DFUCTの囲碁への応用について

吉本晴洋[†1] 岸本章宏[†2]
金子知適[†3] 美添一樹[†4]

UCT アルゴリズムは、モンテカルロ碁で利用されている代表的な最良優先探索である。本論文では、深さ優先探索を利用して、UCT を改良したアルゴリズム DFUCT(Depth-First UCT) を提案する。囲碁の次の一手の実験によると、DFUCT は UCT と同等の解答能力を保ちつつ、3%程度の速度の向上が見られた。

Depth-First UCT and Its Application to Go

Haruhiro Yoshimoto ,[†1] Akihiro Kishimoto ,[†2]
Tomoyuki Kaneko [†3] and Kazuki Yoshizoe [†4]

The UCT algorithm is a representative best-first search algorithm that has been popularly used in Monte Carlo Go. This paper presents the DFUCT (Depth-First UCT) algorithm, which is an efficient depth-first variant of UCT. Experimental results in Go show that DFUCT achieves 3% improvement in running time, while the solving abilities of DFUCT and UCT are comparable.

1. はじめに

将棋やチェスをはじめとする完全情報ゲームでは、先読みを行うことは、良い手をプレイする上で重要な要素である。α-β 法をはじめとするゲーム木探索では、局面の優劣を数値化する評価関数を利用して、先読みを行い、次の一手を決定している。ゲーム木探索は、完全情報ゲームをプレイするプログラムの多くに適用され、チェスでは人間の世界チャンピオンに勝つ[7] など目覚ましい成果を挙げてきた。

将棋やチェスなどのプログラムの強さは、ゲーム木探索に改良を重ねることによって格段に進歩してきたが、囲碁ソフトウェアは人間のトッププレイヤーにまだまだ及ばない[16]。その最大の要因は、ゲーム木探索で利用できるような正確な評価関数の作成が、囲碁では難しいことにある。将棋やチェスなどでは、駒の価値等の基準を利用することによって、正確な評価関数を作りやすい。ところが、囲碁では、最終的には「地」と呼ばれる自石で囲まれた空点の多いプレイヤーの勝ちであるが、この地になりそうな部分を正確に判定できる評価関数の構築は、現状では困難である。

この評価関数の問題を解決するために、コンピュータ囲碁の分野では、局面の評価にモンテカルロ法を利用するモンテカルロ碁がの研究が盛んである。現在最も有効なモンテカルロ碁は、UCT アルゴリズム[12] に基づくものである。例えば、現在最強のプログラムである MoGo[10] は、UCT に様々なヒューリスティックを搭載している。

UCT は、木探索とモンテカルロシミュレーションから構成されるが、木探索には最良優先探索を利用している。他のゲーム木探索では、最良優先探索の多くは、深さ優先探索を利用すれば探索効率を上げられることが知られている[15),17]。

本論文では、UCT を深さ優先型探索に変換した DFUCT(Depth-First UCT) アルゴリズムを提案する。DFUCT は、UCT とほぼ等価なアルゴリズムであるが、内部節点の再展開数が UCT よりも少ない。実際に、囲碁の次の一手を利用した実験では、DFUCT は、UCT と同等の解答能力を保ちつつ、UCT の内部節点の再展開数を 56〜67%に減らすことができた。この効果によって、UCT に比べて、3%程度の高速化

†1 東京大学大学院情報理工学研究科電子情報学専攻
hy@logos.ic.i.u-tokyo.ac.jp, tau@logos.t.u-tokyo.ac.jp
†2 公立はこだて未来大学システム情報科学部
kishi@fun.ac.jp
†3 東京大学大学院総合文化研究科
kaneko@graco.c.u-tokyo.ac.jp
†4 中央大学研究開発機構
yoshizoe@tamacc.chuo-u.ac.jp

を達成した。

本論文の構成は、以下の通りである。2 節では、関連研究について言及する。3 節では、UCT アルゴリズムの概要について述べる。4 節では、本論文で提案する DFUCT について説明する。5 節では、実験結果を示し、6 節では、本論文のまとめと今後の課題について述べる。

2. 関 連 研 究

2.1 モンテカルロ碁とその背景

モンテカルロ法は、情報科学における代表的な近似アルゴリズムである。モンテカルロ法をゲームに利用した例としては、本論文で述べるモンテカルロ碁だけでなく、ポーカー[2] やブリッジ[11] などの不完全情報ゲームがある。囲碁は完全情報ゲームであるが、近年の研究では、モンテカルロ碁[4] は、評価関数を用いたゲーム木探索に代わる手法として、注目を浴びている。

モンテカルロ碁では、モンテカルロ法に基づくシミュレーション (以下、シミュレーションと呼ぶ) を繰り返すことで、もっともらしく局面の評価を行う。ある局面 P で行われるシミュレーションでは、P から終局局面まで、各プレイヤーは、眼を埋める手を除いた合法手よりランダムに手を選択し、終局時のスコアを計算する。モンテカルロ碁は、このスコアを局面 P の評価基準として利用する。

モンテカルロ碁の研究は、最初に Brügmann[6] によって行われ、Bouzy らによって有効性が示された[3),4)]。初期の単純なモンテカルロ碁は、ゲーム木探索と評価関数を利用する従来の手法には及ばなかった[4] が、Crazy Stone[8] や MoGo[10] などの強いプログラムの開発により、現状では最も有効な手法であると考えられている。特に、MoGo で利用されている UCT アルゴリズム[12] は、最良優先探索とモンテカルロ法を組み合わせた代表的な手法である。

2.2 最良優先探索の深さ優先探索への変換

最良優先探索は、有望そうな節点を中心に探索できるが、多くのメモリを使うという欠点がある。一方、深さ優先探索では、メモリの使用量が少ないが、先に展開すべき節点を後回しにしてしまうことがあるので、探索に時間がかかることが多い。

最良優先探索の多くは、深さ優先探索の閾値に深さ以外の基準を利用することで、等しい振る舞いを行う深さ優先探索に変換できる。この深さ優先探索では、ある節点 n での評価が閾値を超えていない場合には、n 以下の探索を行う。このような変換によって、深さ優先探索と最良優先探索の両方の利点を保持できる。

深さ優先探索への代表的な変換例として、一人ゲームにおいて、最良優先探索 A* を変換した IDA*[13] や RBFS[14]、AND/OR 木探索において、最良優先探索である PN 探索[1] を変換した df-pn 探索[17] などが挙げられる。特に、df-pn 探索では、PN 探索で行われている無駄な内部節点の再展開を減らして、探索効率を上げることに成功した。

本論文では、このような深さ優先探索への変換の成功例を踏まえて、最良優先探索である UCT を深さ優先探索に変換することを試みる。

3. UCT アルゴリズム

本節では、Koscis らの UCT アルゴリズム[12] を囲碁に利用した場合に限定して、説明を行う。

UCT は、節点の評価にシミュレーションを利用した最良優先型の探索手法である。UCT における、ある節点のシミュレーションとは、その節点から終端節点まで、両プレイヤーがランダムにプレイし、そのランダムプレイの勝ち負けの結果 (勝ちは 1 で、負けは 0) を返すことである。

UCT では、各プレイヤーは UCB 値に基づいて子節点の選択を行う。指し手 i の UCB 値とは、指し手 i で到達する子節点 c_i と c_i の兄弟節点を利用して、以下のように定義される。

$$\mathbf{get\_ucb}(i) \equiv \frac{w_i}{n_i} + \sqrt{\frac{2 \ln s}{n_i}} \qquad (1)$$

ここで、w_i は c_i のシミュレーションの勝ち数、n_i は c_i のシミュレーション回数、s は c_i および c_i の兄弟節点のシミュレーション回数の総和である。

UCB 値の計算は、基本的に勝率に基づいているので、UCB 値が大きければ、勝ちやすい指し手であると考えられる。ただし、訪問回数の節点が少ない節点の勝率は信頼できないので、探索によって、より正確な勝率を求める必要がある。UCB 値では、訪問回数の反比例値も考慮に入れることで、勝率の高そうな指し手とそうでない指し手の探索の頻度をうまく制御することができる。UCT では、有望そうな指し手の訪問頻度は、そうでない指し手よりも指数のオーダーで大きくなる。

UCT では、根節点よりシミュレーションが行われたことのない先端節点に至るまで、UCB 値の最も大きな指し手☆を各プレイヤーは選択する。次に、選択した先端節点において、シミュレーションを行い、そ

☆ シミュレーションの行われたことのない節点に至る指し手の UCB 値は、非常に大きい値となる。

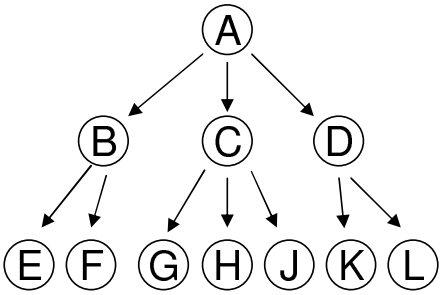

図 1 UCT の探索木

の節点を評価し、探索木にその節点を追加する。この評価値を利用して、根節点よりその先端節点に至る経路を逆向きに手繰り、経路上の UCB 値を更新しながら、根節点まで戻る。UCT では、この木探索とシミュレーションの過程を何度も繰り返して、最も有効な次の一手を選ぶ。

4. DFUCT アルゴリズム

4.1 改良のアイデア

UCT は、PN 探索と同様に最良優先探索であるので、根節点から順に最も有望な節点を選んで移動し、たどりついた葉節点を展開することを繰り返す。例えば、図 1 の UCT の探索木において、次にシミュレーションを行う節点が E であったとする。この場合には、UCT は $A \to B \to E$ と展開し、E でシミュレーションを行った後、$E \to B \to A$ と経路を戻り、UCB 値を計算しなおす。次に、シミュレーションを行う節点が F であったときには、UCT は、$A \to B \to F$ と節点を展開するので、$A \to B$ という節点展開を再度行うことになってしまう。

DFUCT では、df-pn と同様に深さ優先探索を行うことで、必ずしも根節点まで戻らずに、探索を継続できる場合は継続する。つまり、DFUCT では、図 1 の探索木で同じ節点を展開するときには、$A \to B \to E$(シミュレーション) $\to B \to F$(シミュレーション) となる。このようにして、DFUCT では、UCT で行われていた $B \to A$ と $A \to B$ に関する計算を省略できる。

4.2 深さ優先探索の探索継続の条件

UCT 改良のアイデアを実現するためには、途中の節点で親節点に戻らずに子節点の探索を継続できることを判断する必要がある。df-pn 探索の場合には証明数と反証数に関する閾値を利用するが、UCT の探索継続条件は簡単な閾値では表現できない。そこで、DFUCT ではスタックを用意し、各先祖の情報を保存する方法を採用した。具体的には、UCB 値最大の節点と 2 番目の節点についての勝ち数 w_i と試合数 n_i、ならびに兄弟ノードの合計試合数 s を保存する。これにより、探索が進んだ後でも二つの節点の最新の UCB 値を計算できる。

DFUCT で固定の閾値を利用できない理由は、UCB 値の複雑さにある。まず UCB 値は式 (1) にあるように w_i, n_i, s の 3 種類の変数から決まるため、第一項だけ考える場合でも「x 回シミュレーションしたら終了」というような単純な条件では表現できない。この点はスタックを用いる本手法では解決できている。さらに、変数 s が兄弟節点のシミュレーション数であるため、葉節点でのシミュレーションがその親節点および兄弟節点の UCB 値に再帰的に影響を与えるという難しさがある。こちらを簡単な計算で対処することは難しい。

4.3 提案アルゴリズム

提案アルゴリズムを以下に示す。

(1) 根節点から開始する。

(2) 子節点のうち UCB 値が最大のものを選んでそれを新しい探索節点とする。その際に UCB 値が最大の子節点の情報と、2 番目の子節点の情報をスタックに保存する。先端節点に到達するまで繰り返す。

(3) 先端節点でシミュレーションを行い、その後、親節点に戻るべきかを確認する。
- スタックの内容を深さの深いエントリから順に更新する。具体的には、勝数と試合数を適切に 1 増やす。それらを元に、二つの節点の UCB 値を計算する。
- UCB 値が最大だった節点の UCB 値が、2 番目だった節点の UCB 値より小さくなっていたら、その節点まで戻る。戻りながら、元の UCT 同様に節点の情報を子節点の情報を元に更新する。2 に戻る。

(4) 親節点に戻る必要がなければ、2 へ戻る。

このアルゴリズムは、「スタックを調べることで、PV の各節点について最大の UCB 値を持つ子節点であることを確認できる」(*) 場合には、元の UCT と等価な探索を行う。例えば、変数 s が変化しない場合はこれは自明である。

図 2 に DFUCT の疑似コードを示す。

DFUCT で問題となるのは、シミュレーションが進むにつれ変数 s が変化し、UCB 値の大きさの順序が 3 位以下だった節点が 2 位に浮上する場合である。スタックには過去に 1,2 位だった節点のみが記録されて

```
# 探索継続条件を格納するスタックを用意
cond = Array.new()
# スタックにある節点の UCB 値を更新しながら、
# 探索継続条件をチェックする
# 戻り値: 根節点に向かって探索木を戻る深さ
def checkAndUpdateCond(win_or_loss)
  coumter = 0
  cond.each{ |c|
    counter += 1
    # 探索継続条件を更新する
    c.best.win.update(win_or_loss)
    c.total += 1
    # 最も UCB 値が高かった節点の更新後の値を得る
    ucb_best = calculateUcbValue(c.best,c.total)
    # 二番目に UCB 値が高かった節点の更新後の値を得る
    ucb_second = calclulateUcbValue(c.second,c.total)
    if ucb_second > ucb_best
      # 二番目の節点の UCB 値が一番目の節点の値を
      # 越えたので、探索を切り上げて親節点に戻る
      return cond.size() - counter
    end
  }
  # 子節点の探索を続行する
  return 0
end

# DFUCT 探索を行う
def search(node)
  # 現在の節点の合法手を生成する
  moves = node.getValidMoves
  # まだ全ての子節点を展開していないとき
  while node.childNodes.size < moves.size
    # 展開していない節点を生成する
    newnode = node.makeMove(
     moves[node.childNodes.size])
    # シミュレーション結果が勝ち:true, 負け:false
    win_or_loss = newnode.doMonteCarloSimulation()
    #シミュレーションの勝利数を保存する
    newnode.win.update(win_or_loss)
    #訪問回数を保存する
    newnode.nb = 1
    #現在の節点に子節点を追加する
    node.addChildNode(newnode)
    #現在の節点の勝利数を更新する
    node.win.update(win_or_loss)
    #現在の節点の訪問回数を更新する
    node.nb += 1
    # 子節点の探索を続けるかどうかを判定する
    rollback = checkAndUpdateCond(win_or_loss)
    if rollback > 0
      cond.pop()
      # 探索を切り上げて親節点に戻る
      return rollback - 1
    end
  end
  while true
    total = node.childNodes.sumOfNb()
    # 最も UCB 値の高い子節点を選択する
    best = node.childNodes.
     bestUcbNode(total)
    # 二番目に UCB 値の高い子節点を選択する
    second = node.childNodes.
     secondBestUcbNode(total)
    # 探索継続条件を設定する
    newcond = makeNewCondition(
     best,second,total)
    cond.push(newcond)
    # UCB 値の最も高い子節点を探索する
    rollback = search(best)
    if rollback > 0
      # 探索を切り上げて親節点に戻る
      cond.pop()
      return rollback - 1
    end
  end
end
```

図 2 DFUCT の疑似コード

いるため、この場合には、上記 (*) の条件は満たされず、結果として親節点に戻るべき場合にも戻らないということが起こりうる。しかし、後で述べる実験では、そのようなケースはほとんどなかった。

5. 実験結果

5.1 実験条件

DFUCT と UCT を Intel Xeon dual 2.4GHz (Linux, メモリ 2GB) マシン上に実装し、150 問からなる問題集[18] を両手法に解答させた。各手法が返す次の一手が、問題集の解答と一致していれば、正解とした。一問当たりのモンテカルロシミュレーション数を 100,000 回から 700,000 回まで、100,000 ずつ増やし、DFUCT と UCT の正解数と内部節点の再展開回数の変化を調べた。

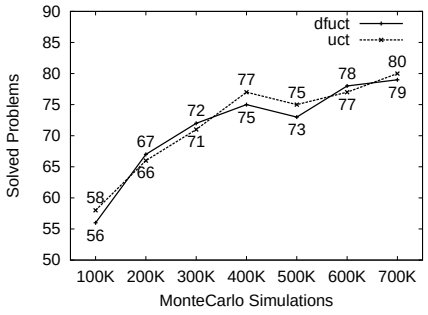

図 3 正解数

5.2 結　　果

図 3 に、各サンプル数に対する DFUCT と UCT の

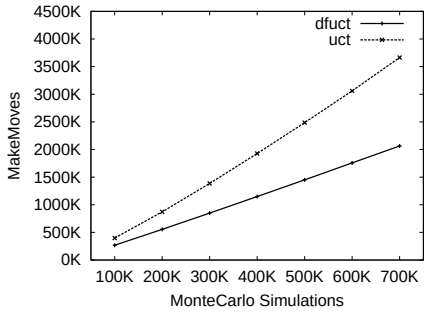

図 4 内部節点の再展開回数

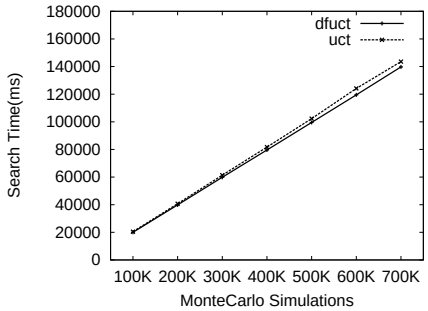

図 5 探索時間

解答能力を示す。横軸がシミュレーション数で、縦軸が正解数である。この図より、DFUCT と UCT の正解数には殆ど差がなく、ほぼ互角の解答能力であると言える。全体的にはシミュレーションの回数が増えるほど正解数が増える傾向にあるが、80 付近でほぼ頭打ちになっている。さらに、UCB 値の大きさが順序が 3 位以下のものが 2 位に浮上する確率を調べたところ、シミュレーション数に関係なく、平均で 2.53%から 2.57%であった。UCT と DFUCT の解答能力比較から、この性質から生じる DFUCT への悪影響は少ないと考えられる。

図 4 に、1 問当たりの内部節点の平均再展開回数を示す。横軸がシミュレーション数で、縦軸が内部節点の再展開回数である。再展開回数は DFUCT の方が UCT よりも少なく、さらにシミュレーション数が大きくなるほど UCT の再展開の手間が DFUCT に比べて大きくなることが分かる。例えば、シミュレーション数が 100,000 回ときの DFUCT の内部節点の再展開数は、UCT の 67%であるが、700,000 回では、56%にまで減少している。

図 5 に探索にかかった時間を示す (なお、表 1 はこの図を表にしたものである)。横軸がシミュレーション数で、縦軸が探索にかかった時間である。DFUCT は、UCT と比較して、およそ 3%程度の速度向上が

シミュレーション数	UCT	DFUCT
100K	20,466	20,035
200K	40,713	39,858
300K	61,377	59,844
400K	816,88	79,603
500K	102,300	99,543
600K	124,165	119,514
700K	143,581	139,755

表 1 探索時間 (ミリ秒)

シミュレーション数	UCT	SIM (UCT)	DFUCT	SIM (DFUCT)
100K	204.7	178.0	200.4	176.0
200K	203.6	176.2	199.3	175.0
300K	204.6	176.1	199.5	175.1
400K	204.2	175.3	199.0	174.4
500K	204.6	174.9	199.1	174.3
600K	206.9	177.4	199.2	174.2
700K	205.1	174.1	199.7	174.2
800K	205.7	175.0	199.0	173.6
1000K	208.5	177.0	198.4	172.9
1200K	206.0	173.1	199.9	173.3
1400K	207.1	174.0	197.7	171.6
1600K	207.9	173.7	198.9	172.5

表 2 1 シミュレーション当たりの探索時間 (μ 秒)

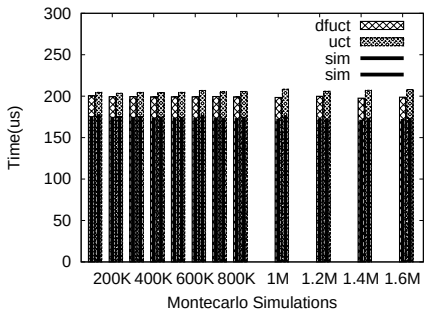

図 6 1 シミュレーション当たりの探索時間 (μ 秒)

見られた。

表 2 と図 6 に、1 シミュレーション当たりの探索時間を示す。この実験では、シミュレーション数を 160 万まで増やした。表 2 では、UCT と DFUCT は、各手法において、1 シミュレーション当たりに費した探索時間 (内部節点の展開やシミュレーション等を含む) であり、SIM(UCT) と SIM(DFUCT) は、そのうちシミュレーションのみに費した時間である。図 6 は、表 2 を図示したものである。黒く塗り潰された部分がシミュレーション自体に費した時間で、残りが内部節点の再展開などに要した時間である。

この図と表より、シミュレーション数の増加にしたがって、両手法がシミュレーション自体に費す時間は減少することが確認できる。この理由としては、シミュレー

ション数の増加によって、深い探索が行われていることが挙げられる。探索の深い節点では、浅い節点よりも終局までに要する手数が短くなるので、シミュレーションにかかる時間が減少する。
また、DFUCT は UCT に比べて、単位シミュレーション当たりの探索時間が短く、この傾向は、シミュレーション数が増えるほど強くなる。この振舞いの違いは、DFUCT と UCT の性質の違いから生じる。UCT では、各シミュレーションごとに根節点まで戻り、次の先端節点を選択するので、探索が深くなれば、それに比例した時間がかかる。一方、DFUCT では、内部節点の途中から先端節点へ向かって探索を行うことが多いので、探索が深くなっても、単位シミュレーション当たりの探索時間の増加を抑えられる。

6. ま と め

最良優先探索 UCT とほぼ同等のことを行う深さ優先探索 DFUCT を、UCT 探索の子節点の探索継続条件をスタックで管理することで、実現した。DFUCT では、UCT と同じ問題解答能力を保ちつつ、探索木の再展開を大幅に減らすことに成功したが、速度向上は 3%程度であった。

今後の課題としては、DFUCT の並列化が挙げられる。並列探索の際には、UCT 探索木が大幅に大きくなるため、探索木の内部節点展開に必要な手間が増大することが、本論文の実験結果より推測できる。そのため、DFUCT と UCT の性能差は、本論文で達成した値よりも、大きくなることが期待できる。また、ゲーム木探索の並列化では、深さ優先探索を並列化したものが多く、これらの先行研究[5),9)] を適用できれば、DFUCT は有用なアルゴリズムになると考えられる。

参 考 文 献

1) L.V. Allis, M.vander Meulen, and H.J. vanden Herik. Proof-number search. *Artificial Intelligence*, Vol. 66, No. 1, pp. 91–124, 1994.
2) D. Billings, A. Davidson, J. Schaeffer, and D. Szafron. The challenge of poker. *Artificial Intelligence*, Vol. 134, No. 1-2, pp. 201–240, 2002.
3) B. Bouzy. Associating domain-dependent knowledge and Monte Carlo approaches within a Go program. *Information Sciences*, Vol. 175, No. 4, pp. 247–257, November 2005.
4) B. Bouzy and B. Helmstetter. Monte Carlo Go developments. In *Advances in Computer Games. Many Games, Many Challenges*, pp. 159–174. Kluwer, 2003.
5) M. Brockington. *Asynchronous Parallel Game-Tree Search*. PhD thesis, University of Alberta, 1998.
6) B. Brügmann. Monte Carlo Go. Technical report, Physics Department, Syracuse University, 1993.
7) M. Campbell, A. Joseph Hoane Jr., and F.-h. Hsu. Deep Blue. *Artificial Intelligence*, Vol. 134, No. 1-2, pp. 57–83, 2002.
8) R. Coulom. Computing elo ratings of move patterns in the game of Go. In *Computer Games Workshop*, 2007.
9) R. Feldmann. *Game-Tree Search on Massively Parallel Systems*. PhD thesis, University of Paderborn, 1993.
10) S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with patterns in Monte-Carlo Go. Technical report, INRIA, 2006. RR-6062.
11) M. L. Ginsberg. GIB: steps toward an expert-level bridge-playing program. In *Sixteenth International Joint Conference on Artificial Intelligence*, pp. 584–589, 1999.
12) L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *European Conference on Machine Learning*, pp. 282–293, 2006.
13) R. E. Korf. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, Vol. 27, No. 1, pp. 97–109, 1985.
14) R. E. Korf. Linear-space best-first search. *Artificial Intelligence*, Vol. 62, No. 1, pp. 41–78, 1993.
15) R. E. Korf and D. M. Chickering. Best-first minimax search. *Artif. Intell.*, Vol. 84, No. 1-2, pp. 299–337, 1996.
16) M. Müller. Computer Go. *Artificial Intelligence*, Vol. 134, pp. 145–179, 2002.
17) A. Nagai. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*. PhD thesis, Department of Information Science, University of Tokyo, 2002.
18) 日本棋院（編）. 置き去りの傷を探せ―進級シリーズ〈2〉. 日本棋院, 2002.