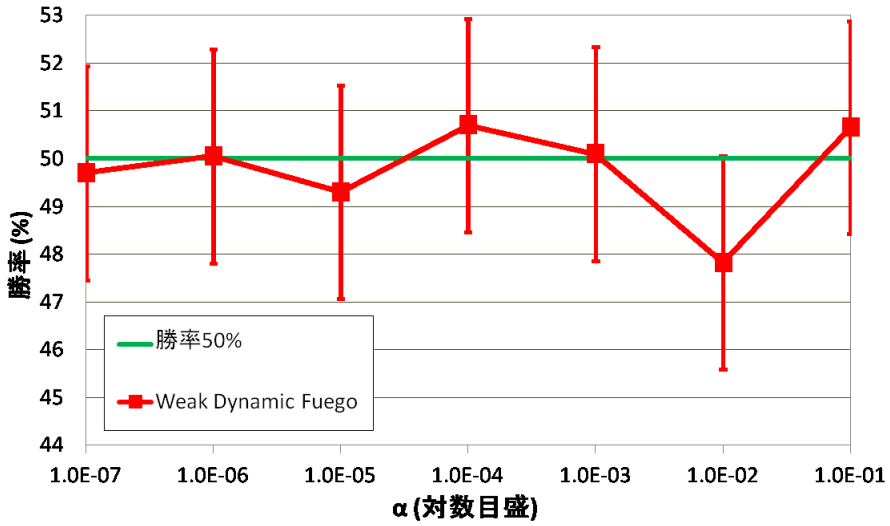


図 5.9 検証実験: Weak Dynamic Fuego - Plain Fuego との対戦結果 (九路盤)

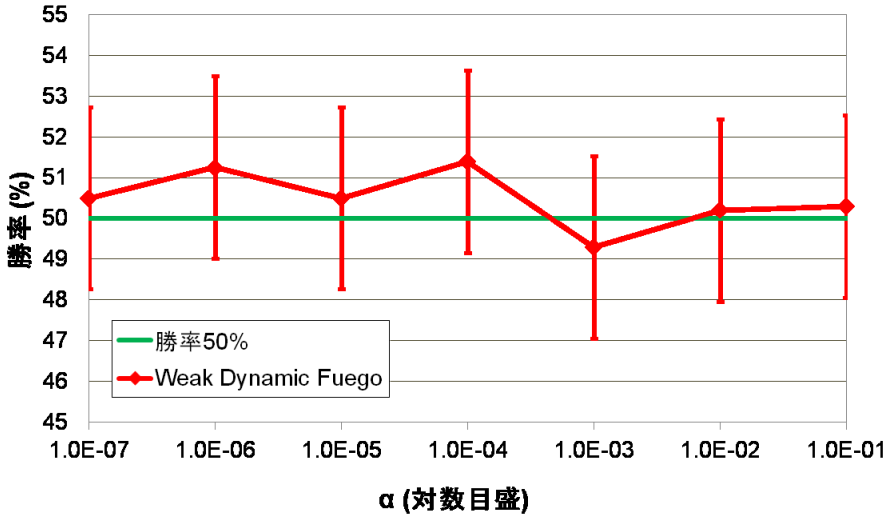


図 5.10 検証実験: Weak Dynamic Fuego - Plain Fuego との対戦結果 (十三路盤)

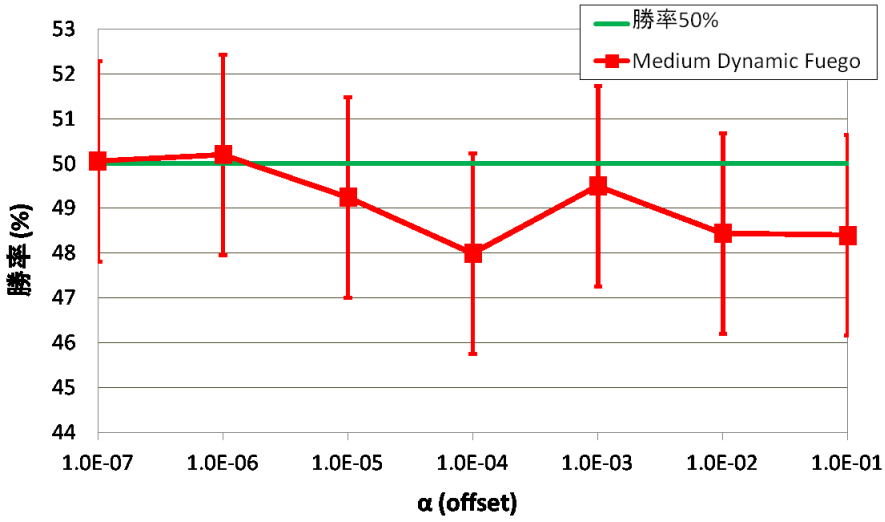


図 5.11 検証実験: Medium Dynamic Fuego - Plain Fuego との対戦結果 (九路盤)

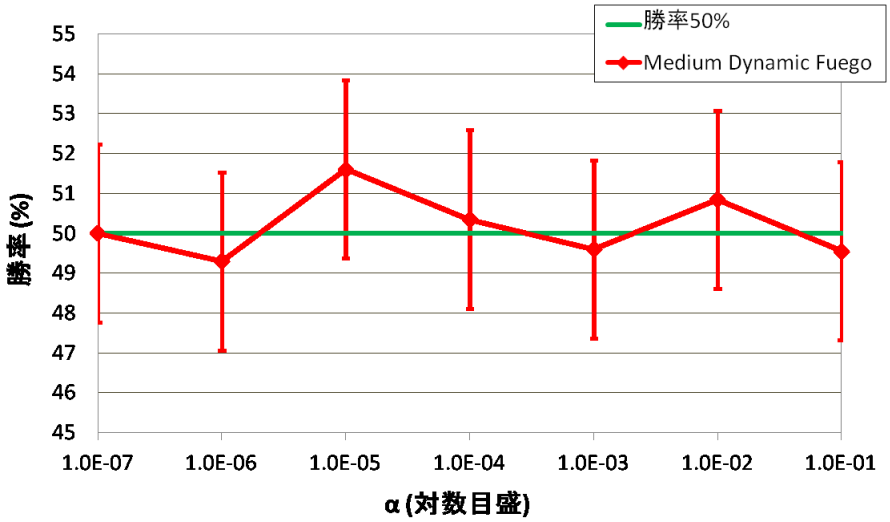


図 5.12 検証実験: Medium Dynamic Fuego - Plain Fuego との対戦結果 (十三路盤)

5.2.2.3　GNU Go との対戦実験

Weak Dynamic Fuego と GNU Go を対戦させた結果を図 5.13 に，Medium Dynamic Fuego と GNU Go を対戦させた結果を図 5.14 に示す．これらはそれぞれ，先手後手を入れ替えて合計 2,000 局対戦した時の結果である．また，ベースラインとなる Plain Fuego と GNU Go の対戦結果は，先手後手入れ替えて 10 万局対戦した時の結果である．

Weak Dynamic Fuego の性能を示す図 5.13 では，全体的に性能向上の傾向が見られず，α の増加に伴って勝率が減少する傾向を示している．これは自作プレイヤに同様に検証用アルゴリズムを実装したものと同様の結果となっている．また，図 5.14 の結果を見ると，Medium Dynamic Fuego は $\alpha = 10^{-6}$〜10^{-3} の範囲では性能が若干上回る傾向を示しているものの，$\alpha = 10^{-2}$ では統計的に有意な差で負け越してしまっている．

5.2.3　対戦実験結果からの考察

自作プレイヤによる実験では，自己対戦，GNU Go との対戦両方において，α の増加と共に性能の低下を示し，また，適切な範囲内に α がある場合でもベースラインと同等程度の性能であった．Fuego をベースとしたプレイヤでは，対戦相手や盤面の大きさによって傾向に差はあるものの，ベースラインから統計的に有意な差を得て勝ち越す結果にはならなかった．

特に，既知の最善手として，自作プレイヤや Weak Dynamic Fuego より精度が高い着手を用いている Medium Dynamic Fuego は，α の増加による性能低下は他より若干抑えられているものの，対戦相手や盤面の大きさによってその傾向に変化がある結果となっている．これは，事前探索から得られる着手の精度は強く影響せず，それを利用した方策の調整方法がより重要であるということを示す結果である．

これまで行ってきた検証実験に用いたシミュレーション方策の問題点として，「周辺状況に関わらず常に着手確率を変化させている」という点が挙げられる．シミュレーション開始後，数手の間は最善手周辺の状況は変化しない場合が多くあるが，シミュレーション中盤から終盤においては，開始時から大きく周辺状況が変化しており，着手確率を高めることが適切でないと考えられる．そこで，次節において，「既知の最善手周辺の状況が変化しない間」のみ着手確率を上げるシミュレーション方策で検証実験を行った結果を示す．

5.3　周辺パターンにも注目した検証実験

5.1 節で使用した，着手そのものについて重み (着手確率) を変える方策では，着手位置しか見ておらず，局面状況を完全に無視してしまっている．つまり，「最善である」と判断された時と周辺状況が異なる場合でも，その着手確率を高くしてしまっていた．そこで，基本的には 5.1 節のアルゴリズムと同様のプレイヤを扱うが，「現局面での良い着手」だけでなく，「その着手を中心とした 3×3 パターン」も同様に与え，「その着手位置について 3×3 パターンが一致している場合」だけ着手確率を高くするようなシミュレーション方策を利用する．これにより，周辺状況が一致している場合のみ，

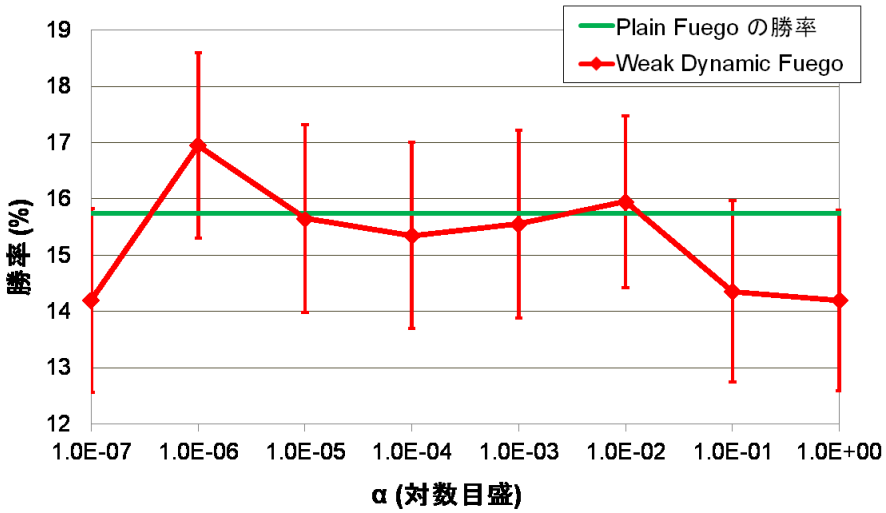


図 5.13 検証実験: Weak Dynamic Fuego - GNU Go との対戦結果

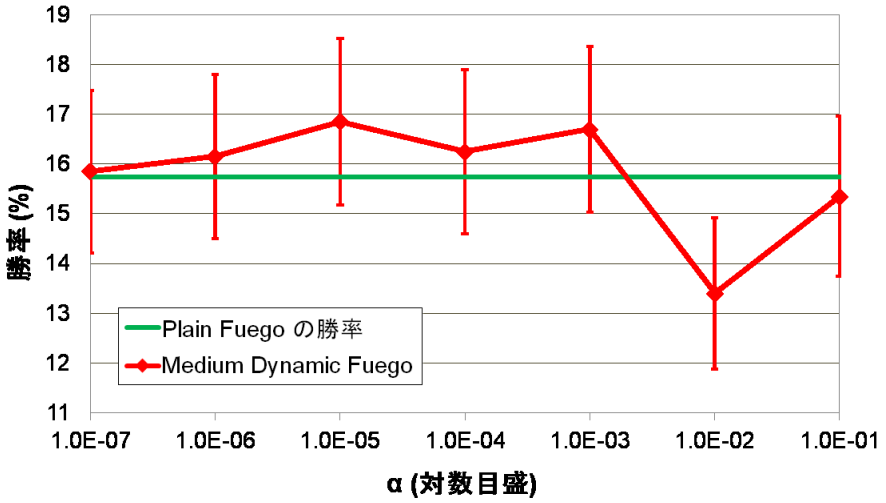


図 5.14 検証実験: Medium Dynamic Fuego - GNU Go との対戦結果

Algorithm 5.2 周辺パターンのチェックを加えた検証用アルゴリズム

```
1: function SelectBestMove(s_0, N)
2:     best ← SearchUCT(s_0, null, null, null, 10N)
3:     pattern ← Extract3x3Pattern(s_0, best)
4:     return SearchUCT(s_0, α, best, pattern, N)
5: function SearchUCT(s_0, α, best, pattern, n)
6:     while repeat n times do
7:         s ← TreeSearch(s_0)
8:         reward ← SimulationWithBestMoveAndPattern(s, best, pattern)
9:         BackUp(s, reward)
10: function SimulationWithBestMoveAndPattern(s, best, pattern)
11:    table ← []
12:    while s is nonterminal do
13:        table[all_moves] ← 1
14:        if pattern == Extract3x3Pattern(s, best) then
15:            table[best] = 1 + α
16:        move ← SelectMoveByTable(table)
17:        s ← Move(s, move)
18:    return EvaluatePosition(s)
```

良いと判断された着手位置への着手確率が高くなるシミュレーションを行うことになる.. このアルゴリズムを疑似コードで表すと Algorithm 5.2 となる．SelectMoveByTable に変更は無いため，詳細は省略している．また，Extract3x3Pattern 関数は，第一引数の局面について，第二引数で与えた着手位置を中心とした 3×3 パターンを抽出する関数である．

5.3.1 自作プレイヤによる対戦実験

このアルゴリズムについて，自作プレイヤを用いてベースラインとの自己対戦，GNU Go との対戦を行った．対戦に用いたプレイヤは次のようになる．

- GNU Go: GNU Go ver. 3.8 level 10
- Plain: シミュレーション方策に一様方策を用いたプレイヤ
- Pattern Dynamic: Algorithm 5.2 を用いたプレイヤ

その他の各種設定は，以前用いた表 5.1 と同様の設定を用いている．

5.3.1.1　自己対戦実験

Plain と Pattern Dynamic の直接対戦による実験の結果を図 5.15 と図 5.16 に示す．前者が九路盤での結果であり，後者が十三路盤での結果となる．勝率は，先手後手を入れ替えて合計 3,000 局の対戦を行った結果となっている．

九路盤での結果を見ると，Plain から統計的に有意な差は得られておらず，ほぼ同等の性能という結果となっている．しかし，α の増加に伴った性能低下が見られず，図 5.4 で示されているパターンのチェックが無い場合に比べて良い傾向になっていることが分かる．また，十三路盤の結果では，九路盤ほど明確ではないものの，パターンチェックを行わない図 5.5 の結果に比べると，パラメータに対する性能低下が抑えられているのが分かる．さらに，$\alpha = 10^{-4}$ の時，若干ではあるが，エラーバーが勝率 50% から離れており，ベースラインから統計的に有意な差を得られたという結果になっている．

5.3.1.2　GNU Go との対戦実験

GNU Go と対戦したけ結果を，図 5.17 に示す．勝率は自己対戦同様，Pattern Dyanmic は先手後手を入れ替えて合計 3,000 局の対戦を行った結果である．また，ベースラインとなる Plain は，先手後手入れ替えて 10 万局の対戦を行った結果である．この結果を見ると，自己対戦の時と同様に，図 5.6 で示されるパターンチェックが無い場合の結果に比べて，α の増加による性能低下が顕著でないことが分かる．

5.3.2　検証実験からの考察

自己対戦実験，GNU Go との対戦実験共に，単純に着手確率を上げるシミュレーション方策に比べ，改善傾向がある結果となった．十三路盤での実験結果は，九路盤での実験結果ほどパラメータの変化に対して頑強ではない結果となったが，これは 3×3 パターンが九路盤では十分な大きさであるのに対し，十三路盤では小さすぎたことが原因の一つであると考えられる．その一方で，十三路盤の場合では，$\alpha = 10^{-4}$ の時，5% の有意水準で統計的に有意な差を得られた結果となっており，パターンチェックが性能改善に貢献しているということが分かる．

また，実際にパターンが一致し，最善手について着手確率がシミュレーション中で上がっている状況は，シミュレーション開始後，九路盤においては平均 2〜3 手，十三路盤においては平均 7〜9 手程度の間であった．これは,「あまり長い間着手確率を変化させない」という目的に合致した結果となる．

以上のように，方向性としては良い手法であるが，この手法だけでは統計的に有意な差を得ることができず，有意に勝ち越すためにはさらに改善を行う必要がある結果となった．

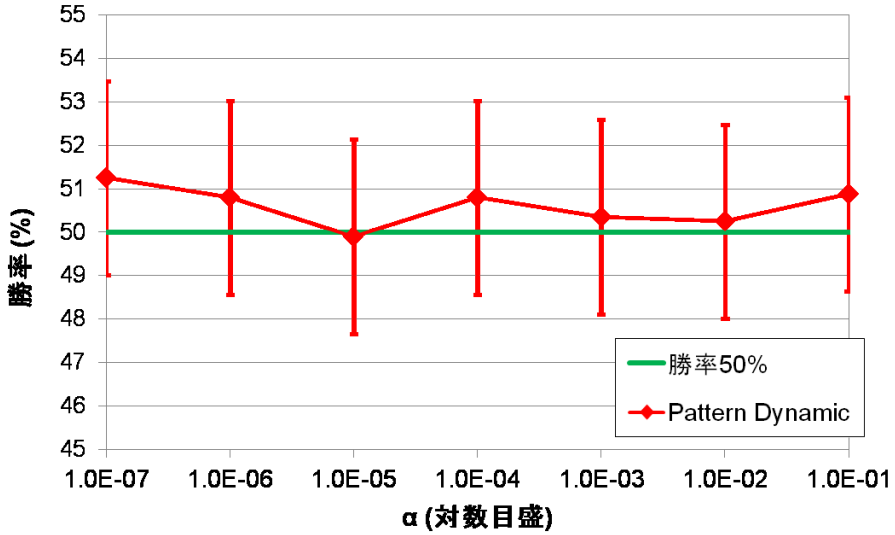


図 5.15 検証実験: Pattern Dynamic - Plain との対戦結果 (九路盤)

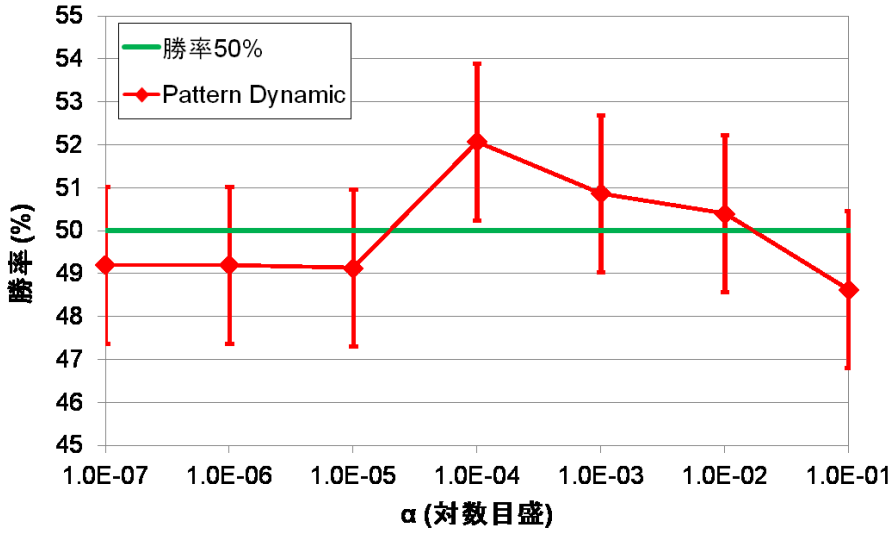


図 5.16 検証実験: Pattern Dynamic - Plain との対戦結果 (十三路盤)

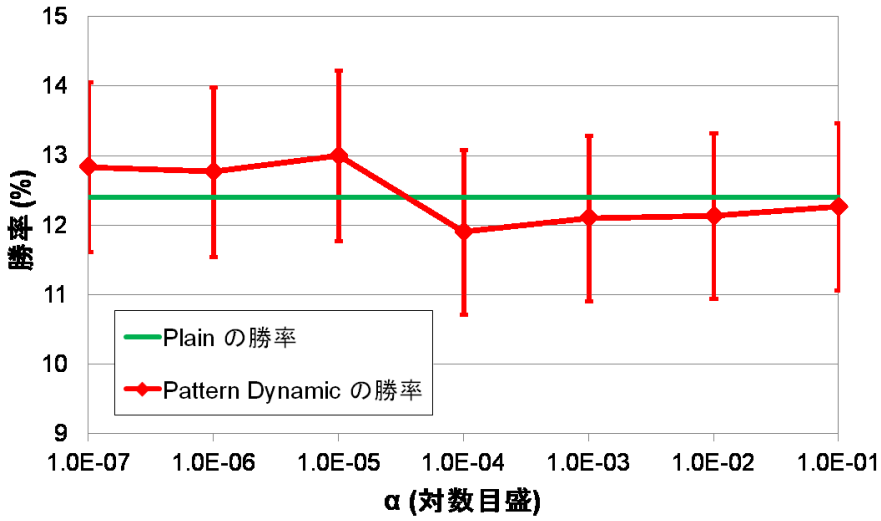


図 5.17 検証実験: Pattern Dynamic - GNU Go との対戦結果

第6章　結論

6.1　まとめ

本研究において，木探索の部分結果を利用し，ゲームの局所性に注目したシミュレーション方策の動的調整手法を提案した．囲碁は局所的な戦いが複数個所で発生し，どの戦いから着手を行おうと，局所的に見ると戦いごとの応手手順に変化はないという仮定での手法である．対戦実験によって手法の検証を行った結果，ベースラインプレイヤに統計的な有意差をつけて勝ち越すことはできなかったが，パラメータによっては性能向上の傾向を見せる結果となった．しかし，考察の結果，提案手法で用いた特徴量では望まない方策の調整が発生してしまうことがあることが分かった．また,「調整に用いるデータの選択」と「方策の調整」を同時に行っており，どちらがどのように結果に影響を与えているか分かりづらく，これらを分離して検証することが必要となった．

以上の結果を受け，より理想的な設定で検証実験を行った.「探索対象の局面における最善手が既知」という設定で,「最善手の着手確率を上げる」というシミュレーション方策によるプレイヤでの実験である．この実験の結果，必ずしも最善手について着手確率を上げることが適切ではなく，最善手が既知である場合にも，その情報の用い方が重要であるという知見が得られた．そこで,「既知の最善手の周辺状況が変化しない間，着手確率を上げる」というシミュレーション方策によって，さらに検証を行った．その結果，九路盤の場合は統計的に有意な差をベースラインにつけることは出来なかったが，パラメータの変化に対して性能の低下が抑えられる結果となった．また，十三路盤の場合は，特定のパラメータにおいてベースラインに対して統計的に有意な差をつける結果となっており，単純に最善手の着手確率を上げるシュミレーション方策に比べ，性能向上の傾向を見せる結果となった．

6.2　課題

今回提案した手法で用いた特徴量は，シミュレーション方策において一般的に用いられる特徴量であったが,「事前の学習では得られない情報を得る」という目的では，このような特徴量だけ用いるのは適切ではなかった可能性がある．そこで,「事前の学習では得られない特徴量 (死活の情報等)」と言ったような特徴量を導入，もしくは，特徴量として導入せずとも，木探索の部分結果から取得できる情報を利用し，その情報に基づいたヒューリスティックスを導入する方法等が必要となる．以上のように，より目的に適した特徴量，ヒューリスティックスの設計が課題の一つとして挙げられる．

また，調整手法についても課題が残っている．提案した確率的勾配法をベースとした方法ではなく，ミニバッチ的な調整手法を検証する必要もある．本研究で使用した手法では，最後に調整に使用したデータの影響が一番強くなってしまい，うまく学習が行えない可能性がある．さらに，調整に用

いる局面と着手の選択についても検証の余地がある．死活情報等がはっきりしている局面と着手を用いたりすることで，さらに改善する可能性がある．

さらに，評価実験や検証・解析については，自作プレイヤの強化，Fuego の機能全ての利用等による，より強いプレイヤでの実験が課題として残っている．自作プレイヤを強化するためには，ツリー方策への種々のヒューリスティックスの導入，RAVE [11] の導入，シミュレーションの高速化等が具体的な課題となる．

参考文献

[1] Nobuo Araki, Kazuhiro Yoshida, Yoshimasa Tsuruoka, and Jun'ichi Tsujii. Move Prediction in Go with the Maximum Entropy Method. In *Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games*, pp. 189–195, 2007.

[2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, Vol. 47, No. 2, pp. 235–256, 2002.

[3] Radha-Krishna Balla and Alan Fern. UCT for Tactical Assault Planning in Real-Time Strategy Games. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pp. 40–45, 2009.

[4] Vincent Berthier, Hassen Doghmen, and Olivier Teytaud. Consistency Modications for Automatically Tuned Monte-Carlo Tree Search. *Technical Report 00437146, INRIA*, 2010.

[5] Tristan Cazenave. Nested Monte-Carlo Search. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pp. 456–461, Jul. 2009.

[6] Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Proceedings of the 5th International Conference on Computers and Games*, pp. 72–83, 2006.

[7] Rémi Coulom. Computing Elo Ratings of Move Patterns in the Game of Go. In *Computer Games Workshop*, pp. 198–208, 2007.

[8] Markus Enzenberger, Martin Müller, Broderick Arneson, and Richard Segal. Fuego - An Open-Source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 2, No. 4, pp. 259–270, 2010.

[9] Hilmar Finnsson and Yngvi Björnsson. Simulation-Based Approach to General Game Playing. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pp. 259–264, 2008.

[10] Hilmar Finnsson and Yngvi Björnsson. Learning Simulation Control in General Game-Playing Agents. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[11] Silvian Gelly. Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go. *Artificial Intelligence*, Vol. 175, No. 11, pp. 1856–1875, 2011.

[12] Silvian Gelly and Silver David. Combining Online and Offline Knowledge in UCT. In *Proceedings of the 24th International Conference on Machine Learning*, 2007.

[13] Sylvain Gelly, Wang Yizao, Munos Remi, and Teyta Oliver. Modification of UCT with Patterns in Monte-Carlo Go. *Technical Report RR-6062, INRIA*, 2006.

[14] Shih-Chieh Huang, Rémi Coulom, and Shun-Shii Lin. Monte-Carlo Simulation Balancing in Practice. In *Proceedings of the 7th International Conference on Computers and Games*, pp. 81–92, 2010.

[15] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo Planning. In *In Proceedings of the 17th European Conference on Machine Learning*, pp. 282–293, 2006.

[16] T. Lai and Herbert Robbins. Asymptotically Efficient Adaptive Allocation Rules. *Advances in Applied Mathematics*, Vol. 6, No. 1, pp. 4–22, 1985.

[17] Richard J. Lorentz. Amazons Discover Monte-Carlo. In *Proceedings of the 6th International Conference on Computers and Games*, pp. 13–24, 2008.

[18] Manuel Loth, Youssef Hamadi, and Marc Schoenauer. Hybridizing Constraint Programming and Monte-Carlo Tree Search : Application to the Job Shop problem. In *Proceedings of the 7th International Conference on Learning and Intelligent Optimization*, pp. 315–320, Jan. 2013.

[19] Jean Méhat and Tristan Cazenave. Combining UCT and Nested Monte Carlo Search for Single-Player General Game Playing. *Computational Intelligence and AI in Games*, Vol. 2, No. 4, pp. 271–277, 2010.

[20] Chaslot M.J-B., Guillaume, Winands H.M., Mark, H. Jaap van den Herik, Uiterwijk W.H.M., Jos, and Bouzy Bruno. Progressive Strategies For Monte-Carlo Tree Search. In *Proceedings of the 10th Joint Conference on Information Sciences*, Vol. 04, pp. 655–661, 2008.

[21] Hootan Nakhost and Martin Müller. Monte-Carlo Exploration for Deterministic Planning. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, 2009.

[22] Yasuhiro Osaki, Kazutomo Shibahara, Yasuhiro Tajima, and Yoshiyuki Kotani. An Othello evaluation function based on Temporal Difference Learning using probability of winning. In *Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games*, 2008.

[23] Diego Perez, Edward J. Powley, Daniel Whitehouse, Philipp Rohlfshagen, Spyridon Samothrakis, Peter I. Cowling, and Simon M. Lucas. Solving the Physical Travelling Salesman Problem: Tree Search and Macro-Actions. *IEEE Transaction on Computational Intelligence and AI in Games (accepted)*, 2013.

[24] Nathan R. Sturtevant. An Analysis of UCT in Multi-Player Games. *International Computer Games Association Journal*, Vol. 31, No. 4, pp. 195–208, 2008.

[25] Christopher D. Rosin. Nested Rollout Policy Adaption for Monte Carlo Tree Sesarch. In *Proceedings of the Twenty Second International Joint Conference on Artificial Intelligence*, pp. 649–654, 2011.

[26] Jann Schafer. *The UCT Algorithm Applied to Games with Imperfect Information*. PhD thesis, Otton-von-Guericke-Unversität Magdeburg, 2008.

[27] David Silver, Richard S. Sutton, and Martin Müller. Sample-based learning and search with permanent and transient memories. In *Proceedings of the 25th International Conference on Machine Learning*, pp. 968–975, 2008.

[28] David Silver and Gerald Tesauro. Monte-Carlo Simulation Balancing. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pp. 945–952, 2009.

[29] David Silver and Joel Veness. Monte-Carlo Planning in Large POMDPs. In *Proceedings of the Neural Information Processing Systems*, pp. 2164–2172, 2010.

[30] Jiao Wang, Shiyuan Li, Jitong Chen, Xin Wei, Huizhan Lv, and Xinhe Xu. 4*4-Pattern and Bayesian Learning in Monte-Carlo Go. In *Proceedings of the 13th International Conference on Advances in Computer Games*, pp. 108–120, 2011.

[31] Mark H.M. Winands and Yngvi Björnsson. Evaluation function based Monte-Carlo LOA. In *Proceedings of the 12th Advances in Computer Games Conference*, pp. 33–44, 2009.

[32] 佐藤佳州, 高橋大介. モンテカルロ木探索によるコンピュータ将棋. 情報処理学会論文誌, Vol. 50, No. 11, pp. 2740–2751, Nov. 2009.

[33] 荒木伸夫, 村松正和. UEC 杯海外勢の先端技術. 人工知能学会誌, Vol. 28, No. 5, pp. 776–780, Sep. 2013.

発表文献

[1] 板持貴之, 三輪誠, 鶴岡慶雅, 近山隆. モンテカルロ囲碁における探索木の情報を利用したシミュレーション方策の動的更新. 第 18 回ゲームプログラミングワークショップ, pp. 122–125, Nov. 2013.

[2] 板持貴之, 三輪誠, 鶴岡慶雅, 近山隆. 歴史の選択問題を解くため必要なフレーム的知識に関する考察. 言語処理学会第 19 回年次大会, Mar. 2013.

謝辞

本研究を進めるにあたり，多くの方々にお世話になりました．

研究テーマがなかなか決まらず，方向性を落ち着かせることができないなかでも，指導教員である鶴岡慶雅准教授には，研究テーマの決定，発表や論文執筆等について様々なアドバイスを頂きました．また，Microsoft Research Asia でのインターンシップを紹介して頂き，修士 1 年の夏を有意義に過ごすことが出来ました．

近山隆教授には，学部時代から 3 年間お世話になりました．広く深い知識と経験をから，研究の本質そのものに関する質問やアドバイスを頂き，有意義な議論を行うことができ，大変助かりました．

マンチェスター大学コンピュータ科学科所属であり，研究室の OB である三輪誠さんには，研究の細かい部分に関しての指摘や，論文執筆，資料作成等についてのアドバイスを多数頂きました．研究生活だけではなく，美味しいお酒や料理等，日常生活についても色々と意見を頂きました．

博士課程の浦晃さんには，研究がうまくいかなかった時に相談に乗って頂いたり，日頃から研究についての方向性を示して頂いたりと，学生視点でのアドバイスを色々と頂きました．また，研究室の計算機資源についてや，サーバ運営に関わる Linux 関連の基本的なことなどを教えて頂き，研究とは関係ない部分でも色々と力になりました．

同期の佐藤美沙さんとは，中国でのインターンシップの時に同僚として働いたり，日常のとりとめのない雑談を行ったりと，協力し合いながら修士生活を過ごせました．また，同じく同期の成川弘樹君には，私が抱えていたサーバ管理関連の雑務を一部引き受けてもらったりと，色々と手助けをしてもらいました．その他にも，研究室を同じくした皆さまには，研究の助言から取り留めのない雑談，研究室のゴミ捨てに至るまで色々とお世話になりました．

卒業論文の時に研究室を同じくしていた同期である，田浦研究室所属の中澤隆久君，中谷翔君，林伸也君，喜連川研究室所属の鈴木恵介君には，たまに研究室に来て気晴らしに雑談を行なったり，私が色々と困っている時に相談に乗ってもらったりと，お世話になりました．

コンピュータ科学専攻所属の鈴木健太郎君，細川航平君，サークルの先輩である三浦敬司さんとは，就職活動についての話，研究についての話から，美味しい焼肉屋の話，好きなアニメや漫画の話など，色々なことを話しました．修士生活に苦しむこともありましたが，ここまで来れたのも 2 人との付き合いが大きかったと感じています．本当にありがとう．

株式会社レピダム 代表取締役である林達也さんには，私のわがままを聞いていただき，修士生活であまりアルバイトに時間が割けない中でありながらずっとアルバイトとして雇っていただきました．また，私の就職活動の際には，お忙しい中色々と相談に乗っていただきました．

修士生活を送る上で心の支えとなってくれた漫画の作者である，なもり先生，原悠衣先生，三上小又先生，タチ先生には大変感謝をしています．

最後に，弱音を吐くことがあっても責めずに受け入れてくれた彼女と，20 代半ばになっても殆ど金銭的収入がない私を金銭的・精神的にサポートしてくれた家族には心から感謝しています.

ここに，お世話になった皆様への心からの感謝の意を表します.

平成 26 年 2 月 6 日