平成17年度　修士論文

コンピュータ詰碁の探索戦略の改良

Improving Search Strategy of Computer Tsumego

指導教官　近山　隆　教授

東京大学大学院 工学系研究科

電子工学専攻 近山・田浦研究室

石井宏和

平成18年2月03日

Abstract

Building computer game players has been studied in the area of artificial intelligence for a long time as a suitable subject in which typical AI technologies such as machine learning or data mining can be applied to and evaluated. Thanks to great increase of computing power and application of many innovative methods, players of some challenging games, such as Othello, Backgammon, and Chess, have reached or surpassed the level of the best human players. In some cases, such as Go-Moku, the game has been solved completely, allowing for perfect play by a rational computer agent. On the other hand, for some other games such as Shogi and Go, building champion-level computer players remains to be a great challenge to the field of artificial intelligence for a variety of reasons.

Checking mates in Shogi and Go have been studied for a long time as a field of computer game players. Great success has been achieved in Shogi, but there are still many challenges in Go. They can be divided into two main problems. One is a large search space. Search space of Go is about ten to the three hundred and sixty power at an estimate, so brute-force approaches which achieved a great success in Chess or Othello can not solve problems in Go. Two is an ambiguity standard of estimation. In the case of Shogi, there is an absolute standard of estimation that someone who gets the king become a winner. On the other hand, Go's standard of estimation take on ambiguity because Go doesn't have an absolute standard of estimation as Shogi. A lot of Go solvers use a costly manual estimation function, so it is hard to solve problems by computer.

We apply a static evaluation function "Bouzy's 5/21 algorithm" which can estimate a state of position to Depth-First Proof-Pumber(Df-pn)+ search which is an advanced approach of AND/OR tree search have achived great successes in checking mates of Shogi and confirms an effect in Tsumego. This thesis presents the result of using this approach to a oneeye problem as a field of Tsumego.

目 次

図 目 次

表 目 次

第1章　序論

本章では、まず本研究の背景を述べ、研究の目的とその動機について示す。次に、本研究の概要について述べ、最後に本論文の構成について概説する。

1.1　研究の背景

コンピュータゲームプレイヤは、人工知能の黎明期から盛んに研究され続けている [1, 4, 6]。人工知能にとって、ゲームをコンピュータに行わせることは人間の「思考」に関する重要な研究課題であり、「思考」実現のためには、機械学習や探索・並列処理・認知科学など様々な要素技術を統合して扱う必要がある。現在、計算能力の向上や最新の手法を通して、いくつかのコンピュータエージェントの技術は、オセロやバックギャモン・チェスなどの複雑なゲームにおいて人間の世界チャンピオンのレベルか、あるいはそれ以上の領域にまで到達した。また、五目並べのようないくつかのゲームにおいては完全に解決された。しかし、探索空間が大きいこと、あるいは知識や判断基準の定義と優先順位の決定を正確に行うことの難しさなどにより、いまだアマチュアレベルに留まっているゲームもある。その中でも、将棋や囲碁は難しいゲームとして知られている。

ゲームにはいくつかの分類が存在する。主な分類方法としては、人数・完全情報かどうか・確定的かどうか・ゼロ和かどうかの 4 つがある。完全情報とは、ゲームの情報がプレイヤに分かっているかどうか、確定的とは、ゲームの進行がプレイヤの着手以外に確率的に不確定かどうか、ゼロ和とは、ゲームの結果として勝ちを 1、負けを -1、引き分けを 0 とした場合に全プレイヤの合計が 0 になるかどうかということである。本研究で扱う囲碁は、ゲームとしては 2 プレイヤ・完全情報・確定的・ゼロ和という分類になる。これらに当てはまるゲームとして、他にも将棋やチェス・オセロのようなものがある。

ゲームでは、そのゲームの問題の一部として、詰めを考える問題がある。将棋や囲碁では、その研究が始まったのと同時期の 1970 年代に、詰めの問題についても研究が始まった。詰将棋は、近年大きな進歩を遂げており、1997 年に現存する最長手数の詰将棋である「ミクロコスモス」(1525 手詰) が解かれ、現在では、300 手以上の手数を要する長編詰将棋が全て解かれている。一方詰碁の方では、まだ有効的な手段は見つかっていない。

詰碁がコンピュータにとって難しいのには大きく三つの理由がある [7]。一つは

前述した探索空間の大きさである。囲碁は最大で盤の大きさが 19×19 の大きさを持ち、駒である石は特徴を持たない。そのため、ゲーム木が非常に大きくなってしまう、いわゆる組合せ爆発という現象を引き起こしてしまう。囲碁の探索空間はある試算では 10^{360} にまで上ると言われており、ここまで大きくなると宇宙の物質全てを使って、宇宙の年齢のあいだ計算しても答えを出すことはできない。詰碁では探索する範囲が限られるためこれ程まで探索空間は大きくはならないが、それでも探索空間は依然大きく、難しさの原因の一つとなっている。もう一つに判断基準の曖昧さがある。将棋の場合、王を取ったものが勝者という絶対的な基準がある。しかし、囲碁の場合このような基準はなく、また将棋で評価の助けとなっている駒の特徴もない。そのため、評価基準が曖昧性を帯びてしまい、難しさの原因の一つとなっている。最後は、囲碁を探索で同じ局面の繰り返しが多くなるという性質を持つということがある。最終的には詰むか詰まないかの 2 種類の評価値しか存在しないような問題には探索を行うが、囲碁は同じ局面の繰り返しが多いため、この点を考慮に入れて探索を行わないと誤った結果を招いてしまう。これらの問題点を踏まえ、現在様々な研究が行われている。

1.2 研究の目的と概要

本研究では詰碁の以下で説明する一眼問題を対象として、効率的に解くことを目的とする。

1.2.1 詰碁

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、攻め手と受け手に手番 (「白番」、「白先」もしくは「黒番」、「黒先」) が示され、どのように打てば受け手は自分の石を生きにもちこめるか、あるいは攻め手は相手の石を取ることができるかという死活を考えるものである。詰碁の目的は黒先黒生・黒先白死・白先白生・白先黒死の 4 種類がある。本論文では、目的を達成できる場合 (黒先黒生ならば、黒が最終的に生きること) を詰みと呼び、達成できない場合を不詰みと呼ぶこととする。将棋では詰将棋がそれに対応する。詰碁は概して以下の三つに大分される。

- 攻め合い問題
  攻撃側と防御側の区別をしないでの勝者の決定が主目的
- 脱出・切断問題
  石の連絡の有無の判定が主目的

- 死活問題
  囲まれた領域内での、被攻撃側の石の眼の有無の判定が主目的

攻め合い問題では、組合せ数学の一分野である組合せゲーム理論という方法を用いての研究が行われており [14]、攻め合いにおけるダメ数の計算・ヨセの解析 [17]・眼形の解析などに適用されている。　しかし、攻め合い問題・脱出・切断問題はその煩雑さや定義が難しいことから、扱いが難しい。本研究における詰碁では、この中でも比較的扱い易い死活問題を対象とする。死活問題は “*life and death*” と呼ばれ、研究の歴史は古い。後述する詰碁プログラム *GoTools* も扱う問題を死活問題に限定している。

詰碁で詰むか詰まないかの判断は、眼の数を持って行われる。目的が生きの場合は、生かす石が眼を二つ以上持てば生きで詰みとなり、眼が一以下であれば死にで不詰みとなる。目的が死にの場合は、死なせる石が眼を一つ以下しかもたないなら死にで詰みとなり、二つ以上持てば生きで不詰みとなる。黒先白死の例を図 1.1、図 1.2 に示す。左の図 1.1 が問題で、右の図 1.2 がその答えとなる。この場合は、図 1.2 にあるように、黒が 1 の場所に打てば、その後白はどう打ったとしても二眼を持つことができず、死にとなる。この場合、黒の 5 が急所の一着となる。

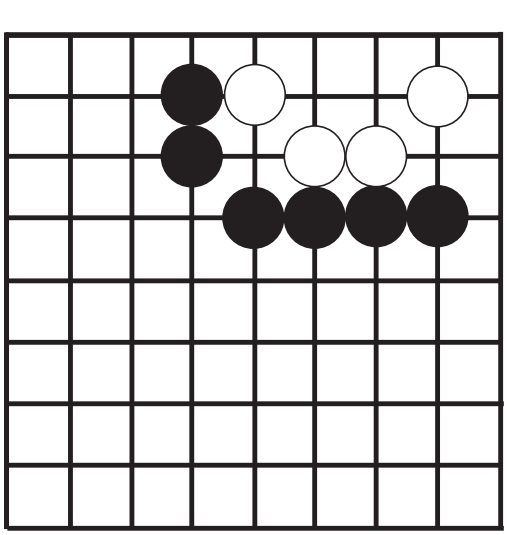

図 1.1: 詰碁の例 (問)

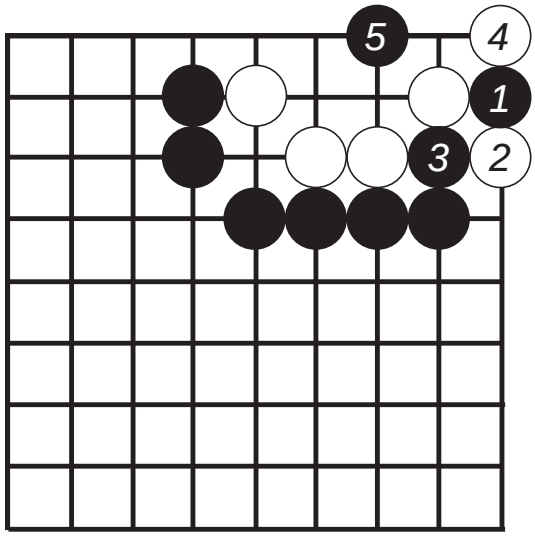

図 1.2: 詰碁の例 (答)

詰碁の探索の行き着く結果は、生き・死にの他にも、セキやコウ・ヨセコウ・長生など循環を伴う状態もあり、問題は複雑となっている。そのため、本研究では以下に述べる通常の問題にさらに制限をかけた一眼問題に対して実験を行った。

1.2.2 一眼問題

一眼問題とは、詰碁の問題にさらに制限をかけた問題を指す [26]。詰碁は前述の通り、目的の石が二眼を持つかどうかを焦点とするが、一眼問題では目的の石が眼を一つ持った状態からスタートする。そこからさらにその目的の石がもう一つ眼を持つことができるかどうかを計算する。つまり、詰碁の問題としては生きだけを扱うことになる。一眼問題が前提とする事項は以下の通りである。

- 始めから生かすべき石が決まっている
- 目的の石は始めから眼を一つ持つ
- 受け手側の石は攻め手側の生きた石に囲まれている
- 打てる範囲が決まっている

一眼問題の例を図 1.3 と図 1.4 に示す。前述の詰碁の例と同様、左の図 1.3 が問題で右の図 1.4 がその答えとなる。両図の盤の左上隅にある眼が、一眼問題の前提となる、石が初めから持つ眼であり、生かすべき黒石は △ の印が付いた石である。また、黒の石を囲む白の石は既に二眼を作っており、生きの状態になっている。よって、これによって囲まれた × の付いた点が着手できる交点である。この場合は、図 1.4 にあるように、黒が 1 の場所に打てば、その後は白がどう打ったとしてもさらに一眼を作り、二眼を持つため生きることができる。この場合、黒の 1 が急所の一着となる。

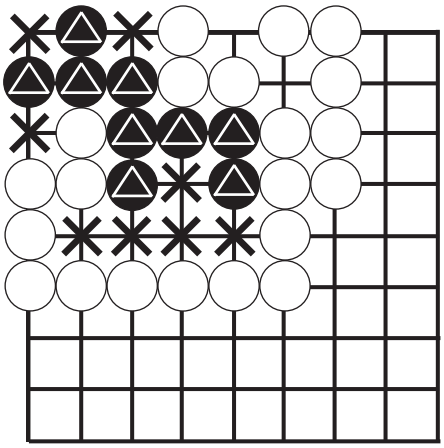

図 1.3: 一眼問題の例 (問)

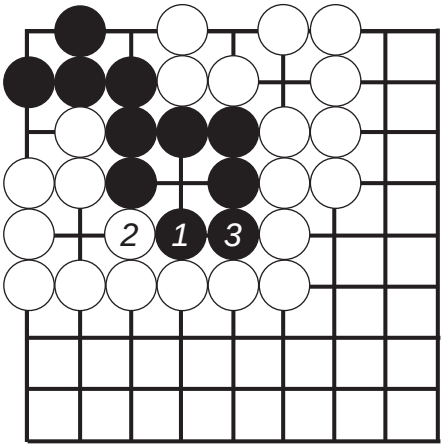

図 1.4: 一眼問題の例 (答)

1.3　本論文の構成

本論文は、本章を含めて 6 章と A・B2 つの Appendix によって構成される。第 2 章では、本研究で用いる探索手法について、既存の研究の内容を説明する。第 3 章では、本研究におけるシステムの実装と、提案する手法の内容を述べる。第 4 章では、本研究を行った実験環境と実験の結果を述べる。第 5 章では、実験結果について詳しく考察する。第 6 章は、本論文で述べている研究の内容とその結果を総括する。また、今後の課題についても述べる。最後に、本研究を行うにあたって指導を受けた方々にたいして謝辞を述べる。Appendix A では簡単に囲碁のルールを紹介する。Appendix B では、本研究で用いた一眼問題のデータを紹介する。

第2章　関連研究

本章では、基本となる探索法とその探索の戦略、また探索で用いられる評価関数と既存の詰碁プログラムについて概説する。

2.1　基本となる探索法

問題に対する知識がない場合には、問題の状態を一定の順序で調べていかなければならない。探索空間が有限で、解が存在するならば、必ず解を見つけることができる。探索空間をグラフで表現した時の探索の順序は、深さ優先探索と幅優先探索の2つに分類することができる[2]。探索空間の例を以下の図2.1に示し、それぞれについて説明する。

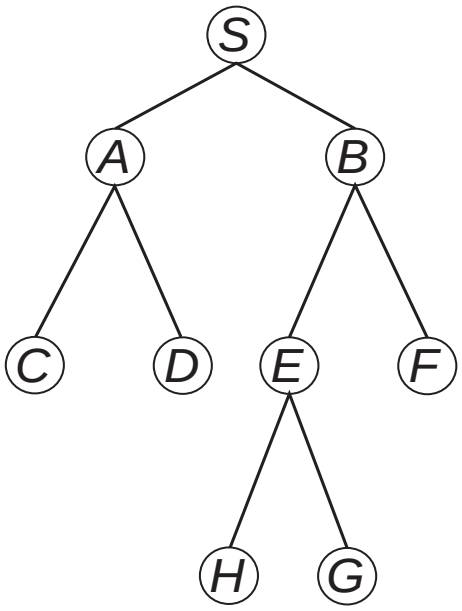

図 2.1: 探索空間

2.1.1 深さ優先探索

深さ優先探索 (depth-first search) は、ルートノードから離れたノードを優先的に調べる探索法である。問題の探索空間が木で表現される場合であれば、木の深いノードから先に調べる。この探索手法を用いると、図においてルートノード S から目標ノード G を見つけるまでの探索の順序は、A・C・D・B・E・H・G となる。調べているノードが目標ノードでない場合、次にその子ノードを調べようとする。もし n が子ノードを持たなければ n は行き止まりであり、その時は n の親ノード m に戻り、まだ調べてない m の子ノードを調べる。この繰り返しによって、ルートノード S から目標ノード G までの経路を調べる。

2.1.2 幅優先探索

幅優先探索 (breadth-first search) は、ルートノードから近いノードを優先的に調べる探索法である。この探索手法を用いると、図においてルートノード S から目標ノード G を見つけるまでの探索の順序は、A・B・C・D・E・F・H・G となる。

効率の比較

木の深さを n、平均の子ノード数 (ブランチングファクター) を b として、記憶容量と時間の効率について深さ優先探索と幅優先探索を考える。

深さ優先探索の場合を考える。記憶容量として、あるノードから子ノードへ探索を進める際はまだ探索をしていない $b-1$ 個の子ノード数を覚えておく必要がある。つまり、末端のノードを探索する際は、$(n-1)(b-1)$ 個のノード数を覚えておかなければならない。さらに末端の b 個のノードを展開することになるので、必要な記憶容量は $(n-1)(b-1)+b = n(b-1)+1$ となる。つまり、深さ n に対して線形に必要な記憶容量は増加していく。一方、処理時間としては、求める解が木の左端にある場合、探索するノード数は n となる。解が右端にある場合、全てのノードを探索しなければならないので、探索するノードの数は、$1+b+b^2+\cdots+b^{n-1} = \frac{b^n-1}{b-1}$ となる。これらの平均を取ると、探索すべきノード数の平均は $\frac{b^n+bn+b-n-2}{2(b-1)}$ となる。n の数が大きい場合、これはほぼ $\frac{b^{n-1}}{2}$ に等しくなる。つまり、平均的には木のおよそ半分を探索することになる。

幅優先探索の場合、末端のノードを探索する直前には末端より深さが 1 つ少ないノードは全てが展開されている必要がある。これにより、必要な記憶容量は b^{n-1} となる。つまり、深さ n に対して指数的に必要な記憶容量は増加していく。一方、処理時間としては、幅優先探索は末端でないノードは全て探索しなければならないため、探索するノード数は $1+b+b^2+\cdots+b^{n-2} = \frac{b^{n-1}-1}{b-1}$ となる。また、末端のノード数は最小で 1 個、最大で b^{n-1} 個のノード数を調べることになるので、平均は

$\frac{b^{n-1}+1}{2}$ となり、これらを足し合わせると、探索するノード数の平均は $\frac{b^n+b^{n-1}+b-3}{2(b-1)}$ となる。つまり、末端より深さが一つ少ないノード全てと、末端の半分のノードを探索することとなる。

これにより、記憶容量と時間両方の観点から見ても、深さ優先探索が有利なことが分かる。(ただし、処理時間だけは、n が大きく木の分岐数が多い場合には両者はあまり差が無い) しかし、求める解の深さが分からない場合は、深さ優先探索ではあまり上手くいかない場合がある。例えば、左の子から順に探索していく場合、右の子に解がある場合、幅優先探索ではすぐに解に辿り着くが、深さ優先探索では解が見つかるまでずっと深さがどこまであるか分からない左の子を掘り進むことになる。

2.1.3 反復深化法

必要な記憶容量が少ない深さ優先探索を用いた上で、幅優先探索のように浅いところに解がある場合でも解が求まる方法が反復深化法 (Iterative Deepning) である。反復深化法は、探索する深さに閾値を設け、それを段階的に増やしていきながら深さ優先探索をしていく方法である。探索するアルゴリズムは以下のようになる。

1. 閾値 n を 1 に設定する
2. 深さ n の深さ優先探索を行う
3. 2 で解が見つかれば探索を終了、見つからなければ n = n + 1 として 2 を行う

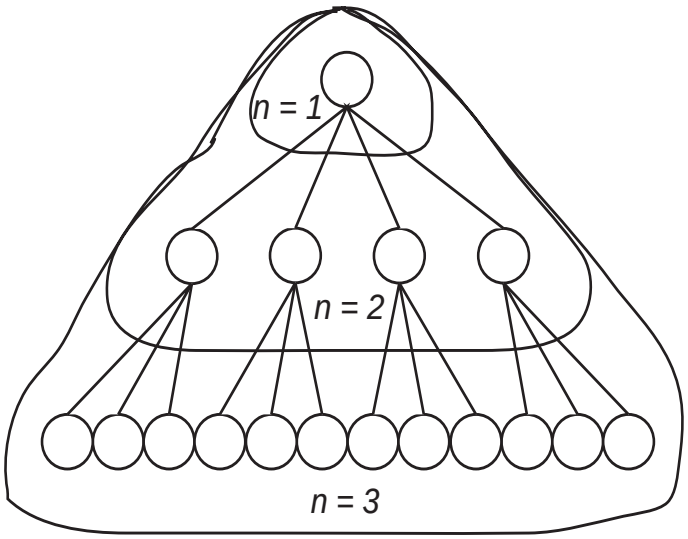

図 2.2: 反復深化法

それぞれで行っているのは深さ優先探索であるので、必要な記憶容量は深さ優先探索と同じである。また、探索する順番は、深さ 1 と 2 のノード、深さ 1 と 2 と 3 のノード $\cdots$ となり、同じものを何回も探索する以外は幅優先探索と同じなので、浅いところに解がある場合も最低限の深さで解が求まる。また、同じところを探索する場合は置換表 [19] を用いて探索の効率化を図る。

2.2 探索の戦略

詰碁は最終的に詰むか詰まないかの 2 種類の評価値しか存在しないような問題であり、そのような問題の探索には AND/OR 木の探索手法を用いることができる。一般的に詰めの問題に対しては木探索が非常に有効な手段となる。囲碁に AND/OR 木探索を行うことは前述した難しさがあるため困難だが、詰碁は盤面全体ではなく局所的な形勢についての判定問題であり、適度なブランチングファクターであることから、現在の技術の適用には最適と言える。

AND/OR 木には、AND ノードと OR ノードが存在する。OR ノードとは、先手番の局面に相当し、その子ノードのどれかが詰みであれば詰むようなノードを意味する。AND ノードとは、後手番の局面に相当し、その子ノードが全て詰んで初めて詰むようなノードを意味する。ゲームは初期の状態から二人が交互に手を指す。AND/OR 木探索も同様に、OR ノードから始まり、AND ノード・OR ノードと交互に展開して最適解を求める。この方法は解がある時には必ず解けるが、実際の問題では解の深さは分からないことがほとんどで、末端ノードを全て含む木を生成することは事実上不可能となる。そのため、探索をする時は深さなどのなんらかの閾値を設け、閾値まで探索し終わった木の末端ノードに対して、なんらかの方法でそのノードを評価して、評価値を決定する。この時にノードを評価する関数を静的評価関数 (static evaluation function) と呼ぶ。以下、ミニマックス法から Depth-First Proof-Number+探索まで、AND/OR 木探索を基本として発展した手法を示す。これらの手法は、最初の手を打つ人は最大の評価値を得る (自分にとって一番良いと思われる手) ことを目標にし、次の手を打つ人は最小の評価値を得る (相手にとって一番悪いと思われる手) ことを目標にすることを前提とする。つまり、双方は最善を尽くすことを前提とする。

2.2.1 ミニマックス法

最初に手を打つ人を Max、次に手を打つ人を Min と呼ぶとする。前述した双方は最善の手を尽くすという前提のもとに、静的評価関数 f から得られた末端ノードの評価値からルートノードにとって最大の評価値を得るような経路を選ぶ。Max のノードを n とし、その子ノードを $n_i(i = 1 \sim m)$、各子ノードを $n_{ij}(j = 1 \sim i_m)$ とすれば、Max ノードは n_i の評価値 $f(n_i)$ を最大化するノードを選ぶ。すなわち、

次式を満足する i を選ぶ。

$$f(n) = \max_i f(n_i) \tag{2.1}$$

同様に、Min はそれぞれの i に対して $f(n_i)$ を最小化するので、次式を満足するように j を選ぶ。

$$f(n_i) = \min_j f(n_{ij}) \tag{2.2}$$

従って、Max は次式によって i を選ぶこととなる。

$$f(n_{ij}) = \max_i \{\min_j f(n_{ij})\} \tag{2.3}$$

ゲーム木の深さを d とするならば、ルートノードは Max ノードなので、ルートノードは

$$f(n) = \max_{i_1} \min_{i_2} \cdots\cdots \{f(n_{i_1,i_2\cdots i_k})\} \tag{2.4}$$

となるように i_1 を選択する。以下に例として図 2.3 を示す。　ノード A をルート

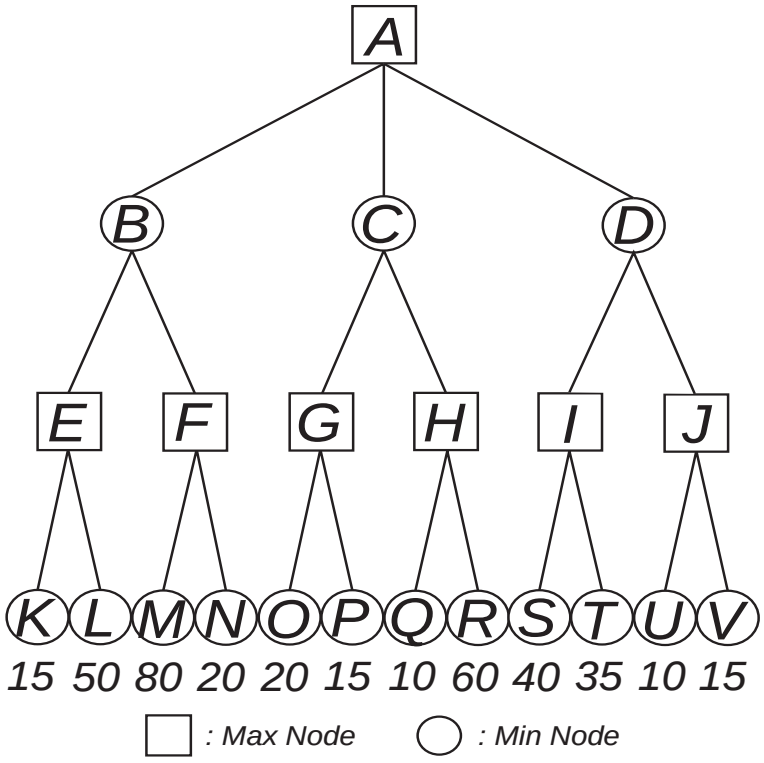

図 2.3: ミニマックス法

ノード、ノード $K \sim V$ を末端ノードとし、評価関数でそれぞれ図 2.3 にあるように 15・50･･･15 と評価値が与えられたとする。末端ノードから深さが 1 浅いノードを考えると、ノードは Max ノードであるため、ノード E の評価値は 50、ノード F は 80、ノード G は 20、ノード H は 60、ノード I は 40、ノード J は 15 となる。次

にさらに深さが 1 浅いノードを考えると、ノードは Min ノードであるため、ノード B は 50、ノード C は 20、ノード D は 15 となる。最後に、ルートノードであるノード A は、Max ノードであるため、50 が評価値となる。よってこの場合、ルートノードにとって最大の評価値 50 を与える A・B・E・L という経路を選ぶ。

2.2.2 $\alpha\beta$ 法

$\alpha\beta$ 法はミニマックス法を基本にして、各ノードにおいて評価値の下界と上界それぞれに α と β という 2 つの閾値を設けて、探索する最中に無駄な探索を行わないように枝刈りを行う探索である。以下に例として図 2.4 を示し、$\alpha\beta$ 法を具体的に説明する。

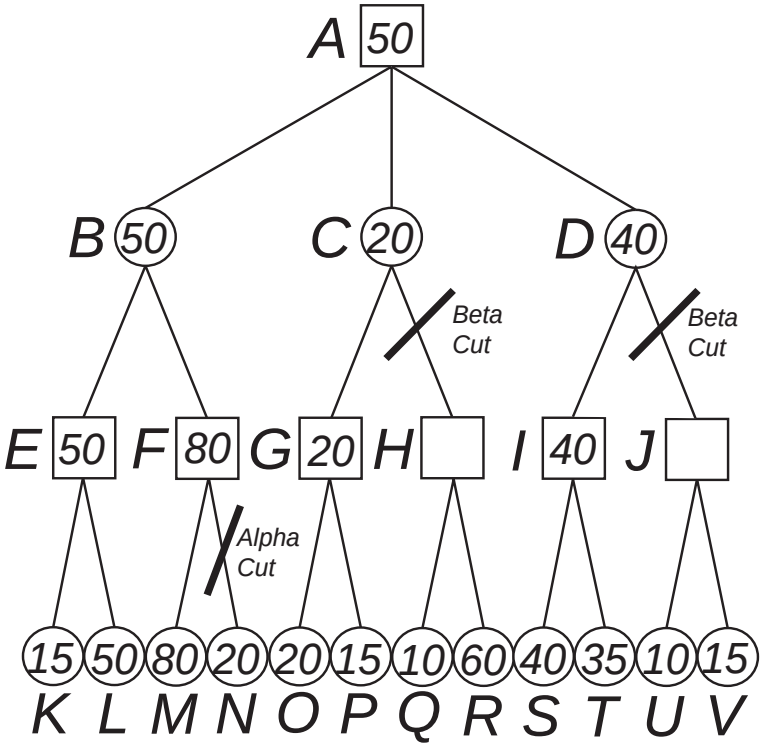

図 2.4: $\alpha\beta$ 法

初期状態ではルートノード A の α と β はそれぞれ $-\infty$ と ∞ である。これを $\alpha\beta(A) = (-\infty, \infty)$ と書くとする。ノード E の評価値が設定されると $\alpha\beta(E) = (50, 50)$ となる。その結果、ノード B では $\alpha\beta(B) = (-\infty, 50)$ となる。もしノード B の子ノードの評価値が $(-\infty, 50)$ の範囲内にないことが分かれば、この場合は 50 以上であれば、その子ノードは解になることがないので探索を打切ることができる。子ノード M を展開した時点で $\alpha\beta(F) = (50, \infty)$ となる。$(50, \infty)$ は $(-\infty, 50)$ の範囲内でないという探索打切り条件を満足しているため、ノード F はこれ以上

子ノードを探索する必要がない。よって、ミニマックス法では探索していたノードNを$\alpha\beta$法では探索せずに済むことができる。この時点で$\alpha\beta(B) = (50, 50)$となり、$\alpha\beta(A) = (50, \infty)$となる。探索が進み、ノードGの評価値が20に決まった時点で、$\alpha\beta(C) = (-\infty, 20)$となり、これは$(50, \infty)$の範囲内でないので、ノードC以下の探索を打切ることができる。よって、ミニマックス法では探索していたノードH・Q・Rを$\alpha\beta$法では探索せずに済むことができる。同様に探索が進み、ノードIの評価値が40に決まった時点で、$\alpha\beta(D) = (-\infty, 40)$となり、これも$(50, \infty)$の範囲内でないので、ノードD以下の探索を打切ることができる。よって、ミニマックス法では探索していたノードJ・U・Vを$\alpha\beta$法では探索せずに済むことができる。

第1の打切りは、ノードFが親ノードBのβの値以上となるため、第2の打切りは、ノードCの値が親ノードAのαの値以上となるため、第3の打切りは、ノードDの値が親ノードAのαの値以上となるために起こる。β値による枝刈りをベータカット、α値による枝刈りをアルファカットと呼ぶ。

最良の場合、$\alpha\beta$法によって調べなければならないノードの数は、ほぼ深さが半分の木をミニマックス法で探索する場合に調べなければならないノードの数が等しい。しかし、最悪の場合にはすべてのノードを調べなければならない。調べなければならないノード数は次式で近似することができる。

$$m^{\frac{k}{2}} < N(k, m) \leqq m^k \tag{2.5}$$

2.2.3 証明数探索

証明数探索は、それぞれのノードに証明数と反証数という2つの閾値を設けて探索を行う手法である。証明数･反証数の元となったのは、1980年代中頃にMcAllesterによって提案された共謀数という概念で、$\alpha\beta$法などの探索方法の対象とするMinimax木に対して提案されたものである[18]。Minimax木の探索においては、先端ノードの静的評価に適用される評価関数は一般に正確ではなく、その評価誤差が上のノードに伝播されて、ルートでの評価の誤りにつながる。しかし、一般には一つの局面だけでの評価誤差がルートでの評価誤差に直接結びつくわけではなく、複数のノードが共謀しなければ、ルートの評価値を変えることができない場合が多い。つまり、共謀しなければならないノードの数が多ければ多いほどルートでの評価値が誤っている可能性が高くなる。この考え方をAND/OR木に適応したものが証明数・反証数の概念である。

証明数とは、各ノードにおいて、その詰みを証明するために必要な子ノードの最小の数を意味する。ノードがORノードであった場合、その各子ノードで示される証明数の値は、ノードを探索する時に最低限必要なリソースの量と考えることができる。よって、ノードを展開して次に選ぶ子ノード中での最も有望なノードを選ぶ判断基準として、有効に働くと考えられる。反証数も同様に有効である

と考えられる。

証明数・反証数は、ノードの種類により計算方法が異なる。ノード n が末端ノードである場合の評価値が**true** つまりそのノードが詰みである時は、詰みであることが証明できているので証明数は 0 となり、不詰みであることは証明できないので、反証数は ∞ となる。末端ノードの評価値が**false**、つまりそのノードが不詰みである場合は、詰みの場合と逆なので、証明数が ∞ で反証数が 0 となる。また、評価値が不明の時は、どのくらいリソースをつぎ込めば詰み、あるいは不詰みが証明できるか分からないので、共に 1 とする。次に、ノード n が内部ノードでかつ OR ノードである場合は、そのノードの詰みを示す場合、子ノードのうちどれか一つが詰みであればよいので、その証明数は子ノードで証明数が最小のノードの証明数となる。また、不詰みであることを証明するためには、子ノード全てが不詰みでなければならないので、子ノードの反証数を全て足したものをそのノードの反証数とする。n が OR ノードの場合の証明数と反証数は、ノードが AND ノードの場合のそれらの計算と逆になる。このような計算を再帰的に行う。AND/OR 木探索における最終的な評価値 (「詰み」と「不詰み」) を**true** と**false** で一般化すると、証明数・反証数の定義は以下のようになる。

定義 [証明数 (反証数)]

1. ノード n が末端ノード

   (a) 最終的な評価値が**true** のとき

$$\mathbf{pn}(n) = 0 \tag{2.6}$$
$$\mathbf{dn}(n) = \infty \tag{2.7}$$

   (b) 最終的な評価値が**false** のとき

$$\mathbf{pn}(n) = \infty \tag{2.8}$$
$$\mathbf{dn}(n) = 0 \tag{2.9}$$

   (c) 最終的な評価値が不明のとき

$$\mathbf{pn}(n) = 1 \tag{2.10}$$
$$\mathbf{dn}(n) = 1 \tag{2.11}$$

2. ノード n が内部ノード

   (a) n が OR ノードのとき

$$\mathbf{pn}(n) = \min_{n_c \in children\ of\ n} \mathbf{pn}(n_{child}) \tag{2.12}$$
$$\mathbf{dn}(n) = \sum_{n_c \in children\ of\ n} \mathbf{dn}(n_{child}) \tag{2.13}$$

(b) n が AND ノードのとき

$$\mathbf{pn}(n) = \sum_{n_c \in children\ of\ n} \mathbf{pn}(n_{child}) \tag{2.14}$$

$$\mathbf{dn}(n) = \min_{n_c \in children\ of\ n} \mathbf{dn}(n_{child}) \tag{2.15}$$

2.2.4 Depth-First Proof-Number 探索

Depth-First Proof-Number(Df-pn) アルゴリズムとは、前述の証明数と反証数という二つの概念を用いる最良優先探索の証明数探索を深さ優先探索に変換し、多重反復深化法 (Multiple Iterative Deepning) という手法を用いたアルゴリズムである [20, 21, 22]。前述したように、深さ優先探索は、メモリの使用量や時間の効率が良く、詰碁のように大きな探索空間を持つ問題には適している。しかし、深さ優先探索では、解の存在する深さが深い場合、はまってしまう可能性があるので、そのような状況を避けるために探索をする時には閾値を設ける。多重反復深化法とは、この与えた閾値をそのまま用いるのではなく、各ノードで段階的に閾値を増やして探索を行う方法である。与えた閾値をそのままで探索を行った場合、不必要なノードを探索してしまうことも多いが、この方法を取ることで探索するノード数を抑えることが期待できる。これらの方法により、比較的少ないメモリ量で最良優先探索と同様の効率的な探索を実現している。

Depth-First Proof-Number 探索は、以下のようなアルゴリズムで探索を行う。ルートノード r での閾値を

$$r.th_p = \infty,\ r.th_d = \infty \tag{2.16}$$

とする。

1. 各ノード n においては、上記の定義に則って計算したノードの証明数 $n.pn$ が閾値 $n.th_p$ 以上になるか、反証数 $n.dn$ が閾値 $n.th_d$ 以上になるまで (これを終了条件と呼ぶ) 自分の下を探索し続ける。

2. n が OR ノードの場合、証明数最小の子ノード n_c、二番目に証明数の小さい子ノード n_2 を選ぶ (証明数最小の子ノードがもう一つあれば、そのノードを n_2 とする)。

$$n_c.th_p = \min(n.th_p, n_2.pn + 1) \tag{2.17}$$

$$n_c.th_d = n.th_d + n_c.dn - \sum n_{child}.dn \tag{2.18}$$

のように閾値を割り当て、n_c を探索する。終了条件を満たすまでこの操作を繰り返す。

3. n が AND ノードの場合、反証数最小の子ノード n_c、二番目に反証数の小さい子ノード n_2 を選ぶ (反証数最小の子ノードがもう一つあれば、そのノードを n_2 とする)。

$$n_c.th_p = n.th_p + n_c.pn - \sum n_{child}.pn \quad (2.19)$$

$$n_c.th_d = \min(n.th_d, n_2.dn + 1) \quad (2.20)$$

のように閾値を割り当て、n_c を探索する。終了条件を満たすまでこの操作を繰り返す。

4. 終了条件を満たしたら、親ノード n_{parent} に処理を戻す。

閾値は、コンピュータがそのノードの探索につぎ込むリソースの量を表したものと考えることができる。ルートノードでは、リソースの全てをつぎ込むと言う意味で閾値を ∞ とする。ノード n が OR ノードの場合、証明数は子ノードのうちどれかが詰みであることを証明できればよい。よって、ノード n_c に許される証明数のためのリソースの量は、許されるリソースの全てか、あるいは二番目に証明数が少ないノード n_2 を証明するためにつぎ込まなければならないリソースの量に 1 を加えたもののうちより少ない方を取ればよい。反証数は子ノードすべてを不詰みであることを示さなければならないので、ノード n_c につぎ込める反証数のためのリソースの量は、許されたリソースの量から自分以外の子ノードの反証数を引いた値となる。ノード n が AND ノードの時は OR ノードの逆を考えればよい。

以下に例として図 2.5 を示し、具体的に説明する。

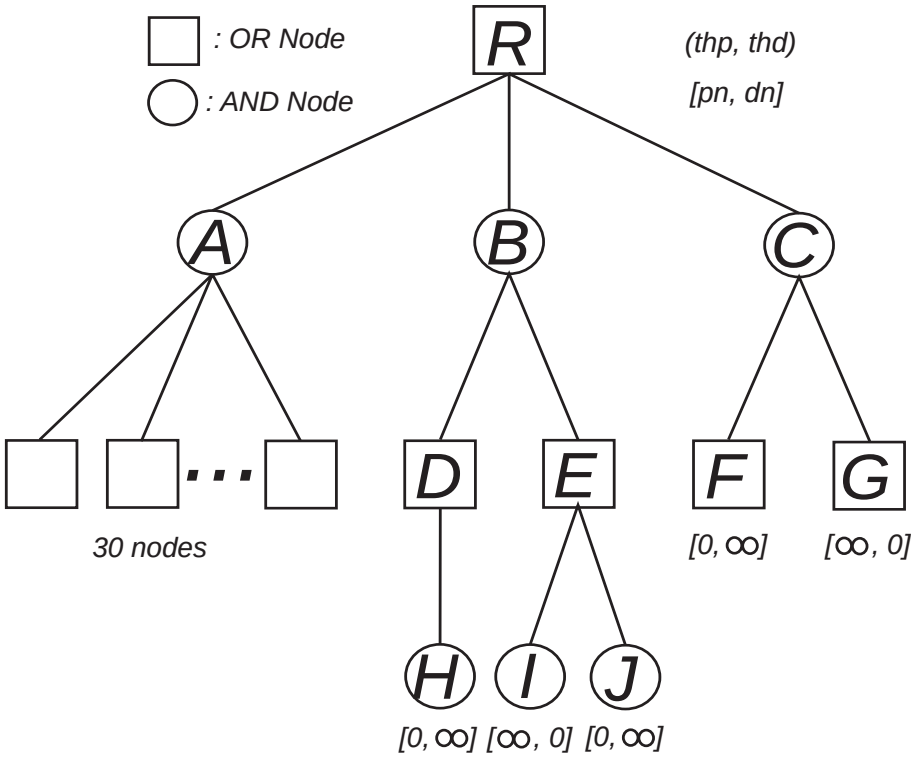

図 2.5: Depth-First Proof-Number 探索

図 2.5 のノード F〜J を末端ノードとし、それぞれの証明数・反証数を図に書かれている値とする。A の子ノードは、子ノードが 30 ノードあるとする。各ノードでの証明数・反証数を [pn, dn] で、証明数・反証数の閾値を (th_p, th_d) で表す。

まず、初めにルートノードに証明数と反証数の閾値にそれぞれ ∞ が与えられる。この時、ルートノードは評価が分からないため、式 (2.10)・(2.11) より証明数・反証数共に 1 が与えられ、共に閾値を超えないのでさらに子ノードを探索する。

ルートノードの子ノードは三つ存在し、それらは全て末端ノードではなく評価値が分からないため、それぞれの証明数・反証数に 1 が与えられる。これにより式 (2.12)・(2.13) からルートノードの証明数は 1、反証数は 3 となる。共に閾値を超えていないので、さらに子ノードを探索する。ルートノードの子ノードの証明数は全て同じであるので、ノード A を探索するとする。ノード A の証明数と反証数の閾値は、式 (2.17)・(2.18) より 2 と ∞-2 となる。ノード A の子ノードは 30 個あり、それら全て末端ノードではなく評価値が分からないため、全て証明数・反証数ともに 1 となる。これにより、ノード A の証明数は 30、反証数は 1 となり、証明数が閾値を超えたため、ノード A での探索が終わり親のノードへ処理が戻される。ノード A の証明数と反証数が変わったことにより再度ルートノードでの証明数と反証数を計算し直すと、式 (2.12)・(2.13) より 1 と 3 になる。共に閾値を超えていないため、さらに子ノードを探索する。

ルートノードの子ノードを見てみると、ノードBとCが証明数が同じため、ノードBを探索することとする。ノードBの証明数と反証数の閾値は式(2.17)・(2.18)から2と∞-2となる。ノードBの子ノードDとEは末端ノードではなく評価値が分からないため、その証明数と反証数は共に1となる。これにより、ノードBの証明数と反証数は2と1になり、証明数が閾値以上になったため、ノードBでの探索が終わり親のノードへ処理が戻される。これよりルートノードの証明数と反証数は式(2.12)・(2.13)より1と3になり、共に閾値を超えていないため、さらに子ノードを探索する。

ルートノードの子ノードを見てみると、ノードCの証明数が1で最小のため、ノードCを探索することになる。ノードCの証明数と反証数の閾値は式(2.17)・(2.18)から3と∞-2となる。ノードCの子ノードFとGは末端ノードで、その与えられた数値から式(2.14)・(2.15)よりノードCの証明数と反証数は∞と0となり、証明数が閾値を超えたためノードCでの探索は終わり、親ノードへ処理が戻される。これによりルートノードの証明数と反証数は式(2.12)・(2.13)より2と2になり、共に閾値を超えていないため、さらに子ノードを探索する。

ルートノードの子ノードを見てみると、ノードBの証明数が2で最小のため、ノードCを探索することになる。ノードBの証明数と反証数の閾値は式(2.17)・(2.18)から31と∞-1となる。ノードBの証明数と反証数は2と1であったので、共に閾値を超えていないのでさらに子ノードを探索する。ノードBの子ノードDとEは共に反証数が等しいため、ノードDを探索することとする。ノードDの証明数と反証数の閾値は式(2.19)・(2.20)から30と2となる。ノードDの子ノードHは末端ノードで、その与えられた数値からノードDの証明数と反証数は0と∞となり、反証数が閾値を超えたためノードDでの探索は終わり、親ノードへ処理が戻される。これにより、ノードBの証明数と反証数は式(2.14)・(2.15)から1と1になり、共に閾値を超えていないため、さらに子ノードを探索する。ノードEの方がノードDよりも反証数が小さいため、次はノードEを探索することになる。ノードEの証明数と反証数の閾値は、式(2.19)・(2.20)より31と∞-1となる。ノードEの子ノードは末端ノードで、その与えられた数値からノードEの証明数と反証数は、0と∞となる。反証数が閾値を超えたため、ノードEの探索は終わり、親ノードへ処理が戻される。これにより、ノードBの証明数と反証数は式(2.14)・(2.15)から0と∞となる。これにより反証数が閾値を超えたため、ノードBでの探索が終わり、親ノードへ処理が戻される。これにより、ルートノードの証明数と反証数は0と∞となり、反証数が閾値を超えたため探索が終わり、これによりこの探索木の探索が全て終了する。以上のようにしてDf-pnは探索を行う。この探索木の場合、結果的にはノードHかJに至る手筋が正解ということになる。

以下にDepth-First Proof-Number探索の擬似コードを示す。なお、ORノードにおける証明数とANDノードにおける反証数は本質的には等価であり、同様にORノードにおける反証数とANDノードにおける証明数も本質的に等価であることから、ϕとδを用いてアルゴリズムの簡略化を行った。

$$\phi = \begin{cases} pn(n) & (n \text{ は } OR \text{ ノード}) \\ dn(n) & (n \text{ は } AND \text{ ノード}) \end{cases} \tag{2.21}$$

$$\delta = \begin{cases} dn(n) & (n \text{ は } OR \text{ ノード}) \\ pn(n) & (n \text{ は } AND \text{ ノード}) \end{cases} \tag{2.22}$$

```
1   //ルートでの反復深化
2   procedure Df-pn(r) {
3     φ(r) = ∞ - 1; δ(r) = ∞ - 1;
4     MID(r);
5   }
6   //ノード n の展開
7   procedure MID(n) {
8     // 1. ハッシュを引く
9     LookUpHash(n, φ, δ);
10    if (φ(n) ≦ φ || δ(n) ≦ δ) {
11      φ(n) = φ; δ(n) = δ;
12      return;
13    }
14    // 2. 合法手の作成
15    if (n が末端ノード) {
16      if ( ( n が AND ノード&&Eval(n) = true) ||
17           ( n が OR ノード&&Eval(n) = false) ) {
18        φ(n) = ∞; δ(n) = 0;
19        } else { φ(n) = 0; δ(n) = ∞; }
20        PutInHash(n, φ(n), δ(n));
21        return;
22      }
23    GenerateLegalMoves();
```

```
24      // 3. 多重反復深化
25      while (1) {
26        // φかδがその閾値以上なら探索終了
27        if (φ(n) ≦ ΔMin(n) || δ(n) ≦ ΦSum(n) ) {
28          φ(n) = ΔMin(n); δ(n) = ΦSum(n);
29          PutInHash(n, φ(n), δ(n));
30          return;
31        }
32        n_c = SelectChild(n, φ_c, δ_c, δ_2);
33        φ(n_c) = δ(n) + φ_c - ΦSum(n);
34        δ(n_c) = min(φ(n), δ_2 + 1);
35        MID(n_c);
36      }
37    }
38    //子ノードの選択
39    procedure SelectChild(n, &δ_c, &δ_c, &δ_2) {
40      δ_c = ∞; δ_2 = ∞;
41      for (各子ノード n_child について) {
42        LookUpHash(n_child, φ, δ);
43        if (δ < δ_c) {
44          n_best = n_child;
45          δ_2 = δ_c; φ_c = φ; δ_c = δ;
46        } else if (δ < δ_2) δ_2 = δ;
47        if (φ = ∞) return n_best;
48      }
49      return n_best;
50    }
51    //ハッシュを引く
52    procedure LookUpHash(n, &φ, &δ) {
53      if (nが登録済み) {
54        φ = Table[n].φ; δ = Table[n].δ;
55      } else {φ = 1; δ = 1; }
56    }
57    //ハッシュに記録
58    procedure PutInHash(n, φ, δ) {
59      Table[n].φ = φ; Table[n].δ = δ;
60    }
```

```
61  //n の子ノードの δ の最小を計算
62  procedure ΔMin(n) {
63    min = ∞;
64    for (各子ノード n_child について) {
65      LookUpHash(n_child, φ, δ);
66      min = min(min, δ);
67    }
68    return min;
69  }
70  //n の子ノード δ の和を計算
71  procedure ΦSum(n) {
72    sum = 0;
73    for (各子ノード n_child について) {
74      LookUpHash(n_child, φ, δ);
75      sum = sum + φ;
76    }
77    return sum;
78  }
```

図 2.6: Df-pn の擬似コード

2.2.5 Depth-First Proof-Number+探索

人間が局面を見て先読みをする時は、いくつかの見込みのある手を絞り、その手を深く先読みする。一方、あまり見込みのないような手は、早々に読むのを止めてしまう。コンピュータによる探索でも、このように見込みのある手・見込みのない手を判別することができれば、探索をより効率的に行うことができる。Depth-First Proof-Number+とは、Df-pn にヒューリスティックスと呼ばれる問題領域固有の知識を使って人間の見込みを表現できるようにした探索法である [27]。

ノード n からその子ノード n_{child} までにかかるコストを $cost_{(dis)proof}(\mathrm{n}, n_c)$、あるヒューリスティックスを用いて静的に評価したノード n の評価値を $h_{(dis)proof}(\mathrm{n})$ と定義し、上記の式 (2.10)〜(2.15)・(2.17)〜(2.20) を以下のように変更する。また、Df-pn ではノードの閾値を決める時に、証明数最小のノードと二番目に小さいノードを選んだが、Df-pn+では、証明数に cost を加えたものの中で最小のものを n_c に、同様の基準で二番目に小さいものを n_2 として選ぶ。

$$\mathbf{pn}(n) = h_{proof}(n) \tag{2.23}$$

$$\mathbf{dn}(n) = h_{disproof}(n) \tag{2.24}$$

$$\mathbf{pn}(n) = \min_{n_c \in children\ of\ n}(\mathbf{pn}(n_{child}) + cost_{proof}(n, n_{child})) \tag{2.25}$$

$$\mathbf{dn}(n) = \sum_{n_c \in children\ of\ n} (\mathbf{dn}(n_{child}) + cost_{disproof}(n, n_{child})) \quad (2.26)$$

$$\mathbf{pn}(n) = \sum_{n_c \in children\ of\ n} (\mathbf{pn}(n_{child}) + cost_{proof}(n, n_{child})) \quad (2.27)$$

$$\mathbf{dn}(n) = \min_{n_c \in children\ of\ n} (\mathbf{dn}(n_{child}) + cost_{disproof}(n, n_{child})) \quad (2.28)$$

$$n_c.th_p = \min(n.th_p, n_2.pn + cost_{proof}(n, n_2) + 1) - cost_{proof}(n, n_c) \quad (2.29)$$

$$n_c.th_d = n.th_d + n_c.dn - \sum(n_{child}.dn + cost_{disproof}(n, n_c)) \quad (2.30)$$

$$n_c.th_p = n.th_p + n_c.pn - \sum(n_{child}.pn + cost_{proof}(n, n_c)) \quad (2.31)$$

$$n_c.th_d = \min(n.th_d, n_2.dn + cost_{disproof}(n, n_2) + 1) - cost_{disp}(n, n_c) \quad (2.32)$$

今まで探索の途中で評価値が不明なノードは評価値を 1 としてきたが、各ノードの局面によって静的に評価を与え、その差異で各ノードの差別化を行う。この評価値を判断するものとして、h を用いる。また、cost を用いて、各ノードにおける証明数・反証数や閾値の計算では、次に探索するノードを選ぶ際に他の兄弟ノードに対して局面に評価値を与えることにより差別化を行い、次に探索するノードの順番を変える。このような h と cost という二つのヒューリスティックスを用いて、人間が行っている見込みの判断を表現する。

以下に Depth-First Proof-Number+探索の擬似コードを Depth-First Proof-Number 探索から変えられた部分のみ示す。それぞれの関数の頭の行数が Depth-First Proof-Number 探索の擬似コードの該当関数の頭の行数と一致する。

```
38  //子ノードの選択
39  procedure SelectChild(n, &δ_c, &δ_c, &δ_2) {
40    δ_c = ∞; δ_2 = ∞;
41    min = ∞; min_2 = ∞
42    for (各子ノード n_child について) {
43      LookUpHash(n_child, φ, δ);
44      if (δ + cost_φ(n, n_child < δ_c) {
45        n_best = n_child;
46        min_2 = min;
47        min = δ + cost_φ(n, n_child)
48        δ_2 = δ_c; φ_c = φ; δ_c = δ;
49      } else if (δ + cost_φ(n, n_child) < δ_2) {
50        δ_2 = δ; min_2 = φ + cost_φ(n, n_child)
51      }
```

```
52        if (δ = ∞) return n_best;
53      }
54      return n_best;
55    }

51    //ハッシュを引く
52    procedure LookUpHash(n, &φ, &δ) {
        ...
55      } else {φ = h_φ(n); δ = h_δ(n); }
56    }

61    //n の子ノードの δ の最小を計算
62    procedure ΔMin(n) {
        ...
66        min = min(min, δ + cost_φ(n, n_child));
        ...
69    }

70    //n の子ノード δ の和を計算
71    procedure ΦSum(n) {
        ...
75        sum = sum + φ + cost_δ(n, n_child);
        ...
78    }
```

図 2.7: Df-pn+の擬似コード

2.2.6 Df-pn 探索の問題点

Df-pn ではノードを展開するかどうかの判断基準は以下のようになっている。

$$n.pn \geqq th.pn \parallel n.dn \geqq th.dn \tag{2.33}$$

Df-pn は上の条件を満たした時に探索は止まり、親ノードに情報を返すが、この Df-pn の判断基準はループを含む探索空間を持つゲームでは問題を起こす可能性がある [23, 24]。以下の図 2.8 を例に考える。

この図 2.8 において A を探索するとする。A を展開するにあたっての閾値は、大抵の場合 A の証明数 (反証数) よりも少しだけ大きな値に設定されている。ここで、A での証明数の閾値を th.pn = A.pn + 1、D の証明数を 1 とする。この状況で E に辿り着いたとすると、E での証明数の閾値は th.pn = A.pn となる。E の証

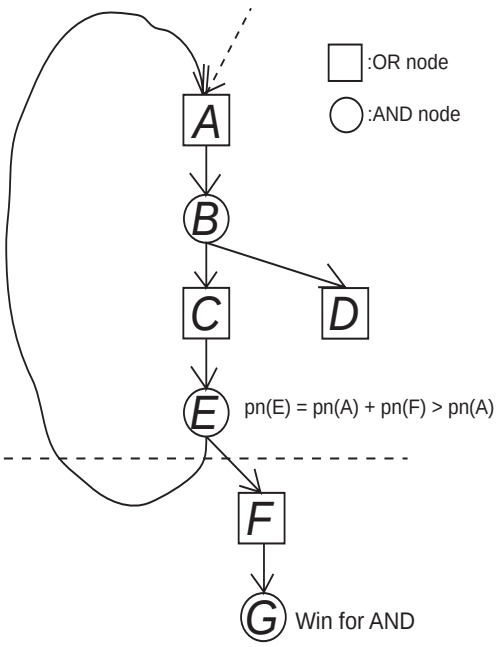

図 2.8: ループを含むゲームでの Df-pn の問題

明数は、F の証明数を 1(まだ展開されていない) と仮定すると

$$E.pn = A.pn + F.pn = A.pn + 1 > A.pn = th.pn \tag{2.34}$$

となる。ここで Df-pn の判断基準を考えると、E の証明数は証明数の閾値より大きいので、E は展開されずに親ノードに戻ることになる。このグラフ自体は F→G を展開すれば、AND ノードの勝ち (不詰め) が証明されるのだが、上の現象が永久に起こり続けるために E 以下のノードは全く展開されない。

この問題を展開するために、ハッシュのエントリーを増やして、ノードを展開するたびに、ルートノードからそのノードまでの最小距離 (minimal distance) を計算して、その最小距離の情報を保持するようにするという方法 (最小距離法) が提案されている。つまり、探索の深さが深くなればなるほど最小距離が大きくなることから、その事実からループを引き起こす可能性のあるノードの証明数を無視するようにする。md を最小距離、$n_i.md >$ n.md であるような n の子ノード n_i(i $= 1 \sim$ l) (以下、普通の子ノードと呼ぶ)、n_j.md $>$ n.md であるような n の子ノード n_j(k $=$ l $+ 1 \sim$ k) (以下、異常な子ノードと呼ぶ) として、AND ノードと OR ノードでの式 (2.12)〜(2.15) のようにしていた証明数と反証数の計算方法を以下のように変更する。

1. n が OR ノードの時

$$n.pn = \min_{1 \leqq i \leqq k} n_i.pn \tag{2.35}$$

$$n.dn = \begin{cases} \sum_{i=1}^{l} n_i.dn \ (if \sum_{i=1}^{l} n_i.dn \neq 0) \\ \max_{l+1 \leqq j \leqq k} n_j.dn \ (if \sum_{i=1}^{l} n_i.dn = 0) \end{cases} \quad (2.36)$$

2. n が AND ノードの時

$$n.pn = \begin{cases} \sum_{i=1}^{l} n_i.pn \ (if \sum_{i=1}^{l} n_i.pn \neq 0) \\ \max_{l+1 \leqq j \leqq k} n_j.pn \ (if \sum_{i=1}^{l} n_i.pn = 0) \end{cases} \quad (2.37)$$

$$n.dn = \min_{1 \leqq i \leqq k} n_i.dn \quad (2.38)$$

また、最小距離法を機能させるためには不十分である。n を AND ノードとして、普通の子ノードと異常な子ノードを持っていて、普通の子ノードはすでに証明 (詰み) されてしまったとする。この時、n の結果を調べるためには、異常な子ノードを展開しなければならない。つまり、n 自体が異常なノードになってしまう (n が OR ノードの時も同様)。この場合、n の最小距離を更新しなけらばならない。岸本は、まだ解かれていない異常なノードの中で、最小距離が一番小さなものに n の最小距離を更新している。

2.2.7 GHI 問題

GHI(Graph History Interaction) 問題とは、最良優先探索において詰む局面を不詰め (あるいはその逆) と誤った判断をしてしまう問題である [25]。これはループと DAG のある木を探索する時に起こる問題である。囲碁には、長生という双方譲らないと同型反復となり無勝負となってしまうというルールがある (将棋でいう千日手)。詰碁では目的が達成できなければ不詰みである。以下の図 2.9 を用いて GHI 問題での誤った判断の例を示す。

ノード D で不詰め、ノード I で詰みであったとする。A-B-E-H-J-K-E と探索した時、E-H-J-K というループを発見する。この時、K を先ほどのルールから不詰みとすると間違いになる。正解手順は A-C-F-H-J-K-E-G-I で、結果は詰みとなる。

GHI 問題を考えない場合、Df-pn ではルートに証明数・反証数共に閾値として ∞ を与えていた。しかし、GHI 問題を考慮しなくてはならないような問題では以下のように変更を加える。

1. ルートで閾値に ∞-1 を与えて探索する。このフェーズは、ループを避けつつ詰・不詰を見つけられるかどうかという探索を意味する。
2. 1 が終了した時、ルートの証明数と反証数がそれぞれ 0 と ∞ ならば問題なく解けたことを意味する。そうでないならば、ループを避けることが出来なかっ

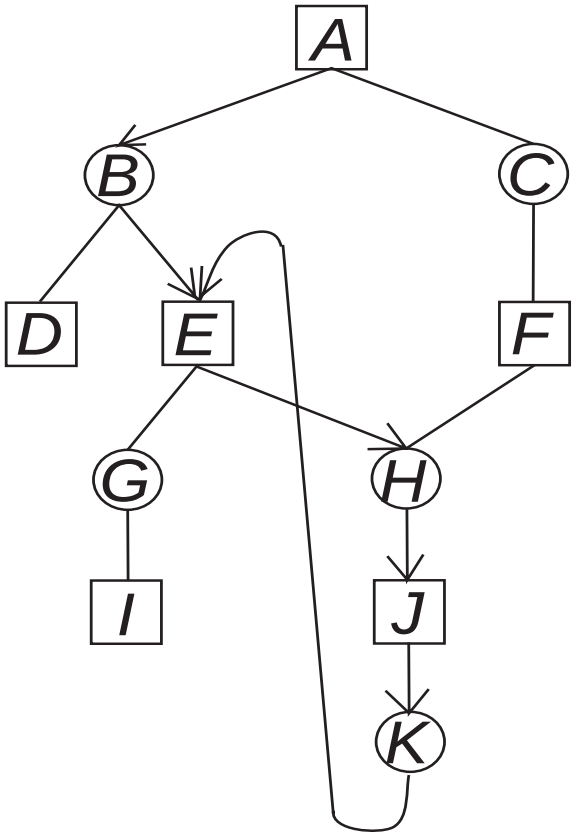

図 2.9: GHI 問題

たことを意味する。そこで、ルートに閾値として ∞ を与えて再度探索する。このフェーズでは、ループは不詰めの扱いをする。

ルート以外の全てのノードについても同様の処理を行う。なお、

$$\sum \text{有限の証明数} < \infty - 1 < \infty \tag{2.39}$$

でなくてはならない。

2.3 静的評価関数

静的評価関数とは、その局面の状態だけから得られる情報で、局面の状態を数値化する関数である。評価関数は、駒の価値やヒューリスティックスを用いて人手で作成する方法と、機械学習などによって機械的に作成する方法の 2 通りがある。この評価関数の正確さは処理速度とトレードオフの関係にある。

2.3.1 Bouzy's 5/21 Algorithm

Bouzy's 5/21 Algorithm は、Brono Bouzy によって作られた、囲碁の局面を静的に評価するアルゴリズムである [28]。このアルゴリズムは、基本的に Dilation と Erosion という 2 つの操作で構成される。アルゴリズムの流れは以下の通りである。

1. 盤上で死んでいる石を取り除く
2. 盤上で残った石のある交点に対して、高い評価値を与える (例えば、黒ならば 128、白ならば-128)
3. 空点に 0 の評価値を与える
4. Dilation を n 回繰り返して行う
5. Erosion を m 回繰り返して行う

Dilation と Erosion の操作は以下の通り。

- Dilation(膨張)
  - 交点の評価値が 0 以上かつ負の評価値を持つ交点が周りに無い
    - 現在の評価値に周りにある正の交点の個数を足す
  - 交点の評価値が 0 以下かつ正の評価値を持つ交点が周りに無い
    - 現在の評価値から周りにある負の交点の個数を引く
- Erosion(侵食)
  - 交点の評価値が正
    - 現在の評価値から周りにある 0 以下の交点の数を引く
  - 交点の評価値が負
    - 現在の評価値に周りにある 0 以上の交点の数を足す
  - それぞれ評価値が 0 になったら Erosion の操作を止める

これらの操作を碁盤上の交点全てに対して行う。以下に Dilation を 3 回、Erosion を 7 回行った流れを例として示す。

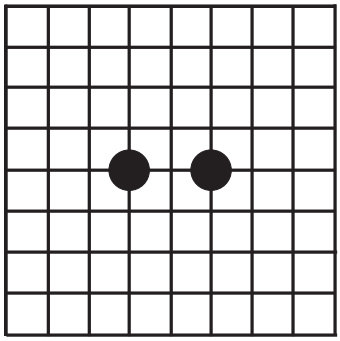

図 2.10: 初期状態

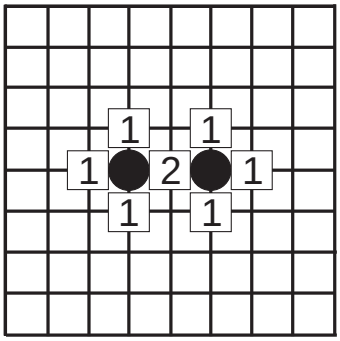

図 2.11: 1 Dilation

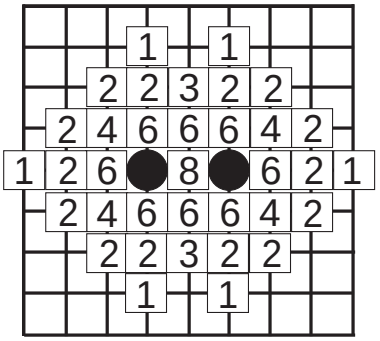

図 2.12: 3 Dilation

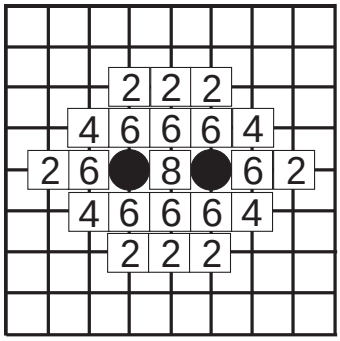

図 2.13: 3 Dil + 1 Ero

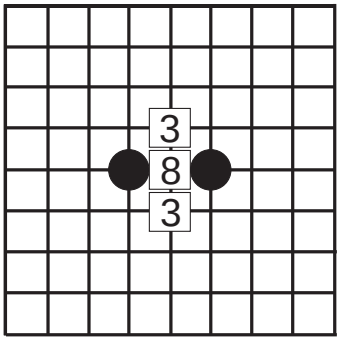

図 2.14: 3 Dil + 6 Ero

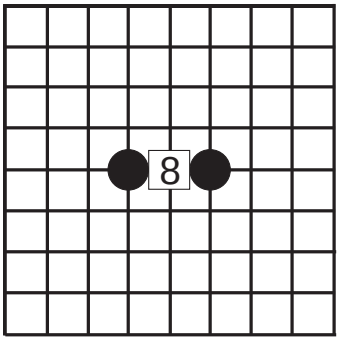

図 2.15: 3 Dil + 7 Ero

Erosion の回数は、m = 1 + n(n - 1) の計算式 [29] から、Dilation の数 n で表現される。また、このアルゴリズムは *GNU Go*[9] のバージョン 2 で使われており、5Dilation+21Erosion を目算、5Dilation+10Erosion を模様、4Dilation+0Erosion を勢力図として用いている。また、現行のバージョン 3 でも連 (石の連なり) の呼吸点 (ダメ) の算出に用いられている。

2.4 既存のプログラム

様々な研究の成果として、囲碁のプログラムは現在多数存在する [9, 10, 11, 12, 13]。ここでは現在最も有名な詰碁の解析プログラムの一つである *GoTools* のアルゴリズムについて紹介する。

2.4.1 GoTools

GoTools は Thomas Wolf によって開発された、一眼問題の条件にあるような完全な敵石に囲まれた局面の死活問題に特化した詰碁解析プログラムである [30, 31]。このプログラムは、探索戦略としては単純な $\alpha\beta$ 法をとっており、ヒューリスティックスを用いて探索が効果的に行われることが予想されるノードから選択し、それ

によって高速化を図っている [32, 33, 34]。*GoTools* に用いられているヒューリスティックスは大きく静的なものと動的なものの 2 つに分かれる。

静的な規則

ヒューリスティックスの大部分がこの静的な規則であり、当該局面の盤上の場所や盤上の他の石の配置などの静的な情報に依存した規則となっている。それら静的な規則は、相手が打てないような箇所には低い優先順位を与え、次に相手が打てる箇所には高い優先順位を与える。

動的な規則

部分木の探索によって得られた情報を元に、動的な規則から着手の整序を行う。以下の図 2.16 に示す部分木を例に動的な規則を説明する。図中における数字は着手の生成する順番を表し、文字は着手する石の場所を表すものとする。

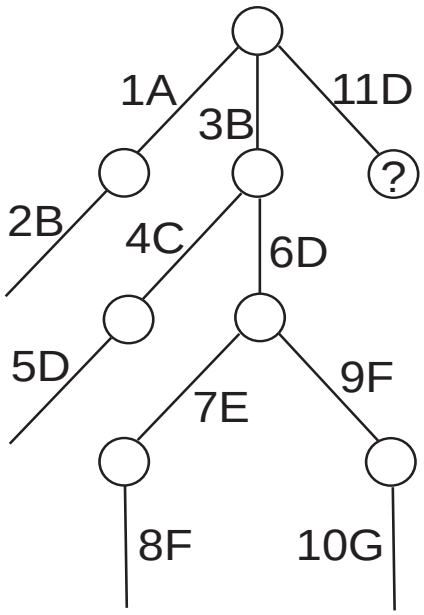

図 2.16: 部分木

ルートノードを白とする。まず白がAに打ち、次に黒がそれに対して後に黒の勝ちを保証するような着手Bを見つけたとする。すると、白は次にAの代わりに、次の黒が勝ちへのチャンスを減らす強烈な着手としてBを試みる。続けて、黒はCに打ち、次に白が勝ちを保証するような着手Dを発見したとすると、黒は次にCの代わりに着手Dを試みる。同様に探索を続ければ、ノード?では着手BとFとGがある程度の信頼を持つ着手として評価ができる。このように、動的な規則は次

の局面において、相手が打ってくるような手を試みる規則であり、このような規則は、着手の良い整列を達成する上でいくらかの効果がある。また、この他にも

- 木のほかの場所で有効であった手はランクを上げて先に試す
- 中間結果をハッシュ表に蓄えて利用する
- 自殺手などは考慮しない (例外あり)
- 各種のパラメタによる木の枝刈り

などを行って処理の効率化を図っている。

また、*GoTools* ではこれらヒューリスティックスのパラメータを遺伝的アルゴリズムを用いて調整している [34]。上記のヒューリスティックスを、静的な情報に関するパラメータ 45 個・動的な情報に関するパラメータ 10 個・枝刈りに関するパラメータ 14 個の三種類にカテゴライズし、それぞれに対して独立に遺伝的アルゴリズムを適用して最適化を図っている。

第3章　研究の概要

本章では、研究の方針と、それに沿ったシステムの実装について述べる。

3.1　提案手法

評価関数は、探索手法とともにコンピュータゲームプレイヤの振る舞いを決める重要な要素である。この評価関数は、囲碁を始めとして将棋やチェスなどの比較的難しいゲームにおいて、人手で作成するのが一般的となっている。評価関数がより正確であれば探索はより正しい結果を導くことができるが、前章でも述べたように、評価関数の正確さは処理速度とトレードオフの関係にある。特に囲碁に関しては、駒に個性が無いなのどゲームの特徴から、正確さを増すために前に説明した *GoTools* のようにかなりの部分を人手で書いており、処理速度の無視できない割合を評価関数の部分が占めている。先行研究として岸本が行った詰碁の解析プログラムにDepth-First Proof-Number探索を用いている例[26]も、評価関数として囲碁の知識を用いて作成したものを使っている。そこで、本研究では前章で紹介した一定の計算式から局面を評価でき、計算時間も比較的軽くて済むBouzy's 5/21 AlgorithmをDepth-First Proof-Number+探索の評価関数として適用して、人手で作られたものだけではなく定式化された評価関数でも有効であることを示し、現状よりも良い結果を出す詰碁解析プログラムを作ることを目指した。

3.2　システムの実装

一眼問題にBouzy's 5/21 Algorithmを適用することを考える。当然、眼となっている部分は評価値が最も高いはずである。詰碁では二眼を作れば生きなのであるから、Bouzy's 5/21 Algorithmを適用した時に、始めからある眼の評価値と同じ評価値を他の交点が持てば、そこがもう一つの眼となる可能性が非常に高い。このことから、本研究では局面にBouzy's 5/21 Algorithmを適用し、その中で二番目に高い評価値を最大化するような着手に最大の評価を与える評価関数h(n)を作成した。h(n)の値の範囲は-A $\leqq$ h(n) $\leqq$ Aとした(Aは定数)。また、注目するべき二番目に高い評価値がある交点に対して打つような着手は評価が分からないのでh(n) = 0とした。これを用いて、(2.21)と(2.22)の式を以下のように変更した。

$$\mathbf{pn}(n) = 1 + A - h(n) \tag{3.1}$$
$$\mathbf{dn}(n) = 1 + A + h(n) \tag{3.2}$$

A ∓ h(n) に 1 を加えたのは、末端ノードにおいて最終的な評価値が 0 ではないのに、証明数 (反証数) が最大 (最小) の評価値を得た時に証明数 (反証数) が 0 となって、詰み (不詰み) と判定をしないようにするためである。例えば、h(n) が最大の評価の A となるような着手 (一手前の局面においてその一手で詰ませることができると思われる着手) の場合の証明数は 1・反証数は 1 + 2A となり、h(n) が最小の評価の-A となるような着手 (一手前の局面においてその一手で不詰みににさせることができると思われる着手) の場合の証明数は 1 + 2A・反証数は 1 となる。

一眼問題の 10.sgf(Appendix B 参照) を用いて例を示す。この局面において黒が次の着手として可能性があるのは、a-1・c-1・d-1・f-1・d-2・e-2 の計 6 手である。10.sgf に Bouzy's 5/21 Algorithm を n = 5 で適用すると、下の図 3.1 のようになる。このアルゴリズムを n = 5 で適用した場合、h(n) の最大値 A は、n1 回で一つの交点が与えられる最大値が 4 であることから A = 20 となる。

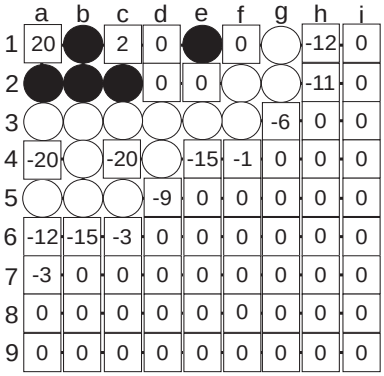

図 3.1: 10.sgf

これを見ると、初めからある眼の場所の a-1 の交点の評価値は 20 で最高となっており、二番目に高い評価値は c-1 の 2 となっている。下の図 3.2〜図 3.7 がそれぞれ可能性のある着手に対して Bouzy's 5/21 Algorithm を適用した後の図である。

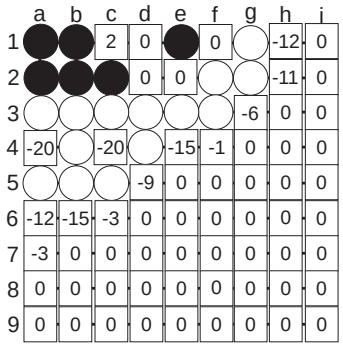

図 3.2: a-1

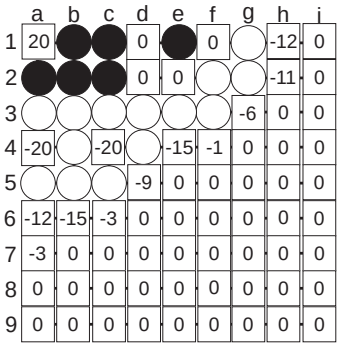

図 3.3: c-1

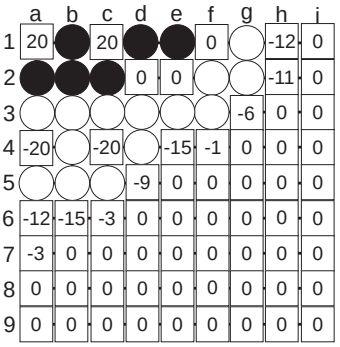

図 3.4: d-1

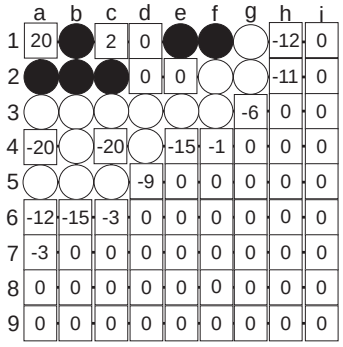

図 3.5: f-1

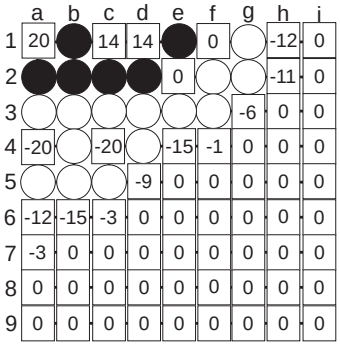

図 3.6: d-2

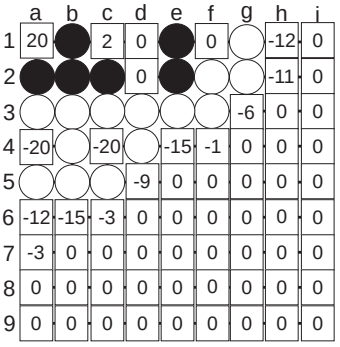

図 3.7: e-2

それぞれの着手に対して Bouzy's 5/21 Algorithm を適用した結果を次の表 3.1 に示す。

表 3.1: h(n) の例

着手	評価値
a-1	0
c-1	0
d-1	18
f-1	0
d-2	12
e-2	0

これを見ると、d-1 が 18 で 6 手の中では最大の評価で、次に d-2 となっている。この 10.sgf の問題の解は d-2 であるので、解である d-2 は優先的に探索することができ、Bouyz's 5/21 Algorithm を使わない場合と比べて無駄な探索を行わずに解を得ることが期待できる。

また、実装には最小距離法で問題点とされている部分についても考慮を行った。図 3.8 を見てみると、ループがないのにもかかわらず、この状態では反証数が足しこまれなくなってしまっている。

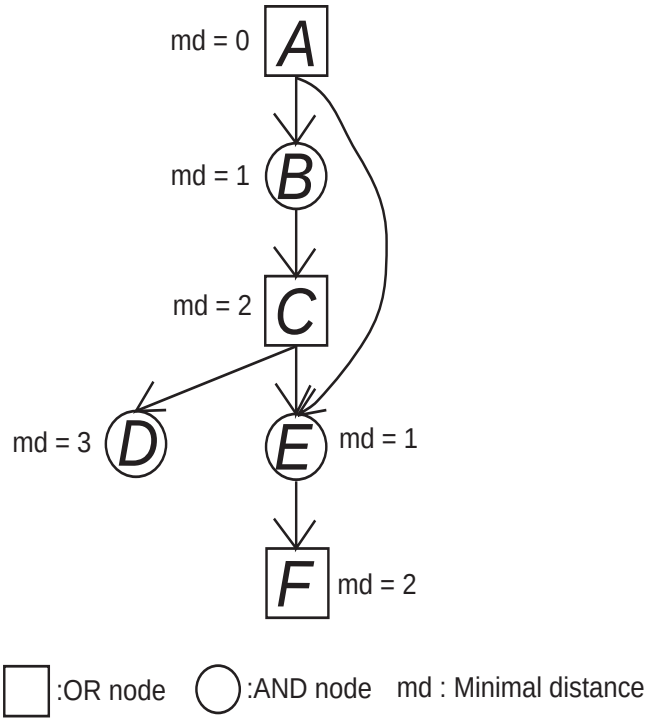

図 3.8: 最小距離法の問題点

C の反証数は C.dn = D.dn + E.dn であるべきだが、D が反証されない限りは、C.dn = D.dn となってしまう。ここで本研究の実装では、探索で当該探索ノードまでの履歴を保存することによりループを発見する関数を作り、それによりループを発見した場合は足しこまず、発見しないでこの状況が起こった場合は足しこむという方法を取ることにより、この問題の回避を行った。

第4章　実験結果

本章では、実験を行った環境と、その結果について述べる。

4.1　実験環境

実験は、Intel Pentium 4 1.90GHz・メモリ 512MB のマシン上で行った。実装には C++言語を用いた。

4.2　実験結果

提案手法との比較検討のために、A:反復深化法に子ノードの優先順位付けとして Bouzy's 5/21 Algorithm を適用したもの、B:Depth-First Proof-Number 探索、C:Depth-First Proof-Number 探索で pn・dn が同じだった場合に Bouzy's 5/21 Algorithm を使って子ノードの優先順位付けをしたものの 3 つを用意し、これと本研究で提案する提案手法:Depth-First Proof-Number+探索に Bouzy's 5/21 Algorithm を適用したものの計 4 つに対して先行研究である [26] に用いられていた一眼問題の一部と、それに訂正を加えたものの計 40 問に対して実験を行った。なお、実験に用いた一眼問題は Appendix B に示す。結果、手法 A の正解数は 32 問で、その他の手法では 40 問全て正解した。手法 A〜手法 C の探索ノード数をそれぞれを縦軸とし、提案手法の探索ノード数を横軸とした散布図を以下の図 4.1〜図 4.3 に示す。また、詳しい実験の結果を以下の表 4.1〜表 4.4 に示す。なお、表中の＊は解けなかった問題を表す。

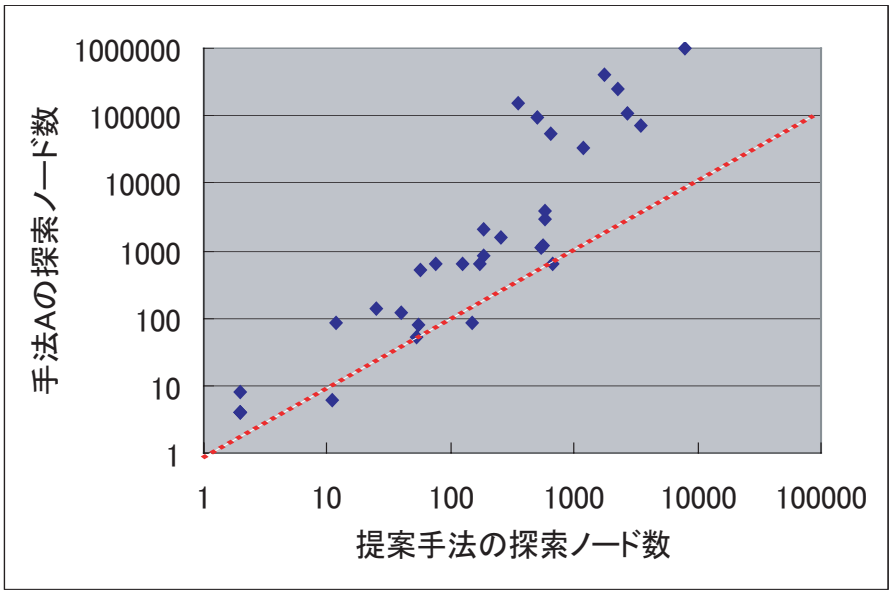

図 4.1: 手法 A と提案手法の探索ノード数の散布図

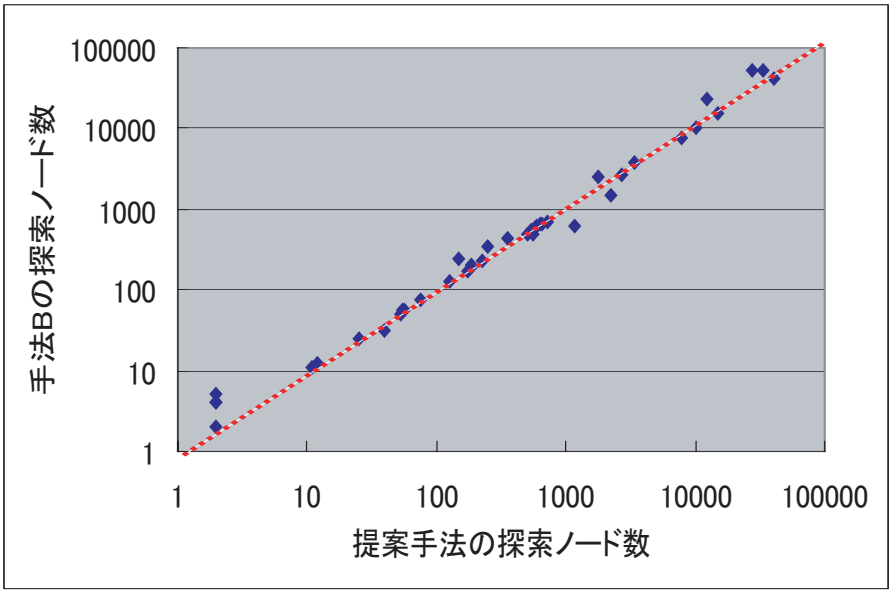

図 4.2: 手法 B と提案手法の探索ノード数の散布図

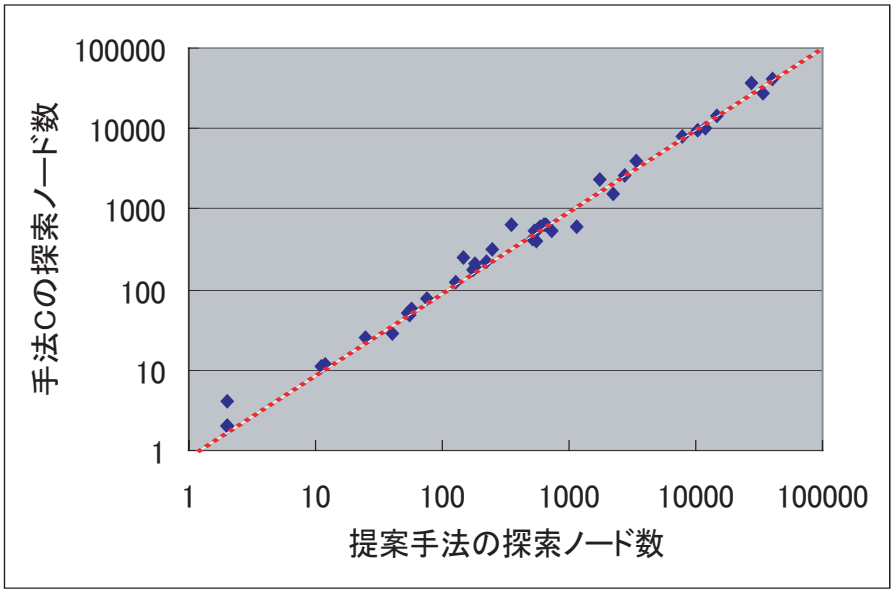

図 4.3: 手法 C と提案手法の探索ノード数の散布図

問題番号	時間 [sec]	探索ノード数
1	0.00	4
2	0.00	4
3	0.00	4
4	0.01	8
5	0.00	6
6	0.05	87
7	0.08	141
8	0.06	119
9	0.02	53
10	0.04	77
11	0.30	521
12	0.36	628
13	0.37	627
14	0.05	83
15	0.38	622
16	0.45	836
17	∗	∗
18	1.18	2114
19	0.89	1562
20	59.23	97299
21	90.50	156422
22	2.35	3747
23	0.67	1230
24	1.78	2932
25	0.36	627
26	33.50	55916
27	0.49	1101
28	∗	∗
29	16.68	32790
30	250.52	402932
31	150.34	249517
32	58.76	108032
33	40.81	70822
34	598.85	998598
35	∗	∗
36	∗	∗
37	∗	∗
38	∗	∗
39	∗	∗
40	∗	∗

表 4.1: 手法 A

問題番号	時間 [sec]	探索ノード数
1	0.00	2
2	0.01	4
3	0.02	5
4	0.03	4
5	0.18	11
6	0.05	12
7	0.16	25
8	0.19	43
9	0.26	51
10	0.19	56
11	0.60	57
12	0.77	76
13	1.32	127
14	1.19	246
15	1.61	175
16	1.38	208
17	2.01	225
18	0.94	188
19	5.35	346
20	7.23	485
21	6.57	433
22	5.09	574
23	2.90	491
24	16.09	606
25	5.95	659
26	7.83	656
27	2.07	540
28	6.54	692
29	3.21	628
30	24.30	2556
31	13.59	1530
32	13.49	2599
33	28.42	3867
34	61.54	7829
35	116.41	10380
36	204.23	23556
37	123.74	14994
38	479.36	51108
39	563.36	42338
40	628.82	51299

表 4.2: 手法 B

問題番号	時間 [sec]	探索ノード数
1	0.01	2
2	0.01	2
3	0.01	2
4	0.05	4
5	0.26	11
6	0.07	12
7	0.24	25
8	0.25	39
9	0.38	52
10	0.24	47
11	0.85	57
12	1.14	76
13	1.86	127
14	1.68	246
15	2.17	175
16	1.84	205
17	2.89	225
18	1.34	188
19	6.72	317
20	8.11	421
21	12.75	634
22	6.79	574
23	3.28	392
24	19.42	590
25	7.73	658
26	10.19	640
27	3.00	539
28	6.83	532
29	4.59	617
30	29.00	2362
31	18.34	1511
32	18.60	2577
33	39.64	4044
34	100.99	7785
35	129.27	9276
36	125.76	10321
37	158.19	14756
38	313.85	26816
39	692.48	40764
40	525.44	36526

表 4.3: 手法 C

問題番号	時間 [sec]	探索ノード数
1	0.02	2
2	0.03	2
3	0.04	2
4	0.06	2
5	0.78	11
6	0.20	12
7	0.68	25
8	1.03	40
9	1.05	53
10	0.79	55
11	2.48	57
12	3.48	76
13	5.08	127
14	2.44	149
15	5.61	175
16	3.88	184
17	9.31	225
18	3.33	184
19	14.85	251
20	26.43	510
21	20.18	351
22	14.12	574
23	10.41	554
24	50.11	590
25	18.86	660
26	26.70	641
27	6.62	532
28	19.91	726
29	17.58	1169
30	50.15	1762
31	54.93	2248
32	40.36	2704
33	71.91	3420
34	177.46	7843
35	313.13	10218
36	313.92	12104
37	301.89	14723
38	744.35	33919
39	1455.7	40018
40	1060.2	27433

表 4.4: 提案手法

第5章　考察

本章では、実験の結果の考察を行う。なお、今回の実験では、碁盤プログラムが完成には遠いため、一手の生成や着手の戻しの処理速度が遅く、計算の時間では正確な比較検討が出来ないことから、検討事項から計算時間を除いて考える。

図 4.1 を見てみると、特に探索ノード数が増えれば増えるほど提案手法の方が良い結果となっている。一方、図 4.2 と図 4.3 を見てみると、どちらも散布図からはあまり変化をみることは出来ない。

次に、手法 A〜C のそれぞれと提案手法で各問題毎に探索ノード数を比較し、提案手法よりも探索ノード数が多かった問題の数・同数だった問題の数・少なかった問題の数を次の表 5.1 に示す。

表 5.1: 探索ノード数から見た問題の数

	手法 A	手法 B	手法 C
多かった問題の数	36	21	12
同数だった問題の数	1	12	16
少なかった問題の数	3	7	12

手法 A を見てみると、提案手法よりも探索ノード数が多かった場合が圧倒的となっており、散布図からみても提案手法の方が優れているということができる。手法 B を見てみると、問題のうち約半分が提案手法よりも探索ノード数が多くなっている。しかし、一方で少なかった問題の数も 4 分の 1 に上り、おおむね提案手法の方が良い結果を出しているが、局面によっては悪い結果が出ることがあるという結果になった。手法 C を見てみると、多かった問題の数と少なかった問題の数は同数であり、提案手法との差は見らず、手法 C とは優劣がほぼないという結果になった。

これらの問題の探索ノードの数から、提案手法の出す結果についていくつかの傾向が見えてきた。探索する領域がある程度まとまった大きさの空点である場合、特に問題 6.24 や 6.26 のように、探索領域の中に受け手の石がいくつかある場合は提案手法が優れた結果を出した。これは、本研究の方針が問題によくはまった場合であると考えられる。しかし、一方で提案手法では上手くいかない場合も見つかった。

上手くいかなかったのは、その問題の解が受け手の守ろうとしている石と離れている場合である。問題 6.34 や 6.37 がこのような場合に当たる。Bouzy's 5/21 Algorithm は、目算や領地の計算として用いられているように、自分の陣地であると見なせるような場所、つまり自分の石がたくさんあるような場所に高い評価値を示す。そのため、石と解が離れていると、その交点に対しては高い評価値は出ず、その場所に対する着手は他の着手より低い評価値となり、余計な部分を探索してしまうことになる。

第6章　結論

本章では、本研究での結果をまとめ、さらに今後の課題について述べる。

6.1　まとめ

本研究は、詰碁を効率的に解くことを目的とし、現在最も成功しているAND/OR木探索であるDepth-First Proof-Number探索をさらに改良したDepth-First Proof-Number+探索に、一定の計算式から局面を評価でき、計算時間も比較的少なくて済むBouzy's 5/21 Algorithmを評価関数として適用して、詰碁の一分野である一眼問題に対して実験を行った。提案手法の比較検討のために、手法A:反復深化法に探索順序制御としてBouzy's 5/21 Algorithmを用いたもの・手法B:Depth-First Proof-Number探索・手法C:Depth-First Proof-Number探索に探索順序制御としてBouzy's 5/21 Algorithmを用いたものの三つを用意した。その結果、提案手法は手法Aよりも優れた結果を出すことができた。手法Bと比べるとおおむね良い結果を出したが局面によっては悪い結果も出していた。手法Cとはほぼ優劣がつかないという結果になった。

6.2　今後の課題

今後、まずは碁盤プログラムの改善をしなければならない。その上で、今回プログラムの速度が出なかったため検討できなかった計算時間についても再度検討していきたい。

次に、今回Bouzy's 5/21 Algorithmでは Dilationを5回、Erosionを21回行ったが、この計算の回数を変えて実験を行うことを考えている。前述したように、Erosionの回数はDilationの回数から計算されているが、これは経験則に基づいたものであるので、この計算方法を崩して色々な回数で実験を行い、Dilation5回とErosion21回よりも良い結果が出せる回数の組合せを探してみたい。また、Erosionを行わず、Dilationだけを使った評価関数というのも考えてみたい。

次に、Bouzy's 5/21 Algorithmの評価関数への適用の仕方の変更も考えてみたい。本研究では評価する局面と一手前の局面にBouzy's 5/21 Algorithmを適用し、一手前の局面の全ての交点の中で二番目に高い評価値を持つ交点の評価値を最大化するような着手に対して最大の評価を与える評価関数を用いて実験を行った。今

度は、局面に対して Bouzy's 5/21 Algorithm を適用した後、探索可能領域の全ての交点の評価値の和を最大化するような着手に最大の評価を与えるという評価関数を作ってみたいと考えている。下の図の問題 (6.41) を例に、この評価関数について考える。

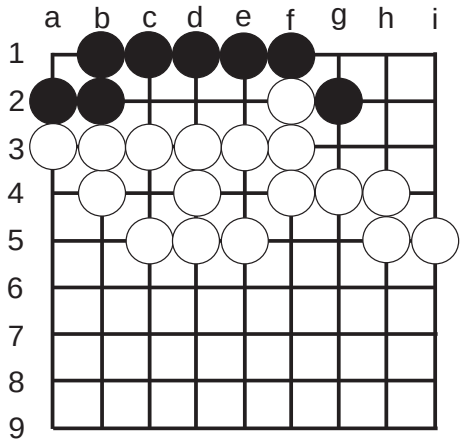

図 6.1: 36.sgf

この問題の解は i-2 であり、解が受け手の石と離れている提案手法の上手くいかなかった場合である。ここで、評価値の和を見る評価関数を考える。いくつかの Dilation と Erosion 組合せを試した表に次に示す。

表 6.1: 評価値の和を見る評価関数

	a-1	g-1	h-1	i-1	c-2	d-2	e-2	h-2	i-2	g-3	h-3	i-3	i-4
Dil:1	0	3	8	6	4	4	4	6	7	5	7	8	7
Dil:2	1	7	15	16	9	9	9	16	17	12	19	21	17
Dil:2, Ero:1	0	7	15	16	8	8	8	16	17	10	17	20	14
Dil:2, Ero:2	0	6	14	14	8	8	8	16	17	9	16	19	12
Dil:2, Ero:3	0	5	14	14	8	8	8	16	16	9	13	18	11
Dil:3	5	13	22	26	17	17	17	28	29	23	36	39	30
Dil:4	10	19	29	36	26	26	26	40	41	35	55	56	45
Dil:5	15	25	36	46	35	35	35	52	53	47	74	75	60

この表で、Dilation2 回・Erosion2 回行った場合を見てみると、この場合では解が 2 番目に高い評価値を得ており、他の着手に比べて優先的に探索されることが分かる。これにより、探索の結果も良くなることが期待できる。この方法を使え

ば、提案手法では上手くいかなかった場合でも上手くいく可能性が見込めると考えている。

参考文献

[1] 松原仁, 竹内郁雄 編著:ゲームプログラミング, 共立出版,1998.

[2] ゲーム木の探索問題: `http://ray.sakura.ne.jp/search_problem/index.html`

[3] 白井良明:人工知能の理論 (増補), コロナ社,2003.

[4] Bruno Bouzy and Tristan Cazenave. *Computer Go: an AI Oriented Survey.* June 2001.

[5] 清愼一, 川嶋俊明:探索プログラムによる四路盤囲碁の解,「ゲーム情報学」研究報告,2000.

[6] Dev Purkayastha:*A Survey and Synthesis of Recent Advances in Go-playing Agents.* January 13,2003.

[7] Martin Muller:*Not Like Other Games - Why Tree Search in Go is Different*,In Proceedings of Fifth Joint Conference on Information Sciences (JCIS 2000), pages 974-977, 2000.

[8] Akihiro Kishimoto:*TRANSPOSITION TABLE DRIVEN SCHEDULING FOR TWO-PLAYER GAMES*, M.Sc.Thesis,University of Alberta ,January 2002.

[9] GNU Project: *GNU Go*, `http://www.gnu.org/software/gnugo/gnugo.html`

[10] 河龍一:HARUKA のプログラム構造,CGF Journal Vol.5.

[11] 清愼一, 川嶋俊明:記憶に基づく推論を使った囲碁プログラム「勝也」の試作,the 3rd Game Programming Workshop, pp.115-122, 1996.

[12] 清愼一, 川嶋俊明:「局所パターン」知識主導型の囲碁プログラムの試み,the 1st Game Programming Workshop, pp.97-104, 1994.

[13] Markus Enzenberger:*The Integration of A Priori Knowledge into a Go Playing Neural Network*,1996.

[14] 滝沢武信: 数理ゲーム理論の囲碁への応用, ゲーム情報学 1-5 pages 39-46, June 24, 1999.

[15] Elwyn Berlekamp:*"Idempotents Among Partisan Games"*,More Games of No chance,Cambridge University Press, pp.3-23,2002.

[16] 中村貞吾:組み合わせゲーム理論と囲碁, CGF Journal Vol.5. January 2003.

[17] 中村貞吾: 組合せゲーム理論を用いた囲碁の攻め合いの解析, ゲーム情報学 GI-9-5, March 2003.

[18] 脊尾昌宏: 共謀数を用いた詰将棋の解法, コンピュータ将棋の進歩 2, pp.1-28, 共立出版, 1998.

[19] Seal Software: アルゴリズム解説, `http://fujitake.dip.jp/sealsoft/thell/algorithm.html\#transpositiontable`

[20] Ayumu Nagai: A NEW DEPTH-FIRST-SEARCH ALGORITHM FOR AND/OR TREES, A Master Thesis, the University of Tokyo, February 1999.

[21] 長井歩: df-pn アルゴリズムと詰将棋を解くプログラムへの応用, アマ 4 段を超える -コンピュータ将棋の進歩 4, pp.96-114, 共立出版, 2003.

[22] A. Kishimoto, M. Muller: Df-pn in Go: Application to the one-eye problem. In *Advances in Computer Games. Many Games, Many Challenges*, pages 125-141. Kluwer Academic Publishers, 2003.

[23] Akihiro Kishimoto: A Correct Algorithm to Prove No-Mate Positions in Shogi, 9th Game Programming Workshop in Japan (GPW2004), pages 1-8, November, 2004.

[24] Akihiro Kishimoto: About the Tsume-Shogi Solver of ISshogi, at Computer Shogi Association Bulletin Vol. 16, 2004.

[25] Akihiro Kishimoto and Martin Muller: A General Solution to the Graph History Interaction Problem, Nineteenth National Conference on Artificial Intelligence (AAAI'04), pages 644-649, AAAI Press, 2004.

[26] Akihiro Kishimoto: CORRECT AND EFFICIENT SEARCH ALGORITHMS IN THE PRESENCE OF REPETITIONS, A Doctor Thesis, the University of Alberta, Spring 2005.

[27] Ayumu Nagai, Hiroshi Imai: Applicaqtion of df-pn+ to Othello Endgames. In *The 5th Game Programming Workshop (GPW'99)*, pages 16-23, 1999.

[28] Brono Bouzy: Mathematical morphology applied to computer go. IJPRAI, vol 17, no.2, March 2003.

[29] Brono Bouzy: [coumputer-go] Bouzy/Zobrist algorithms, `http://computer-go.org/pipermail/computer-go/2004-June/000593.html`

[30] Thomas Wolf: *GoTools*, `http://alpha.qmw.ac.uk/~ugah006/gotools/t.wolf.html`

[31] Thomas Wolf: Investigating tsumego problems with "RisiKo". In D.N.L. Levy and D.F. Beal, editors, *Heuristic Programming in Artificial Intelligence2*, pages 153-160. Ellis Horwood, 1991.

[32] Thomas Wolf: About problems in generalizing a tsumego program to open positions. July 12, 1996.

[33] Thomas Wolf: Foward pruning and other heuristic search techniques in tsume go. *Information Sciences*, 122(1): 59-76, 2000.

[34] Thomas Wolf, Matthew Pratola: Optimizing GoTools' search heuristics using Genetic Algorithms. ICGA Journal March 2003.

発表文献

1. 石井 宏和, 横山 大作, 近山 隆. Bouzy's 5/21 algorithm を用いた Df-pn+探索の詰碁への適用. 情報処理学会第 15 回ゲーム情報学研究会, 2006. (発表予定)

謝辞

本研究を進めるにあたり、近山隆教授・田浦健次朗助教授・遠藤敏夫特任助手・横山大作助手・鈴木彦文助手 (現・信州大学総合情報処理センター助教授) には温かいご指導・ご鞭撻を頂きました。特に近山教授には、研究の岐路で貴重な助言を頂きました。また横山助手には、研究はもとよりプログラミングの基礎から丁寧にご指導を頂きました。心より感謝申し上げます。

また、研究を進めていくにあたり様々な助言や協力をいただきました諸先輩方々に深く感謝します。特に、同室で博士の三輪誠さんと同輩の今竹春彦君の 2 人には大変お世話になりました。重ねて感謝申し上げます。

院から情報系に移り、ほとんど何も分からない状態から出発した自分を暖かく迎えてくださり、楽しく 2 年間の研究生活を送ることができたのも、一重に研究室のみなさんのおかげです。みさなんがいなければ、ここまで来ることはできませんでした。本当に、本当にありがとうございました。

平成 18 年 2 月 3 日

Appendix A

囲碁のルール

囲碁とは、縦横最大 19 本の線の交点に、黒と白の石を交互に置いていき、最終的に自分の色の石で囲った交点の多いほうが勝ちとするゲームである。ここで囲った交点のことを地と呼ぶ。また、盤上の交点 1 つを 1 目と数え、囲碁では先手を持つ黒が有利とされている為、後手側にコミと呼ばれる調整のための 6 目半の地がプラスされる。一局の流れは大まかに序盤 (布石)・ 中盤・ 終盤 (ヨセ) と表現される。

囲碁では重要なルールが三つ存在する。まず一つ目は、相手の石を自分の石で上下左右 4 方向に囲めば、その石を取り上げることができるというルールである。4 方向に盤の端を含む場合は、そこを除いて考える。辺では 3 方向・ 角では 2 方向を埋めるだけで石を取れることになる。ここで取り上げた石は捕虜として、終局後に相手の地に埋めて計算する。石を取るのに必要な最低限の数をダメと呼ぶ。また、石を取るのに必要な最低限の手数のこともダメと呼ぶ。さらに、下図 6.2 の白石ように、あと一つ相手の石を置かれただけで相手に石を取られてしまうような状態のことをアタリと呼ぶ。一石で相手の石の二つ以上のアタリがかかることもあり、その状態のことを両アタリと呼ぶ。

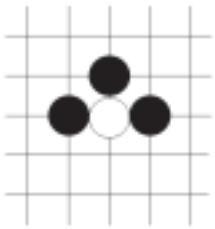

図 6.2: アタリ

囲碁では、石のない場所に石を打っていくが、その石を打ったときに、その石のダメが 0 になってしまい、打った瞬間に全て取られてしまうような交点には

打つことができない。これが二つ目の重要なルールである。ただし、そこに打つことによって、隣接する相手の石のダメを 0 にするような場合にのみ、その交点に打つことができる。

下図 6.3 を見ると、真ん中にある黒石は白石からアタリの状態にされている。そこで白石が黒石の右側に打つと、今度は打った白石が逆に黒石にアタリの状態にされてしまっている。このように同型で、お互いに石の取り合いが限りなく続き、対局が終了できなくなってしまう形をコウという。このような状況になった場合は、一手ほかのところへ打たなければならない。これが三つ目の重要なルールである。コウがあるために他のところへ打つことをコウダテという。また、コウが三つ以上出てきてしまった場合は、勝負が着かなくなってしまうので、無勝負となる。

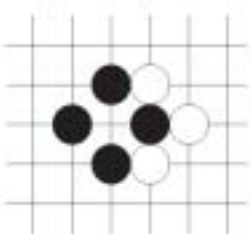

図 6.3: 劫

以下に、ルール以外で重要になってくる事柄を紹介する。

眼

囲碁の戦略を考える上で、最も重要な概念が眼である。この眼は石の生き死にを考える際に用いる。眼とは、石で囲った部分のことを指す。眼が一つしかできなかったと場合は、その眼を持った石は相手に必ず取られてしまう。しかし、下図 6.4 の黒石のように石が眼を二つ持つことができた場合、相手はそのどちらの空点に置いたとしてもダメを 0 にすることができない為に石が取れず、終局時に自分の地となる。二眼を持つ石をどれだけ持てるかが、勝敗を左右する。

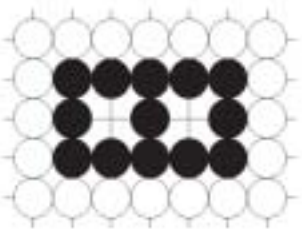

図 6.4: 二眼

模様

囲碁では、布石においては、大抵の場合自分の地を確保するのに有利であることから、盤の中央よりも、星と呼ばれる角から縦横にそれぞれ四つ移動した点の周辺から打ち始める。その後、自分の地が多くなるように大きく打っていく。その過程でできる、まだ完全に地とはなっていないが、影響力の大きいところを模様と呼ぶ。

セキ

図 6.5 のように、双方の石が囲いあっていて、二眼を持っていないが双方とも生きているような形をセキと呼ぶ。終局時はここを無視して地を計算する。死活問題では生きとする。

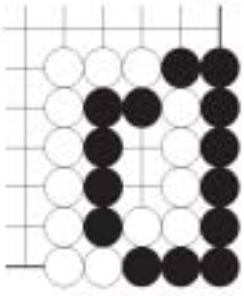

図 6.5: セキ

Appendix B

一眼問題

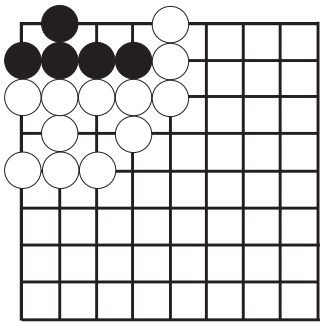

図 6.6: 1.sgf

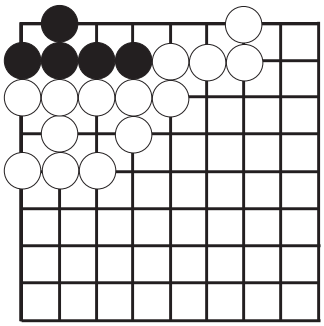

図 6.7: 2.sgf

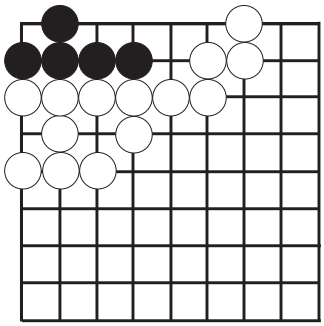

図 6.8: 3.sgf

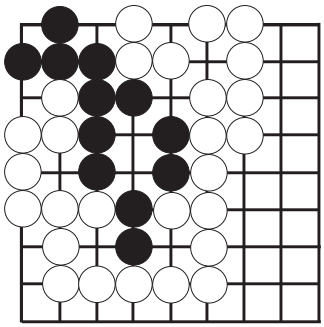

図 6.9: 4.sgf

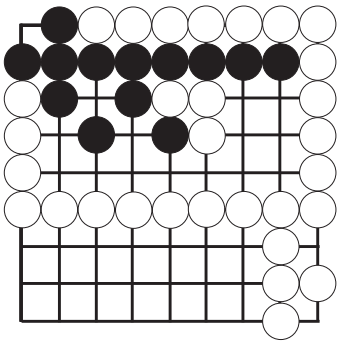

図 6.10: 5.sgf

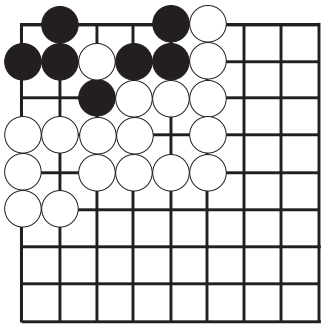

図 6.11: 6.sgf

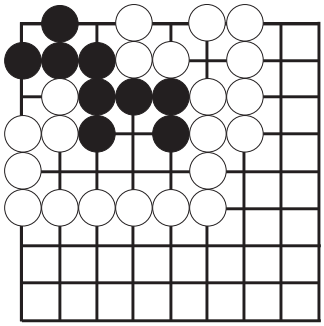

図 6.12: 7.sgf

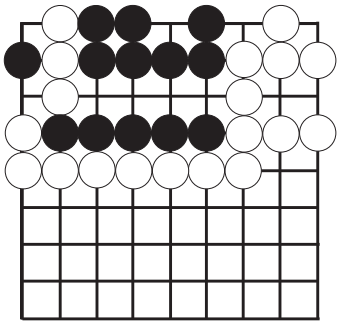

図 6.13: 8.sgf

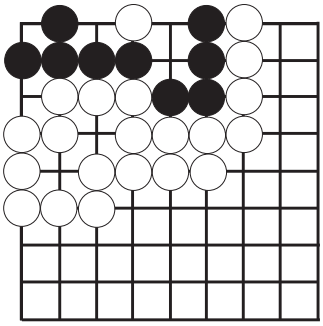

図 6.14: 9.sgf

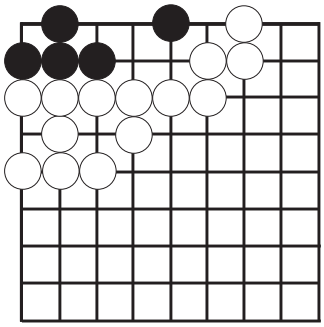

図 6.15: 10.sgf

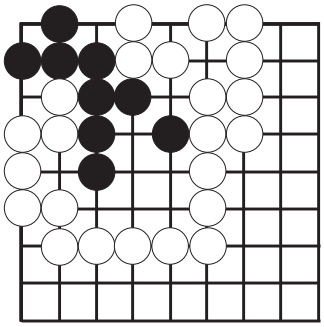

図 6.16: 11.sgf

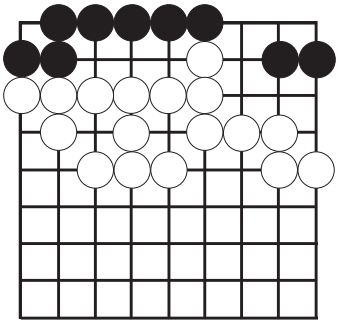

図 6.17: 12.sgf

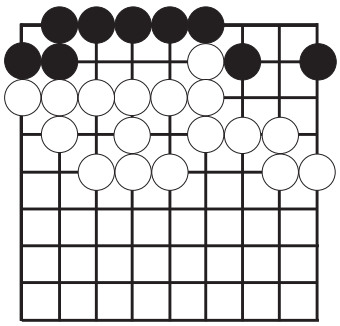

図 6.18: 13.sgf

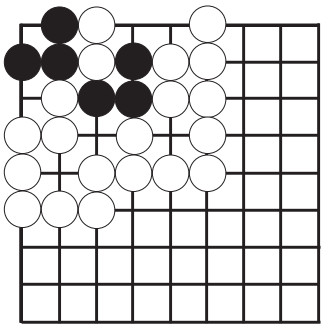

図 6.19: 14.sgf

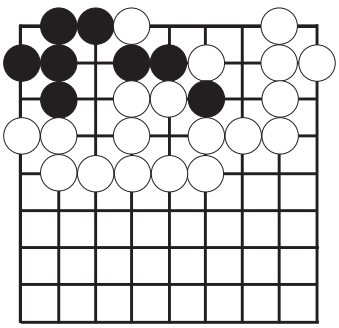

図 6.20: 15.sgf

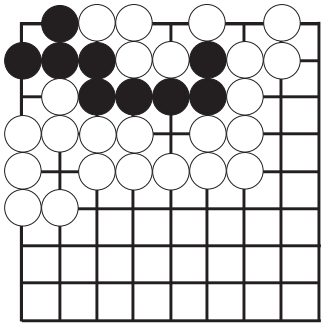

図 6.21: 16.sgf

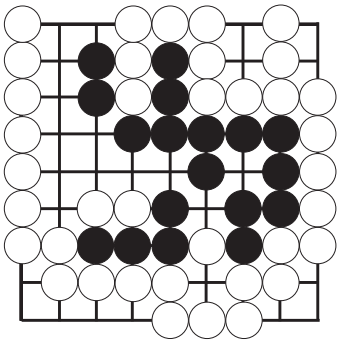

図 6.22: 17.sgf

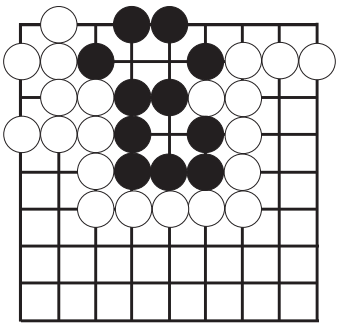

図 6.23: 18.sgf

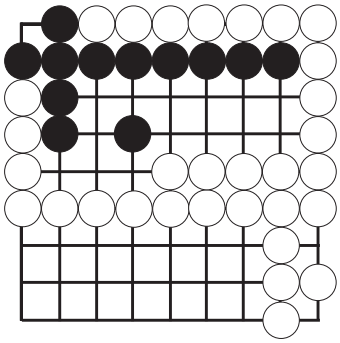

図 6.24: 19.sgf

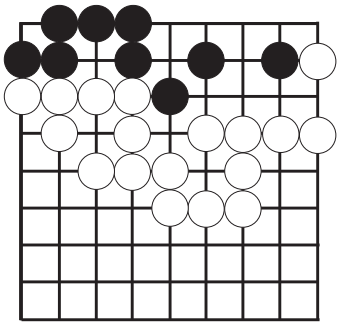

図 6.25: 20.sgf

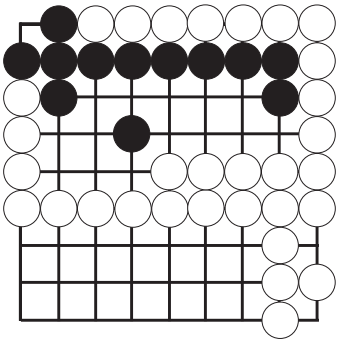

図 6.26: 21.sgf

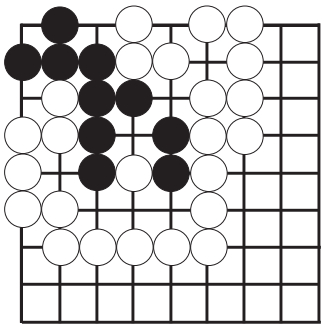

図 6.27: 22.sgf

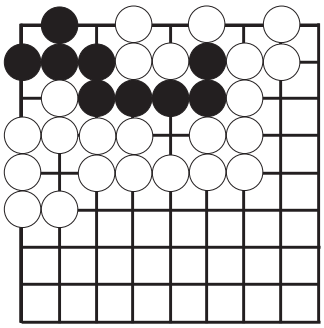

図 6.28: 23.sgf

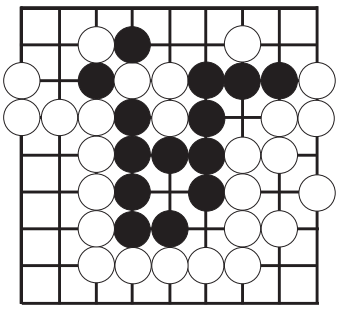

図 6.29: 24.sgf

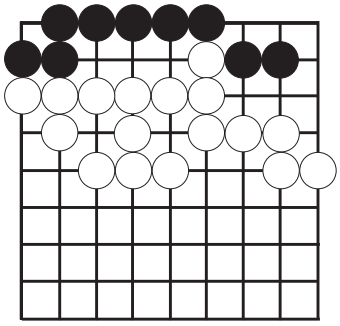

図 6.30: 25.sgf

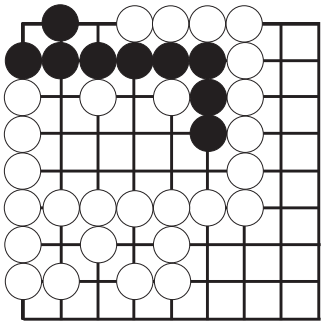

図 6.31: 26.sgf

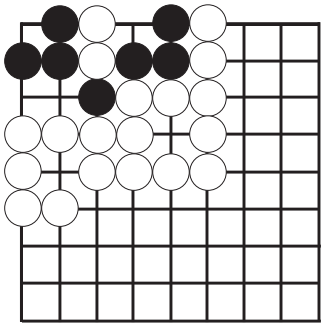

図 6.32: 27.sgf

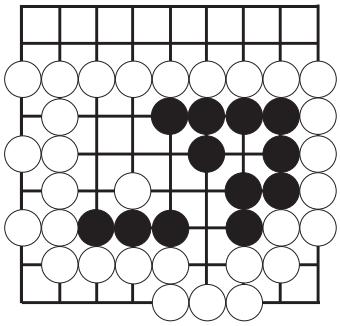

図 6.33: 28.sgf

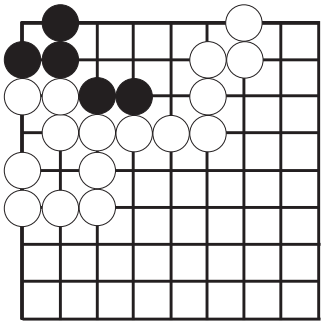

図 6.34: 29.sgf

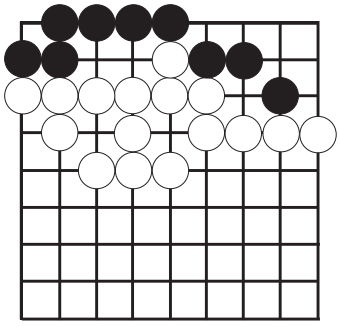

図 6.35: 30.sgf

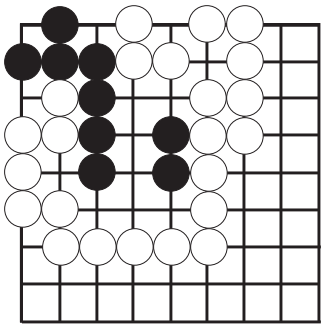

図 6.36: 31.sgf

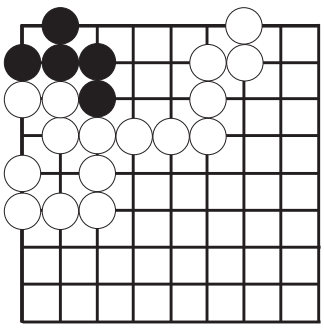

図 6.37: 32.sgf

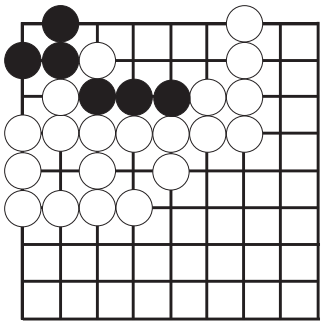

図 6.38: 33.sgf

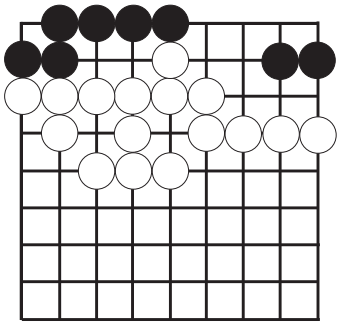

図 6.39: 34.sgf

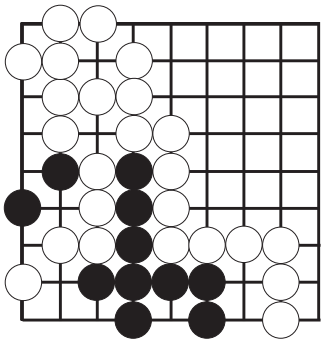

図 6.40: 35.sgf

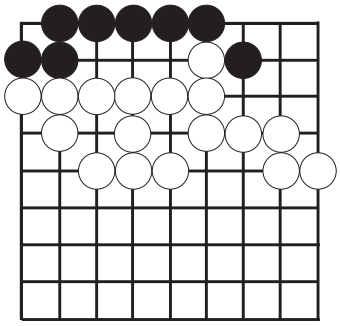

図 6.41: 36.sgf

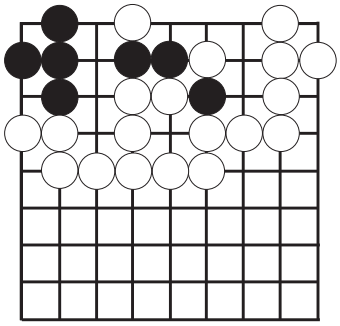

図 6.42: 37.sgf

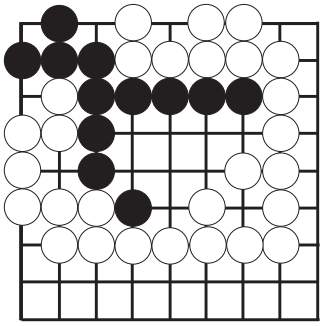

図 6.43: 38.sgf

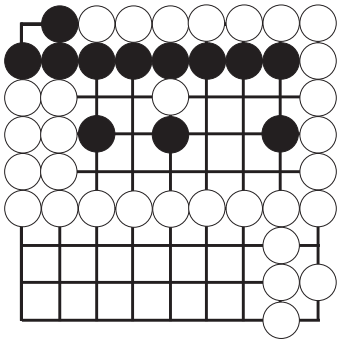

図 6.44: 39.sgf

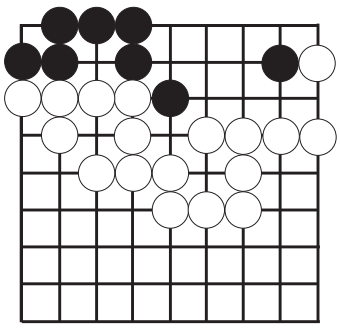

図 6.45: 40.sgf