コンピュータ将棋におけるモンテカルロ法の可能性

橋本隼一[†], 橋本剛[†], 長嶋淳[†]
[†] 北陸先端科学技術大学院大学

概　要

2006 年に行われた Computer Olympiad の囲碁の部 (9 路盤) では昨年に続きモンテカルロ法を用いたプログラムが好成績をおさめ，この手法が効果的であることが実証された．本論文ではモンテカルロ法の有効性の根拠について考察するとともに，コンピュータ将棋の分野におけるこの手法の可能性を探った．将棋は囲碁とは違い，ランダムな指し手では自然に終局することが殆どない．この問題への対処を中心にアルゴリズムの設計・改良を進めた．

A potential application of Monte-Carlo Method in Computer Shogi

Junichi Hashimoto[‡], Tsuyoshi Hahsimoto[‡], Jun Nagashima[‡]
[‡]Japan Advanced Institute of Science and Technology

Abstract

Even this year, a program based on Monte-Carlo method played outstandingly in the Go section (9x9) of Computer Olympiad 2006, and showed the efficacy of the method. In this paper, we discuss the reasons of this efficacy and investigate potential application of this method in Computer Shogi. Unlike Go, Shogi hardly ends by a sequence of random moves therefore this is a main problem in the design and improvement of algorithm.

1　はじめに

モンテカルロ法は乱数を用いたシミュレーションを行うことで正確な数値計算を肩代わりさせる手法であり，よく知られた例としては円周率πや重積分の計算等に用いられることがある．ランダムシミュレーションを利用した最適化手法である焼き鈍し法が発案されると，物理シミュレーションや組み合わせ最適化問題にも応用され，1993 年にはじめて囲碁プログラム Gobble に用いられた [1].

Brügmann はその冒頭で”How would nature play go?”と述べて，ルールに基づいた知識の表現やパターン認識，機械学習とは全く異なる方向から囲碁プログラムを捉えることを試みている．その後この手法は主流からはずれていたが Bouzy らによる見直し，改良がなされ [2][3]，ついに今年行われた第 11 回の Computer Olympiad の囲碁部門（9 路盤）においてモンテカルロシミュレーションと minimax アルゴリズムを統合させたプログラム Crazy Stone[4] が優勝し，ゲーム分野におけるモンテカルロ法の利用が再び注目を集めている．

本研究では主に [4] で Rémi Column が述べているアルゴリズムを元にその有効性について考察するとともに，この手法の将棋における可能性を探った．

2　モンテカルロ法の有効性

2.1　評価値の誤差補正

二人完全情報零和ゲームの思考アルゴリズムを構成する手法としては，ゲーム木を走査する探索アルゴリズムと探索の末端局面の優劣を評価する評価関数を用いる方法が確立している．

正確な評価関数が得られればこのフレームワークの中で十分な成果が得られるが，そのような評価関数を得ることは難しい．探索の末端局面の多

くは勝ち負け引き分けが確定するような終局ではないため，評価関数は先読みすること無しに後続の変化の可能性を含めて局面を評価しなくてはならない．このことは評価関数が本質的に不確実さ（以下誤差と呼ぶ）を含むことを意味している．

モンテカルロ法を探索に用いる第一の利点は，ランダムなシミュレーションの統計を用いることでこの誤差の影響を抑え得る点である．

一方でランダムなシミュレーションを用いることには大きな問題も存在する．二人ゲームでは双方が勝利を目標に指し手を決定するため，実際のゲームには確定的な手順が現れることも多い．特に将棋ではこのような傾向が強く，ランダムに選んだ指し手が最善手となることは殆どない．このように最善手が確定的である局面についてはminimax アルゴリズムで扱うのが合理的であると考えられる．

ランダムシミュレーションによる評価値の補正と minimax アルゴリズムの長所を融合したのがCrazy Stone で用いられた Coulom のアルゴリズムである．

Coulom のアルゴリズムではゲーム木の各ノードは内部ノードと外部ノードに分類される．最善手は root ノードを唯一の内部ノードとして以下のシミュレーションを反復することで得られる．

- 内部ノードでは，それまでのシミュレーションから得られた勝敗についての統計を元にした確率で指し手が決定され，その手が最善手に取って代わる確率をもとに，費やれるシミュレーション回数が決定される．
- 外部ノードでは終局まで至るランダムシミュレーションを行う．ノードを通過するシミュレーション回数が一定の条件を満たすと外部ノードから内部ノードに変わる

シミュレーションが進むにつれ重要なノードは徐々に内部ノードとなり，また内部ノードでは最善手に費やされるシミュレーション回数が増加するため内部ノード全体は minimax 木に近づくように成長する．

2.2 探索コストの制御

このアルゴリズムはモンテカルロ法と minimax アルゴリズムの融合もさることながら，内部ノードにシミュレーション回数を動的に割り当てる部分が注目に値する．

一般に与えられた計算時間内でより良い結果を得るには正確な評価を要する部分にコストを割き，そうでない部分ではあまりコストをかけないように探索を制御することが望ましい．コンピュータ将棋においても 0.5 手延長アルゴリズム [5] や，実現確率探索における評価値更新時の再探索 [6] などの手法によって部分的にはこれが実現されているが，Coulom のアルゴリズムでは探索全体でコストの制御が行われる．

探索コストの制御はモンテカルロ法の利用と必ずしも一致しないが最善手になる確率をもとに探索コストを決定するアイディアには妥当性があり，モンテカルロ法と密接に結びついている．

3 将棋におけるモンテカルロ法

3.1 特有の問題

将棋は取得駒の再使用が認められているゲームであり，囲碁やオセロなどの盤面を埋めてゆくゲームとは違い自然な収束が見られない．実際，詰みにより終局を判定するプログラムを使って適当な局面からシミュレーションを行ったところ手数が 1000 を超えたため断念した．モンテカルロ法を多数のシミュレーションを行うことで評価関数の精度を補うものであると考えれば，終局までシミュレーションを行うことは必須ではない．筆者らはシミュレーションの深さを浅くし（10 程度），末端で単純な評価関数を利用し，回数を確保することでこの問題への対処を試みた．

次節で設計したアルゴリズムの概要を述べるが，多くの点では Coulom のものを踏襲している．ただし [4] では細部が十分に示されていないことと，実装の困難さからいくつかの部分では異なる．とくに最善の子ノードを選択する箇所ではシミュレーションの平均を直接用いずに分散とシミュレーション回数で補正を行うようにした点

と，外部ノードが内部ノードとなる条件をシミュレーションの通過回数ではなく，複数回のシミュレーションを終えた後で他ノードと比較して決定する点は Coulum のものとは大きく異なる．

3.2 モンテカルロ将棋のアルゴリズム

内部ノードの探索と外部ノードのシミュレーション部分の擬似コードを図 1 に示した．アルゴリズム全体としては与えられた局面を root ノードとして search が反復して呼ばれることで探索がおこなわれる．

各内部ノードはその局面での指し手とシミュレーション回数 t，評価値 v の和 $S := \Sigma v$，2 乗和 $S2 := \Sigma v^2$，及び後続する内部ノードのリストを持つ．内部ノードリストは，$t, S, S2$ から得られる平均を μ, 標準偏差を σ としたときの

$$v' := \mu \mathrm{t}^{\,2}/\sigma \qquad (1)$$

で得られる値によって降順にソートされており，先頭は最善の子ノードとみなされる．直観的には平均値 μ でソートを行うのが妥当と思われるが，標準偏差が小さいほど，またシミュレーション回数が多いほど，μ の値は信憑性が高いと考え σ と t とで補正した (1) 式を採用した．

以下では search 関数内での処理について述べる．(1)〜(4) とした箇所は図 1 に対応を記したほか，次節で検討する．

与えられたノードのすべての子ノードについて，

- 内部ノードであるときは前回の探索結果を元にして，そのノードの評価値が最善ノードを上回る確率に比例して再帰探索される．
- 外部ノードであるとき（すなわち内部ノードリストに含まれないとき）は，内部ノードと同様に求められた回数分のシミュレーションが行われる．シミュレーションは (2)固定深さで打ち切られ，末端では (3)駒割のみを評価する eval が呼ばれる．シミュレーションの後，is_good の判定に基づいて内部ノードとなる条件を満たしているかを判定し，あてはまれば新しい内部ノードとしてリストに追加する．

ここで関数 is_good は v' の値について最善ノードのものと比較し，(4)差がある負値以上であるか を判定しており，v' が最善ノードに及ばないものについてもある程度は内部ノードとみなすようにした．

すべての子ノードの処理を終えると，update_values が呼ばれて，ノードリストの先頭ノードをもとに値が更新される．具体的には $a :=$ **ノードのシミュレーション回数/最善ノードのシミュレーション回数**として，search が negamax アルゴリズムとして動作するように S には最善ノードの値を $-a$ 倍した値が，$S2$ には a 倍した値がセットされる．

反復終了後，root ノードから内部ノードリストの先頭をたどることで最善応手手順が得られる．

3.3 評価と改善すべき点

アルゴリズムを実装したプログラムに次の一手問題（[7]，100 問強）を解かせたところ，正答率は 3 %程度であった．内部ノードを見ると読むべき手はおおむね含まれているようであるが，探索の浅い手がリストの上部に集まろうとする傾向が見られるため，前節の (1)〜(4) を調整することでこの傾向を抑えることができればより良い結果が得られると考えられる．

(1) μ/σ にはそれなりの妥当性があると考えられるが，t の乗数はアドホックに定めたものでありさらに検討する必要がある．

(2) シミュレーションの深さを変化させることで，解ける問題が変わることが確認できたため，動的に変更するか，あるいは (3) とあわせて静止探索を導入するのが効果的かもしれない．

(3) 評価の粗さについては 2.1 節で述べた誤差補正が働くと考えあまり問題視していないが，駒割のみの評価では同じ評価値が多数現れてしまうことがあり，シミュレーションのランダム性を失わせる傾向が見られた．

(4) v' の値ではなく評価値の信憑性の指標を t, σ から算出してこれを判断基準にすれば，低

い評価のノードを再度シミュレートすることが少なくなり効率を上げることができる可能性がある．

このほか，2 節では触れなかったが，モンテカルロ法の長所として各ランダムシミュレーションは他とは独立に行えるため，単純な並列化によって性能の向上が見込める．

4 まとめと今後の展望

9x9 囲碁で効果が認められた [4] の手法を元に，固定深さで打ち切るシミュレーションと簡単な評価関数を利用するコンピュータ将棋用のアルゴリズムを設計し，その性能を元に改良が可能と思われる点について考察した．

結果はいまひとつではあるもののモンテカルロ法の可能性が否定されたとは考えていない．理由のひとつは 2.2 節で述べたように，コストの割り当てが動的に行われることである．また評価関数部分のみに限定してシミュレーションを利用するなど，思考アルゴリズムにおけるランダムシミュレーションと統計量を用いる手法はむしろ多くの可能性を秘めていると考えられる．

図 1 モンテカルロ将棋の擬似コード

```
// 局面
Position p

// n の評価値が best のそれを上回る確率と，
// t とに比例した正数値を返す
int clac_times(int t, Node best, n){...}

// (4)n が内部ノードとなるかどうかを判定する
bool is_good(Nodes list, Node n){...}

// 外部ノードのシミュレーション
// t: シミュレーション回数
simulate(int t, Node n) {
  p.makemove(n.move)
  s1 = s2 = 0

  for ( t ) {
    Position work = p

    // (2) 個定深さのシミュレーション
    for ( MaxDepth )
      work.makemove(work.randome_move())

// (3) 駒割のみの評価
    s1 += eval(work)
    s2 += eval(work) * eval(work)
  }

  n.set(t, s1, s2)
  p.unmakemove(n.move)
}

// 内部ノードの探索
// t: シミュレーション回数の最大値
search(int t, Node n) {
  p.makemove(n.move)

  // 内部ノードは再帰的に探索
  foreach ( Node in; n.innerNodes ) {
    search(clac_times(t, n.best, in), in)
    // (1) v' の値で降順になるように追加
    searchedNodes.sort_insert(in)
  }

  // 外部ノードはシミュレーションを行って
  // 内部ノードにする価値があれば追加
  foreach ( Move m; p.all_unsearched_moves() ) {
    Node ex = new Node(m, UNSEARCHED)
    simulate(clac_times(t, n.best, ex), ex)

    if ( is_good(searchedNodes, ex) )
      searchedNodes.sort_insert(ex) // (1) 同上
  }
  // ノードのデータを更新
  n.innerNodes = searchedNodes
  n.update_values()
  p.unmakemove(n.move)
}
```

注) `for(n)`, `foreach(type t; typelist l)` はそれぞれ n 回のループ，リスト `l` の全要素について `t` に代入しながらのループの意．

参考文献

[1] Bernd Brügmann, "Monte Carlo Go", Unpublished technical report, 1993. http://www.ideanest.com/vegos/MonteCarloGo.pdf

[2] Bruno Bouzy, Tristan Cazenave, "Computer Go: An AI Oriented Survey", Artificial Intelligence, vol.132, pp.39-103, 2001.

[3] B.Bouzy, B.Helmsetter, "Monte-Carlo Go Development", in H.J. van den Heric, H.Iida, and E.A.Heinz, editors, Proceedings of the 10th Advances in Conputer Games Conference, 2003.

[4] Rémi Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search", Proceedings of Computers and Games Conferemce 2006 (CD-ROM), 2006.

[5] 山下 宏, YSS-そのデータ構造，およびアルゴリズムについて，コンピュータ将棋の進歩 2（松原 仁編著），共立出版, ISBN:4320028929, pp.112-142, 1998.

[6] Yoshimasa Tsuruoka, Daisaku Yokoyama and Takashi Chikayama, "Game-tree Search Algorithm based on Realization Probability", ICGA Journal, Vol. 25, No.3, pp.145-152, 2002.

[7] 中級必修次の一手 105,（週刊将棋編）毎日コミュニケーションズ, ISBN:4839908370, 2002.