3 将棋プログラムの現状と未来

鶴岡 慶雅
科学技術振興機構
tsuruoka@is.s.u-tokyo.ac.jp

はじめに

コンピュータ将棋の実力はプロ棋士のレベルに近づきつつある．その理由の1つは，ハードウェアの進歩により探索を高速に実行できるようになったことにあるが，ソフトウェアの面での進歩も大きい．本稿では，第15回世界コンピュータ将棋選手権で優勝した将棋プログラム「激指（げきさし）」の探索手法を中心にして，現在トップレベルにある将棋プログラムの中身，さらにコンピュータが今後名人のレベルに到達することが可能なのか，そのためには何が必要なのかについて解説する．

ゲーム木探索

コンピュータはどのようにして次の一手を「考えて」いるのだろうか？　コンピュータの思考内容は，**図-1**のように木の形で表現することができる．一番上にあるノードがルートノードと呼ばれ，現在の局面を表す．それより下にあるノードが，これから起こり得る局面，つまりコンピュータが「頭の中で」考えている局面である．ノード間を結ぶエッジは，指し手に対応し，その指し手によってある局面から別の局面に変化することを示す．

図-1は2手先まで読むとした場合の探索木の例である．コンピュータは，末端の局面（2手先の局面）で，その局面が先手にとって（あるいは後手にとって）どれくらい有利かを，評価関数によって数値化する．先手が有利であればプラス，後手が有利であればマイナスの数値をとる．ここで，お互いのプレイヤーが，各局面において自分にとって最も有利な指し手を選択すると仮定すると，末端ノードからルートの方向へ逆算していくことによって，探索木中のすべてのノードについて評価点を付与することができる（Min-Max法，**図-2**）．そして，コンピュータはルートノードの直下のノードのうち，最も高い点数のノードにつながる指し手を選択すればよい．

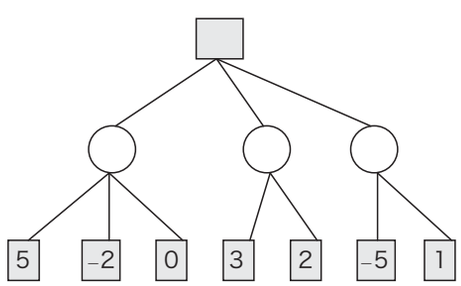

図-1　探索木の例．四角が先手番の局面，丸が後手番の局面を表す．末端ノードについているのは評価関数によって計算されたその局面の評価値．

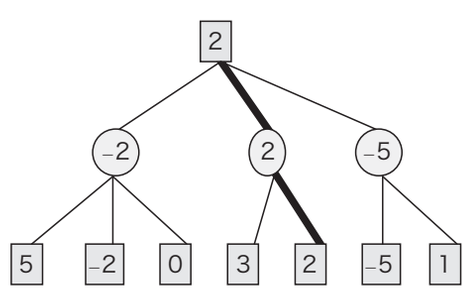

図-2　指し手の決定方法（Min-Max法）．後手番の局面では子供ノードのうち，最も評価値が小さいノード（後手によって有利な局面），先手番の局面では子供ノードのうち，最も評価値が大きいノードを選ぶ．

ゲーム木探索の基本的な仕組みはこれだけである．これに加えて，無駄になるノード展開を排除するための枝刈り手法（$\alpha\beta$法）と，同一局面の探索結果を再利用するためのトランスポジションテーブルを実装してしまえば，とりあえずはまともに将棋が指せるプログラムは作れるといってよい．

では何が強いプログラムとそうでないプログラムを分けているのだろうか？　難しい点は2つある．1つは探索範囲の制御である．図-1では2手先までノードを展開するとしたが，すべての合法手をある一定の深さまで探索するという方法（全幅探索）では，探索に時間がかかりすぎて強い将棋プログラムは作れない．どのような展開をどこまで読むのか，ということが将棋プログラムでは決定的に重要である．

もう1つの大事な要素は，評価関数の精度である．図-1の末端ノードにつけられている点数が，先手・あるいは後手の有利不利を正確に数値化したものでなければならない．

つまり将棋プログラムでは，「探索範囲の制御」と「評価関数の設計」が強さの大きな鍵となる．

探索範囲の制御法

探索範囲の制御における重要な問題の1つは，各ノードでどういう指し手を生成するかという問題である．コンピュータチェスでは，全幅探索といって，その局面での合法手をすべて生成するという，いわば力任せの手法が比較的有効であり，それに，singular extension[1)]，☆1 や null-move forward pruning[2)]，☆2，futility pruning[3)]，☆3 などの，経験的知識とはある程度独立な枝刈り手法を組み合わせることで，かなり強力なプログラムを作ることができる．

ところが将棋の場合，持ち駒が使えるというルールによって，チェスよりもはるかに中終盤が複雑になっている．合法手の数は，序盤では数十程度であるが，終盤になると二百を超えることも珍しくない．探索にかかる時間は分岐数の指数オーダーになるため，終盤近くになると，全幅探索による手法では現実的な時間では10手先まで読むことも困難であり，高段者並みの指し手を実現することは難しい．

指し手のタイプ	遷移確率
駒得で取り返す	58〜89%
駒得で駒を取る	16〜42%
駒が逃げる	12〜69%
駒得で王手をかける	43%
飛車が成る	21%
角が成る	20%
桂馬が成る	20%
⋮	⋮

表-1　指し手の種類と遷移確率

そのため将棋プログラムでは，経験的知識（ヒューリスティクス）に基づいて，可能な指し手の中から有望そうな指し手に絞って探索を行うという手法が広く利用されている．代表的なヒューリスティクスとしては，「末端に近いところでは駒損☆4 する手を読まない」とか，「末端に近づくほど生成する指し手の数を減らす」などがある．実際の強いプログラムでは，詳細なヒューリスティクスを数多く組み合わせることで候補手の絞り込みを実現している．

また候補手を減らすだけではなく，特定の場面では探索を延長する（そのノード以下の探索の残り深さを増やす）ことも有効である．たとえば「王手をされているときは探索を1手延長する」というのは有力なヒューリスティクスである[4)]．また，将棋の手筋としてよく出現する手順を3手一組のパターンで記述しておき，それらが探索木中で実現しそうなときは探索を延長するという手法が強豪プログラムの1つであるIS将棋で利用されている[5)]．

上記の手法では，基本的に深さを基準とした探索範囲の制御になっているが，激指では少し異なった手法を利用している[6)]．激指では，ノードの深さの代わりに，ノードごとに計算される「実現確率」を基準にして探索範囲を決定する．あらかじめ，大量のプロ棋士の棋譜から，どういうタイプの指し手がどのぐらいの確率で指されるのかを集計しておき（**表-1**），それをノードからノードへの「遷移確率」として利用する．ノードの実現確率は，親ノードの実現確率から再帰的に，

（親ノードの実現確率）×（遷移確率）＝
（子ノードの実現確率）

として計算することができる（ルートノードの実現確率は1）（**図-3**）．そして，深さの代わりに，実現確率を探索範囲の基準とする，つまり，ノードの実現確率がある値を下回った時点で末端ノードとするわけである．このようにすることで，実際に実現する可能性の高い局面を中心に探索するという，人間の思考方法に近い探索範囲

☆1 子ノードのうち他と比較して特に評価値が高いノードの探索を延長する手法．
☆2 ノードの評価値の下限を，一手パス＋浅い探索で見積もって枝刈りする手法．
☆3 末端ノードに近いノードで，評価関数を直接呼び出してそのノードの評価値の上限を見積もって枝刈りする手法．
☆4 動かした駒が相手にただで取られてしまう手や，大駒を相手の小駒と交換してしまうような手．

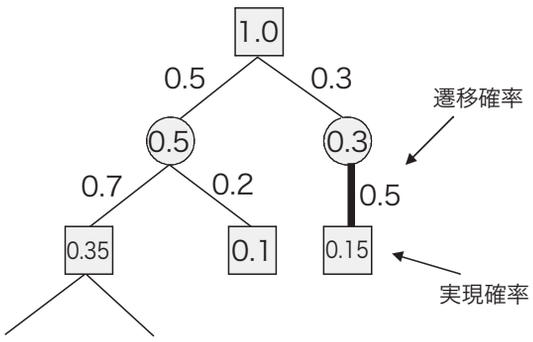

図-3　実現確率と遷移確率

の制御を目指している．

■指し手の並び替え

Min-Max 法を利用した探索では，ある条件を満たしている部分は探索を省略することができる．たとえば図-1 で左から右へ探索を行っている場合，右端から 2 番目のノードの評価値が −5 だと分かった時点で，右端のノードの値がどんな値であれルート局面での指し手選択に影響がなくなる．つまり右端のノードは実は探索しても意味がないことが分かる．$\alpha\beta$ 法ではこれを一般化し，探索木の中で，探索を省略してもかまわない部分を枝刈りして探索効率を大幅に向上させることができる．$\alpha\beta$ 法によって枝刈りを行う場合，各ノードにおいてどういう指し手が探索されるかだけなく，どの順番で探索されるか，ということが探索の全体的な効率に大きくかかわってくる．$\alpha\beta$ 法では，それぞれのノードで（Min-Max の意味で）最善の手を最初に探索した場合に枝刈り効率が最も高くなることから，効果的な枝刈りを行うためには，どういう手が良い手なのかを実際に探索する前に見積もって，良さそうな手から順に探索する必要がある．

指し手に関するヒューリスティクスは並び替えにおいても有用である．たとえば，「直前に動いた駒を取る手」や，「直前の手で狙われた駒を逃げる手」などを，他の手より先に探索すると探索の効率が上がることが知られている [5]．

また，浅い探索の結果を利用する方法も広く利用されている．つまり，あるノードにおいて，残り探索深さが d であるときに，残り深さを d − 1 に設定して探索した結果得られた最善手を最初に探索するのである．もちろん，浅い探索をするためのコストは余計にかかることになるが，それよりも枝刈りの効率が上がることによる効果が大きいため，多くのプログラムではこの方法を採用している．

キラー手といって，兄弟ノードで最善手とされた手を利用する手法も効果的である．激指では，兄弟ノードのうち，親ノードでパスをした場合の兄弟ノードの最善手をキラー手として他の手より優先して探索する．つまり，兄弟ノードは，現在の局面と似たような局面であるため，最善手が同一であることが多いという性質を利用している．

さらに激指では，兄弟ノード以外での類似局面での最善手も利用することを試みている．一般に，コンピュータで類似局面を検出することは簡単ではない．局面の一部をキーにしてハッシュテーブルを構成する方法も考えられるが，どの部分をキーにするべきかが状況によって変わってしまうために実現することは難しい．そこで，盤面をキーにするのではなく，手順をハッシュキーとして利用する．すなわち，ある局面で最善手が得られたら，直前の手をキーとして最善手をハッシュテーブルに保存する．そうするとまったく別な局面でも直前の手が同一であれば，その最善手を使いまわすことができる．ある指し手に対する最善の応手というのが，別な局面においても最善の応手になっていることが多いという性質を利用しているといえる．

人間がどのようにして頭の中で探索を行っているかは分からないが，類似した局面の探索結果をかなり有効に再利用していることは間違いない．将棋プログラムにおいて，類似局面での探索結果の利用は，今後追求すべき課題のひとつだと思われる．

Mix→Minに直しました

評価関数

もしコンピュータの性能が十分に高く，すべての局面でゲームの終了状態まで探索することができるのであれば，評価関数は，先手か後手どちらが勝ったかの 2 値を返すだけでよい．また逆に 1 手しか先読みができないとしたら，たとえば，王手飛車取りをかけられて飛車がただで取られてしまう可能性とか，大駒が追い詰められて捕獲される可能性などを評価関数の中に織り込んで評価しなくてはいけなくなるため，非常に複雑な評価関数を設計する必要があるだろう．

このことから分かるように，評価関数の設計は探索と不可分な関係にある．つまり，ある特定の評価関数を取り上げて，これが「正しい」評価関数である，と主張することは無意味である．したがって，これから述べる評価関数の各評価要素は，あくまでも現在の将棋プログラムの探索手法とコンピュータの性能において，それなりに有効な評価要素は何か，を示していることに留意する必要がある．

今までのところ経験的には，最も重要な評価項目は，駒の損得であることが分かっている．**表-2** に激指にお

駒種	価値
王	∞
飛	950
角	800
金	600
銀	550
桂	400
香	400
歩	100

駒種	価値
竜	1300
馬	1150
成銀	600
成桂	600
成香	600
と	600

表-2　激指の評価関数における駒の価値

けるそれぞれの駒の価値を示す．金であれば歩の 6 倍の価値，飛車であれば歩の約 10 倍の価値というわけである．駒の損得に関する評価値は，先手の駒についてこれらの値を合計し，それから後手の駒について合計したものを引くことによって算出することができる．

面白いのは，駒の価値が駒のききの数におおむね比例していることである．将棋が，「盤面を支配する」ゲームだと思えば，駒の価値がそのききの数に比例するのも自然なことなのかもしれない．

終盤になってくると，駒の損得だけでなく駒の働きを評価することが重要になってくる．終盤で，端の桂馬や香車を取りにいって負けるというのは，駒の価値だけしか評価していない場合の典型的なコンピュータの負けパターンである．

金や銀をはじめとする動きの小さい駒の働きを評価する手法に関しては，YSS で提案されている方法[4]が広く使われている．すなわち，自玉あるいは相手玉に近い駒は高く評価し，自玉からも相手玉からも遠い駒は低く評価する．

大駒は移動力が大きいので，現時点の位置よりも，その駒の自由度や，ききが相手玉の周辺に届いているか，などを評価する．

駒の働きを評価するうえで難しいのは，いつ終盤に入ったのかを検出することである．以前は，終盤であるかどうかを判定するルーチンによって，それぞれの局面が終盤であるかどうかを判定し，終盤であるならば駒の働きを評価するといった方法が用いられていた．ところがこのように離散的に判定してしまうと，特に終盤の入り口あたりで問題が起こる．ひとつには，評価値の整合性の問題で，終盤だと判断された局面と，そうではない局面が，探索木中に混在している場合，評価値の整合性（順序関係）を正しく保つことが難しくなる．それを避けるために，ルート局面のみで終盤であるかどうかを判定し，探索木のすべてのノードでその判定結果に従うようにすることも可能であるが，そうすると今度は，ルート局面では終盤ではなかったのに，探索木の深い部分では終盤になるような場合に，駒の働きを無視したぬるい指し手を選択してしまうといったことが起きる．

激指ではそのような問題を避けるために，序盤･中盤･終盤であるかの判定は 1 つの連続的な数値で行うようにしている．駒がどれくらい敵陣に近づいているか，という指標を足し合わせて，盤面全体の進行度を計算する．このように，局面の進行度が連続的な数値で表現されていれば，駒の損得と駒の働きの評価のバランスを連続的に変化させることが可能になるために，評価関数の整合性を保つことが簡単になる．

評価関数ではこのほかに，将棋に関するさまざまな知識が表現されている．たとえば，「端に桂馬がはねた場合は減点」とか「歩が位をとっている場合は加点」といった具合である．これらの評価要素は，1 つ 1 つの評価点としては小さいが数多く存在し，特に序盤の差し回しは，これらの細かな評価要素に支えられている．

評価関数の設計は，開発者の勘と経験によるところが大きいが，多少システマティックに評価関数を改良する手段がないわけではない．1 つの方法としては，プログラムの対戦中の評価値の遷移を観察しておき，負けた棋譜のなかから，自分の側が途中までは圧倒的に有利だと思っていたのにもかかわらず負けた，という棋譜をピックアップする．多くの場合，そのような将棋では評価関数が良くなかったがために自分側に有利な評価をしていることが多く，どの評価項目のパラメータがまずかったのか，あるいは，どういう評価項目を追加しなくてはならないのかが見えてくる．

詰め将棋

詰みの発見はそのまま勝ちにつながるため，詰み探索ルーチンの性能は終盤において非常に重要である．

詰み探索アルゴリズムの最も基本的な手法は，単純な反復深化による方法である．攻め方は王手だけ，守り方は王手を防ぐ手だけ生成すればよいので，通常の探索と比較して分岐数がはるかに少なく，ヒューリスティクスによる指し手の絞り込みを組み合わせることである程度の性能を持つ詰めルーチンを作ることができる[4]．

長手数の詰みを反復深化による手法で見つけることは難しい．そのような問題に対して高い性能をあげる探索アルゴリズムとして注目を集めたのが，証明数を利用したアルゴリズムである．玉の逃げ方が少なくなるような攻め手順を優先的に探索することで，かなり長い詰み手順でも発見することができるようになった．いまでは，多くの将棋プログラムが，このアルゴリズムやその発展版である df-pn アルゴリズム[7]を採用しており，コンピュータ同士の戦いで 20 手以上の詰み手順を見ることはまったく珍しくなくなった．

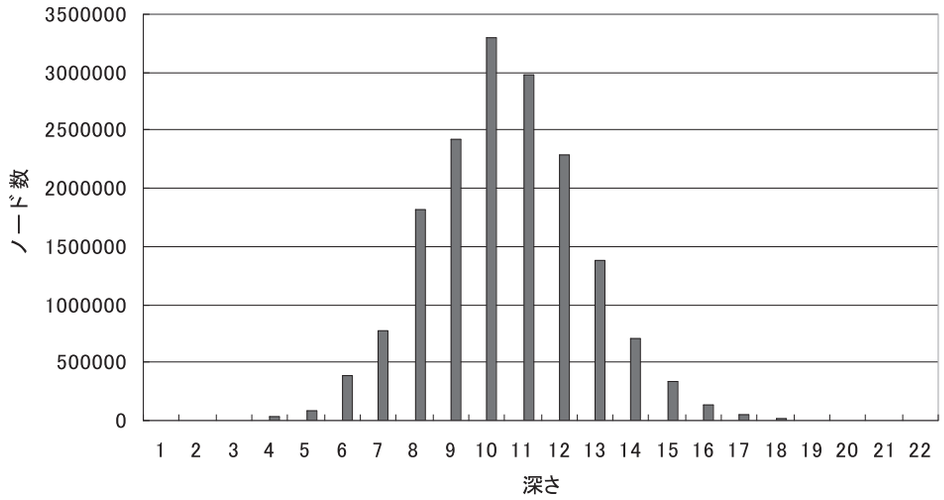

図-4　探索の深さとノード数

25→25分に
直しました

■必至探索

終盤で詰みについで重要な概念が「必至」である．これは，玉方がどうやっても次に詰まされてしまうことを防げない状態をいう．将棋のレベルが高くなってくると，詰ませて勝ちというよりも，相手玉に必至をかけて勝ちということが増えてくるため，必至を効率的に発見することは，そのまま終盤力の向上につながるといってよい．

実戦的な必至発見手法としては，IS 将棋で提案されているシミュレーションに基づく手法が非常に有効である[5]．必至を見つけることに失敗することの大きな原因は，必至をかけた瞬間，相手から連続的に王手をされて，相手玉を詰ます部分が探索範囲の外に追いやられてしまうことにある．シミュレーションに基づく手法では，連続王手をかける前の局面とかけた後の局面を類似局面とみなして，同様の手順で詰みがあるかどうかを高速にチェックすることで，必至の検出精度を大きく向上させることができる．

探索の実際

さて，実際のところ，現在のコンピュータの性能と，これまで述べてきたような探索アルゴリズムによって，コンピュータはどのくらい先の局面まで読んでいるのだろうか．**図-4** に，ある中盤の局面において激指で約 1 分間探索させた場合のノード数の分布を示す（CPU は Opteron 2.6GHz）．最も深いところで 21 手先まで読んでおり，ノード数が多いのは 10 手前後の局面であることが分かる．ちなみに 20 手先というのは，アマチュア初段前後の筆者の棋力からいうと相当に先の局面であり，自分が将棋を指すときに 20 手先まで読むことはほとんどない．総ノード数は，約 1,700 万局面なので，1 秒間に約 30 万局面を探索していることになる．

コンピュータに与えられる時間は有限である．たとえばコンピュータ将棋選手権では，25 分切れ負けといって，勝負がつくまでの自分側の思考時間をトータルで 25 分以内に収める必要がある．そのため，実際の将棋プログラムの強さには探索速度が大きくものをいう．探索アルゴリズムに関する手法は，基本的にはノード数をいかに少なくするかというための工夫であるが，それに対して，1 ノードあたりの探索にかかる時間をいかに短くするのか，というのはゲーム木探索に関する理論とは別の問題である．

速さに対する要請のためか，将棋プログラムの多くは C 言語（あるいは C++）で書かれている．以前は部分的にアセンブリ言語で書かれたプログラムも存在したが，現在ではさすがにそこまでしているプログラムはほとんどない．激指に関しては，並列処理のためのロックの部分を除けばすべて C++ で書かれている．探索の高速化に関しては，各プログラムとも相当の努力がなされていて，かなりの開発時間がプログラムの高速化のために割かれている．

探索速度と強さ

探索速度が速くなる，つまり一定時間内に探索できるノード数を増やすと，プログラムの強さが上昇することは経験的に分かっている．しかし，どれくらい速くなるとどれだけ強くなるか，という定量的な性質についてはまだ分かっていない部分が多い．激指に関していえば，3 倍の探索量をかけると，自己対戦での勝率が 8 割前後に上昇する．これはレーティング☆5 換算でいうと 200 点以上の上昇ということになるが，自己対戦での勝率と

☆5　持ち点の差を勝率に対応付けることで計算される強さの尺度．あるインターネット将棋道場のレーティングでは，アマチュア初段で 1,600 点前後，プロのトップで 3,000 点前後といわれる．

いうのは明らかに過大評価なので，実際の勝率上昇はそれよりも少ないと考えられる．YSS の開発者である山下氏の実験によると，約 2 倍に高速化した場合，他のソフトに対する勝率が 1 割弱上昇という結果が得られている．探索アルゴリズムの性質や，対戦相手，持ち時間など諸条件によって変わってくるが，3 倍の高速化によってレーティングにして約 100 点上昇というのが筆者の個人的な見積りである．

現在の激指のレーティングは，早指しの条件であれば，2,500 点前後と考えられる．単純に考えると，プログラムの高速化のみでプロのトップである 3,000 点近くに到達するためには，$3^4 = 81$ 倍の高速化を行えばよいことになる．

■並列化による高速化

完全にタスクが独立であれば並列計算の効果は大きい．すなわち 100 台投入することによって 100 倍の速度向上が可能である．しかし一般には，タスクが独立ではなかったり，並列化によるオーバーヘッドがあるために，そのまま台数分の効果が出ることは少ない．特にゲーム木探索においてはその傾向は顕著である．$\alpha\beta$ 法では，部分の探索の結果を利用して次の探索範囲を狭めていくという動作になっているため，逐次的に処理をしたときに最も枝刈りの効率が高くなる．そのため，プログラムを並列化して単位時間あたりの探索ノード数を増やすことができても，探索ノードの総数が増えてしまって，実効的な並列効果がでないということが起こる．実際のところ，激指では 2 プロセッサで約 1.5 倍の速度向上，YSS では 4 プロセッサで 2.2 倍の実効的な速度向上というのが現状である．したがって，100 個のプロセッサを搭載した並列マシンがあれば今すぐにでも名人に勝てるというわけではない．もっとも，だからこそ将棋の並列探索は，挑戦しがいのある課題といえる．

おわりに

現在のところ将棋プログラムのほとんどの部分は，設計者の知識がハードコーディングされたような形になっている．ある意味ではヒューリスティクスの塊である．つまり，少なくとも開発者の側から見ると，プログラムの全体を把握した状態で開発をしているわけで，ブラックボックスになっている部分はほとんどない．

このことは一見，プログラムの強さというものが，開発者の将棋に関する知識によって非常に強く制約されているのではないかという印象を与える．つまり，プログラムを強くするためには，開発者もプログラムと同じくらい将棋が強くなくてはならないのではないか，ということである．

ところが幸運なことに現実の状況はそうではなく，プログラムのほうが開発者より強いというのはまったく珍しいことではないし，また逆に，開発者の棋力がプロ並みだからといってプロ並みの強さのプログラムが作れるわけでもない．その大きな理由は 2 つある．1 つには，コンピュータ上のプログラムとして表現できる知識がそれほどリッチではないということである．たとえば，評価関数として実現されているものは，ひいき目にみてもアマチュア級位者程度の大局観である．人間のパターン認識的な判断能力をプログラムの形で表現するのは非常に難しく，プログラムで記述できているのはそれの非常に大雑把な近似にすぎない．もう 1 つの理由は，量が質にダイレクトに結びつくという性質である．探索量を増やせば増やしただけ強くなるというのはきわめて重要で，このおかげで，探索の効率化・高速化といった，将棋の知識とは独立な軸での改良が意味を持つのである．

自然言語処理をはじめとする他の知識処理において，量が質に最後まで直結するという幸運な性質を持っているテーマはそう多くない．プログラムが自分を上回る答えをはじき出したときの感動はコンピュータ将棋の大きな魅力である．

参考文献

1）Anantharaman, T. et al.: Singular Extensions: Adding Selectivity to Brute-Force Searching, Artificial Intelligence 43, pp.99-109 (1990).
2）Donninger, C.: Null Move and Deep Search: Selective-, Search Heuristics for Obtuse Chess Programs, ICCA Journal, Vol.16, No.3, pp.137-143 (1993).
3）Heinz, E. A.: Extended Futility Pruning, ICCA Journal, Vol.21, No.2, pp.75-83 (1998).
4）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），pp.112-142 (1998).
5）棚瀬　寧：IS 将棋のアルゴリズム，コンピュータ将棋の進歩 3（松原　仁編著），pp.1-14 (2000).
6）Tsuruoka, Y., Yokoyama, D. and Chikayama, T.: Game-tree Search Algorithm based on Realization Probability, ICGA Journal, Vol.25, No.3, pp.145-152 (2002).
7）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，コンピュータ将棋の進歩 4（松原　仁編著），pp.96-114 (2003).

（平成 17 年 6 月 14 日受付）