平成 20 年度電気通信大学大学院情報工学専攻修士論文

モンテカルロ碁に対する
詰碁を解くための改良

指導教官　村松 正和 教授

平成 21 年 1 月 30 日

電気通信大学大学院情報工学専攻

0731049
李 輝

目 次

1 はじめに **3**
1.1 背景 3
1.2 構成 4

2 詰碁とは **5**

3 モンテカルロ碁 **6**
3.1 ランダムシミュレーション 6
3.2 UCB 8
3.3 UCT 8

4 詰碁を解くための改良 **10**
4.1 捕獲率の導入 10
4.2 局所的なプレイアウト 10
4.3 詰碁専用のパターン 12
4.3.1 3 × 3 パターン 13
4.3.2 5 × 5 パターン 14
4.3.3 詰碁専用パターンと対局用のパターンとの比較 15

5 手順を得るための試み **19**
5.1 Min-Max ルート 19
5.2 モンテカルロ碁を用いた木探索 22

6 実験結果と考察 **26**
6.1 一手目の正解を求める実験 27
6.1.1 ランダムシミュレーション回数 10,000 回の場合 27
6.1.2 ランダムシミュレーション回数 100,000 回の場合 28
6.1.3 ランダムシミュレーション回数 500,000 回の場合 28
6.1.4 セキとコウ 30
6.2 正解手順を求める実験 32

7 結論 **33**

8 謝辞 **34**

1 はじめに

1.1 背景

四千年の歴史を持つ囲碁は、ルールは単純であるが、その局面数は 10^{300} [5] と言われており、同じボードゲームであるチェスや将棋と比べても格段に複雑なゲームである。表 1 に主なボードゲームにおけるコンピュータの強さを表す。

表 1: 主なボードゲームにおけるコンピュータの強さ

ゲーム	コンピュータの強さ
チェッカー	1994 年に世界チャンピオンに勝利
リバーシ	1996 年に世界チャンピオンに完勝
チェス	1997 年に IBM の DeepBlue が当時世界チャンピオンの Kasparov を破る
将棋	アマトップレベルの強さと言われている
囲碁	アマ初段を超えるプログラムがようやく現れた

現時点で最も優れたコンピュータ囲碁プログラムはアマチュア程度である [2] 。1 秒間に 1 億手 (10^8) の検証ができるとして、総当たり方式ですべての手を検証する時間は、3.17 $\times$ 10^{344} 年がかかる。このため、総当たり方式で必勝法を探し出すことは不可能である。その複雑さから、囲碁は今や人工知能の最も難しい挑戦の一つであると言われている。

囲碁は多くの面で将棋やチェスと異なっている。その違いの第一は、局面数の多さである。つまり、次の着手を選ぶ選択肢はチェスや将棋と比較にならない程多い。チェスや将棋において威力を発揮した様々な探索手法は、囲碁の巨大のゲーム木の前では効果が激減する。第二は正確な盤面評価が困難なことである。その理由として以下のものが挙げられる。

1. 石の価値は表面上平等である。チェスや将棋のように駒の価値などを用いることができない。
2. 領域の広さを競うゲームであるが、領域が確定するのはゲームの最後である。
3. リバーシにおける隅のような明らかに特徴のある箇所が少ない。
4. 局所的な最善手が全局的な最善手になりにくい。

これらの理由から、チェスや将棋で威力を発揮したα β探索はほとんど役に立たない。

最近コンピュータ囲碁の分野において、モンテカルロ碁という考え方が登場し、従来の知識ベースプログラムよりも強くなってきている。モンテカルロ碁は、基本的に次のようなプロセスで着手を決定する。

1. 現在の盤面にランダムに着手を行い、終局まで打つ。(これをランダムプレイアウト、あるいは単にプレイアウトと呼ぶ。)

2. 着手可能な点すべてを起点としてプレイアウトを大量に行い、最も勝率が高い着手を選ぶ。

このように非常に単純な戦略を利用することは、次の考えによって正当化されている：ある局面から、同じ棋力のプレイヤーが最後まで打てば、優勢な方が勝つ確率が高い。

モンテカルロ碁はランダムシミュレーションを基本とするので平均的には強いが、石の生死にはある程度直線的で深くヨミが必要であるので、石の生死の読みは不得意と言われている。

本研究では、モンテカルロ碁を用いて、ある特定の石の生死についてシミュレーションを行い、モンテカルロ碁に対する詰碁を解くため改良を行い、その検証を行う。主な改良は以下のものである。

1. ランダムシミュレーションの範囲を小さくして、シミュレーション速度を上げる。

2. 詰碁専用のパターンを自動生成してランダムシミュレーションの精度を上げる。

3. モンテカルロ碁を用いた木探索を用いて直線的で深くヨミができるようにする。

1.2 構成

本論文の構成は以下の通りである。

2 章では、詰碁の定義、種類、および詰将棋と比較しそれぞれの特徴を述べる。3 章では、従来研究であるモンテカルロ碁の構成、特徴を説明し、ランダムシミュレーションと UCT を紹介する。また、本研究に適した実装も述べる。4 章では、詰碁問題を解くための改良について述べる。ランダムシミュレーションの速度向上のために行った局所的なプレイアウト、精度向上のために行った詰碁専用パターンの抽出および適応について説明する。5 章では、詰碁問題の正解手順を得るために必要な Min-Max ルート、モンテカルロ碁を用いた木探索アルゴリズムを解説する。6 章では、詰碁問題の正解を求める実験結果と考察について述べる。

2　詰碁とは

詰碁とは、白黒の石が置かれた囲碁の盤面の一部または全部と、手番（「白番」、「白先」もしくは「黒番」、「黒先」）が示され、どのように打てば自分の石を生きに持ち込めるか、または相手の石を殺すことができるか、すなわち死活を考えるものである。一手目がわかればあとは一本道で進んでいく問題もあれば、途中の対応が難しいものもある。

詰碁問題の種類は黒先活、黒先白死、白先活、白先黒死、黒先コウ、白先コウなどがある。図１は黒先白死の例である。

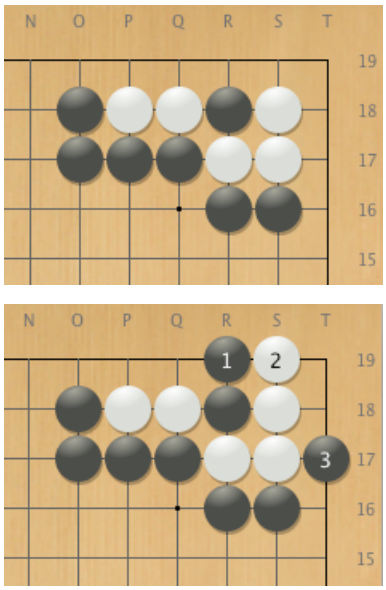


図 1: 黒先白死

黒１とまず逃げるが、白２ですぐにつかまるので損のように見える。しかし攻めのねらいはそこにあり、黒３で白はT18に打つことができず、白死となる。黒１を打った時点で白がすでに死んでおり、実戦でそれ以上打つことに意味はないが、最も強い抵抗として白は白２を打つ。

この例で、白２を「最も強い抵抗」と表現したが、この概念は自明のものではなく、主観的なものである。どんな抵抗をしても、死には変わりないが、人間には考える道筋があり、その中で最も面白いと感じるのが、「最も強い抵抗」と言われるのである。

これは、詰碁と詰将棋が異なる点である。詰将棋では、攻め方は最短手順を、受け方は最長手順を試みるので、詰るまでのルートが一意に決まる。

これに比べ、詰碁では決めるのは (原則的に) 次の一手のみである。しかし、一手のみを示したのでは、本当に生死がわかっているのか判定できず、何より詰碁の解として不適切である。これは、どの詰碁の本を見ても、解答に一手しかないものはほとんどないことから明らかであろう。

そこで本研究では、次の一手の正解を得るのみならず、白の抵抗も予想して、それへの対応を探すという、詰碁を解く ”手順 ”を得ることも目標の一つとする。

3　モンテカルロ碁

モンテカルロ碁に関して [1, 2, 3] などの従来研究を基に説明する。モンテカルロ法を用いた囲碁プログラムは、UCT [2] 部分とランダムシミュレーション部分から成る。その関係を図 2 で示す。

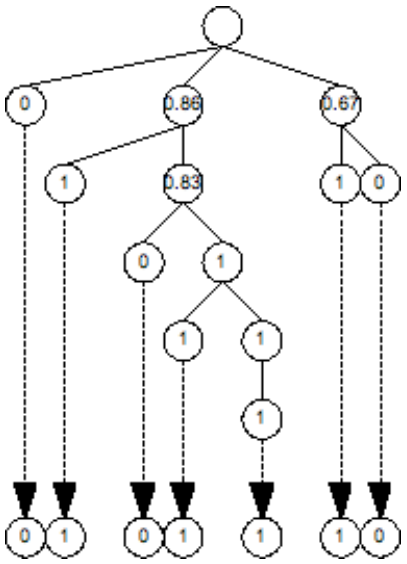


図 2: UCT とランダムシミュレーション

図 2 のように、下の部分はランダムシミュレーションで、上は UCT 探索木となる。
以下で UCT とランダムシミュレーションについて解説する。

3.1　ランダムシミュレーション

ランダムシミュレーション部分は、目を定義し、自分の目を埋めないようにランダムで手を選び、盤面上のすべての位置に置けなくなるまで石を打つ。置けなくなった最後の盤面を評価し、評価値を得る。

ランダムシミュレーションで実行速度が重要である。ランダムシミュレーションの数を増やすほど、一般には手が絞り込まれ、強いプログラムとなる。できるだけ多くのシミュレーションをすることに大きな意味がある。よって、プログラムのデータ構造を考えることが重要である。

ここで連について説明する。連とは、縦か横にくっついている同色の石の極大集合をいう。プログラムに連を導入することで、石のダメの計算、石が取られるときの処理などさまざわな面で計算コストを減らすことができる。

本実験のデータ構造を表 2 に示す。

表 2: 囲碁プログラムのデータ構造

連構造体	
id	連の id
color	連の色
size	石の数
dame size	ダメの数
stone table	連の石のテーブル
dame table	連のダメのテーブル

連リスト構造体	
kou	コウの位置
get stone	前回取った石の位置
next id	次の連 id
gs id table	連 id テーブル
gs list	連リスト

プログラムの強さに影響するもう一つの要素は、プレイアウトの定義にある。プレイアウトでは、目の定義をして、自分の目に埋めないようにする。盤面上のすべての位置にランダムに石を置き、石が置けなくなるまでこれを繰り返す。最後の盤面で石の数と石で囲まれた空点の数で、評価値を計算して返す。もし目の定義をしないと、相手の石を取ったり自分の石が取られたりが繰り返され、ゲームは永遠に続く。

囲碁において、目は一つの連で囲まれた空点、多数の連で囲まれた空点などいろいろな場合がある。囲碁ルール通りに正確に目を定義することは困難な問題である。それは不可能ではないが、ここではプレイアウトの速度を考慮した上で、目の定義を簡略化する。本実験での目の定義は

同じ色の石で囲まれた空点

とする。この定義だと欠目になる可能性がある。欠目とは、必要な連結点を相手に占められているため、目にならない空点という。図 3 は欠目の例である。しかし、欠目が出る確率の低さと最後の盤面の結果に与える影響の少なさから、このように目の定義を簡略化してプレイアウトの速度を上げるほうが、性能的に有利と考えられる。

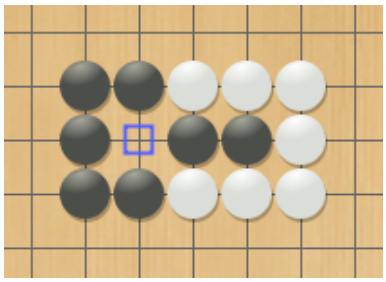


図 3: 欠目

3.2 UCB

プレイアウトを行う際、すべての可能な候補手に関して同じ回数のシミュレーションを行うのは効率が悪い。なるべくならば、勝つ確率が高そうなところを集中的に調べたいが、あまり集中させるといろいろな手を候補に入れることができなくなる。このバランスを取るために、UCB（Upper Confidence Bounds）というものが広く使われている。

この技法は、パラメータ $c > 0$ に関して着手 i に関する UCB 値を計算し、この値が最も高い手を候補手（次にプレイアウトを始める手）とする。

$$UCB_i = \frac{W_i}{N_i} + c\sqrt{\frac{\log t}{N_i}}$$

ここで、W_i, N_i はそれぞれ着手 i を初手とした場合の勝ち数とプレイアウト数、また t はすべての着手に関するプレイアウト数の合計である。

UCB 値の定義より、勝つ確率が高いものか、または長い間訪ねていなかった手のどちらかがプレイアウトの候補手に選ばれやすい。

3.3 UCT

UCT（Upper Confidence bounds for Tree）は UCB を木構造に対して拡張した技術である。その考え方は、各ノードに対して UCB 値が最も高い子ノードを選んで、その子ノードに対して同じことを行い、リーフまでそれを繰り返す。各ノードを一つずつ繰り返し処理するのではなく、ルートから始まりあるリーフで終わるシーケンスを一定時間プレイする。

UCT アルゴリズムの擬似コードを表 3 と表 4 に示す。一度に一つのシーケンスをプレイし、それを繰り返す（表 4 ）。これは各候補手に対して一度選ばれることを保証する UCB 値を用いて子ノードを選択する（表 3 ）。

表 3: 一つのシーケンスを行う擬似コード

```
function playOneSequence(rootNode)
  i = 0;  node[i] = rootNode;
  while(node[i] はリーフでない)
    node[i+1] = descendByUCB(node[i]);
    i = i + 1;
  end while;
  通ったノードの値を更新する;
end function;
```

表 4: 子ノードを選ぶ擬似コード

```
function descendByUCB(node)
  if (探索していない候補手があれば)
    その候補手で新しいノードを作る;
    このノードをリーフとする;
    ランダムシミュレーションを行う;
  else
    すべての子ノードの UCB 値を計算する;
    UCB 値が最も高い子ノードを選ぶ;
    子ノードを返す;
  end if;
end function;
```

一つのシーケンスをプレイ後、通ったノードの値を、リーフからルートまで更新する。ランダムシミュレーションの評価値を反映させることによって探索木を成長させていく。一般に、各ノードの値はシミュレーション回数の増加につれて真の最小（最大）値に収束する。

UCT が α β 探索より優れている点は主に以下の三つがある。

1. いつでも止めることができ、その結果はそれなりに良い可能性がある。
2. 不確実性を自動的に滑らかにするので頑健である。各ノードで計算された値は各子ノードの値を探訪頻度で重み付けた平均値である。探訪頻度は推定値とこの推定の信頼度の差に依存するので, この値は最大値の滑らかな推定になる。
3. 木は自動的に最良優先的に成長する。良い手をより深く探索する。

4　詰碁を解くための改良

詰碁問題の特徴に基づいて、ランダムシミュレーションの速度向上、精度向上、および詰碁の解答の ”手順 ”を求めるための改良を行った。

4.1　捕獲率の導入

詰碁は石の死活問題とほぼ等しいことから、ランダムシミュレーションの中で、指定した石が取られるかどうかという石の捕獲率を導入する。

石の捕獲率とは、指定した石がその盤面で取られる確率と定義する。

$$\text{捕獲率} = \frac{\text{捕獲数}}{\text{プレイアウト数}}$$

ランダムシミュレーション部分で、シミュレーション中に指定した石が殺されれば、捕獲数に 1 を加える。殺されなかった場合、捕獲数は変わらない。

本実験において、UCT 木は勝ち負けでなく、この捕獲率に基づいて成長させていく。

4.2　局所的なプレイアウト

詰碁問題は石の数が多いものもあるが、そのほとんどは盤面の一部しか使わない問題である。離れた点は全く問題と関係していないと言える。

よって、UCT 探索は局所的に行うようにしてモンテカルロ碁の効率をあげることが可能である。

例えば次の図 4 の例では、ヒューリスティックより、正解が範囲 N〜T,19〜15 にあり、それ以外は考えられないとされる。

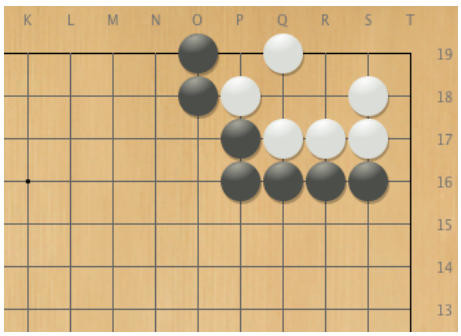


図 4: 詰碁問題

さらに、UCT 木を成長させる時だけではなく、ランダムシミュレーションの中でも、N〜T,19〜15 範囲外の点はほとんど無関係なので、これを黒（又は白）石で埋め尽くしたほうが効率があがる。

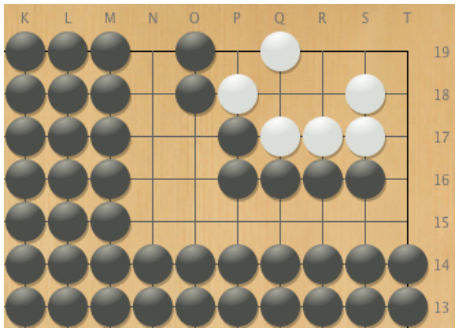


図 5: 探索範囲を小さくする

また、図 5 の黒石を埋めても、シミュレーションの中で黒石が死んでしまう可能性は小さいが、決してないとは限らない。よって、埋めた黒石が死んで捕獲率に悪い影響を与えることを防ぐために、埋めた黒石に適当な場所で目を二つ作ることで、完全に生きるようにしてシミュレーションの精度を向上させる。その様子は図 6 で示している。

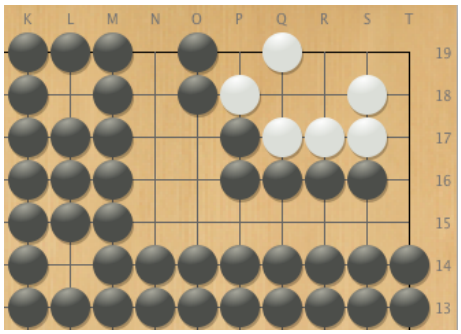


図 6: 埋めた石に目を作る

本研究における詰碁プログラムはこの技法を用いた。

4.3 詰碁専用のパターン

ランダムシミュレーション中、パターンを使うことでシミュレーションの精度をあげる従来の研究がある。MoGo [2] は次の手が打たれる中央の点が空いた 3 × 3 の点をパターンとして、いくつかのパターンを導入しており、パターンがランダムシミュレーションの性能をいかに改善しているかに関して明確な結果を出している（表 5 参照)。

表 5: MoGo パターンなしモードとパターンありモードの比較 [2]
（70,000 ランダムシミュレーション/手.9 路）

ランダム・モード	黒番の勝率	白番の勝率	全勝率
パターンなし	46%（250）	36%（250）	41.2% ± 4.4%
パターンあり	77%（400）	82%（400）	80% ± 2.8%

本研究では、詰碁専用のパターンを作成した。具体的には、「ポケット詰碁 200」[6] から詰碁問題 200 問を取り出し、詰碁の棋譜を入力し、それから詰碁専用のパターンを自動抽出した。パターンをサイズで分けて、3 × 3 パターンと 5 × 5 パターンの二つのサイズのパターンを生成した。詰碁問題を SGF 棋譜に変換するのに GoGui [8] を使用した。また、パターンを抽出するため、SGF 棋譜からパターンを自動抽出するプログラムを作成した。

図 7 の棋譜からどんなパターンを取れるかについて例として説明する。

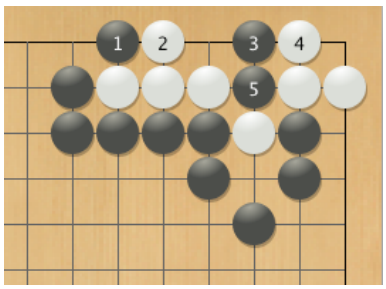


図 7: 詰碁専用パターンを取るための棋譜

図 7 は「ポケット詰碁 200」の応用編問題 26 の棋譜の一例である。この詰碁問題の正解手順は 5 手あり、一手ずづ 3 × 3 パターンと 5 × 5 パターンを抽出する。この棋譜から図 8 の 3 × 3 パターンおよび図 9 の 5 × 5 パターンを取得することができる。また、四角マークはパターンの中心を示しており、図 9 (1)(2)(3)(4) は上に盤外 2 マスを含むものである。

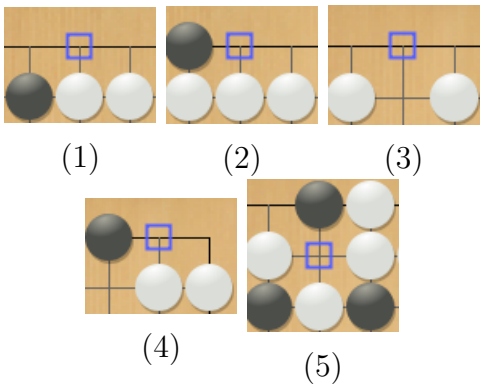


図 8: 詰碁の棋譜から抽出した 3 × 3 パターン

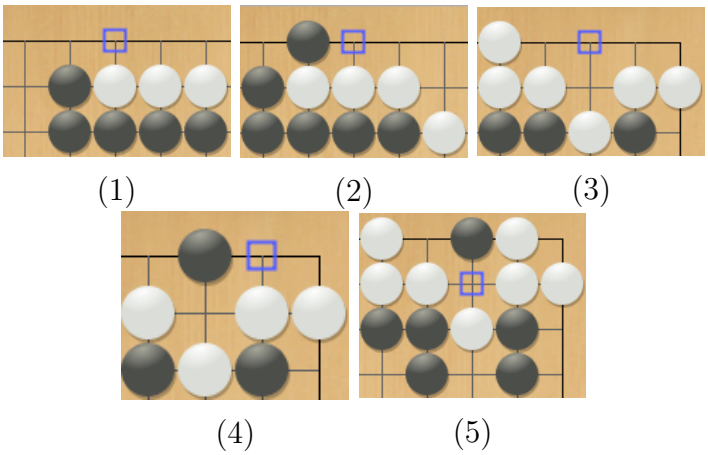


図 9: 詰碁の棋譜から抽出した 5 × 5 パターン

以下で 3 × 3 パターンおよび 5 × 5 パターンの実装について述べる。

4.3.1 3 × 3 パターン

パターンマッチの速度を考える上で、ビット列を使ってパターンを表現する。盤面の状態を自分の石、相手の石、空点および盤外として分けることができる。表 6 はそれぞれの状態のビット表現を示す。

表 6: 盤面上の点の状態のビット表現

自分の石	相手の石	空点	盤外
01	10	00	11

3 × 3 範囲の点に順序を表 7 のように付ける。位置 5 は着手点の位置を示す。

表 7: 3 × 3 範囲の順序

1	2	3
4	5	6
7	8	9

18(= 9 × 2) ビットで 3 × 3 パターンを表現することができる。これを int 型で表現し、下位の 18 ビットを使い、上位の 14 ビットを 0 で埋めるようにする。図 10 のキリのパターンから表 8 のビットパターンが得られる。

表 8: 3 × 3 ビットパターン

順序	1	2	3	4	5	6	7	8	9
ビットパターン	00	10	01	00	00	10	00	00	00

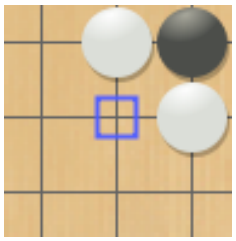


図 10: 3 × 3 パターンの例（黒石が自分、白石が相手とする）

すべて出現可能なパターンの数は 2^{18} ある。このサイズなら配列で持つことが可能なので、サイズ 262144 の int 配列でビットパターンを持つようにする。また、パターンマッチするときに、配列の添字で検索を行う。

人間が同じ形と認識する、回転または対称変換したパターンはコンピュータでは違うパターンと認識するので、一つのパターンを取るときに回転、対称変換を行う必要がある。それらの処理を実装し、一つの形から八つの回転、対称変換したパターンを生成する。

4.3.2 5 × 5 パターン

5 × 5 パターンは範囲を拡大したパターンである。範囲を拡大したことで、3 × 3 パターンといくつかの処理が異なる。

5 × 5 範囲の点に表 9 の順序を付ける。位置 13 は今の着手の位置とする。

5 × 5 パターンを表現するのに、50 ビットが必要となる。long long int 型（64 ビット）の下位 50 ビットを使い、上位の 14 ビットを 0 で埋める。

表 9: 5 × 5 範囲の順序

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

5 × 5 パターンの出現可能の数は 2^{50} あるので、配列で持つことは効率的ではない。よって、ハッシュ表を使って、5 × 5 ビットパターンを保持する。

4.3.3 詰碁専用パターンと対局用のパターンとの比較

詰碁の棋譜から得られたパターンの特徴を求めるため、対局用のパターンと比較を行う。詰碁のパターンデータは本実験で取ったものを使い、対局用のパターンデータは同研究室の渡部の実験 [4] で取ったデータを使う。

詰碁専用パターンは 200 局の棋譜から生成し、平均 5 手についてパターンを抽出した。

対局用のパターンは 9365 局の棋譜から生成し、平均 200 手についてパターンを抽出した。

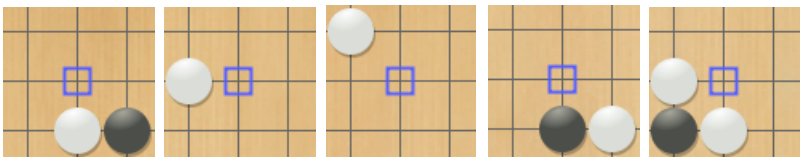
それぞれのパターンの総数は表 10 に示す。

表 10: パターンの総数

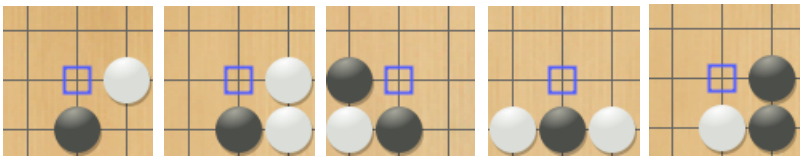
詰碁専用パターン	対局用のパターン
1,528	6,866,438

普通の棋譜から抽出したパターンと詰碁棋譜から抽出したパターンのそれぞれの特徴を比較するために、出現回数順で上位 10 位までのパターンを比較する。

出現回数が最も多い 10 個のパターンを比較する。対局用の 3 × 3 パターンを図 11 に示す。図 11 の対局用のパターンは、ツゲ、ハネ、ノビ、キリなど名前のある手が多いことがわかる。表 11 はそれぞれのパターンの出現回数である。



(1) (2) (3) (4) (5)



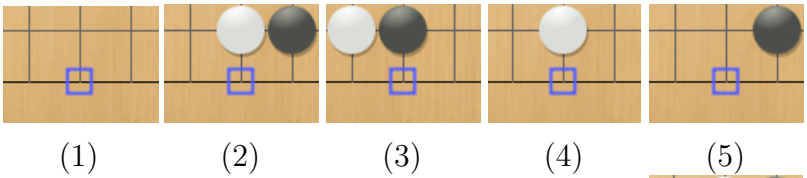
(6) (7) (8) (9) (10)

図 11: 対局用の 3 × 3 パターンの出現回数上位 10 位（黒石が自分、白石が相手とする）

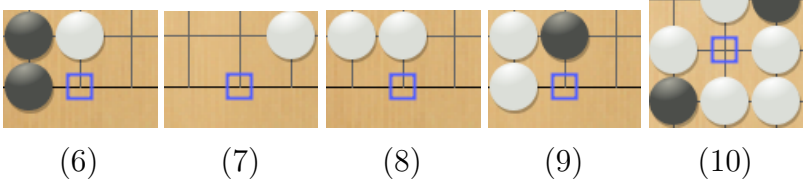
表 11: 対局用の 3 × 3 パターン出現回数

番号	(1)	(2)	(3)	(4)	(5)
出現回数	137,804	135,958	131,748	86,034	58,743
番号	(6)	(7)	(8)	(9)	(10)
出現回数	51,789	47,350	37,425	34,566	33,563

図 12 は詰碁専用パターンの出現回数上位 10 位を表したものである。表 12 は図 12 の詰碁専用パターンの出現回数である。対局用の 3 × 3 パターンの出現回数上位にあるのは、ほとんど盤面の中央に位置するパターンである。これと違って、詰碁専用 3 × 3 パターンの出現回数上位 10 個はほとんど盤面の端にある。また、図 12 (8) の目を奪う手と図 12 (9) の石を取る手のような詰碁に役に立つ可能性が高い手が出てくる。



(1) (2) (3) (4) (5)



(6) (7) (8) (9) (10)

図 12: 詰碁専用 3 × 3 パターンの出現回数上位 10 位（黒石が自分、白石が相手とする）

表 12: 詰碁専用 3 × 3 パターン出現回数

番号	(1)	(2)	(3)	(4)	(5)
出現回数	26	22	19	14	13
番号	(6)	(7)	(8)	(9)	(10)
出現回数	13	11	11	10	10

次に 5 × 5 パターンについて比較する。まず対局用のパターンの出現回数上位 10 位のパターンを図 13 で示す。表 13 はその出現回数である。対局用の 5 × 5 パターンの出現回数上位 10 位までは、碁盤中央のパターンしかなく、石の数は少ない傾向にある。石が多くなると、同じ形が出にくいことがわかる。

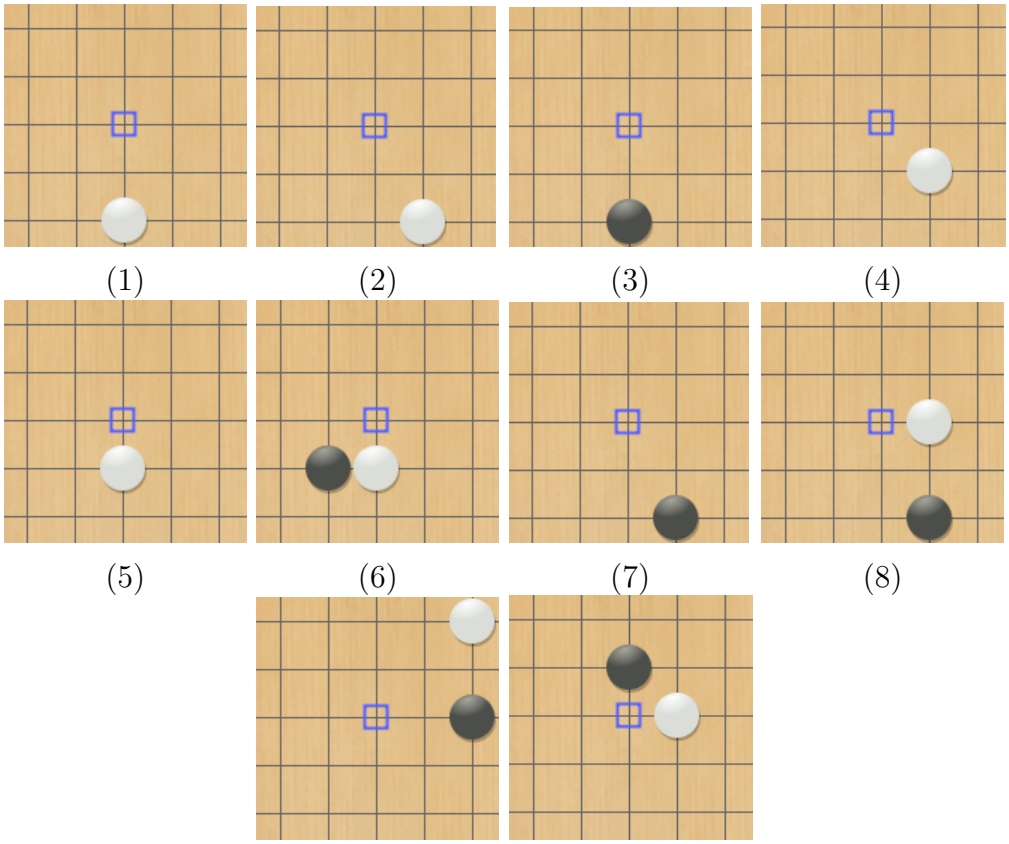


図 13: 対局用の 5 × 5 パターンの出現回数上位 10 位（黒石が自分、白石が相手とする）

表 13: 対局用の 5 × 5 パターン出現回数

番号	(1)	(2)	(3)	(4)	(5)
出現回数	49,660	23,523	15,180	13,456	12,424
番号	(6)	(7)	(8)	(9)	(10)
出現回数	9,342	8,070	6,942	6,534	5,504

詰碁専用 5 × 5 パターンの出現回数上位 10 位を図 14 で示す。表 14 はその出現回数である。対局用の 5 × 5 パターンに対し、詰碁専用 5 × 5 パターンは主に二つの特徴がある。

1. 石の数が多い。
2. 碁盤の端に位置するパターンがほとんどである。

また、形から見ると、自分の目を作る手と相手の目を奪う手が最も多いことがわかった。

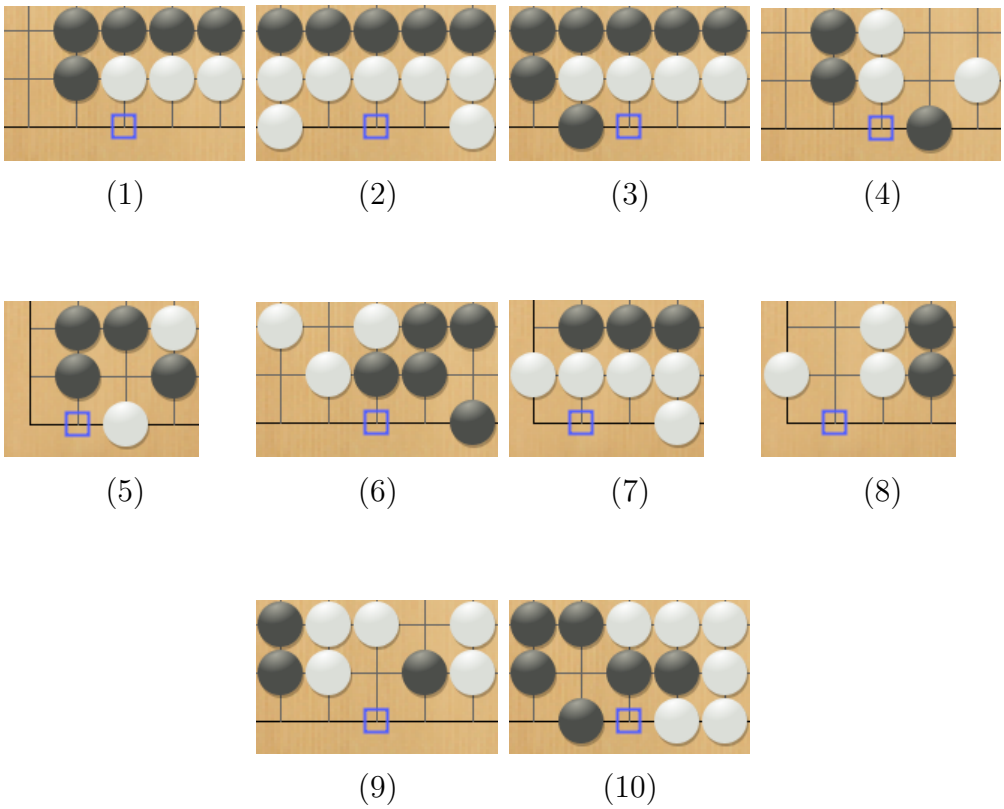


図 14: 詰碁専用 5 × 5 パターンの出現回数上位 10 位（黒石が自分、白石が相手とする）

表 14: 詰碁専用 5 × 5 パターン出現回数

番号	(1)	(2)	(3)	(4)	(5)
出現回数	6	3	3	2	2
番号	(6)	(7)	(8)	(9)	(10)
出現回数	2	2	2	2	2

5　手順を得るための試み

詰碁問題は次の一手が一通りしかなく、その手が正解となる。しかし、通常の詰碁問題の解答はその一手だけではなく、相手の最も強い抵抗とともに、何手かの正解手順が示されていることがほとんどである。

正解手順を求めることこそ、問題を解くにあたって最も（手筋を明らかにする）面白いところである。この章では、正解の一手だけではなく、その手順を出すことを試みる。

5.1　Min-Max ルート

まず UCT 探索木を上から、捕獲率の高いところをたどる道が正解手順であることが考えられる。これを Min-Max ルートと呼ぶことにする。

例として、Min-Max ルートを図 15 の赤い線で示す。自分の手の場合 Max の 0.86 を選ぶ、相手の場合 Min の 0.83 を選ぶ。このようにリーフまでの Min-Max ルートを出すことができる。

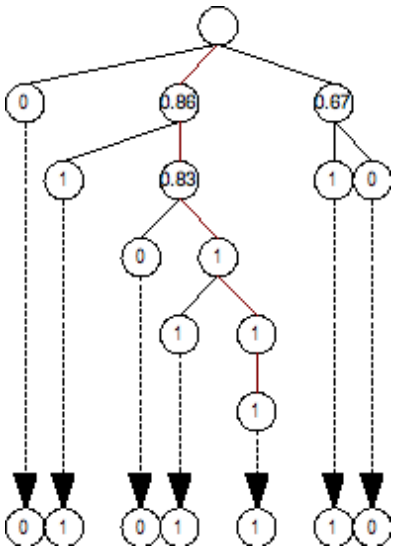


図 15: Min-Max ルートの例

次の詰碁問題の例で、Min-Max ルートを探してみる。

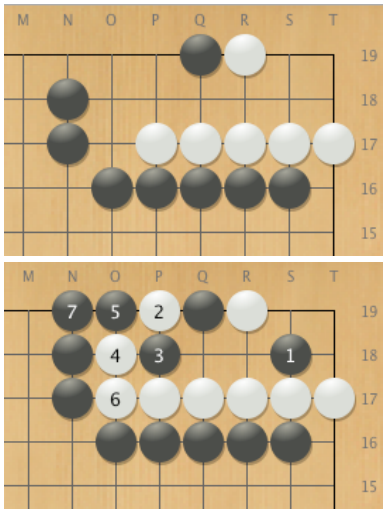


図 16: 詰碁問題とその解答

図 16 の上は詰碁問題で、下は人間の考える正解手順である。この問題についてランダムシミュレーション回数を 10,000 回にして解いてみると、図 17 はプログラムの解答であり、表 15 は一手ずつの捕獲率とシミュレーション回数を示す。

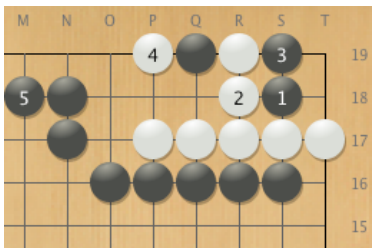


図 17: プログラムの解答（シミュレーション回数 10,000 回）

表 15: Min-Max ルート（シミュレーション回数 10,000 回）

順番	位置	捕獲率	シミュレーション回数	色
1	S18	0.453651	3150	黒
2	R18	0.383436	652	白
3	S19	0.483516	91	黒
4	P19	0.300000	10	白
5	M18	1.000000	1	黒

一手目は正解とあっていたものの、二手目白の抵抗は最も強いところから外れている。この方法では、手の数が少なめの詰碁（中手など）は、シミュレーション回数を増やすとよい手順が得られる。しかしこの例のように、手の数が多い問題だと、シミュレーション回数を限界まで増やしても 2、3 手までしか信用できる値が得られない。図 16 の問題に対して、シミュレーション回数を 500,000 回に増やし、解いた結果を図 18 で示し、表 16 は一手ずつの捕獲率とシミュレーション回数を示す。3 手目までの正解手順を得ることができた。

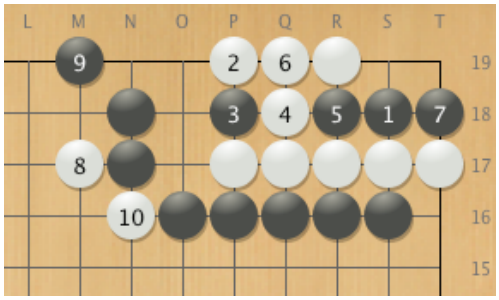


図 18: プログラムの解答（シミュレーション回数 500,000 回）

表 16: Min-Max ルート（シミュレーション回数 500,000 回）

順番	位置	捕獲率	シミュレーション回数	色
1	S18	0.509422	479810	黒
2	P19	0.502314	16205	白
3	P18	0.527831	9971	黒
4	Q18	0.499547	6624	白
5	R18	0.536378	4357	黒
6	Q19	0.492340	2872	白
7	T18	0.546358	302	黒
8	M17	0.444444	45	白
9	M19	0.833333	12	黒
10	N16	0.000000	1	白

5.2　モンテカルロ碁を用いた木探索

一手目の正解だけではなく、詰碁を解く手順を得るための方法を考える。モンテカルロ碁で得られた Min-Max ルートは、深さ 2 までが信用できる値と考える。詰碁の手順の長いものは、7,8 手かかる場合も普通にある。より深く探索できるようにするには、次のモンテカルロ碁を用いた直線探索の方法が考えられる。

1. モンテカルロ碁を行い、次の 2 手を得る。
2. 得られた 2 手を実際に打って、次の盤面を得る。
3. 捕獲率が 1 か 0 になるまで 1. と 2. を繰り返す。

つまり、何回かモンテカルロ碁を連続して行うことで、詰碁の手順を求める方法である。図 19 はモンテカルロ碁を用いた直線探索の流れ図である。

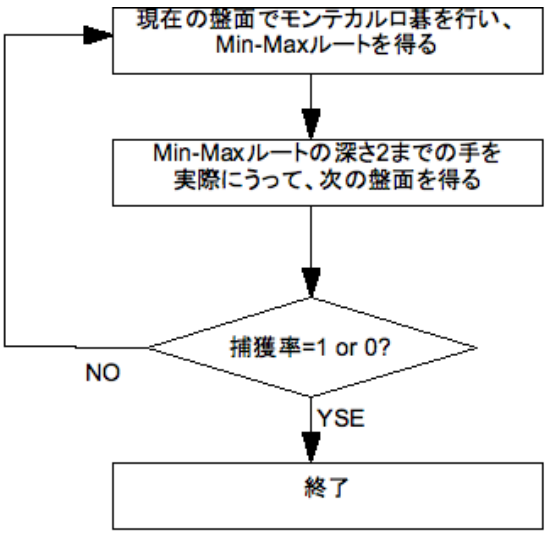


図 19: モンテカルロ碁を用いた直線探索の流れ図

この方法で図 20 の詰碁問題を解くと、次の結果が得られる。シミュレーション回数は各 2 手あたり 10 万回である。プログラムで得た解答は表 17 で示す。

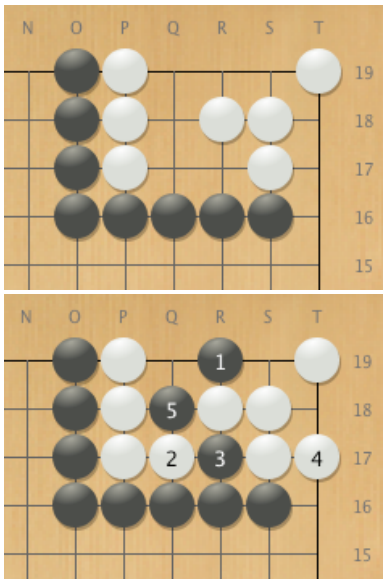


図 20: 詰碁問題とその解答

表 17: プログラムで得た解答

順番	位置	捕獲率	シミュレーション回数	色
1	R19	0.658068	66519	黒
2	Q17	0.644826	40414	白
3	R17	0.715695	96632	黒
4	T17	0.707995	7380	白
5	Q18	0.911251	99471	黒
6	Q19	0.896009	38734	白

この問題について、モンテカルロ碁を用いた直線探索の方法を使ったプログラムの解答が人間の解答と一致することを確認した。

しかし、直線探索途中、モンテカルロ碁で得た Min-Max ルートの深さ 2 までの手から構成するノードは必ず人間の答えと一致するとは限らない。木探索のなかで一つでも間違うと、正確な手順を得ることができない。より高い確率で正確な手順を得るために、一つのノードを選択するのでなく、二つのノードを調べると考える。この方法は、正解の手は高い確率で、捕獲率の最も高い二つの手のどちらかにあるだろうという考え方に基づいている。以下はモンテカルロ碁を用いた木探索について述べる。

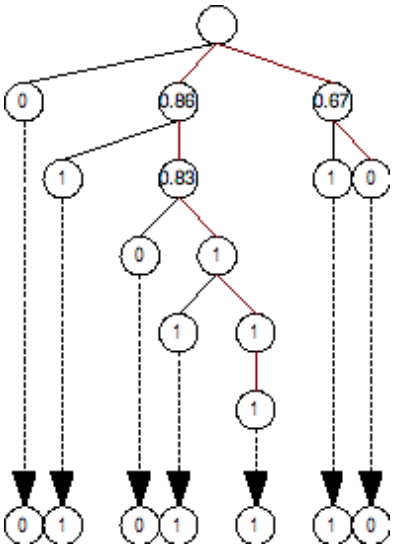


図 21: モンテカルロ碁の例

モンテカルロ碁を用いた木探索のノードは、自分の手と相手の抵抗の手と、二手のセットからなる。図 21 の UCT 探索木から 2 つの二手セットノードを作成できる。

Max: 0.86 Min: 0.83

Max: 0.67 Min: 0

このように、一回の UCT 木探索を行うことで、2 つの二手セットノードが得られる。得られたノードを使い、木を作成する。図 22 は作成した木の例である。ただし、矢印は探索の順序、〇の中の数字はノードの一手目の捕獲率、×はその枝が刈られていることを表す。

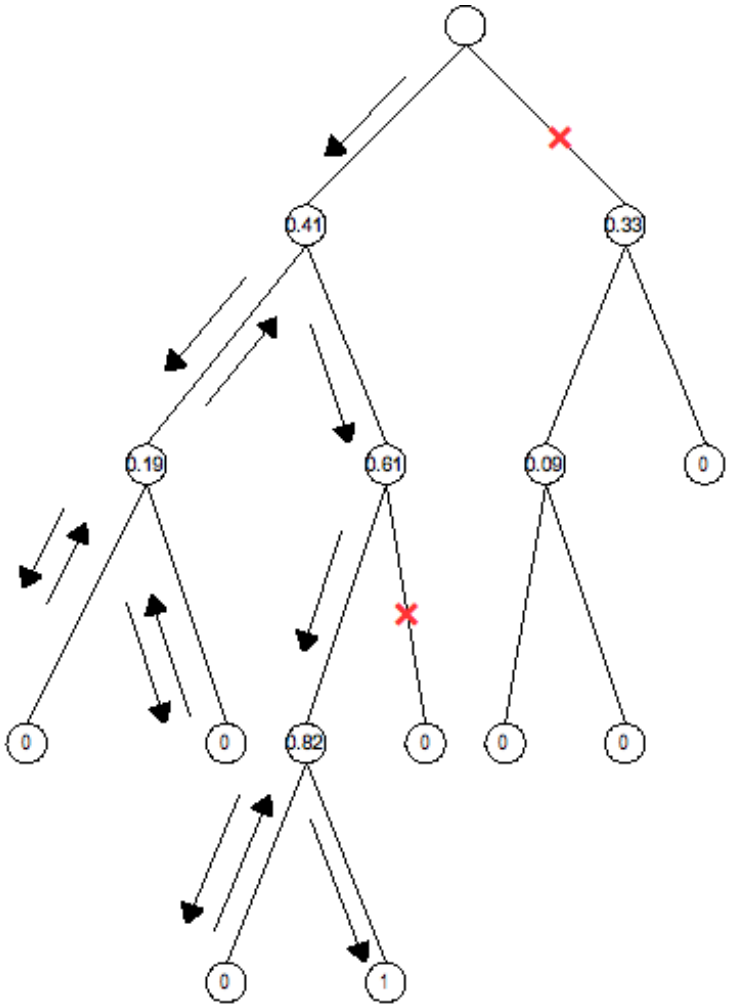


図 22: モンテカルロ碁を用いた木探索

モンテカルロ碁を用いた木探索アルゴリズムを表 18 で示す。また、配列変数 node は手順を表している。

表 18: モンテカルロ碁を用いた木探索の擬似コード

```
function MonteKi(rootNode)
  i = 0;
  node[i] = rootNode;
  while(node[i] の捕獲率が 1 でない)
    if(node[i] を通ったことがない)
      node[i] についてモンテカルロ碁を行い、
      二つの子ノードを作る;
      i = i + 1;
      node[i] = 左の子ノード;
    else
      i = i + 1;
      node[i] = 右の子ノード;
    end if;
    if(node[i] の捕獲率が 0 である)
      i = i - 1;
      while(node[i] の二つの子ノードが調べ済み)
        i = i - 1;
      end while;
    end if;
  end while;
  return node;
end function;
```

6 実験結果と考察

実験環境は以下の通りである。

- OS: MAC OS X 10.5.5
- CPU: Intel Core Duo 2G
- MEM: 2G

実験用の詰碁問題は「もっとひとめの詰め碁」[7] からの 50 問と、goproblems [9] からの 60 問（難易度は簡単、中程度、難しいの 3 種類から、それぞれ 20 問）を使用した。どちらもセキとコウを除いた問題とする。

実験は二段階に分けて行う。一段階目は、ランダムシミュレーションの精度改良と速度向上を行ったプログラムで詰碁問題の一手目の正解を求める。二段階目は、モンテカルロ碁を用いた木探索アルゴリズムで正解の手順を求める。

6.1 一手目の正解を求める実験

まず一手目の正解を求める実験で、局所的なプレイアウトありとなしの両方について実験を行った。両方ともパターンありである。ランダムシミュレーション回数を 10,000 回、100,000 回、500,000 回に分けて詰碁を解ける確率を求めた。表 19 はそれぞれの平均実行時間である。

表 19: ランダムシミュレーション回数ごとの平均実行時間

シミュレーション回数	局所的なプレイアウトあり (秒)	局所的なプレイアウトなし (秒)
10,000	2.794	6.652
100,000	29.009	71.043
500,000	160.673	363.272

6.1.1 ランダムシミュレーション回数 10,000 回の場合

ランダムシミュレーション回数 10,000 回の場合、局所的なプレイアウトなし平均実行時間は 6.652 秒であり、局所的なプレイアウトあり平均実行時間は 2.794 秒である。一手目の正解率を表 20 で示す。また、正解率を表す値の ± の右部分は信頼区間を表している。

表 20: ランダムシミュレーション回数 10,000 回の場合の正解率

モード	一手目の正解率
局所的なプレイアウトなし	19.1% ± 7.7%
局所的なプレイアウトあり	35.5% ± 9.3%

ランダムシミュレーション回数 10,000 回の場合、手数の少ない簡単な詰碁である中手の問題を解くことができた。三目中手から六目中手までの基本的な形をすべて解けることを確認した。また、手数の少ない簡単な詰碁を解くことができた。図 23 に解けた詰碁問題の例を示す。

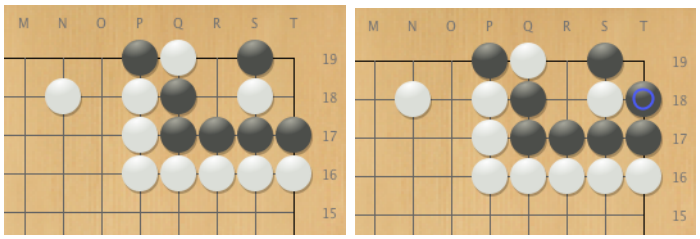


図 23: 解けた詰碁の例（黒先活、ランダムシミュレーション回数 10,000 回）

6.1.2　ランダムシミュレーション回数 100,000 回の場合

ランダムシミュレーション回数 100,000 回の場合、局所的なプレイアウトなし平均実行時間は 71.043 秒であり、局所的なプレイアウトあり平均実行時間は 29.009 秒である。一手目の正解率を表 21 で示す。

表 21: ランダムシミュレーション回数 100,000 回の場合の正解率

モード	一手目の正解率
局所的なプレイアウトなし	37.3% ± 9.4%
局所的なプレイアウトあり	83.6% ± 7.3%

図 24 に解けた詰碁問題の例を示す。

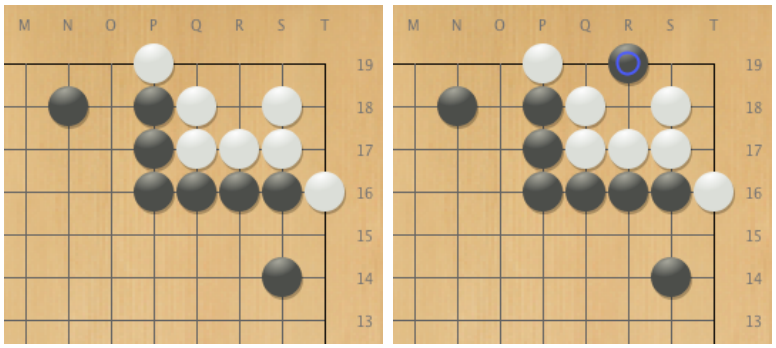


図 24: 解けた詰碁の例（黒先白死、ランダムシミュレーション回数 100,000 回）

6.1.3　ランダムシミュレーション回数 500,000 回の場合

ランダムシミュレーション回数 500,000 回の場合、局所的なプレイアウトなし平均実行時間は 363.272 秒であり、局所的なプレイアウトあり平均実行時間は 160.673 秒である。一手目の正解率を表 22 で示す。

表 22: ランダムシミュレーション回数 500,000 回の場合の正解率

モード	一手目の正解率
局所的なプレイアウトなし	42.7% ± 9.6%
局所的なプレイアウトあり	88.2% ± 6.5%

図 25 に解けた詰碁問題の例を示す。

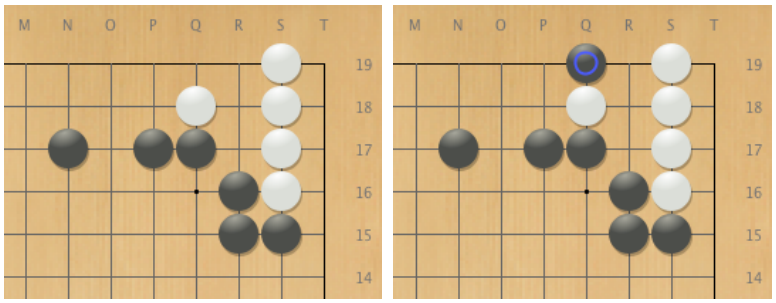


図 25: 解けた詰碁の例（黒先白死、ランダムシミュレーション回数 500,000 回）

この例は手数が多くて深く先読みしないと生死がわからない局面である。この詰碁問題は局所的なプレイアウトなしで 500,000 回シミュレーションしても正解を得ることができなかったが、局所的なプレイアウトを使うことで解くことができた。

局所的なプレイアウトなしモードと局所的なプレイアウトありモードにおいて、ランダムシミュレーション回数 10,000 回、100,000 回、および 500,000 回の正解率を図 26 と図 27 にまとめた。

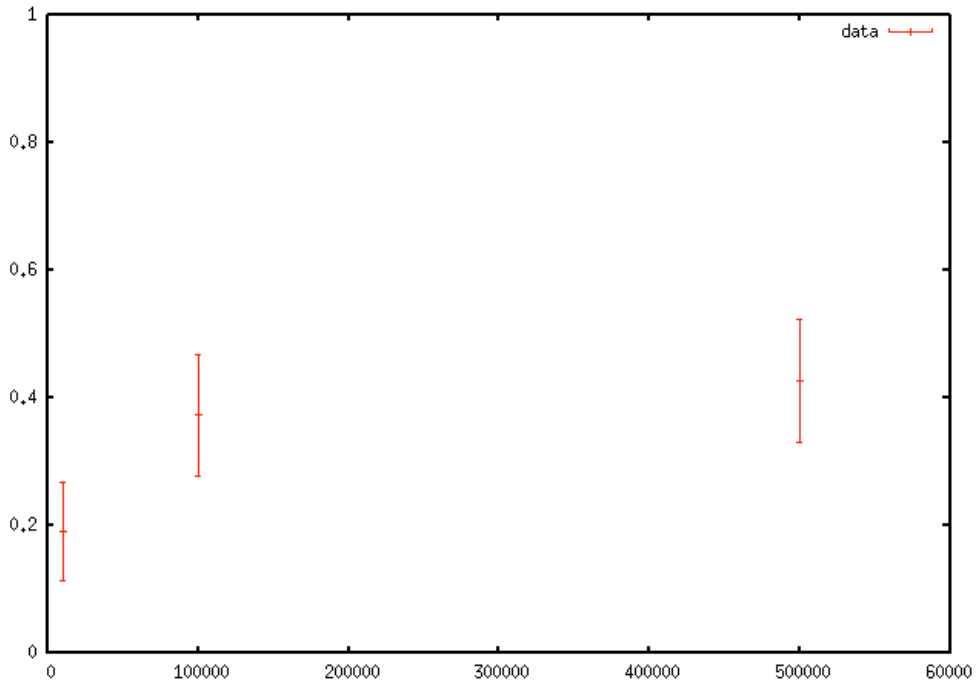


図 26: 局所的なプレイアウトなしモードの正解率

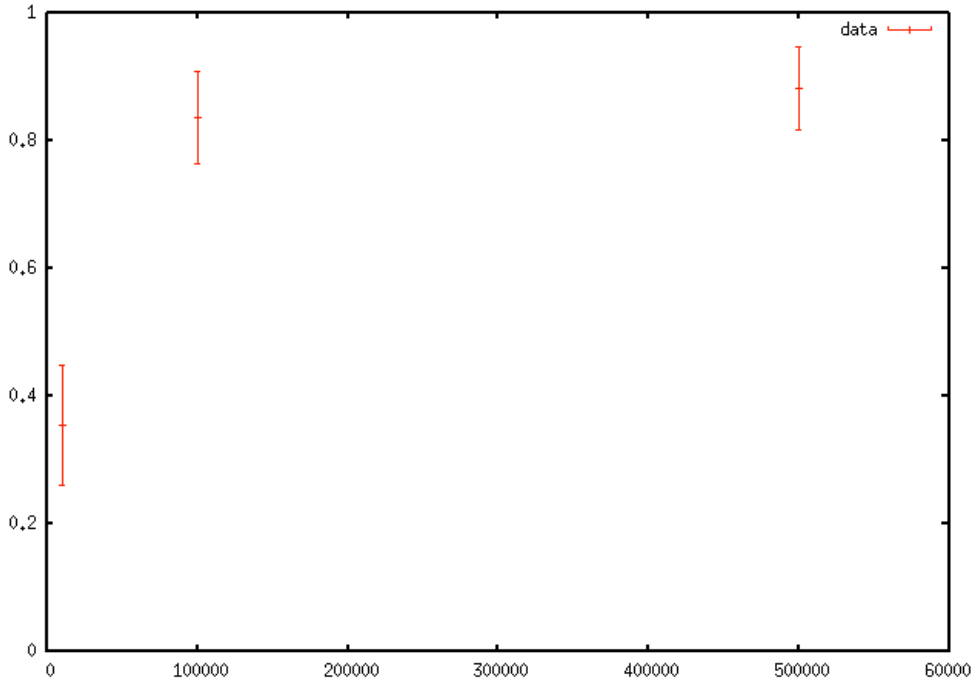


図 27: 局所的なプレイアウトありモードの正解率

6.1.4 セキとコウ

囲碁では、石を取りに行った方が逆に取られてしまうために両者とも手が出せない状態がある、このことをセキという。セキは両方の石が生きていると考えられる。

詰碁の正解はセキとなるものもある。図 28 は黒先セキの例である。

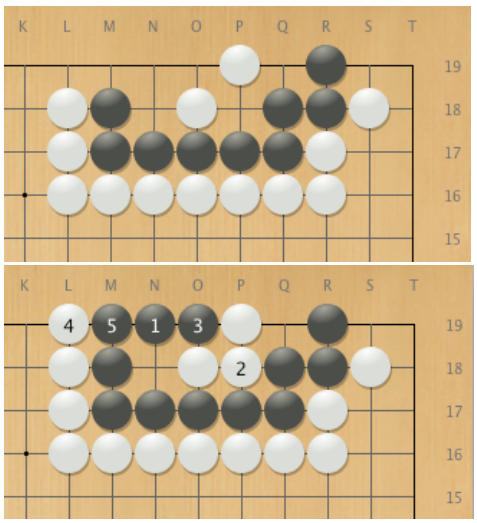


図 28: 黒先セキ

セキは特殊な形である。本実験のプログラムはランダムシミュレーション中、知識として目の判定しか入れていない。従って、ランダムシミュレーション中にセキの形が出たら、必ず黒か白かどちらか一方がセキを埋める手を打つ。最終盤面はセキが残らない。しかし、本実験の詰碁プログラムでは、セキの判定をしなくても、図 28 の正解を得られることを確認した。そして、最初の一手の捕獲率が 50%に近いことを確認した。この値は黒を取る確率と白を取る確率が同じであることを示していると考えられる。

また、ほかの黒先セキ詰碁問題 4 問を解いてみて、一手目の捕獲率が 50%に近い一定の範囲にあることを確認した。

コウは囲碁のルールの一つで、お互いが交互に相手の石を取り、無限に続きうる形である。しかし、下記のルールで、無限反復は禁止されている。

- 対局者の一方がコウの一子を取った場合、もう一方は他の部分に一手打ち、相手がそれに受けたときに限り、コウの一子を取り返すことができる。

図 29 は黒先コウの例である。

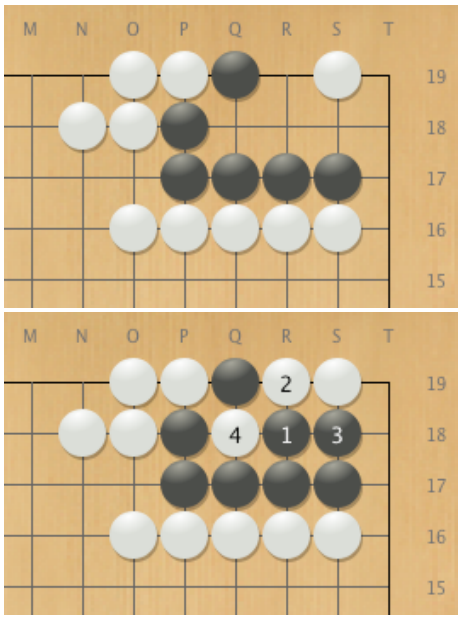


図 29: 黒先コウ

本実験の詰碁プログラムを用いてこの問題を解けるかどうかを確認してみた。プログラムで解いたときの一手目は T17 で、捕獲率が 0.438934 である。コウの詰碁問題を解くことができなかった。その原因は、ランダムシミュレーション中、コウの場所を取ったり取られたり、またいつ埋めるかいろいろな可能性があって、捕獲率が不安定であるからだと考えられる。

6.2 正解手順を求める実験

直線探索と木探索との両方を用いて、詰碁問題の正解手順を求める実験を行った。両方とともに局所的なプレイアウトと詰碁専用パターンを使用する。表 23 に実験結果を示す。また、正解率を表す値の ± の右部分は信頼区間を表している。

表 23: 直線探索の正解率と木探索の正解率との比較

モード	一手目の正解率	手順の正解率
直線探索	83.6% ± 7.3%	71.8% ± 8.8%
木探索	90.9% ± 5.8%	79.1% ± 8.0%

直線探索と木探索とを比較するため、両方の一手目の正解率を図 30 に、手順の正解率を図 31 に示す。

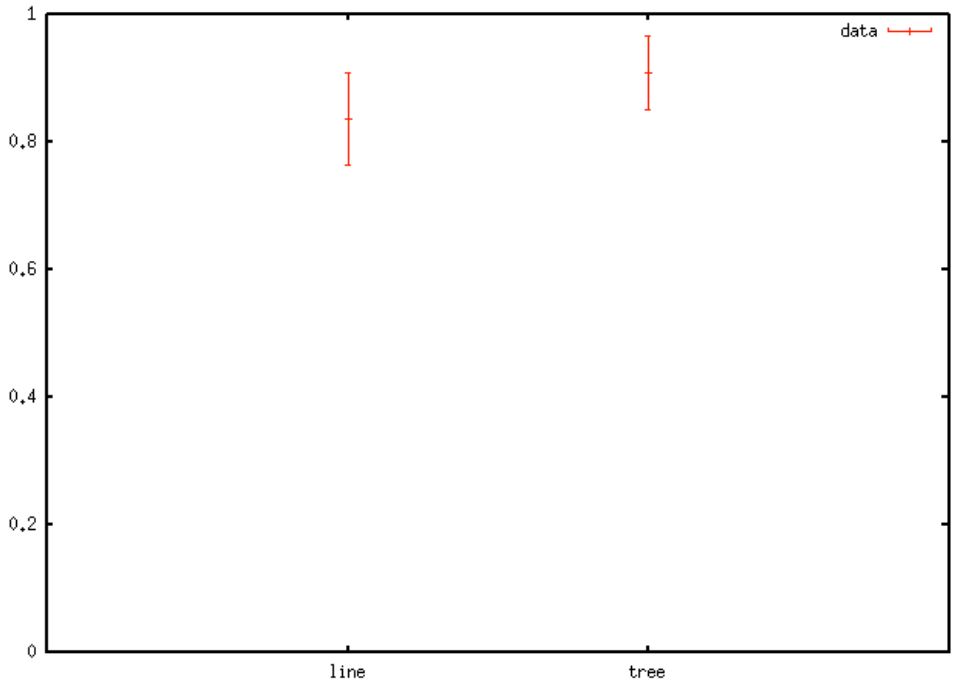


図 30: 一手目の正解率の比較

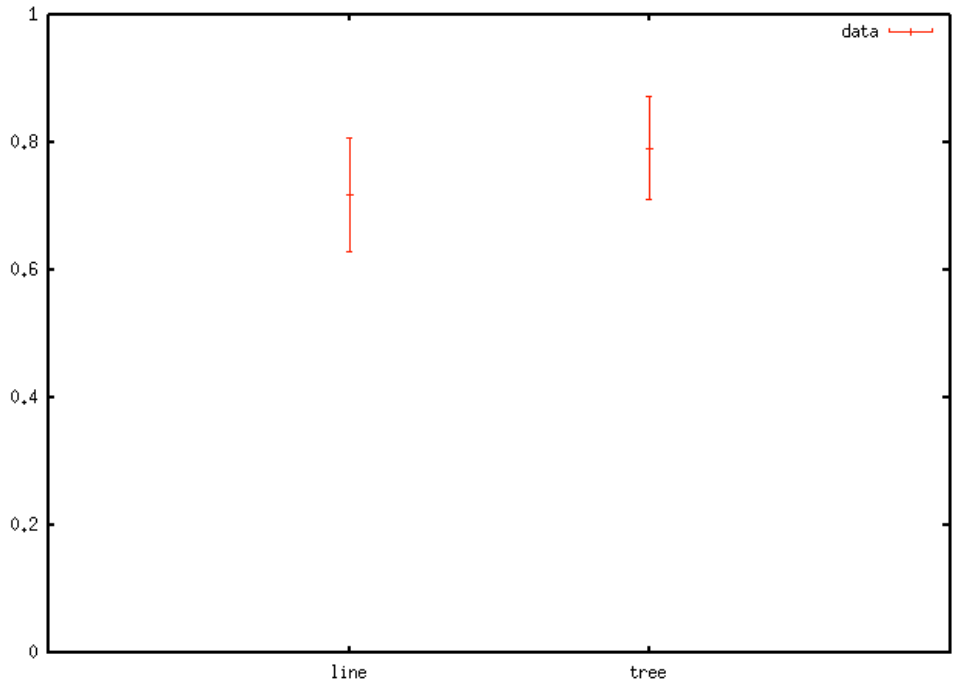


図 31: 手順の正解率の比較

詰碁問題の正解手順を求める実験では、一手目と二手目の抵抗が正解であったら、その次の手が正解から外れることが少ない。木を深く探索する程、捕獲率が高くなり正解から外れる確率が低くなると考えられる。

実験結果から見ると、直線探索を木探索に拡張することによって、詰碁問題の正解手順が求められる確率を高めることを確認した。これは詰碁問題におけるモンテカルロ碁では、捕獲率が二番目のノードも有望な手である可能性があることを意味すると考えられる。

7　結論

本研究では、モンテカルロ碁に対する詰碁を解くための改良をいくつか行った。まず詰碁問題はほとんど生死の問題であるからランダムシミュレーションに捕獲率を導入した。捕獲率はシミュレーション中指定した石を殺せる確率を表す。UCT 探索はこの値について行う。詰碁専用パターンを自動抽出するプログラムを作成し、ランダムシミュレーションの精度をあげることを試した。次に、詰碁問題は盤面上に一定の範囲にしか石がないことから、モンテカルロ碁に局所的なプレイアウトという技法を使用し、ランダムシミュレーションの速度を 2.3 倍程向上させることに成功した。また、この技法で正解率を 2 倍程向上させたことを確認した。

詰碁問題の正解手順を得るために、モンテカルロ碁における最も捕獲率が高い道を Min-Max ルートと定義し、Min-Max ルートと正解手順との関係について調べた。その結果、

Min-Max ルートが正解手順と一致することが確認したが、ランダムシミュレーション回数の限界があることによって Min-Max ルートの 2、3 手までのノードしか信用できない。正解手順をより深く探索するため、モンテカルロ碁と木探索とを併用することを考えた。木のノードはモンテカルロ碁の捕獲率順上位の手とそれに対する抵抗の手からなる。本研究では捕獲率順上位 2 位までの手を木に加えた。この方法を用い、実験用詰碁問題の手順の正解率を大幅に高めることに成功した。また、解けない問題の中では、手数が多く、一本道で進んでいけず途中の対応が難しいのが多く現れた。

今後、探索木のノード数をさらに増やし、モンテカルロ碁の捕獲率順上位 3 位またはそれ以上のノードを使用し木を構成することで、一手ずつのシミュレーション数と木のノード数とを調節し、正解率がさらに上がる可能性があると考える。

8 謝辞

村松研究室に所属してから 3 年間にわたり、熱心なご指導をしてくださった村松正和教授に心より感謝いたします。また、普通対局パターンのデータを提供した渡部庸史氏に深く感謝いたします。最後に、苦楽をともにした村松研究室の皆様、どうもお世話になりました。

参考文献

[1] B.Bouzy, B.Helmstetter, *Monte-Carlo Go Developments*, Advances in Computer Games, pp.159-174(2003)

[2] Sylvain Gelly, Yizao Wang, Rémi Munos, Olivier Teytaud, *Modification of UCT with Patterns in Monte-Carlo Go*, Technical Report RR-6062, INRIA (2006).

[3] Rémi Coulom, *Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search, In P.Ciancarini and H.J. van den Herik, editors, Proceedings of the 5th International Conference on Computer and Games*, Turin, Italy (2006).

[4] 渡部 庸史, モンテカルロ碁における大きなパターンを用いたシミュレーションの改良, 電気通信大学 電気通信学研究科 情報工学専攻, 修士論文 (2009)

[5] 清 愼一, 山下 宏, 佐々木 宣介, コンピュータ囲碁の入門, 共立出版 (2005)

[6] 金川 正明, ポケット詰碁 200, 日本棋院 (2001)

[7] 趙治勲，もっとひとめの詰め碁，毎日コミュニケーションズ (2007)

[8] GoGui
http://gogui.sourceforge.net/

[9] goproblems
http://www.goproblems.com/