モンテ・カルロ碁における UCT のパターンによる改良

Sylvain Gelly[*], Yizao Wang[†], Rémi Munos[‡], Olivier Teytaud[§]

Thème COG — Systèmes cognitifs
Project TAO

概要: 多腕バンディット問題の UCB1 アルゴリズムは既にミニマックス木の探索用 UCT (Upper bound Confidence for Tree) アルゴリズムへと拡張されている. 我々はモンテ・カルロ碁プログラム, MoGo (UCT を使った最初のコンピュータ囲碁プログラム), を開発した. UCT に加えた囲碁用の修正と, パターンを用いた知的なランダム・シミュレーション (これは MoGo の性能を著しく改善した) について述べる. 大きな碁盤用に枝刈り技法を組合わせた UCT, さらに UCT の並列化を議論する. MoGo は現在, 9路と13路でトップレベルのプログラムである. 〈訳注: 2006年12月現在, MoGo の KGS でのランクは9級. なお GNU Go は6級, 彩は9級である. 以下, 〈〉 内は訳注〉.

Keywords: Computer Go, Exploration-exploitation, UCT, Monte-Carlo, Patterns

訳注: 本文書は RR-6062.pdf (20 December 2006版) の和訳である. 本翻訳ならびにその公開は専ら日本のコンピュータ囲碁界への貢献を目的としており, 原著者らの権利を侵害する意図はない. 論文等での引用 (citation) に際しては必ず原報告を引用し, (必要なら) 本翻訳の引用を付加すること. 本翻訳の公開を快諾してくれた原著者に感謝する. 訳者は細心の注意を払ったが, もし誤訳等を見つけた時は mailto:gg@nue.ci.i.u-tokyo.ac.jp まで連絡して頂ければ幸いである.

[*] project TAO, INRIA-Futurs, LRI, Batiment, 490, Université Paris-Sud 91405 ORSAY CEDEX, France
[†] Centre de Mathématiques Appliquées, École Polytechnique, 91128 PALAISEAU CEDEX, France
[‡] Centre de Mathématiques Appliquées, École Polytechnique, 91128 PALAISEAU CEDEX, France
[§] project TAO, INRIA-Futurs, LRI, Batiment, 490, Université Paris-Sud 91405 ORSAY CEDEX, France

1. はじめに

囲碁の歴史はおよそ四千年におよび，今でも世界中で多くの人々を楽しませている．そのルールは単純だが (http://www.gobase.org に包括的な入門がある) その複雑さは70年代遅くから始まった優れたコンピュータ囲碁プログラムを作る幾多の試みを退けてきた [4]．現時点で最も優れたコンピュータ囲碁プログラムのレベルは弱いアマチュア程度である．今や囲碁はチェスとその役割を交代し，人工知能の最も難しい挑戦の一つであると思われる．

囲碁は多くの面でチェスと異なっている．第一に木のサイズと分岐因子が非常に大きい．典型的な碁盤の大きさは (チェスの8路に対し) 9路から19路の間で，潜在的な手の数はチェスの数十に対し数百におよぶ．第二に局面のミニマックス値を近似する効率的な評価関数が手に入らない．これらの理由から，コンピュータ・チェスのプログラムで使われている強力な $\alpha\beta$ 探索 ([14] を見よ) は囲碁には力不足だった．

最近，囲碁の局面の評価に関してモンテ・カルロ法 [6] による進展があった (第2章で詳しく述べる)．しかしこの評価手続きの精度には限界がある; 各局面で最もスコアが高い手を打つことが最終的な勝利に直結しない．むしろそれは各ステップでの候補手の数を制限せざるを得なくさせる．さらに，良い囲碁プログラムに要求される探検対収穫〈exploration *versus* exploitation; EvE〉探索戦略の実施に標準的な強化学習を利用するアプローチ [14] は，(離散的な) 探索空間の大きさゆえに困難である．

そこで本論文では他の探検対収穫戦略の道具立てとして，ゲーム理論から生まれた多腕バンディット問題を検討する．多腕バンディット問題は，全報酬の最大化を目的として過去の選択と報酬に基づいてスロットマシンを選ぶギャンブラをモデル化したものである [2]．多腕バンディットの枠組みで Auer らが提案した UCB1 アルゴリズム [1] は，最近 Kocsys らによって木構造探索空間へと拡張された (UCT アルゴリズム) [12]．

我々が提示する囲碁ソフト (MoGo) の主な貢献は，(i) UCT アルゴリズムの囲碁向けの修正，(ii) モンテ・カルロ評価関数でのパターンの利用 (我々のオリジナル) の二つである．アルゴリズム (動的な木構造 [9] と並列化の実装) やヒューリスティック (シンプルな枝刈りヒューリスティクス) についても議論する．MoGo は囲碁ソフトとして比較的良いレベルに達した: 2006年8月以降，MoGo は9路のコンピュータ囲碁専用サーバ (CGOS[1]) において142プログラム中の1位である．また，国際的な Kiseido Go Server[2] で2006年の10月と11月に開かれた全トーナメント (9路と13路) で優勝した．

本論文の構成は以下のとおり．第2章で関連する研究をざっと紹介する (読者には基本的な囲碁知識を仮定する)．第3章では我々の貢献に焦点をあて，MoGo で用いている広い探索空間用の UCT の実装と，パターンに基づく事前知識を用いてモンテ・カルロ評価にバイアスを与える手法に

---

[1] http://cgos.boardspace.net
[2] http://www.weddslist.com/kgs/past/index. html

ついて述べる．第4章で我々の実験結果を報告し議論する．本論文を MoGo の改良に関する知識とコンピュータ〈のパワー〉に焦点をあてた展望で結ぶ．

2. 過去の関連研究

我々のアプローチはモンテ･カルロ碁と多腕バンディット問題に基づいており，各々2.1節と2.2節で述べる．2.3節で UCT アルゴリズムを与える．これは多腕バンディット技法をミニマックス木の探索に応用したものである．読者はミニマックス木と $\alpha\beta$ 探索を良く知っているものとする．

2.1 モンテ･カルロ碁

モンテ･カルロ碁，初出は1993年 [6]，は近年非常に多くの注意を引き付けている．モンテ･カルロ碁は，特に9路で驚異的に効率が良い; Rémi Coulom が開発した CrazyStone [9] (これは確率的なシミュレーションを用いたプログラムで，囲碁の知識はほとんど持ってない) が最も良く知られている[3]．

我々のプログラムもモンテ･カルロ碁の二つの原則にしたがっている．一，碁盤の状況を，ランダムなゲームを終局 (ここではスコアが容易く高精度で計算できるだろう) までシミュレートすることで評価する．二，モンテ･カルロ評価とミニマックス木探索とを組み合わせて用いる．我々のプログラムも CrazyStone の木構造を使っている．

注1　我々が言う「木」は実際にはほとんどの場合方向付きグラフだが，用語として「木」が広く用いられている. Graph History Interaction (GHI) 問題 [11] に関しては，我々はこれを，特にコンピュータ囲碁の他の難しさと比べ，非常に重大であるとは考えていないので，無視している．

2.2 バンディット問題

*K*腕バンディットは，伝統的なスロットマシン (腕が1本のバンディット) の腕が2本以上の版との類似に基づく簡単な機械学習問題である．プレイ時，各腕はその腕固有のある分布から定まる「報酬」を提供する．ギャンブラの目的は，インタラクティブなプレイ[4]を通じて集めた報酬の和を最大化することである [4]．伝統的に，ギャンブラは最初は腕に関する知識を何も持たず，試行の繰り返しを通じて最も報酬の高い腕に集中することができるようになるものとされている．

バンディット問題の論点は，既に得られている知識に基いて報酬を最大化することと，知識を増やすために新しい行動を取ることとのバランスを取る問題に関係しており，強化学習では「収穫と探検のジレンマ〈exploitation-exploration dilemma〉」として知られている．つまり，バンディット問題における「収穫」はこれまでに集めた知識から現時点で最善の腕を選ぶことであり，他方「探検」は他の腕を選んでその腕に関する知識を増やすことに対応している．

---

[3] CrazyStone は第 11 回コンピュータ･オリンピアードの 9 路碁で, GNU Go，彩, Go Intellect などの強豪を破り金メダルに輝いた．

[4] 一般的な多碗問題の時は「腕をプレイする」，囲碁の時は「手を打つ」と言う．囲碁では「プレイ」はゲームではなく手に対して使う．

K腕バンディット問題はランダム変数 $X_{i,n}$ $(1 \le i \le K,\ n \ge 1)$ で定義される. ここで i はスロットマシン (バンディットの腕) の番号. マシン i での連続プレイは, ある未知の法則にしたがい未知の期待値 μ_i をもち同一で独立な分布の報酬 $X_{i,1}$, $X_{i,2}$, … を生む. ここで独立性はマシンをまたぐ報酬にも成立している; つまり $X_{i,s}$ と $X_{j,t}$ $(1 \le i < j \le K,\ s, t \ge 1)$ は独立である (多分同一分布でもない). アルゴリズムは, これまでのプレイで得た結果から次にプレイするマシンを選ぶ. T_i (n) をマシン i が, 初めの n プレイの後プレイされた回数とする. アルゴリズムが常に最善の選択を行うとは限らないので, その損失の期待値を考える. N 回のプレイ後の後悔〈損失の期待値〉は次式で与えられる.

$$\mu^* n - \sum_{j=1}^{K} \mu_j E[T_j(n)],\ \text{ここで}\ \mu^* = \max_{1 \le i \le K} \mu,$$

$E[]$ は期待値を示す. Auer and Al. は, シンプルなアルゴリズム UCB1 を与えた [1] で, 報酬が [0, 1] の範囲の時, 最適なマシンは一様に他のマシンより指数関数的に多くプレイされることを保証している.

$$\overline{X}_{i,s} = \frac{1}{s} \sum_{j=1}^{s} X_{i,j},\quad \overline{X}_i = \overline{X}_{i,T_i(n)}$$

に注意して以下を得る.

アルゴリズム1. 決定論的な方策 (policy): UCB1

- 初期化: 各マシンを1回プレイする.
- 繰り返し: $\overline{X}_j + \sqrt{\frac{2 \log n}{T_j(n)}}$ が最も大きくなるマシン j をプレイする. ここで n はこれまでの総プレイ回数.

[1] に, より良い実験結果を得た式がある.

$$V_j(s) = \left(\frac{1}{s} \sum_{\gamma=1}^{s} X_{j,\gamma}^2 \right) - \overline{X}_{j,s}^2 + \sqrt{\frac{2 \log n}{s}}$$

をマシン j の分散の上界の推定値とし, 最大化すべき新しい値を得る.

$$\overline{X}_j + \sqrt{\frac{\log n}{T_j(n)} \min\{1/4, V_j(T_j(n))\}} \quad (1)$$

Auer and Al. によると, 各腕の経験値の分散も考慮して (1) を最大化する方策 (UCB1-TUNED) は彼らの実験の全てで UCB1 よりはっきり良かった. これは我々の初期の結果とも一致するので, 我々のプログラムでは常に UCB1-TUNED 方策を使っている[5].

---

[5] 今後これを縮めて UCB1 と呼ぶ.

2.3 UCT: 木探索用 UCB1

UCT [12] は UCB1 [1] をミニマックス木探索用に拡張したものである. その考え方は, 各ノードを独立したバンディットと, そしてその子ノードを独立した腕と見なすことである. 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バンディットのシーケンスを一定時間プレイする.

UCT アルゴリズムを表1に示す[6]. このプログラムは一度に一つのシーケンスをプレイし, それを繰り返す (行1から行8). 行9から行21の関数は UCB1 を用いて腕 (UCT では子ノード) を選んでいる. 行15はさらなる探索の前に各腕が一度選ばれることを保証する. 行16は UCB1 の式を計算している. 一つのシーケンスのプレイ後, プレイされた腕の値を, リーフの親ノードからルートまで (行22から行29の関数 updateValue に書かれている) 式 UCB1 にしたがって更新する[7]. ここでミニマックス処理が行われている. 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.

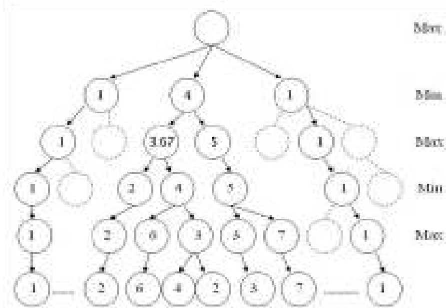

図1. UCT 探索. 木の形は非対称的に大きくなる. 既に探訪されたノードの更新された値 (node[i].value) だけが示されている.

ミニマックス木探索問題で我々が探しているのはたいてい, ルート･ノードにおける最適な枝である. 特に囲碁の様に木が非常に深く分岐が多い時は短時間で最適な枝を見つけるのは非常に難しいので,往々にして最適な枝に近いスコアの枝が見つかればよいこともある.

この意味で UCT は $\alpha\beta$ 探索より優れている. 実際, 大きな利点を3つ挙げることができる. 第1に, UCT は「いつでも」流 〈anytime manner〉 で動く. このアルゴリズムは「いつでも」停めることができ, その性能はそれなりに良い可能性がある. これは $\alpha\beta$ 探索にない特徴である. 図2は, $\alpha\beta$ アルゴリズムを早目に停めた時に第 1 レベルの手がいくつか探索されないままになる様子を示している. つまり, 選ばれた手は最適から遥かに遠い可能性がある. もちろん反復深化 〈iterative deepening〉 でこの問題を部分的に解決することはできる. しかし依然, 「いつでも」性は UCT の方が強く, 細かい時間制御は UCT アルゴリズムの方が容易である.

---

[6] これは説明を明確にするための擬似コードで, 最適化は考慮してない.
[7] ここではアルゴリズム 1 の元の式を用いている.

```
 1:  function playOneSequence(rootNode)
 2:      node[0] := rootNode; i = 0;
 3:      while(node[i] is not leaf) do
 4:          node[i+1] := descendByUCB1(node[i]);
 5:          i := i + 1;
 6:      end while;
 7:      updateValue(node, 1 – node[i].value);
 8:  end function;

 9:  function descendByUCB1(node)
10:      nb := 0;
11:      for i := 0 to node.childNode.size() – 1 do
12:          nb := nb + node.childNode[i].nb;
13:      end for;
14:      for i := 0 to node.childNode.size() – 1 do
15:          if node.childNode[i].nb = 0
                 do v[i] := ∞;
16:          else v[i] := 1 – node.childNode[i].value
             / node.childNode[i].nb
             + sqrt(2*log_e(nb)/(node.childNode[i].nb)
17:          end if;
18:      end for;
19:      index := argmax(v[j]);
20:      return node.childNode[index];
21:  end function;

22:  function updateValue(node, value)
23:      for i := node.size() – 2 to 0 do
24:          node[i].value := node[i].value + value;
25:          node[i].nb := node[i].nb + 1;
26:          value := 1 – value;
27:      end for;
28:  end function;
```

表1. ミニマックス木用 UCT の擬似コード

第2に, UCT は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供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. したがって, もしある子ノードが他より非常に大きな値を持ち, 推定が良ければ, この子ノードは他より非常に高い頻度で探索され, UCT はほとんどの時間‘最大’の子ノードを選ぶ. しかし, もし2つの子ノードの値がほぼ等しいか信頼度が低ければ値は平均値に近づく.

第3に, 木は非対称的に成長する. UCT は良い手をより深く探索する. しかもこれは自動的に行われる. 図1に例を示す.

図1と2は時間が限られた時の二つのアルゴリズムで探索された木の違いを明確に示している. しかしながら UCT の理論的な解析は発展途上である [13]. 我々はこの点に関する意見を本節の終

わりに記すにとどめる. UCT に関わるランダム変数が同じ分布でも独立でもないことは自明であり, これが収束の解析を複雑にしている. 実際我々は腕 i のバイアスを次式で定義できる:

$$\delta_{i,t} = \left| \mu_i^* - \frac{1}{t}\sum_{s=1}^{t} X_{i,s} \right|,$$

ここで μ_i^* はこの腕のミニマックス値である. リーフ・レベルで $\delta_{i,t} = 0$ は明らか. 同様に,

$$\delta_{i,t} \le K^D \frac{\log t}{t}$$

が証明できる. ここで K は定数, D は腕の深さ (ルートから降る向きに数える). これは, 深いレベルからルートへ伝わる時にバイアスが増幅されるという事実に対応しており, このアルゴリズムがルート・ノードで最適な腕を素早く見つけるのを妨げる.

UCT の利点の一つは, それが自動的に'真の'深さに順応することである. ルートの枝の'真の'深さとは, そこから先は $\delta_{i,t} = 0$ になる場所の深さである. これらの枝のバイアスの上界は d ($d < D$) を真の深さとして $K^d \frac{\log t}{t}$ である. こういう枝の値は他の枝の値より速く収束するので, UCT は他の面白そうな枝により長い時間を割く.

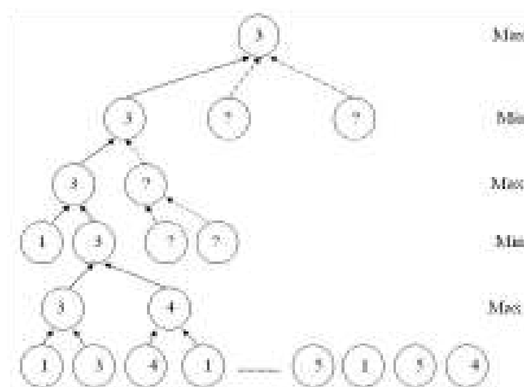

図2. 時間が限られた $\alpha\beta$ 探索. '?' はまだ探索されてないノード. これは完全な探索が不可能な巨大な木の探索ではよく起きる. 反復深化はこの問題を部分的に解決する.

3. 主な成果

本章では UCT アルゴリズムを用いた我々のプログラム MoGo について説明する. 3.1節で UCT の適用について述べる. その後, 強いモンテ・カルロ碁に重要な2つの側面, シミュレーション (つまりはスコアの推定) の質と木の深さについて検討し, 続く2節で各々に対する我々の改良点を示す. 3.2節でパターンを用いた手順っぽい (sequence-like) ランダム・シミュレーションについて述べる. 3.3節で大きな碁盤での木探索の枝刈りのアイデアを与える. 3.4節で未探訪ノードを探索する順序の変更について述べる. 最後, 3.5節は並列化である.

```
1:   function playOneSequenceInMoGo(rootNode)
2:       node[0] := rootNode; i := 0;
3:       do
4:           node[i+1] := descendByUCB1(node[i]); i := i + 1;
5:       while node[i] is not first visited;
6:       createNode(node[i]);
7:       node[i].value := getValueByMC(node[i]);
8:       updateValue(node, 1 – node[i].value);
9:   end function;
```

表2. MoGo の UCT の擬似コード

3.1 コンピュータ囲碁への UCT の適用

MoGo は図3に示すように大きく2つ，木探索部とランダム・シミュレーション部から構成される．木の各ノードは碁盤のある状況を，子ノードは対応した手を打った後の状況を，各々表現している．

コンピュータ囲碁への UCT の適用は，碁盤の各状況は，各合法手が未知だがある一定の分布の報酬を持つ腕に対応したバンディット問題である，という仮説に基づいている．勝ちと負けの 2種類の腕しかないものとし，各々に報酬 1 と 0 を割り当てる．囲碁では引き分けは滅多に起こらないので無視する．

木探索部はメモリを節約するために，CrazyStone [9] と同様の動的な木構造を導入した UCT の節約版を用いている．したがって以下で説明する様に，各シミュレーションの後でノードを1つ加えることにより，木はインクリメンタルに創られる．これは [12] のものと違い，シミュレーションでより少ないノードしか生成せず，より効率的である．別の言い方をすると, 3 回以上探訪したノードしか保存しないのでメモリが大幅に節約され，シミュレーションも加速される．この擬似コードを表 2 に示す．繰り返すが，最適化は考慮してない．

各シミュレーション中，MoGo はメモリに保存されている木のルートからスタートする．MoGo は各ノードで UCB1 の式 1 にしたがって手を選ぶ．続いて MoGo は，選ばれた子ノードに降りて新しい手を選ぶ (ここでも UCB1 にしたがう) ことを，そういう生成されたノードがなくなるまで繰り返す．この部分は行1から行5のコードに対応している．木探索部〈の実行〉は，この新しいノード (実はリーフ) の生成で終わる．これは createNode で完了する．続いて MoGo は，このリーフ局面のスコアを与えるためにランダム・シミュレーション部，対応する関数 getValueByMC (行 7), を呼ぶ．

ランダム・シミュレーション部では，ランダム・ゲームを，当該局面から終局 (ルールにしたがってスコアを素早く正確に計算できる) までプレイする．このランダム・シミュレーションで訪れたノードは保存しない．ランダム・シミュレーションを行い，スコアを受け取り，このシミュレーションで通ったノードの値を更新する[8].

[8] 木探索部中に終局局面に達する可能性がある．この場合スコアを直ちに計算でき，そのノードを生成する必要もランダム・シミュレーション部を呼ぶ必要も無い．

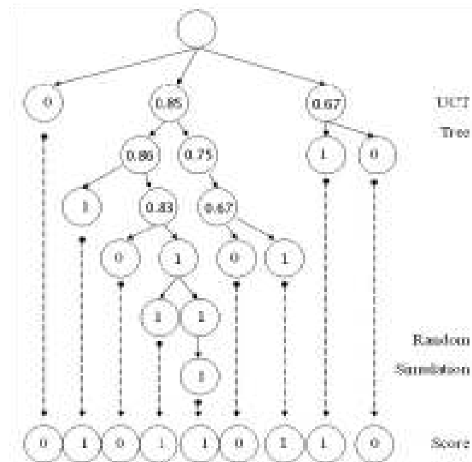

図3. MoGo は UCT を使った木探索部とスコアを与えるランダム・シミュレーション部から成る. 一番下の数値はゲームの最終スコア (勝/負) に対応し, ノード中の数値はそのノードの更新された値 (node[i].value) である.

注2 スコアの更新では, この方がずっとずっと頑健なので, 地の代わりに 0/1 スコアを用いている. ゆえに各ノードの真のミニマックス値は 0 か 1 のはずである. しかし実際には, UCT は各ノードを [0, 1] の範囲のある重み付き平均値で近似している. 通常この値は勝つ確率と考えられている.

3.2 囲碁の知識を利用したシミュレーションの改良

本節では我々の, パターンを用いた手順っぽいランダム・シミュレーションを紹介する. 古典的なランダム・シミュレーション (純粋ランダム・モードと呼ぶ) と比べて, その利点は明らかである. 以下では我々の改良ランダム・モードとパターンの実装について述べる.

ランダム・シミュレーション部では, 信頼できるスコアを与える賢いシミュレーションであることが非常に重要である. Indigo [3] (同様のパターンは [15] にもある) に触発されてシンプルな 3×3 のパターンを用いたところ, 我々のランダム・シミュレーションは前より意味のある手順になったようで, MoGo のレベルは著しく向上した.

我々のと, 他のパターンを用いたモンテ・カルロ碁プログラム ([3]) とのパターンの使い方の本質的な違いは, 我々はパターンを, ランダム・シミュレーション中ローカルな応手を見つけることにより, 意味のある手順を生成するために使っている点である. 着手が大局的に良い手である必要はない. モンテ・カルロ評価をより正確にするのに, 良い手より良い手順を得る方がより重要であることは自明ではない. しかし我々の経験は, 改善は主にローカルな応手の利用に因ることを示している. 面白い手を見つけるために, 同じパターンを直前の手の近くではなく全盤面で使うと, 精度は低下する. 我々はこの主張は自明ではなく, MoGo の主な貢献の一つであると信じている. また我々は, パターンを木の枝刈りには使ってない. 我々は Gnu Go の様な他のプログラムに実装されている, より

洗練されたパターンは調べてない.

我々の純粋ランダム･モードでは, プログラムが自分の眼を埋めるのを阻止する規則が少しだけあるが, 合法手は盤上一様ランダムに打たれる. 我々はまた石を取る手を優先している. 厳密にこのモードを使っている MoGo というボット〈囲碁プログラム･プレイヤのこと〉は CGOS でランク･スコア 1647 ELO[9] に達した. 現在の MoGo のランクは 2200 ELO 弱である.

我々は, ほとんどいつも無意味なゲームしか与えない純粋ランダム･シミュレーションには満足できなかったので, より合理的な手を得るためにローカルなパターンを導入した. 我々のパターンは石が打たれる中央が空いた 3×3 の点〈intersection〉で定義される. 我々のパターンは, その手がそういうローカル (3×3) な状況で面白い〈interesting〉か否かを調べるいくつかの関数から成る. 詳しく言うと, その手が囲碁の古典的な形, 例えばキリやハネなど, になっているか否かを調べる.

さらに, 我々は面白い手を, 直前に打たれた手の周りでしか探さない. この理由は, ローカルに面白い手は直前の手の応手になっていることが多いようで, ゆえにローカルに面白い手が続けて何手か (調べられた後) 打たれるとローカルな手順が姿を現すからである.

改良ランダム･モードが手を生成する方法を簡単に述べる. 最初に, 直前の手がアタリかどうか調べる; もしアタリでかつアタリになっている石を (石を取るかダメを増やすかのどちらかで) 助けることができるなら, 助ける手の一つをランダムに選ぶ; そうでなければ, 直前の〈相手の〉手の周囲 8 箇所から面白い手を探し, あれば〈その中から〉ランダムに一つ選び打つ; そうでなければ盤上の石を取る手を探し, あればそれを打つ. 最後に, もしまだ手が一つも見つかってなければ盤上ランダムに打つ. 確かに, 実際の MoGo のコードを細かく見ると, 囲碁の知識を実現する, 手で書かれた多数の小さな関数で複雑になっている. しかし我々はここで述べた主要な枠組みが手順っぽいシミュレーションに最も重要であると信じている.

異なるモードを用いた2つのランダム･ゲームの最初の30手を図4に示す. 改良ランダム･モードが生成した手は明らかにずっと意味がある. ここで我々のパターンの詳細を述べる. パターンは次の手が打たれる中央の点 p が空いた 3×3 の点である. 各パターンは次に p に打つ手が面白か否かという問いに答える論理関数である. 真が返る (パターンがマッチした時) のは, 盤上の各位置の状態がそのパターンの対応する位置の状態と同じ時, あるいは対応する位置が × である (これはその位置の状態を無視することを意味する) 時に限る. 通常, 次の手の色に関する制約はない (黒の良い手は白にとっても良い手だろう). 特殊な場合はその時に説明する. パターンの対称性, 回転, および石の色の入れ替えは考慮されている. 手をパターンを使って調べるのは, それが合法でかつ自分をアタリにしない時に限る.

---

[9] ELO (http://en.wikipedia.org/wiki/Elo_rating_system) は勝率がランクの差と関係するレーティング･システム. 〈日本の wikipedia〉, 〈日本棋院幽玄の間のレーティング･システム〉

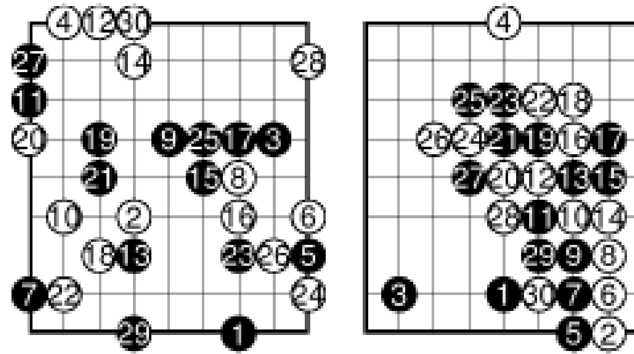

図4. 左: 純粋ランダム・モードでシミュレートしたゲームの序盤. 手はバラバラでほとんど意味がない. 右: パターンを使ったランダム・モードでシミュレートしたゲームの序盤. 5手目～29手目でこみ入った手順が生まれている.

我々は MoGo の開発中いくつかのパターンを試し, 最終的に図5～8のパターンを実装した. 図中, □ は次の手が打たれる場所を示している. 我々はパターンを手でコーディングして実装した. しかし, これを学習で達成できればもっと面白いだろう. 他のアプローチとして, Bouzy[5] に Bayesian generation を用いたものがある.

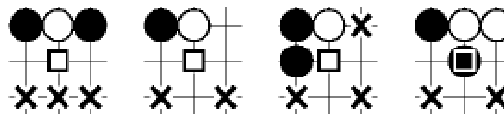

図5. ハネ・パターン. どれかにマッチすれば真を返す. 右端のパターンで □ が乗った黒石は, 周囲8点がマッチしかつ黒番の時に限り真を返すことを意味する.

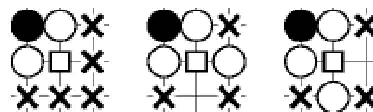

図6. キリ1パターン. キリ1のパターンは3パターンから成る. 最初のパターンがマッチし残りの2つがマッチしない時真を返す.

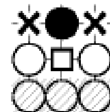

図7. キリ2パターン. 上の6点がマッチし下の3点が白でない時真を返す.

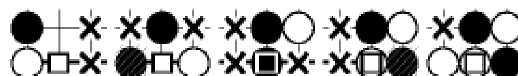

図8. 盤端パターン. どれかにマッチした時真を返す. 右の3パターンの□付きの黒 (白) 石は, 周囲がマッチし手番が黒 (白) の時に限り真を返すことを意味する.

注3　我々はランダム・モードでより‘良い’パターンを使うことが常に良いとは信じてない．すなわち，より重要なのはランダム・シミュレーションが何らかの意味で意味のある手を頻繁に生成することである．この主張に関してははもっと実験する必要がある．

表3はパターンが総合的な性能をいかに改善するかを明確に示している．

ランダム・モード	黒番の勝率	白番の勝率	全勝率
純粋	46% (250)	36% (250)	41.2%±4.4%
手順っぽい	77% (400)	82% (400)	80%±2.8%

表3. 異なるモードの比較. 70,000ランダム・シミュレーション/手. 9路.

3.3 枝刈り付き UCT のアイデア

この節では巨大な木のサイズを小さくする我々のアイデア (ヒューリスティクス) を示す．これは MoGo を広い碁盤で相対的に強くする．そこで我々は, UCT の大局的な視点を失うのと引き換えに, 探索木で局所的により大きな深さを手に入れる．枝刈りヒューリスティクスが準最適解を導く可能性があるのは自明である．最初に，木探索の分岐因子を減らすために囲碁の知識からグループを定義する．次にグループからゾーン分割を導く．これはより正確なスコアを得るのを助ける．我々はグループとゾーン・モードを13路と19路の碁で用いた．図9と図10に説得力があると思われる例を示す．

注4　我々は囲碁の知識を用いたプログラミングに精通しているわけではなく，またこれにあまり時間も割けなかったが, GNU Go や彩あるいは [7] [8] のモンテ・カルロ・プログラムのような他のプログラムはもっと賢い枝刈り技法を持っていると信じている．そういうわけで，スペースの制約のために我々はこの部分の擬似コードを示すことはできないが，我々は既に枝刈り付き UCT の勇気付けられる実験結果を得ている．

初めにグループを，確かな囲碁知識，例えば「大きな活きているグループとその近くの敵を集める」, にしたがって盤上のストリングと空点の集合として定義する．我々はグループの計算を助けるために我々のプログラムに運命共同体グラフ (common fate graph; CFG) [10] を実装した．この方法はストリング1個から始め，近くの空点とその〈空点の〉傍のストリングを，あるパラメタで制御される距離内でこれ以上傍のストリングが見つけられなくなるまで，再帰的に付け加える．

グループ・モードの場合，木探索部では碁盤全体ではなくそのグループ内の手だけを探索する．ランダム・シミュレーション部にはそういう制約はない．グループを用いることで序盤の分岐因子を50未満に減らすことができた．その時の木の深さは多分7～8位だろう．表4は MoGo が，グループ枝刈り技法を使うことで大きな碁盤でも戦えるようになったことを示している．しかし，コンピュータ囲碁

プログラムのレベルを上げるために洗練された枝刈り技法が必要なことは疑いない.

相手	黒番の勝率	白番の勝率	全勝率
グループなし vs. GnuGo level 0	53.2% (216)	51.8% (216)	52%±4.8%
グループなし vs. GnuGo level 8	24.2% (300)	30% (300)	27%±3.6%
グループあり vs. GnuGo level 0	67.5% (80)	61.2% (80)	64.3%±7.5%
グループあり vs. GnuGo level 8	51.9% (160)	60% (160)	56%±5.5%

表4. MoGo のグループ・モード・ヒューリスティックの有無. 70,000シミュレーション/手, 13路.

続いて我々はゾーン・モードを開発した. 上で説明した様に, グループ・モードでは木探索部の手の選択は制限されるが, ランダム・シミュレーション部には何の制約もない. ゾーン・モードでは, ランダムな手を碁盤全体ではなくある一定のゾーンの中でだけ生成する. この理由は, 大きな碁盤でのランダム・シミュレーションは, 信頼できるスコアを返すには長すぎる (19路で平均400手を超える) からである. このゾーンで得られたスコアはこの後全局向けに調整される.

調整というアイデアは, 我々のゾーン・モードでは非常に重要である. さもなければ, ランダム・シミュレーションによる局面の全局的な評価, これは大きな碁盤ではほとんどいつも非常に不正確である, がローカルな判断を大きく偏らせてしまう. 詳しく言うと, UCT 〈の適用〉 の前に, 現在の状況 (直前の相手の手を含む) から決定したゾーンのスコア $\bar{s}_0$ を, そしてゾーン外の領域のスコア $\bar{s}_1$ を, 一定回数 (我々の実験では5,000回) のモンテ・カルロ・シミュレーションにより推定する. そして s_1 を次式で調整する.

$$\hat{s}_1 = \alpha(\bar{s}_1 + \bar{s}_0 - komi) - (\bar{s}_0 - komi) \quad (2)$$

その後, ゾーン内の各シミュレーションの後, ローカル・スコア s_0 と合わせた $s_0 + \hat{s}_1$ を, シミュレーションの最終スコアとして返す. このやり方でスコアは常に $s_0 + \bar{s}_1$ で得られるものよりクロス・ゲームに近づく 〈白と黒のスコアが接近する〉. 調整しないとそのゾーンで一方が過大に評価されてしまうようである. これは MoGo が局所的により普通に指すのを助ける.

訳注: ここではスコアとして 0/1 ではなく地そのものを使っている. また, スコアに二種類あることに注意. 一つは UCT を用いた木探索の**前**に計算される $\bar{s}$, もう一つは UCT 木のノードで計算される s である. ローカル・スコア s_0 は UCT 木のノードの評価関数として計算される. なお, 2006年12月現在, 著者らは既にゾーン・モードを使ってない (私信). もう使ってないし, 説明も分かりにくいからゾーン・モードに関する記述は削除してはどうかと示唆されたが,.そうするとオリジナルのレポートと読み比べた時に混乱する可能性があること, これに触発されてより良いゾーン・モードを開発する読者がいないとも限らないことなどを考慮し, ここに残した.

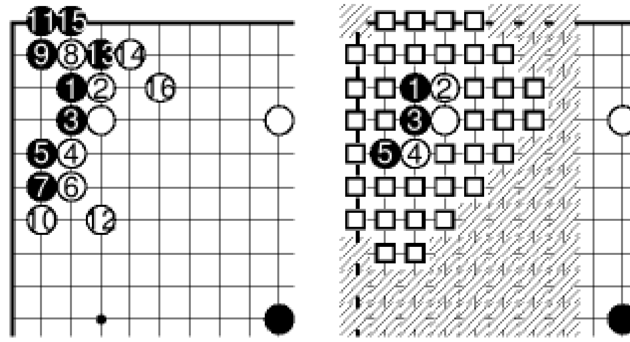

図9. MoGo (白) の KGS での試合の一部. 左図で MoGo はグループ・モードとゾーン・モードを使って左上隅でほぼ定石通りの応手を見つけた. 右図で直前の2手 (4 と 5) に関連した2つのグループの和 (union) に □ で印を付けた (空点のみ). この瞬間のゾーンの他の領域 (石とグループ領域以外) には影を付けた.

3.4 未探訪ノードを探索する順序の変更

UCT は, 探検と収穫のトレード・オフが UCB1 の式でうまく処理されてそのノードが頻繁に訪れられた時は非常にうまく働く. しかし, ルートから遠く離れたノードではシミュレーション回数が非常に少なく, UCT は探検に偏りすぎる傾向がある. これは, ある局面で可能な手は UCB1 の式を使う前に全て探検されていると想定されているためである. それゆえ, 深いノードの手の値はあまり意味がない. なぜなら, その子ノードは未だ全て探検されておらず, また, こちらの方がより悪い時もあるが, 探訪済みのノードは決まった順序で選ばれているから. これは悪い手順を予測する可能性がある.

より良い順序を得るために修正を二つ加えた.

先頭打着緊急度

UCB1 アルゴリズムは式 (1) を使う前に各腕を一度探検して始まる. これは時として, 特に試行回数が腕の数と比べて十分多くない場合, 効率を悪くする可能性がある. これは木に非常に多くのノードがある (探訪回数が手の数と比べて少ない) 場合である. 例えばある腕が常に 1 (勝ち) を返し続けるなら他の腕を探検する理由はない. 我々は先頭打着緊急度 〈first-play urgency〉 (FPU) と名付けた固定定数をアルゴリズムに設定した. 各手に対しその緊急度を式 (1) の値で与える. 最初の探訪の前に各合法手の緊急度に FPU の値を設定する (FPU の省略値は ∞) (表1の行15を見よ). 少なくとも一度探訪されればそのノードの緊急度は UCB1 の式にしたがって更新される. 我々は緊急度が最も高い手をプレイする 〈シミュレートする〉. したがって ∞ の FPU は, 既に探訪済みの手がさらに探訪される前に全ての手が一度探訪されることを保証し, 他方, より小さな FPU は, 初めのシミュレーション 〈複数〉 で緊急度が FPU より大きくなった時 (この場合他の未探訪ノードは選ばれない), 早期の収穫を保証する. これは表7に示した実験の様に MoGo のレベルを改善した.

訳注: この段落は原報告の誤りを, 原著者に確認の上訂正しているので, 原報告と若干異なる.

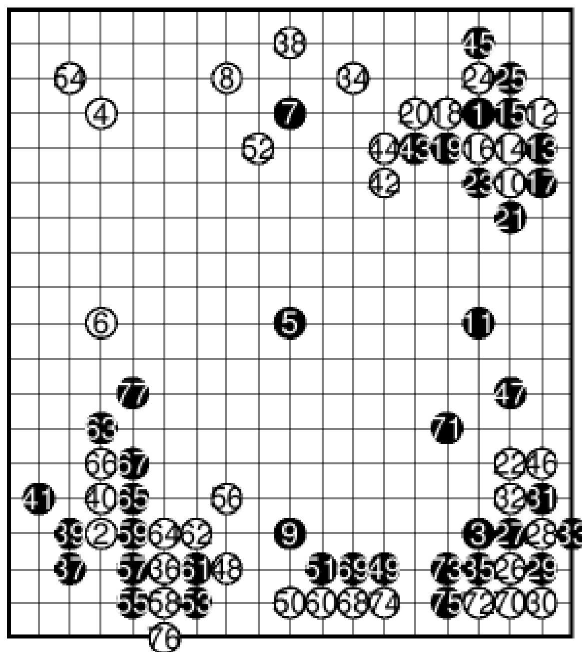

図10. 第18回 KGS コンピュータ囲碁トーナメントでの MoGo と Indigo の対局の序盤.
MoGo (黒) は序盤優勢だったが最後に負けた.

親の情報の利用

囲碁では, ある状況で良い手はそれ以降も良い手であることがけっこうある, と言われている. 我々が新しい状況に出会った時, ある手 m を勝手な順序で探検するのではなく, より良い順序を選ぶのに以前の局面での m の推定値を利用できる. 我々は通常ノードの親の親の推定値を使っている. これは大きな碁盤で MoGo を助けていると我々は信じているが, 重要な結果であると主張するには経験不足である.

3.5 並列化

UCT 〈の性能〉 は時間に良く比例するので, 我々は MoGo をメモリ共有型マルチプロセッサ・マシン上で走らせた. アルゴリズムの変更は全く直線的である. 全てのプロセッサは一つの木を共有し, mutex でロックしてアクセスする. UCT は決定的なので, 全スレッドがリーフを除いて完全に同じ経路を処理する可能性がある. したがって, ここで与えた様なマルチスレッド化 UCT の振る舞いは単一スレッドの UCT とは異なる. その後2つの実験を行った. 最初に我々は1手当たりのシミュレーション回数を等しくして, 単一スレッドとマルチスレッド化アルゴリズムを用いた MoGo のレベルを比較することができる. このような実験全てが, プレイ・レベルに大きな差は無いことを示した[10]. 次に一手当たりの時間を等しくして (マルチスレッド化版の一手当たりのシミュレーション回数は多くな

[10] 我々は4プロセッサのコンピュータしか使えなかったので, もっとプロセッサが多い時の振舞いは大きく異なる可能性がある.

る) レベルを比較することができる. UCT は計算パワーの増加が有効に働くので, マルチスレッド化 UCT は効率的である (4プロセッサで CGOS の ELO が100増える).

4. 結果

この章で我々はこのアルゴリズムの性質を反映した実験結果を示す. 全てのテストは MoGo と GNU Go 3.6 (モードは省略値) との対戦で, コミは7.5目. 表中, MoGo が黒番と白番で対戦した時の勝率にはその色で戦った回数 (括弧内) が添えてある. 信頼区間は95%.

4.1 時間依存性

我々のプログラムの性能は各手に与えられた時間 (シミュレーション回数と等価) に依存する. 表5はこの回数の増加につれてレベルが上がる様子を示している. 2プロセッサと4プロセッサ版 MoGo の傑出した性能もこの主張を支持している.

秒/手	黒番の勝率	白番の勝率	全勝率
5	26%±12% (50)	26%±12% (50)	26%±8.7%
20	41%±6% (250)	42%±6% (250)	41.8%±4.4%
60	53%±7% (200)	50%±7% (200)	51.5%±5%

表5. 純粋ランダム・モードで時間を変えた場合

4.2 パラメタ化 UCT

我々のプログラムで実装した UCT を新しい2つのパラメタ, p と *FPU*, でパラメタ化する. 初めに, UCB-TUNED の式 (1) に係数 p (省略値は1) を追加する. これにより次式を得る: 以下を最大化する j を選べ.

$$\overline{X}_j + p\sqrt{\frac{\log n}{T_j(n)}\ \min\{1/4, V_j(n_j)\}}$$

P は探索と収穫のバランスを決定する. より正確に言うと, p が小さければ木はより深く探索される. 表6に示した我々の実験によれば, UCB1-TUNED はこの意味でほぼ最適である. 〈p が1の辺りで勝率が最高になっているから〉

p	黒番の勝率	白番の勝率	全勝率
0.05	2%±4% (50)	4%±5% (50)	3%±3.4%
0.55	30%±13% (50)	36%±13% (50)	33%±9.4%
0.80	33%±9% (100)	39%±10% (100)	36%±6.7%
1.0	40%±8% (150)	38%±8% (150)	39%±5.6%
1.1	39%±8% (150)	41%±8% (150)	40%±5.6%
1.2	40%±8% (150)	44%±8% (150)	42%±5.7%
1.5	30%±13% (50)	26%±12% (50)	28%±9%
3.0	36%±13% (50)	24%±12% (50)	30%±9%
6.0	22%±11% (50)	18%±10% (50)	20%±8%

表6. 係数 p は探検と収穫のバランスを決定する (純粋ランダム・モード).

次は3.4節で説明した先頭打着緊急度 (FPU) である. 結果を表7に示す. 未探訪のノードの探索順序の修正にはまだ改良の余地があると信じている. 我々は CGOS での MoGo の FPU として最終的に 1.1 を用いた.

FPU	黒番の勝率	白番の勝率	全勝率
1.4	37%±9% (100)	38%±10% (100)	37.5%±7%
1.2	46%±10% (100)	36%±10% (100)	41%±7%
1.1	45%±6% (250)	41%±6% (250)	43.4%±4.4%
1.0	49%±6% (300)	42%±6% (300)	45%±4%
0.9	47%±8% (150)	32%±8% (150)	40%±5.6%
0.8	40%±14% (50)	32%±13% (50)	36%±9.6%

表7. 先頭打着緊急度 (FPU) の影響 (70,000シミュレーション/手).

4.3 CGOS での結果

MoGo は9路のコンピュータ碁サーバ (CGOS) で8月以降1位にランクされている.

5. 結び

MoGo の成功は, ノードが自動的により良い順序で調べられるという意味で, 特に探索時間が非常に制限されている時に, UCT が $\alpha\beta$ 探索より効率的であることを示している. 我々は,コンピュータ碁に関連した UCT の利点を議論した. 現在 UCT を使ったトップ・レベルの囲碁プログラムが増え続けていることは言及する価値がある.

我々は UCT アルゴリズムに加えうる改良点について議論した. 特に UCT は, 未探訪の手の良い順序を選ぶ助けにはならないし, あまり探検されてない手にはそれほど有効ではない. 我々は, この問題を解決するために先頭打着緊急度を調整する方法をいくつか提案した. この方向でさらなる改良が期待される.

我々はパターンに基づく (手順っぽい) シミュレーションを提案した. これは MoGo のレベルを大きく改善した. 我々はより意味のある手順を得るために, ランダム・シミュレーションに 3×3 のパターンを実装した. 我々はこの方向でまだ大きな改善が得られると信じている. 例えば, もしかしたら自動的に生成した, もっと大きなパターンを使うことで. また, シミュレーションの質を高めるために手順を強制する〈sequence-forced〉パターンを実装する方法もある.

我々はまた, 大きな基盤で強いプログラムを得るために UCT に枝刈り技法を組合せる可能性を示した. 勇気付けられる結果を得て, 我々はこの方向でのさらなる改良を固く信じている.

共有メモリ型コンピュータでの UCT の簡単な並列化を行い肯定的な結果を与えた. コンピュータ・クラスタでの並列化は面白いかも知れないが, まだうまい方法が見つからない.

謝辞

Pierre-Arnaud Coquelin には MoGo の開発中の援助に感謝する. Rémi Coulom には, CrazyStone のプログラミング経験を共有したことに特に感謝する. また Computer-Go メーリング・リストでの議論にも感謝する.

参考文献

[1] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47 (2/3) :235–256, 2002.

[2] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. Gambling in a rigged casino: the adversarial multi-armed bandit problem. *In Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 322–331. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[3] B. Bouzy. Associating domain-dependent knowledge and monte carlo approaches within a go program. *Information Sciences*, 2005. To appear.

[4] B. Bouzy and T. Cazenave. Computer go: An AI oriented survey. *Artificial Intelligence*, 132 (1) :39–103, 2001.

[5] B. Bouzy and G. Chaslot. Bayesian generation and integration of k-nearest-neighbor patterns for 19x19 go. *In G. Kendall and Simon Lucas, editors, IEEE 2005 Symposium on Computational Intelligence in Games, Colchester, UK*, pages 176–181, 2005.

[6] B. Bruegmann. Monte carlo go. 1993.

[7] T. Cazenave. Abstract proof search. *Computers and Games, Hamamatsu, 2000*, pages 39–54, 2000.

[8] T. Cazenave and B. Helmstetter. Combining tactical search and monte-carlo in the game of go. *IEEE CIG 2005*, pages 171–175, 2005.

[9] R. 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 Computers and Games, Turin, Italy*, 2006.

[10] T. Graepel, M. Goutri´e, M. Kr¨uger, and R. Herbrich. Learning on graphs in the game of go. *Lecture Notes in Computer Science*, 2130:347–352, 2001.

[11] A. Kishimoto and M. M¨uller. A general solution to the graph history interaction problem, 2004.

[12] L. Kocsis and C. Szepesvari. Bandit-based monte-carlo planning. *ECML'06*, 2006.

[13] L. Kocsis, C. Szepesv´ari, and J. Willemson. Improved monte-carlo search. working paper, 2006.

[14] M. Newborn. *Computer Chess Comes of Age*. Springer-Verlag, 1996.

[15] L. Ralaivola, L. Wu, and P. Baldi. Svm and pattern-enriched common fate graphs for the game of go. *ESANN 2005*, pages 485–490, 2005.

[16] R. Sutton and A. Barto. Reinforcement learning: *An introduction*. MIT Press. , Cambridge, MA, 1998.