博　　士　　論　　文

大規模計算環境を活用する
コンピュータゲームプレイヤの研究

指導教員　　**近山 隆 教授**

東京大学大学院
工学系研究科
電気系工学専攻

氏　名　　**37-117072 浦 晃**

提　出　日　　**平成 25 年 12 月 2 日**

概 要

並列計算環境の並列性は年々増加しており、汎用的な計算機をネットワークでつないだクラスタ環境や、必要なだけ料金を支払って利用できるクラウドコンピューティングサービスなど、大規模な計算環境はより身近なものになってきている。大規模計算環境を有効に活用するためには、様々なアプリケーションに対して、大規模計算環境を用いた研究が行われるべきである。本研究では、アプリケーションとしてコンピュータゲームプレイヤを対象に、強いコンピュータプレイヤを作るために重要な、機械学習と探索の両方の観点から、大規模計算環境を活用する手法を提案する。ゲームとしては将棋を対象とした。数百コア以上からなるクラスタ環境を用いた実験により、提案手法の多くについてその有効性を示した。

コンピュータゲームプレイヤでは、ゲームの局面の有利不利を判断するために、評価関数が用いられる。強いコンピュータプレイヤを作成するためには、精度の良い評価関数が求められるため、機械学習を用いて評価関数のパラメータを自動的に精度良く調整する研究が広く行われている。評価関数のパラメータ調整には、上級者の棋譜を用いた教師あり学習が用いられることも多い。一般に、教師あり学習では、訓練データの数が多いほど精度の高い学習が行えるが、上級者の棋譜の数には限りがある。一方で、将棋などのゲームにおいては、コンピュータプレイヤは、人間の上級者と同程度の強さを達成している。

本研究では、より精度の高い評価関数を得るために、指し手のついていない局面を自動的に生成し、それをコンピュータプレイヤに探索させることで得られる指し手と合わせて、訓練データに追加することを提案する。このとき、コンピュータプレイヤは人間の上級者と同じぐらい「良い」指し手を選ぶ必要があり、そのための探索には時間がかかる。さらに、上級者の棋譜に追加して識別可能な程度の性能向上を実現するためには、棋譜と同程度の数の訓練データを追加しなければならない。したがって、この訓練データの作成には一台の計算機では時間がかかりすぎることになる。しかし、複数の局面に対する探索は完全に独立であり、容易に並列化可能であるため、大規模計算環境を有効に活用できる。訓練データに追加する訓練局面として、コンピュータの自己対戦に現れる局面、探索木のリーフノードに現れる局面、棋譜の局面からランダムに手を進めた局面の性質の異なる三種類の局面を生成した。実験には将棋プログラムの激指を用いた。生成した訓練データを上級者の棋譜と合わせて学習を行ったところ、探索木のリーフノードに現れる局面を追加した場合に、より強いコンピュータプレイヤが得られることを確認した。

学習時間は訓練データの数に比例するので、訓練データの数が増えると学習にかかる時間も長くなる。上で述べた提案により、有用な訓練データの数を増加させたとすると、それだけ学習にかかる時間も長くなることになる。学習時に用いるパラメータの調整などを考えると、学習時間は短い方が有利である。近年、大規模データの分析という需要から、機械学習アルゴリズムの中でも、データを一つ処理するごとにパラメータを更新していくオンライン学習アルゴリズムを、分散計算環境で並列化する研究が盛んに行われている。本研究では、評価関数のパラメータ調整にかかる時間を短縮するために、オンライン学習アルゴリズムの並列化手法を適用することを提案する。実際に並列学習を実装し、学習の進行の指標として学習後のプレイヤが選ぶ指し手の棋譜との一致率を用いて

評価したところ、学習時間を短縮できることを確認した。将棋の評価関数の学習では、学習中に探索が実行されるため、データの入力にかかる時間よりも計算にかかる時間が支配的であり、一つのデータに対する計算時間も大きくばらつくという特徴がある。このような特徴を持つ学習手法に対する、分散並列化手法の適用可能性を示すことができた。

コンピュータゲームプレイヤの探索を高速化するために、ゲーム木探索手法として広く用いられている $\alpha\beta$ アルゴリズムを、大規模計算環境を用いて効率的に並列化することを目的とする。$\alpha\beta$ アルゴリズムでは、すでに終了した探索の結果を用いて、他の部分木を枝刈りすることによって探索空間を大きく削減する。しかし、最も効率の良い枝刈りができたとしても、得られた指し手が最善であることを証明するために、探索しなければならないノードが存在する。$\alpha\beta$ アルゴリズムでは、できるだけそのようなノードだけを探索することが重要である。しかし、どのノードを訪問しなければならないかは当然ながら探索中には分からないため、実際の探索では探索が必要な部分木を予測してから探索を行う。しかし、あくまでも予測であるため、枝刈りされると予測された部分木の探索が必要になることもあり、逆に枝刈りされないと予測された部分木の探索が実は不要だったことが判明することもある。

本研究では、大規模計算環境を用いて、$\alpha\beta$ アルゴリズムを効率的に並列化するために、二つの手法を提案する。一つは、探索が必要だと予測されなかったノードを投機的に探索するために、実行が必要となる可能性に着目してタスクの優先度付けを行う手法であり、もう一つは、探索が必要となる部分木の予測を、探索途中で得られる結果を用いて動的に修正する手法である。

一つ目の提案は、大規模計算環境では、探索が必要だと予測される部分木の探索だけでは、アイドル状態のプロセスが生じてしまうという問題点の解消を目的としたものである。一般に、並列に実行可能なタスクを増やすためには、タスクをさらに細かく分割すればよい。しかし、タスクの粒度を小さくしすぎてしまうと、並列化に伴う様々なオーバーヘッドが相対的に大きくなってしまい、性能上のボトルネックとなってしまう。これは、通信遅延時間が大きい分散計算環境では大きな問題となる。この問題点を解決するための方法として、探索が必要だとは予測されなかった部分木の探索を投機的に実行することが挙げられる。枝刈りの予測は外れることもあるため、探索が必要だと予測されなかった部分木であっても、投機的に探索しておくことには意味があるからである。この投機的実行により、タスクの粒度を小さくすることなく、タスクを増やすことができる。ただし、探索が必要だと予測されなかったノードの中でも、必要になる可能性が高いものと低いものが存在するはずである。本研究では、各部分木の探索が必要となる可能性を見積ることで、実行が必要となる可能性の高いタスクから先に実行することを提案する。部分木の探索が必要となる可能性を見積もる方法としては、次の二つの手法を提案した。一つは、親ノードについて、各子ノードが枝刈りされない確率を予測し、親ノードが枝刈りされない確率に乗じることにより、子ノードが枝刈りされない確率を求める手法である。もう一つは、他の部分木の探索結果を確率分布を用いて予測することで、着目している部分木が枝刈りされない確率を計算する手法である。実験では、将棋の探索を並列化し探索時間を比較したが、提案手法の有効性を示すことはできなかった。しかし、ゲーム木のある深さ以上の部分木の探索時間が全て等しいとみなしたシミュレーションによる評価では、ゲーム木の分枝数が小さい場合に提案手法は有効である可能性があることが分かった。

二つ目の提案は、探索が必要だと予測された部分木を探索した後で、実は探索が不要だったと判

明する可能性を減らすためのものである。逐次探索の場合は、その時点で分かっている全ての情報を用いて、次の部分木の探索を行うことができる。これに対し、大規模な計算資源を十分に活用して並列探索を行うには、逐次探索ならば得ることができたはずの他の部分木の探索結果を得る前に、ある部分木の探索を始めなければならない。したがって、探索が必要となる部分木の予測に探索途中の結果を即座に反映して、探索の必要がない部分木を無駄に探索してしまうことを避けることが重要である。ゲーム木探索では、探索が必要となる部分木の予測に、浅い探索の結果を用いることも多い。よって、並列探索でも、浅い探索の結果を、この予測にすぐに反映させることも重要である。このような予測の修正を行う手法は、すでに先行研究で実装されているが、この予測の修正の性能に対する影響は詳しく評価されておらず、また、この実装も 64 プロセスまででしか評価されていない。本研究では、この先行研究を参考に、情報の更新が即座に反映されるような、より大規模計算環境に適した実装を行った。性能評価では、探索が必要となる部分木の予測精度の高い人工ゲーム木と激指を用いて生成した将棋のゲーム木の二種類のゲーム木を用いた並列探索を実行した。探索が必要となる部分木の予測を浅い探索の結果を用いて動的に修正することにより、並列探索時間を短くできることを示し、その程度は、予測精度が人工木より低い将棋のゲーム木を用いた場合に大きいことを示した。しかし、理想的な線形な速度向上はまだ実現できていなかった。そこで、速度向上を妨げている原因をいくつかの要因に分けて分析し、得られた結果を概ね説明できるモデルを構築した。

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　背景

並列計算環境の並列性は年々増加している。つまり、個々の CPU コアの性能向上が頭打ちになる一方、並列計算環境の計算ノード数や計算ノードあたりのコア数は増加しており、計算性能の向上は主にこの並列性の増加によって実現されている。スーパーコンピュータのランク付けを行っている TOP500 の web ページ [1] で、ここ数年のリストを見比べてみると、総コア数が明らかな増加傾向にあることが分かる。例えば、2008 年 11 月の時点では 1 位のコンピュータが 129,600 コアで、コア数が最大のものは 212,992 コア、トップ 10 位ではコア数が全て 30,000 を越えている程度であったが、5 年後の 2013 年 11 月の時点では 1 位のコンピュータは 3,120,000 コアで、トップ 10 位のものは全て 100,000 コアを越えている。

スーパーコンピュータだけではなく、より身近なところでも大規模な計算環境が利用可能になってきている。例えば、汎用的な計算機をネットワークでつないだクラスタ環境は比較的安価で構成できるため普及している。他にも、Amazon EC2 [2] や Windows Azure [3] といった、大掛かりな初期投資なく、必要なときに必要な時間だけ料金を支払って計算資源を使うこともできるクラウドコンピューティングサービス [12] も、魅力的な選択肢の一つとなっている。

このような大規模計算環境を用いた並列アプリケーションとしては、科学技術計算が主に挙げられる。一般に、シミュレーションの精度を高めようとすると計算量は増加するため、高精度なシミュレーションを高速に実行するために、大規模計算環境が必要とされている。例えば、スーパーコンピュータ「京」 [114, 79] を用いた計算として、分子動力学を用いた薬剤の開発 [39] や、1 兆個の天体の N 体シミュレーション [50] などがある。

大規模計算環境は科学技術計算にとって重要な役割を果たしているが、科学技術計算以外のアプリケーションについても大規模計算環境は有効に活用できるはずである。そのためには、様々なアプリケーションに対して、並列計算による性能向上を目的とした研究が行われる必要があるが、これらの研究に対する注目度は、密行列計算などの従来の科学技術計算より小さかった [63]。しかし、これらのアプリケーションに対しても近年研究が進んでいる。特にグラフ構造を扱うものについては、TOP500 とは異なる指標でスーパーコンピュータの性能をランク付けする Graph500 [4] が登場した。従来とは異なる性質を持つアプリケーションで、大規模計算環境を活用しようとする流れが存在する。

近年急速に注目を浴びている研究分野として、データマイニングや機械学習の並列処理 [16] が挙げられる。インターネットを通して収集された大規模なデータの解析が重要となってきているからである。大規模計算環境を用いた機械学習の高速化手法の研究としては、[41, 69, 34] などが挙げら

れる。また、このような機械学習のために Hadoop [5] や GraphLab [67]、Jubatus [6] などの分散処理基盤の研究開発も行われている。

さらに異なる性質を持つアプリケーションとして、状態空間探索が例として挙げられる。並列探索では、部分木の大きさが著しくばらつくため、負荷分散が難しいといった問題点や、他の部分木の結果を用いて枝刈りを行うため、部分計算の間に制御依存性が存在するという問題点がある。具体的な状態空間探索の並列探索の研究として、A*アルゴリズム [106, 57] や、SAT ソルバ [74]、モデル検査 [36, 78] などの並列化が行われている。また、並列探索における負荷分散に着目した研究 [94, 81, 126] も行われている。

アプリケーションによって異なる性質があるので、並列処理の研究についても様々なアプリケーションについての研究があってしかるべきである。そのような研究によって、各アプリケーションにおける性能向上だけではなく、他のアプリケーションとの共通の性質が明らかになれば、その性質を扱うための並列処理基盤の研究も進むと考えられる。本研究では、人工知能の研究分野の一つであるコンピュータゲームプレイヤを対象に、機械学習と探索の両方の観点から、大規模計算環境を活用する方法について提案する。

1.2 コンピュータゲームプレイヤ

コンピュータゲームプレイヤは、人工知能分野の一つとして研究が進められている。コンピュータプレイヤの強さは、年々着実に成長しており、いくつかのゲームにおいては人間を越える強さにまで到達している。例えば、チェスにおいては、1997 年に IBM のチェスマシンである Deep Blue が当時の世界チャンピオンであった Garry Kasparov に勝利を収めている [29]。ポーカーにおいても、プログラム Polaris が 2000 年に 1 体 1 の対戦でプロのプレイヤに勝利している [23]。

ゲームが研究の対象として適切な理由としては、単純すぎる環境ではなく、だからといって現実世界ほど複雑な環境でもなく、ちょうどよい複雑さをもった環境であることが挙げられる [88]。コンピュータゲームプレイヤはそのような複雑性をもった環境上で行動を決定するエージェントとして捉えることができる。コンピュータゲームプレイヤに適切な行動をとらせるための重要な要素として、探索と学習が挙げられる [103]。探索は、他のプレイヤの手を考慮しつつ、その局面での自分の最善の行動を決定するために行われる。学習は、人間の棋譜や行動履歴のデータから、人間と同じような行動をとらせたり、コンピュータプレイヤ自身の過去の行動とその結果から、より良い行動を決定させたりするために用いられる。

探索、学習ともに、コンピュータゲームプレイヤに限らず、人工知能一般について重要な要素技術である。しかし、コンピュータゲームプレイヤだからこその特徴的な点も存在する。以下では、そのような点を、学習や探索についてそれぞれ述べる。

1.2.1 コンピュータゲームプレイヤにおける探索の特徴

コンピュータゲームプレイヤでは、ゲーム木と呼ばれる木構造を探索する。本研究では、ゲーム木探索のうち、チェスや将棋を中心に広く用いられている $\alpha\beta$ アルゴリズム [59] を対象とする。ゲー

ム木探索や $\alpha\beta$ アルゴリズムには以下の 3 つ特徴がある。

- 一般的な探索問題では解状態を探索することが目的であるが、ある程度以上複雑なゲームに対して、コンピュータゲームプレイヤが探索空間を探索し尽すのは、ゲームの終局が近い場合を除き、探索空間が大きすぎて事実上不可能である。そこで、ゲーム木探索では、探索空間をゲーム木の深さで制限し、その中での最善の状態を求めるという問題設定とする。ある部分木について決められた深さまで探索が終了すると、その部分木の探索結果は確定するため、後述する枝刈りなどに用いることができる。
- 枝刈りが重要である。ゲーム木探索に限らず、他の探索問題についても枝刈りは重要な要素である。$\alpha\beta$ アルゴリズムにおいて特徴的なのは、最も効率良く枝刈りを行えたとしても、得られた解が最善であることを保証するために、訪問しなければならないノードが存在することである。しかし、当然ではあるが、この探索しなければならないノードは前もっては分からない。したがって $\alpha\beta$ アルゴリズムの性能向上のためには、いかにして訪問しなければならないノードだけを訪問するようにして、探索ノード数を減らすかということが重要になる。
- ゲームに対する知識を導入することで、探索性能の改善が可能である。組み合わせ最適化などに現れる一般的な探索問題では、問題に関する知識は前もっては与えられず、任意の問題を解かなければならないことが多い。これに対して、ゲームの場合にはルールが前もって与えられるため、そのルールの範囲で利用可能な知識を探索にも用いることで、効率的な探索が可能となる。具体的には、例えば、前項で述べたような訪問しなければならないノードを、ゲームに対する知識を用いて予測してから、探索することができる。

1.2.2 コンピュータゲームプレイヤにおける学習の特徴

コンピュータゲームプレイヤにおいては、局面の有利不利を正しく判断するために、局面からその有利さの程度を表す評価値を得るための評価関数というものを用いる。コンピュータゲームプレイヤの中で、機械学習の技術が最も重要な役割を果たしているのは、評価関数のパラメータの自動調整についてであり、多くの研究が行われている [28, 47, 54, 66, 100, 108]。評価関数の学習についての特徴として、以下の点が挙げられる。

- 上級者の人間の棋譜や行動履歴のデータの数が限られている。新しいゲームやマイナーなゲームにおいてはそもそも上級者が存在しない場合もある。
- ゲームをプレイすることにより勝ち負けの情報が得られるため、その情報から学習させることが可能である。具体的には、勝ち負けの情報を報酬とした強化学習を用いた研究 [101, 109] や、勝率を適応度に用いた進化計算を用いた研究 [22, 38] がある。
- 学習時に探索を行う手法が多く、探索にかかる時間が支配的である。例えば、コンピュータプレイヤの勝ち負けの情報から学習するためには、コンピュータプレイヤがゲームをプレイする必要があり、これには探索も含まれる。また、将棋で成功を収めた比較学習 [100] では指し手を精度良く評価するために探索を行う。一般的に機械学習では、データの入力にかかる時間が支配的であるものも多いが、コンピュータゲームプレイヤでは、データの入力より、探索にかかる時間の方が支配的であると言える。

1.3 本研究の提案

本研究では、評価関数の学習と探索について大規模計算環境を活用する手法をそれぞれ提案する。なお、本研究では、ゲームとして将棋を対象とし、将棋プログラムとして激指 [7] をツールに用いる。

1.3.1 大規模計算環境を活用する学習手法

本研究では、コンピュータゲームプレイヤの評価関数の学習に大規模計算環境を活用する手法として、以下の二つを提案する。

- 大規模計算環境を用いたコンピュータゲームプレイヤ自身による訓練データの生成
- 学習アルゴリズムを並列化する手法 [77, 117] の、コンピュータゲームプレイヤの評価関数の学習への適用

一つ目は、上級者の棋譜が少ないという問題点を解決するための提案である。コンピュータゲームプレイヤは学習した評価関数だけではなく、それを探索と組み合わせて行動を決定する。つまり、学習結果から単に行動を選ぶよりも、探索によって良い行動が選べる。したがって、それを学習の訓練データに追加することで、訓練データの数を増やし、コンピュータゲームプレイヤの学習結果をより良いものにできると考えられる。このアプローチの最大の問題点が、良い行動を選ぶための探索には時間がかかることである。既存の訓練データに追加して, 識別可能な程度までの性能の向上を実現するには、それと同程度の数の訓練データを新しく追加する必要がある。一つの訓練データを作るための探索に時間がかかる上に、さらに多くの訓練データを作るのは一つの計算機では時間がかかりすぎる。しかし、ある訓練データを作るための探索は、他の訓練データを作るための探索と、完全に独立に実行することができるため、容易に並列化可能である。これは、大規模計算環境を有効に活用することが可能であることを意味する。

二つ目は、評価関数の学習を高速に実行することを目的としたものである。一般に、機械学習にかかる時間は訓練データの数に比例する。したがって、一つ目の提案によって有用な訓練データの数を増加させたとすると、それだけ学習にかかる時間も長くなることになる。よって、学習のプロセスを並列化することにより、学習の時間を短縮させることは有益である。機械学習は、学習の結果を用いるときには、アプリケーションの性能に直結するため速度が重要であるが、学習は前もって行うプロセスであるため、学習にかかる時間はそれほど問題視されない。しかし、学習パラメータの調整にかかる時間などを考慮すると、学習にかかる時間が短縮されることには意味がある。さらに、コンピュータゲームプレイヤでは、学習のときにも探索が実行される手法が多いことを述べた。このときたの探索の深さは時間的制約によって決められていることが多い。したがって、学習の並列によって学習が高速に行えるようになれば、この探索の深さをさらに増やすことが可能になる。

1.3.2 大規模計算環境を活用する並列探索手法

$\alpha\beta$ アルゴリズムを並列化する研究はこれまでも行われている [29, 37, 24, 55]。本研究では、並列 $\alpha\beta$ アルゴリズムの性能を、大規模計算環境を活用して向上させるために、以下の二つを提案する。

- 枝刈りされない可能性の見積りを利用したタスクの優先度付け
- 探索途中の結果を利用した、枝刈りされない計算の予測の動的な修正

大規模計算環境を有効に活用するためには、同時に実行可能なタスクの数を増やす必要がある。タスクの数を増やすためには、問題を細かく分割するのが一般的である。しかし、タスクの粒度を小さくしすぎてしまうと、並列化に伴う様々なオーバーヘッドが相対的に大きくなり、問題になりやすい。これは、分散計算環境ではさらに問題となる。プロセス間の通信遅延時間が大きいため、タスクの粒度を小さくしにくく、結果としてタスクの数を増やすのが難しくなるからである。大規模計算環境は必然的に分散環境であるため、これは考慮するべき問題点である。

$\alpha\beta$ アルゴリズムでは、探索が必要だと考えられるノードを予測してから探索を行う。これは並列探索でも同様である。しかし、探索が必要だという予測は、あくまでも予測であり、必要ではないと予測された計算でも、実際には必要になることがある。したがって、探索が必要だと考えられるノードだけでは並列に実行可能なタスクが十分な数だけ生成できない場合、上述の問題を避けるために、タスクの粒度を小さくするのではなく、探索が必要だと予測されなかった部分の計算を投機的に実行することは有効だと考えられる。

一つ目の提案は、この投機的実行の制御に対する提案である。探索が必要だという予測に含まれなかった部分木の中でも、探索が必要になる可能性の高いものと可能性の低いものが存在する。投機的実行をするのであれば、その中でも探索が必要になる可能性の高いものを実行するべきである。本研究では、部分木が枝刈りされない可能性を見積もることで、その枝刈りされにくさをタスクの優先度に用いることを提案する。

二つ目の提案は、枝刈りされずに探索が必要となる部分木の予測に、探索途中の結果を即座に反映させることで、より正確な予測をより早く得るためのものである。逐次探索の場合は、その時点で分かっている全ての情報を用いて次の部分木の探索を行うことができる。これに対し、並列探索では、逐次探索では得ることができたような他の部分木の最終的な探索結果を得る前に、その部分木の探索を始めなければ十分な並列性を得られない。したがって、探索途中の結果を用いて即座に、枝刈りされない部分木の予測を修正することが重要である。本研究では、このような予測の修正を行う既存手法 [24] をベースに実装を行った。実装に際しては、途中で得られる結果をなるべく早く予測に反映させるための工夫を行った。

1.4 本研究の貢献

本研究の貢献としては、以下の点が挙げられる。

タスクの優先度付けに関する、計算が必要となる可能性を考慮した、一般的な手法の提案 本研究では、部分計算が必要となる可能性をタスクの優先度として用いることを提案し、その有効性の評価を行った。計算の必要性が確定しないのは、ゲーム木探索だけではなく、多くの探索問題に共通する性質である。したがって、並列ゲーム木探索で得られた知見は他の並列探索手法にも有効だと考えられる。特に、A*探索は、問題に対する知識を用いる探索手法であり、本研究

に強く関係している。本研究は、このような問題に対する並列探索手法において、一般的な手法を提案したと言える。

問題に対する知識を用いた、並列探索の制御に関する知見の提供 タスクの優先度付けは、各計算が本質的に投機的である探索問題にとっては重要であると考えられる。そのような問題を効率的に並列化するための並列処理系の研究に対しても知見を与えることができたと考える。この知見は、例えば、問題に対する知識を導入可能な処理系や、タスクの優先度に基づいた負荷分散の研究にも活用できる。

大規模計算環境を用いた並列 $\alpha\beta$ 探索の性能に対する詳細な分析 本研究では最大で 1,536 コアを用いて並列 $\alpha\beta$ 探索の実験を行い、その性能について分析した。100 コアを越える環境での並列 $\alpha\beta$ 探索の性能を評価し、その分析を行っている研究は数が少ない。本研究は大規模計算環境を用いた場合の並列 $\alpha\beta$ 探索の性能について、人工的に作成した木と、将棋プログラムを用いた場合について、それぞれ評価し、その性能差も分析対象とした。また、今回行った実装のボトルネックを詳細に調べ、さらなる研究に必要なデータを提示した。

評価関数の学習のための大規模計算環境の新たな活用方法の提示 本研究では、大規模計算環境を活用して、コンピュータプレイヤ自身に学習に有用な訓練データを生成した。このアイデア自体は単純なものではあるが、大規模計算環境を評価関数の学習に有効に活用するための新たな手法を提示したことになる。

評価関数に対する並列学習手法の適用可能性の実証 本研究では評価関数の学習に既存の並列学習手法を適用することで、学習時間を短縮することに成功した。これにより、データの入出力よりも計算が支配的である具体的なアプリケーションについて、並列学習手法の適用可能性を示したことになる。

1.5 本論文の構成

本研究では、コンピュータゲームプレイヤの学習と探索それぞれに、大規模計算環境を活用する方法を提案する。全体の流れとしては、ゲーム木探索と評価関数について説明した後に、大規模計算環境を用いて評価関数の学習を行うための提案手法を述べる。その後、並列 $\alpha\beta$ 探索手法を紹介した後に、大規模計算環境を用いて並列 $\alpha\beta$ 探索を効率的に行う提案手法について述べる。論文の具体的な構成は以下の通りである。

2 章で、ゲーム木について説明した後に、ゲーム木探索手法であるミニマックス探索と、それに対して枝刈りを用いた改良手法である $\alpha\beta$ アルゴリズムを紹介する。また、ミニマックス探索に対して優先度を用いて部分木の探索順序を決定する研究も紹介する。

3 章では、評価関数について説明した後に、コンピュータプレイヤ自身の探索結果を用いた学習手法を紹介する。さらに、本研究で用いる激指の学習手法について説明を行う。

4 章では、コンピュータプレイヤの評価関数の学習に大規模計算環境を活用する方法を述べる。まず、コンピュータプレイヤ自身によって訓練データを生成し、それをプロ棋士の棋譜に追加して学習

する手法とその評価を述べる。次に、激指の学習を分散環境で並列化する手法を述べ、それにより学習時間を短縮できることを示す。

5 章では、並列探索の関連研究について述べる。ゲーム木探索に限らず、状態空間探索全般における手法について説明した後に、並列 $\alpha\beta$ 探索の先行研究について述べる。

6 章では、並列 $\alpha\beta$ 探索において、探索が必要と予測されなかった部分木の投機的な探索を効率よく行うために、実行が必要となる可能性に着目したタスクの優先度付けを行う手法とその評価を述べる。

7 章では、並列 $\alpha\beta$ 探索において、探索が必要となる部分木の予測を、浅い探索の結果を用いて動的に修正する手法とその実装について述べる。実際に並列探索を行うことで、そのような動的な修正が性能改善につながっていることを示す。さらに、並列探索の高速化の妨げになっている要因を分析する。

8 章では、本研究をまとめた上で、本研究で得られた知見が今後どのような研究に活用されるかという展望について述べる。

第2章　ゲーム木探索

現在広く用いられているゲーム木探索の手法は、大きく分けて 2 種類ある。チェスや将棋などのゲームで用いられているミニマックス探索と、局面の有利不利を正確に判定するのが難しい囲碁などのゲームで用いられているモンテカルロ木探索 [26] の 2 つである。本研究ではミニマックス探索を取り扱う。

まず 2.1 節でゲーム木とミニマックス探索について説明する。その後、2.2 節でミニマックス探索に枝刈りを導入した手法であり、一般的に用いられている $\alpha\beta$ 探索について説明する。$\alpha\beta$ 探索は通常深さ優先探索であるが、これに対して、ミニマックス探索の際にゲーム木の各ノードに優先度を付け、その優先度が高い順に探索する研究を 2.3 節で紹介する。本研究では、並列探索においてタスクに優先度を付け、その優先度が高い順に実行することを提案しているため、関連研究として優先度を付ける手法を挙げておくことは有用であると考えた。

2.1　ゲーム木とミニマックス探索

まずゲーム木について説明する。例としてオセロのゲーム木を図 2.1 に示す。ゲーム木は各ノードがゲームの各局面を、各エッジがゲームの各合法手による局面間の遷移を表現した有向グラフである。リーフノードはゲームの決着がついた終局面となり、勝ち、負け、引き分けが判明している局面である。

しかし、ゲーム木を末端まで探索することは、ある程度以上複雑なゲームの場合は、終局に近い局面を除き、計算量が大きすぎて事実上不可能である。そこで、実際のゲーム木探索ではゲーム木をある適当な深さのノードで探索を打ち切り、そこをリーフノードとみなして探索が行われる。このとき、リーフノードにはその局面の有利不利を表す数値が割り当てられる。これを評価値と呼ぶ。局面を入力とし評価値を出力する関数を評価関数と呼ぶ。通常、評価値は整数であり、本論文でも評価値は整数だとする。

ミニマックス探索は、このようにリーフノードに評価値が入ったゲーム木を、自分も相手も常に最善の指し手を指すという仮定の下で、ルートノードの局面において、最終的に到達するリーフノードの評価値を最大にするような最善の着手を調べるための探索手法である。図 2.2 にミニマックス探索の例を示す。ゲーム木において自分の手番を表すノードを Max ノード、相手の手番を表すノードを Min ノードと呼ぶ。Max ノードにおいては、子ノードのうち最大の評価値の子ノードの選び、そのノードの評価値とし、Min ノードにおいては、子ノードのうち最小の評価値の子ノードを選び、そのノードの評価値とする。これをリーフノードからボトムアップに繰り返し、最終的にルートノードにおいて、どの指し手を指すのが最善かを求めるのがミニマックス探索である。ミニマックス探索

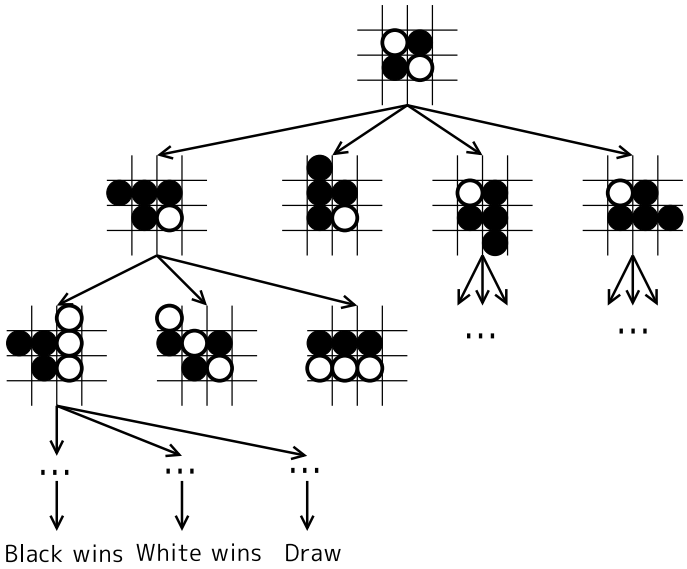

図 2.1: オセロのゲーム木

によって、着手後の局面、すなわちルートノードの子ノードの優劣を、各子ノードに評価関数を直接適用するよりも正確に予測できる。探索の深さが深いほど、優劣の予測はより正確に行うことが可能になる。

全てのノードで合法手数が b の一様なゲーム木の場合、探索深さを d とすると、ミニマックス探索の計算量は $O(b^d)$ となり、指数関数的に増大する。そのため、枝刈りによって計算量を削減することが非常に重要となる。

2.1.1 実現確率探索

実現確率探索 [104] は有望な手を深く探索する手法の一つであり、本研究で用いる将棋プログラムの激指で用いられている。図 2.3 に実現確率探索の例を示す。ある局面に対して、それぞれの手が指される遷移確率 (その手の妥当性に相当) を棋譜から計算する。ルートノードの実現確率を 1 とし、そこからあるノードまでの遷移確率をかけあわせていったものを、そのノードの実現確率とする。実現確率は対数をとって絶対値をとることによって探索深さとみなすことができる。つまり、実現確率探索は妥当な手は残り探索深さをあまり減らさず、そうでない手は残り探索深さを大きく減らすような、残り探索深さの減少が指し手によって可変である探索手法と見ることができる（図 2.4 に例を示す)。以後本論文では「深さ」という用語は、1 手が深さ 1 を表す通常の深さの意味でも、この実現確率による深さの意味でも用いる。特に区別する必要がある場合は、後者を「確率深さ」と呼ぶことにする。

実現確率探索は、有望でない部分木は浅くしか探索しないというものである。しかし、その部分

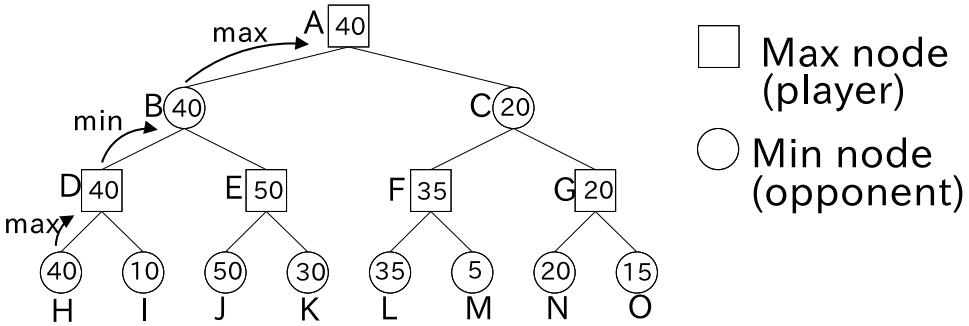

図 2.2: ミニマックス探索の例

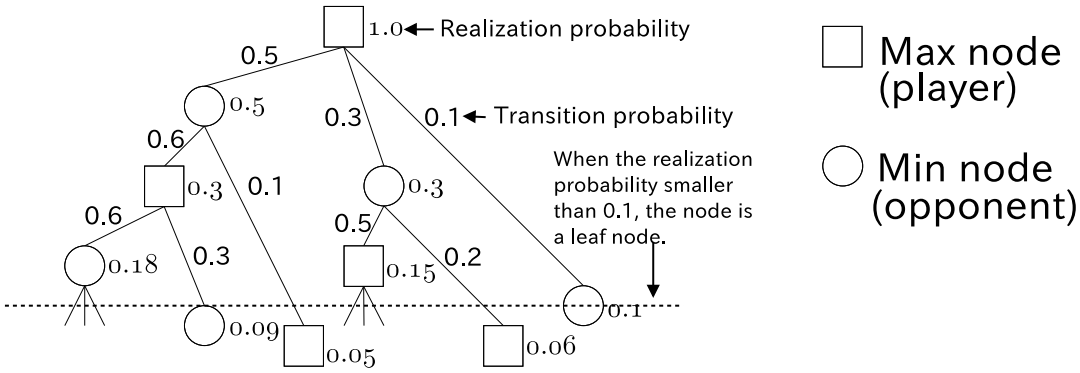

図 2.3: 実現確率探索の例

木の評価値が良かった場合、浅くしか探索していない結果をそのまま用いると、評価値の精度が低いため、悪い手を指してしまう可能性がある。そこで、実現確率探索では、図 2.5 のように、はじめから有望だと考えられる指し手（例えば浅い探索での最善手）は、遷移確率を用いずに深く探索する。深く探索する場合は、確率深さの減少値を最小値である 1 に設定する。他の指し手については、はじめは有望ではないと考えられたため、遷移確率を用いて探索するが、もし評価値が良さそうならば、深く再探索するということを行う。

2.1.2 トランスポジションテーブル

実際のゲーム木は木ではなくて、異なる経路を経て同じ局面に到達する合流が、図 2.6 のように存在することがある。これを木とみなして探索すると、同じ局面を複数回探索することにある。このとき、同じ局面の 2 回目以降の探索は無駄である。この無駄をなくすため、実際のゲーム木探索では、探索した結果を保存しておき、必要になったらその結果を再利用するといったことを行う。この結果を保存しておくテーブルをトランスポジションテーブルと呼ぶ。トランスポジションテーブルは、

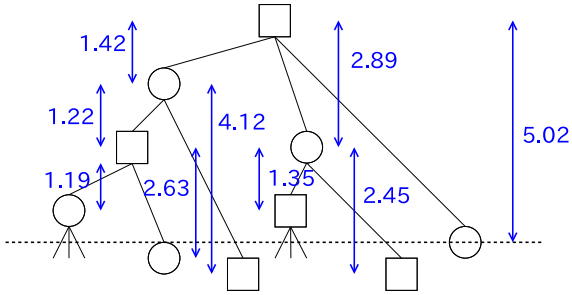

図 2.4: 実現確率探索を確率深さを用いて表現

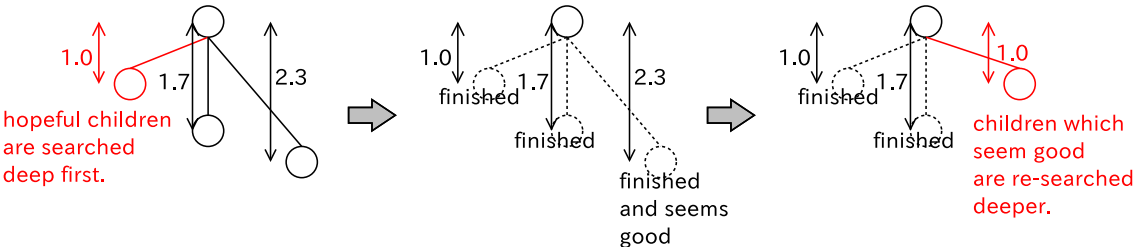

図 2.5: 実現確率探索における再探索

局面をキーとしたハッシュテーブルで実装される。エントリの内容としては、その局面を探索したときの探索深さと、そのノードの評価値や最善手などの探索結果が格納される。浅い探索の結果が必要なときに深い探索の結果が利用可能だった場合、より信頼のおける深い探索の結果を使うとしたならば、探索結果のミニマックス値が変わることがある。

2.2 $\alpha\beta$ アルゴリズム

$\alpha\beta$ アルゴリズム [59] はミニマックス探索の計算量を枝刈りによって削減した探索手法であり、ゲーム木探索において一般的に用いられている。すでに探索した結果から、探索する必要がないと判明したノードの枝刈りを行う。$\alpha\beta$ 探索で行われる枝刈りはミニマックス探索の結果を変えることがなく、$\alpha\beta$ 探索とミニマックス探索の結果は等しくなる。

$\alpha\beta$ 探索の枝刈りの例を図 2.7 に示す。すでにノード B 以下の部分木の探索は終了しており、ノード B の評価値は 40 ということが分かっていたとする。ノード D の探索が終了し、D の評価値が 35 だということが分かった時点で、ノード E の探索は必要ないことが判明する。なぜなら、ノード A の評価値はノード C の評価値が 35 だと分かった時点で、ノード E の評価値を x とすると、$\max(40, \min(35, x))$ となり、これは x に関係なく 40 となるからである。

$\alpha\beta$ 探索には探索窓という概念がある。探索窓は (α, β) で表され、現在の探索を評価値が α 以上 β 以下のものに限って探索をするというものであり、探索窓の範囲外の評価値ならば、その評価値の値

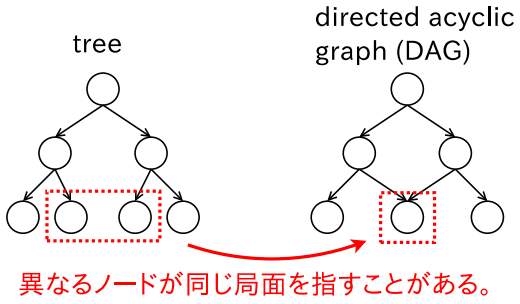

図 2.6: 異なるノードが同じ局面を表している例

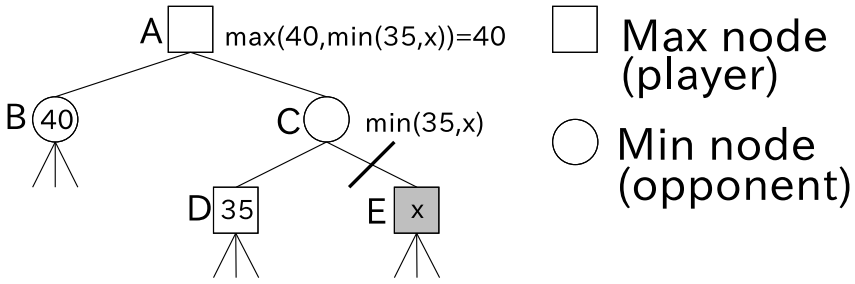

図 2.7: $\alpha\beta$ 探索における枝刈り

を正確に求めなくてもよい。ルートノードでは最終的な評価値がまったくわからないので $(-\infty, \infty)$ で探索を開始させる。探索窓は子ノードにも伝搬されていく。各ノードでは各子ノードの探索結果を元に、Max ノードでは最大値を求めるので下界となる α を、Min ノードなら逆に上界となる β を更新することで探索窓の範囲を狭めていき、$\beta \leq \alpha$ となったとき、つまり関心のある評価値が存在しえないときに残りの枝を枝刈りする。

$\alpha\beta$ 探索の擬似コードを図 2.8 に示す。このコードには以降で説明する多重反復深化を含んでいる。記述を簡略化するために、Max ノードと Min ノード、α と β がそれぞれ対称的であることを利用して、Min ノードの符号を反転した、Negamax 形式と呼ばれる方法で記述してある。このとき、探索窓の更新は常に α だけに関して行い、それが β を上回ったら枝刈りが発生する。子ノードの探索を行うときは、相手の手番になるので、α と β を入れ替えて負号をとったものを探索窓とする。

$\alpha\beta$ 探索では探索窓の範囲を越えた場合のノードの評価値は不正確であり、ミニマックス探索のときと異なる点に注意する。図 2.2 と同じゲーム木を $\alpha\beta$ 探索したときの様子を図 2.9 に示しているが、例えば、図 2.2 ではノード C の評価値は 20 であったが、図 2.9 では 40 となっている。これは図 2.9 ではノード C を探索窓 $(40, \infty)$ で探索しており、40 を下回る値には関心がないからである。ある探索窓 (α, β) を用いて探索して得られた評価値 $v_{\alpha\beta}$ と、ミニマックス探索のときの評価値（ミニマックス値）v_{mm} との間には次のような関係がある。

```
int AlphaBeta(position, depth, alpha, beta){
  if(d == 0 || position is a terminal position)
    return Evaluate(position);
  AlphaBeta(position, depth−2, alpha, beta);
  clist = SortChildren(position);
  foreach(child of clist){
    alpha = max(alpha, −AlphaBeta(child, depth−1, −beta, −alpha));
    if(beta <= alpha) return beta;
  }
  return alpha;
}
```

図 2.8: $\alpha\beta$ アルゴリズムの擬似コード（多重反復深化を含む）

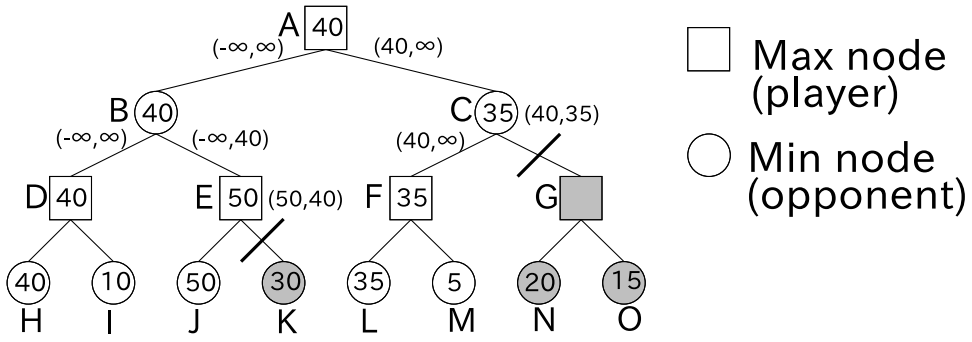

図 2.9: 図 2.2 と同じゲーム木を $\alpha\beta$ 探索

- $v_{\alpha\beta} = \alpha$ ならば $v_{\mathrm{mm}} \leq \alpha$
- $\alpha < v_{\alpha\beta} < \beta$ ならば $v_{\mathrm{mm}} = v_{\alpha\beta}$
- $v_{\alpha\beta} = \beta$ ならば $v_{\mathrm{mm}} \geq \beta$

つまり、探索窓の内部の値が返ってきた場合は、それはミニマックス値に等しい。探索窓の値が返ってきたら、ミニマックス値は探索窓の外の範囲にあり、α や β はその上界や下界となる。

トランスポジションテーブルに評価値を格納するときと、格納されている結果を用いるときは、評価値の範囲を考慮しなければならない。ある部分木の探索が終了すると、ミニマックス値が得られるか、その上界（α）か下界（β）が判明する。つまり、探索によって得られた評価値の下界値と上界を v_l と v_u とすると、この二つは、以下にように表される。

- $v_{\alpha\beta} = \alpha$ のとき、$v_l = -\infty$ かつ $v_u = \alpha$
- $\alpha < v_{\alpha\beta} < \beta$ のとき、$v_l = v_{\alpha\beta}$ かつ $v_u = v_{\alpha\beta}$
- $v_{\alpha\beta} = \beta$ のとき、$v_l = \beta$ かつ $v_u = \infty$

また、トランスポジションテーブルにすでに格納されている評価値の下界値が l、上界が u だとする

（格納されていない場合は $-\infty$ と ∞ だとする）。このとき、トランスポジションテーブルの情報は次のように更新される。

- $l \leftarrow \max\{l, v_l\}$
- $u \leftarrow \min\{u, v_u\}$

トランスポジションテーブルの結果を用いるときは、現在の探索窓の値を α と β、格納されている評価値の下界値と上界を l と u として、以下のように行う。

- $u \leq \alpha$ のとき
  関心のある範囲に評価値がないことが分かるので、α を返す。
- $\beta \leq l$ のとき
  関心のある範囲に評価値がないことが分かるので、β を返す。
- $u = l$ のとき
  評価値が正確に分かっているので、その値 $u = l$ を返す。
- それ以外のとき
  結果を直接使って値を返すことはできない。しかし、格納されている結果を用いて、
  $\alpha = \max\{\alpha, u\}$、$\beta = \min\{\beta, l\}$ のように探索窓を狭めることができる。

実際のコンピュータゲームプレイヤのトランスポジションテーブルのエントリには、評価値の上界値と下界値を両方格納する代わりに、一つの値とそれが上界値か下界値か正確な値かのフラグを格納するという実装もある。後者は情報量が減ってしまうが、使用メモリ量を減らすことができる。激指のトランスポジションテーブルのエントリは後者の実装である。

さらに、実際にトランスポジションテーブルを使うときには、探索の深さも考えなければならない。トランスポジションテーブルのエントリには一つの局面には一つの探索深さの結果しか格納しないのが一般的であり、異なる探索深さの結果を格納しようとしたときに、どちらの結果を残すべきかという問題が生じる。使用メモリ量の制約から全てを残すわけにはいかないとすると、これに対する広く受け入れられている解決策はないが、以下から総合的に判断してどの結果を残すかを決定する [123]。

- 深い結果ほど残す。深い探索の結果は、浅い探索の結果よりも精度が高いと期待されるので、代わりに用いても問題は生じにくい。
- 上界値と下界値の差が小さく、評価値の範囲が絞れている結果を残す。さらに言えば、評価値が確定している結果を残す。評価値を直接使える可能性が高いからである。
- より新しい結果を残す。再び同じ局面に到達する可能性が高いため、参照される可能性も高いと考えられるからである。

2.2.1 指し手の並び替え

$\alpha\beta$ 探索の重要な性質として、探索する子ノードの順序によって枝刈りの効率が大きく変わることが挙げられる。探索窓の更新において、評価値が良い手ほどより探索窓を狭めることができ、探索

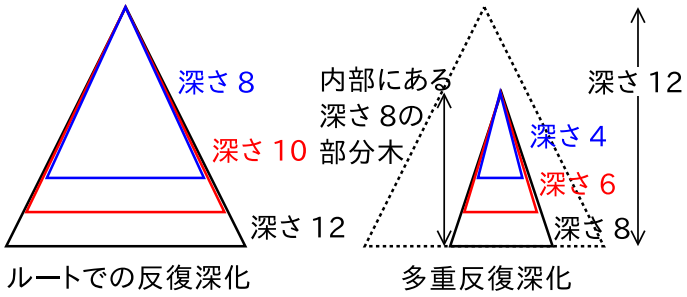

図 2.10: 多重反復深化の例

窓が狭いほど枝刈りが頻繁に発生するようになるからである。つまり、評価値が良いと思われる手を先に探索するすることが探索を効率化するために必須である。そこで実際の探索においてはあらかじめ合法手を並び替え、有望な手から探索を行う。この指し手の並び替えのことを move ordering とも言う。

指し手の並び替えを行うための代表的な手法として反復深化が挙げられる。これは深い探索の前準備として、まず浅い探索を行い、その結果を利用して合法手を並び替えるということを、少しずつ深さを深くしていきながら繰り返すというものである。具体的に述べると、6 手読みの探索を行いたいとき、まずは (例えば)4 手読みで探索を行い、その結果を用いて手を並び替え、6 手読みの探索を行うことになる。これは余計な探索を行うので一見効率が悪くなりそうである。しかし、ゲーム木の計算量は探索深さに対して指数関数的に増大するので浅い探索のコストは小さく、それに対して手の並び替えによる探索効率の改善が大きいことが多いので、有用な手法である。

反復深化はルートノードのみで行われることが多いが、すべてのノードで反復深化を行う場合もあり、そのような反復深化は多重反復深化と呼ばれる。多重反復深化の例を図 2.10 に示す。図 2.8 の 4 行目では、深さを 2 だけ減らした浅い探索を行っているが、これが多重反復深化である。その後、5 行目で子ノードのリストを生成して、並び替えを行っているが、このとき、浅い探索で最善だった子ノードが最初に探索されるように並び替える。激指はこの多重反復深化を採用している。

指し手の並び替えの手法としては他にも存在する。以前に実行した他の部分木の探索において、枝刈りを引き起こしたり、枝刈りが起きなかったときは最良の評価値をもたらした手を記憶しておき、現在の局面においてそのような手が合法手であるならば、その指し手を優先的に探索する手法がある。同じ深さのノードでこれらの指し手 (killer move) を使う方法をキラーヒューリスティックと呼ぶ。また、このような指し手に、ゲーム木の中での深さで重みをつけて足し合わせて重要度として用いる、ヒストリーヒューリスティック [91] という手法もある。

2.2.2　最小探索木

$\alpha\beta$ 探索では有望な指し手を先に探索することで効率よく探索することができると述べた。しかし、最も効率良く探索できた場合でも、最善手が最善であることを保証するために、訪問しなければなら

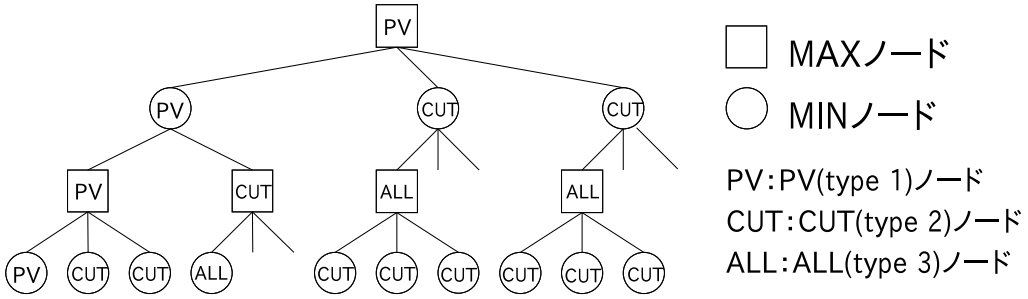

図 2.11: 最小探索木の例

ないノードが存在する [59]。このようなノードからなるゲーム木を最小探索木 (minimal tree) という。最小探索木の例を図 2.11 に示す。最小探索木のノードは次の 3 つのノードに分類される [59, 73][1]。

PV(type 1) ノード 最終的な評価値を与えるリーフノードまでのパス上にあるノード。このパスは最善応手手順 (principal variation) と呼ばれているため、PV ノードと呼ばれる。PV ノードの子ノードは必ずすべて探索され、最左の子ノードは PV ノード、残りの子ノードは CUT ノードとなる。

CUT(type 2) ノード 子ノードを一つ探索すれば、他の子ノードは枝刈りされるノード。その子ノードは ALL ノードとなる。

ALL(type 3) ノード PV ノード以外ですべての子ノードを探索しなければならないノード。子ノードはすべて CUT ノードとなる。

ここで、全てのノードで子ノードの数が同じである一様なゲーム木について、最小探索木を探索した場合の計算量を求めると、$O(b^{\frac{d}{2}})$ となる。つまり、$\alpha\beta$ 探索では、最善の場合には、単純なミニマックス探索の 2 倍の深さまで探索することができる。実際の $\alpha\beta$ 探索でも、指し手の並び替えの精度が高いことが多いため、最小探索木に近いゲーム木を探索できていることが多い。

なお、最小探索木という単語は、細かく言えば複数の意味で用いられている [24]。例えば、以下のようなものが挙げられる。

- 常に最善の子ノードを選択した場合に訪問されるノードからなる木のこと。
- 訪問ノード数が最小の木のこと。一様でないゲーム木については、上の意味での最小探索木と異なることがある。
- 必ず探索しなければならないノードの組み合わせからなる木のこと。この定義では最小探索木は複数存在する [19]。なぜなら、CUT ノードでは最善の子ノードを選ばなくても枝刈りが起こることがあるためである。

[1] 文献 [59] では type 1 などと呼んでいるが、文献 [73] では PV、CUT、ALL と呼んでいるため、両方示す。

2.2.3 Null Window Search

Null window とは $(\alpha, \alpha+1)$ の探索窓のことである。この探索窓を用いて探索することで、評価値が α より大きいかどうかを通常の探索窓を用いた場合よりも高速に調べることができる。null window は探索窓が最も狭い探索窓であるので、通常の探索窓を用いた場合より枝刈りが高頻度で発生し、探索時間が短縮されるからである。ただし正確な評価値を求めることはできない。

Null widnow を利用して $\alpha\beta$ 探索を改良した手法が PVS(Principal Variation Search) [71] と NegaScout [87] である。この二つは別々に提案されたが、基本的に同じアルゴリズムである。手続きは次のようになる。まず、最左の子ノードを通常の探索窓で探索を行う。その後枝刈りが発生せず、残りの子ノードを探索する必要があれば、残りの子ノードを null window で探索する。その結果、最左以外の子ノードの中に、最左の子ノードより良い解が存在しうるかどうかが分かる。もし、最左以外の子ノードが最左の子ノードより良いことが分かれば、探索窓を通常の探索窓に戻して、その子ノードを再探索し、正確な評価値を求める。

PVS や NegaScout は、指し手の並び替えの精度が十分良ければ、ほとんどの探索が null window を用いて行われるので $\alpha\beta$ 探索より少ないノード数で探索が可能である。逆に指し手の並び替えの精度が悪ければ、通常の探索窓による再探索が頻繁におき、null window の探索が無駄になるので、$\alpha\beta$ 探索よりノード数が増えてしまう可能性もある。

null window を利用した別の手法として MTD(f)(Memory-enhanced Test Drivers(first)) [86] がある。MTD(f) では null window だけを用いて探索を行う。初期値に適切な値を選び、その値より大きいか小さいかを調べることで、下界値と上界値を狭めていき、最終的に下界値と上界値が一致したところを評価値とする。MTD(f) では同じノードを何回も探索するのでトランスポジションテーブルを利用して、重複探索を回避している。

2.3 ミニマックス探索に対する探索順序付け

$\alpha\beta$ 探索は深さ優先探索であるが、ミニマックス探索の際に各ノードに優先度を付け、優先度の高いノードから順に探索する手法も存在する。本研究では、並列探索において、探索を早く終了させるためには「どこから探索するか」という優先度付けを提案しているため、このような先行研究との関連を議論することは有意義である。

本節で紹介する手法は、深さを決めて探索するわけではなく、時間が許す限り探索を続けるというものである。深さを決めているわけではないため、ある部分木の探索が終了するということはない。コンピュータプレイヤが良い行動を決定するためには、どの部分木を探索していくのがいいのか、という点に着目して探索が進められる。つまり、重要な部分木を優先的に深く探索したいというのが動機である。このとき「重要さ」を測るための指標を明確に決める必要がある。本節で紹介する各手法の違いは、この指標の決め方の違いとなる。

なお、深さ優先探索においても良さそうな手を深く探索する手法の研究は進められている [20] [104]。しかし、これらはいわば探索する前からどこを深く探索するかを決めている手法であり、探索中に得られる情報も用いて深く探索する手の決定を行っているわけではない。

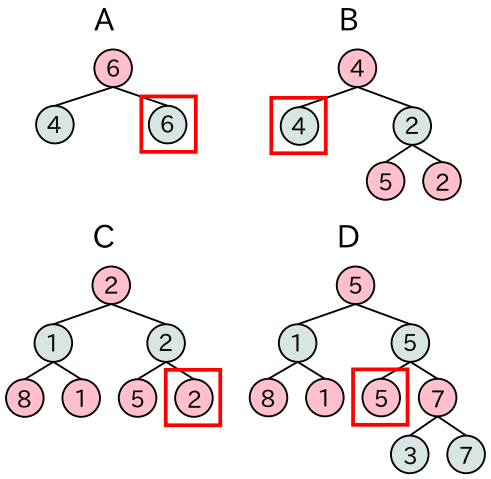

図 2.12: Best-first minimax search の例

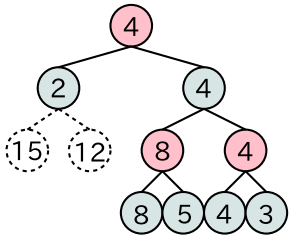

図 2.13: Best-first minimax search の問題点

2.3.1　評価値の不確かさ

評価関数は局面の有利不利を推定して数値化したものであるが、推定値である以上、評価関数から得られる評価値には必ず不確かさが含まれる。ゲーム木探索はこの不確かさを減少させることで、より精度のよい評価値を得る操作であるといえる。深く探索すればするほど評価値の不確かさが増加する例外もまれに存在する [112] が、基本的にはゲーム木を深く探索すれば、評価値の不確かさは減少し、結果的に強いゲームプレイヤを作ることができる。

また、評価値は一般的に数値、すなわちスカラーで表されるが、これは評価値の分布の平均値だけを考慮していると考えることが出来る。評価値をスカラーではなく分布で表現することで、評価値の不確かさも含めて表現することが可能となるが、計算量が増加する。

以下で紹介する各手法は、評価値の不確かさに着目して、次に探索するノードを選んでいる。

2.3.2　Best-First Minimax Search

Best-first minimax search(BFM) [62] は、常に最善の指し手列を表すゲーム木中のパスの先のリーフを展開していく探索手法である。このリーフはその評価値がルートの評価値を決定しているという意味において重要であり、その部分を重点的に探索するのは合理的であるといえる。

図 2.12 を用いて具体的な手続きを説明する。ルートノードは Max ノードである。まず A が初期状態である。このときルートの評価値 6 は右側の子ノードが決定しているので、まずはその子ノードであるリーフを展開する。展開して評価値を更新したのが B である。ルートの右側の部分木は左側の部分木より評価値が小さいということがわかったので、次は左側の子ノードであるリーフを展開して C となる。ルートの左側の部分木の評価値が下がり、再び右側の部分木の評価値の方が大きくなったため、次は右側の部分木でルートの評価値が 2 であることを決定づけているリーフを展開することになる。その結果が D である。このようにして探索を続けていく。

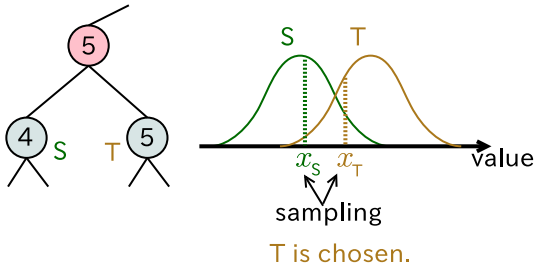

図 2.14: Randomized best-first minimax search におけるノード選択

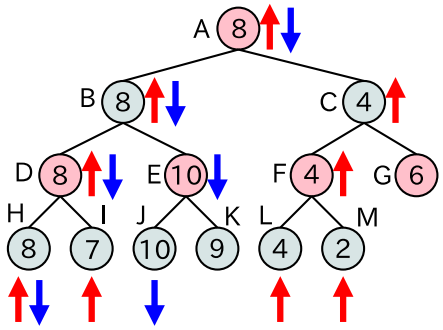

図 2.15: Randomized best-first minimax search で選ばれる可能性のあるノード

BFM では最も良い部分しか注目しないので、始めは運悪く低い評価しか与えられていないが、もっと深く探索すると非常に良い手であることがわかる場合に対応できないという問題点がある。例えば図 2.13 では、ルートの左の子ノードを展開すればよい評価値が得られるにも関わらず、それに気づくことなく右側の部分木をずっと探索することになってしまう。

Randomized best-first minimax search(RBFM) [93] はこの問題点に対応するために、どの部分木を詳しく探索するかについて確率的に選択を行う。例えば、図 2.14 のように、S と T のどちらの子ノードを次に詳しく探索するかを決定する場面を考える。それぞれの子ノードの評価値は、その不確かさから確率分布に従っていると仮定する。この確率分布は正規分布などを用いあらかじめ学習させておく。このとき、それぞれの S と T の評価値の確率分布から評価値を一つずつサンプリングし、x_S と x_T とする。S と T の親ノードは Max ノードなので、x_S と x_T のどちらが大きいかを調べ、大きい値がサンプリングされた方の子ノード、すなわち T を選択する。ルートから開始して以上の操作を繰り返し、リーフに到達したらそのリーフを展開する。

RBFM では、ノードの選択を上で述べたように確率的に行うこともあれば、必要に応じて最善の子ノードを決定的に選択することもある。図 2.15 を用いて説明する。この図は、親ノードの評価値を大きく、あるいは小さくする場合に子ノードの評価値が大きくなればいいか、小さくなればいいかを示したものである。Max ノードである A の評価値を変化させるならば、評価値が最大の子ノードの評価値が変化するか、あるいはそれ以外の子ノードの評価値が大きくなればよいということを示す。また、Min ノードである C の評価値を大きくしたい場合は、評価値が最小の子ノードの評価値を大きくするしかなく、他の子ノードの評価値がどう変化しても C の評価値は大きくはならないことがわかる。このノード C のように、ある子ノードの値がどう変化しても、親ノードの評価値に望まれる変化を与えられない場合は、そのノードは選択しないことにする。これは $\alpha\beta$ アルゴリズムにおける枝刈りを考慮していることに相当する。

$\alpha\beta$ アルゴリズムと比較した RBFM の性能を述べる。$\alpha\beta$ アルゴリズムを用いたチェスプログラムである Crafty と RBFM を用いたチェスプログラムを、RBFM の持ち時間を変化させて対戦させたところ、RBFM は Crafty の 7 倍遅いと報告されている。RBFM のプログラムが Crafty の 7 倍の持ち時間で、やっと強さが互角になったということである。

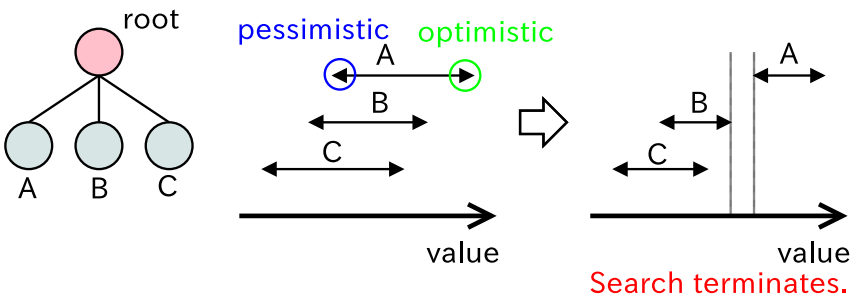

図 2.16: B* search の終了

2.3.3 B* Search

B* search [17] はノードの評価値の不確かさを考慮することで、最善だと考えている指し手が、評価値の不確かさの範囲を考慮しても最善だと判明するまで探索を行う最良優先探索手法である。

B* search では評価値に上界値と下界値を設定する。それぞれ optimistic value と pessimistic value と呼ばれる。このとき探索開始時にはルートの子ノードそれぞれの評価値がとりうる値の範囲は重なり合っているが、探索を進めていったときに、最善の子ノードの pessimistic value が残りの子ノードのどの optimistic value よりも大きくなったときに、最善の子ノードが間違いなく最善であると判断し、探索を終了する (図 2.16)。

B* search では次に展開するリーフの選択方法として、前節で述べた BFM とほぼ同じやり方でリーフを選択している。しかし、これでは最善の子ノードの下界値を他の子ノードの上界値より大きくする、という目標に正しく向かっているとはいえない。以下では、このような目標を達成するようにリーフを選択する方法を考えるために、B* search の改良手法である、B* probability based search [18](以下、確率的 B*と呼ぶ) について見ていく。

確率的 B*では各ノードは次の 4 つの値を保持している。ただし、ルートノードの手番は自分の手番で、Max ノードである。

- RealVal:ノードの評価値。
- OptVal:そのノードの手番のプレイヤから見て最も楽観的な評価値。
- PessVal:そのノードの手番でないプレイヤから見て最も楽観的な評価値。直接計算されることはなく、親ノードや子ノードの OptVal から伝搬されてくる。
- OptPrb:探索が進むことで、ある決められた目標値 TargetVal に RealVal が到達する確率。

確率的 B*の処理は大きく二つのフェーズに分けられる。一つは SELECT フェーズで、もう一つは VERIFY フェーズである。探索は SELECT フェーズから始まり、B*の終了条件をを満たすように探索を行う。VERIFY フェーズでは、SELECT フェーズの結果、最も RealVal が大きかったノードが、相手側の手番から考えても確かに最も RealVal が最大であることを確認する。もしそうでなければ SELECT フェーズに戻って繰り返す。

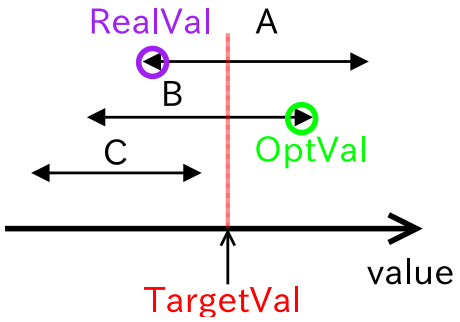

図 2.17: SELECT フェーズ中の TargetVal

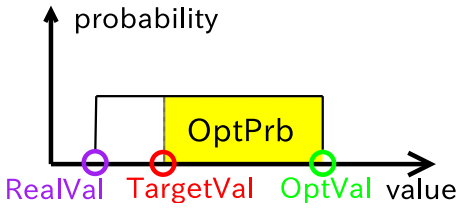

図 2.18: リーフノードの OptPrb

SELECT フェーズでは自分の手番である Max ノードにおいては終了条件を満たすために最も有望な子ノード、つまり最も OptPrb が大きいノードを探索し、相手の手番である Min ノードにおいては、最も RealVal が小さいノードを探索していく。逆に VERIFY フェーズでは相手の手番である Min ノードにおいては最も OptPrb が大きいノードを探索し、自分の手番である Max ノードにおいては、最も RealVal が大きいノードを探索していく。このように、SELECT フェーズと VERIFY フェーズでは共に片方のプレイヤが OptPrb が最も大きいノードを探索し、もう片方のプレイヤが RealVal が最もミニマックス探索で最善な値をとるノードを探索していくことになる。

SELECT フェーズの目標は、ルートの子ノードのうち最大の RealVal を、残りのすべての子ノードの OptVal より大きくすることである。この目標を達成するために、RealVal がひとまず目指すべき目標値 TargetVal を、ルートの子ノードの中での RealVal の最大値と、2 番目に RealVal が大きい子ノードの OptVal の平均値に設定する (図 2.17)。SELECT フェーズは、ルートの最善以外のすべての子ノードの OptVal が、あるしきい値より小さくなったときに終了する。

VERIFY フェーズの目標は、相手の手番から考えて、SELECT フェーズで最善だと判断された子ノードが、実はルートの子ノードの中で RealVal が最大ではなかったということを示すことである。この目標を達成するために、RealVal が目指すべき目標値 TargetVal を、子ノードのうち 2 番目に大きい RealVal から 1 を引いたものに設定する。もし探索途中で SELECT フェーズで最善の子ノードだと判断された子ノードの RealVal が TargetVal を下回ったら、SELECT フェーズで最善だと判断された手は最善ではないと分かったため、VERIFY フェーズを終了し SELECT フェーズに戻る。逆に、ルートの最善の子ノードのすべての子ノードの OptVal がしきい値より小さくなれば、現在最善だと考えている手が悪くなる可能性は十分小さいと考え、探索を終了し、指し手を決定する。

次に OptPrb について説明する。OptPrb はそのノードの RealVal が TargetVal を達成する確率を表している。SELECT フェーズの Max ノードと VERIFY フェーズの Min ノードがリーフになった場合には、まず OptVal が推定される。そして OptPrb を次の式で計算する。

$$\mathrm{OptPrb} = \frac{\mathrm{OptVal} - \mathrm{TargetVal}}{\mathrm{OptVal} - \mathrm{RealVal}} \tag{2.1}$$

これは、リーフの評価値が RealVal から OptVal まで一様分布していると仮定して、図 2.18 の黄色の面積に相当する。

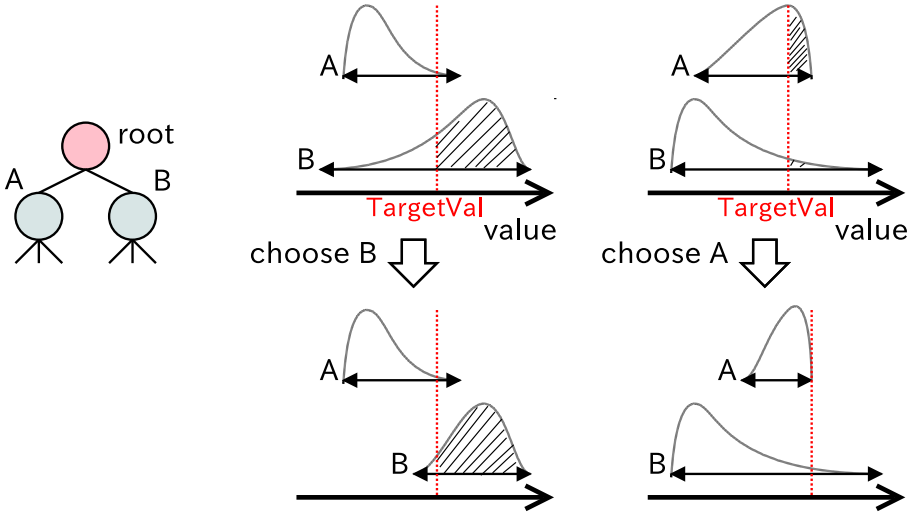

図 2.19: B* probability based search におけるノード選択

RealVal はミニマックス探索のように子ノードから親ノードに値を伝搬させていく。PessVal、OptVal、OptPrb に関しても子ノードから親ノードに値を伝えていく。

確率的 B*のノードの選択について、直感的な説明を与える。ある程度探索が進んで、ルートの子ノードの評価値が図 2.19 の上のような確率分布になっているとする。図の分布の斜線部の面積が OptPrb である。このとき、A と B のどちらを選ぶかを考えるとき、左上の分布の場合には RealVal が OptVal に近づく確率は B の方が高く、右上の分布の場合には RealVal が OptVal に近づく確率は A の方が高いのは、直感的に理解できる。したがって、左側の場合は B を、右側の場合は A を選択する。下側の図は選択した後の分布の変化し、A と B の分布が離れた場合の例を示している。実際の確率的 B*では確率分布を直接用いる代わりに、OptPrb を用いることで分布を考慮している。

$\alpha\beta$ アルゴリズムと比較した場合の確率的 B*の性能だが、チェスにおいて、同じ持ち時間で対戦させたところ、$\alpha\beta$ アルゴリズムを用いたプログラムに対する、確率的 B*を用いたプログラムの勝率は 41.7%であった。

2.3.4 共謀数探索

この章では、共謀数探索 (conspiracy search) [75] を紹介するが、探索そのものの説明を行う前に、まず共謀数 (conspiracy number) の説明を行う。共謀数は、あるノード n の評価値を v に変化させるために必要な、評価値を変更しなければならないリーフの数で定義され、ここでは $C(n, v)$ と書くことにする。例えば、図 2.20 において、ルート A の評価値を 3 まで大きくするためには F か H が 3 になればいいので、$C(A, 3) = 1$ である。同様に A の評価値を 8 まで大きくしたければ、E と F の両方を 8 にするしかなく、$C(A, 4) = 2$ である。逆に A の評価値を 0 まで小さくしたければ、例えば

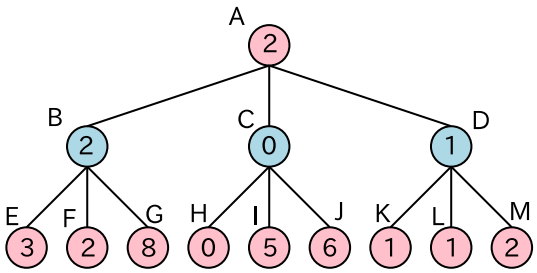

図 2.20: 共謀数探索の例

F と K の両方が 0 にならなければならないため、$C(A,0)=2$ となる。

ここまではノードの評価値をある値にするために共謀数を定義したが、逆に共謀数からノードの評価値がどれだけ変わりうるかということを決定することもできる。あるノード n の共謀数 c が与えられたとき、n の評価値が変化しうる最小値と最大値をそれぞれ $V_{\min}(n,c), V_{\max}(n,c)$ と書くことにする。再び図 2.20 で考えると、共謀数が 1 の場合、つまりどれか 1 つのリーフの値だけを変更できる場合は A の評価値は 1 から 5 まで変化しうるので $V_{\min}(A,1)=1, V_{\max}(A,1)=5$ となる。同様に共謀数が 2 の場合は、$V_{\min}(A,2)=0, V_{\max}(A,2)=8$ となる。

次に共謀数探索のアルゴリズムについて説明する。まず探索パラメータとして、共謀数 c と目標とするルートの評価値の変動幅を Δ を入力して探索を開始する。共謀数が与えられたときの評価値の変動幅 $V_{\max}-V_{\min}$ によって、評価値の不確かさを表現できていると考えられるため、共謀数探索ではルートにおいて、$V_{\max}(\mathrm{root},c)-V_{\min}(\mathrm{root},c)\leq\Delta$ となるまで評価値の不確かさの原因となっているリーフを選択し展開を続ける。展開するリーフを選択するために、まず $V_{\max}(\mathrm{root},c)$ と $V_{\min}(\mathrm{root},c)$ のうち、ルートの現在の評価値から離れている方を選択する。図 2.20 では共謀数が 1 の場合も 2 の場合も $V_{\max}$ の方が離れている。その後、その評価値の変動を防ぐために、その原因となっているリーフを展開する。図 2.20 で共謀数が 1 のときは、H の変動がルートの評価値が 5 まで大きくなる原因となっているので H を展開する。同様に共謀数が 2 のときは、E と F のどちらかを展開する。リーフ選択の手続きを細かく述べると、あるノード i の評価値が v まで大きくなるのを防ぐ場合には、Max ノードでは子ノードの中から $C(j,v)$ が最大となる子ノード j を選択、Min ノードでは $C(j,v)>0$ を満たす子ノード j をどれか一つ選びながら木を下っていけば、展開するべきリーフにたどり着ける。あるノード i の評価値が V まで小さくなるのを防ぐ場合は、Max ノードと Min ノードを逆にすればよい。

共謀数の考え方を $\alpha\beta$ に取り入れた深さ優先探索である alpha-beta-conspiracy search [76] という手法も提案されている。

共謀数探索そのものの研究はほとんどないものの、近年でも盛んに研究されている詰探索のアルゴリズムは、共謀数から考案された証明数探索 [11] が基本となっている。

2.3.5 Bayesian Search

ノードの評価値の不確かさを分布で表現した最良優先探索の手法として bayesian search [13] を紹介する。Bayesian search では、ルートの評価値の期待値と子ノードの評価値の期待値の差に注目する。この評価値の差が大きいほどルートの評価値が探索する深さとともに大きく変わってしまう不安定なものだと考えられる。この差を大きく変化させるようなリーフを情報が多く得られる重要なリーフだと判断し、展開していく。

具体的な例を使って baysian search の探索手法を説明する。なお、文献 [107] が理解しやすく、ここではそれに基づいて説明する。

Bayesian search ではリーフの評価値を確率分布で表現する。この確率分布はいくつかの評価値にスパイクが立っているような確率質量分布であり、これは前もって学習させておく。そしてリーフから順に、子ノードの評価値の分布をミニマックス探索を考慮して次々と親ノードへとルートまで伝搬させていく。ここで、式 (2.2) が親ノードが Max ノードの場合、式 (2.3) が親ノードが Min ノードの場合に、親ノードの確率分布を計算する方法である。

$$P_p(X = x) = P_c(X = x) \prod_{i \neq c} P_i(X < x) \tag{2.2}$$

$$P_p(X = x) = P_c(X = x) \prod_{i \neq c} P_i(X > x) \tag{2.3}$$

ただし、X を評価値を表す確率変数、x を一つの評価値、$P_n(X)$ をノード n の評価値の確率分布、p が親ノード、c が評価値 x をとる可能性のある子ノードの一つ、i は c 以外の子ノードをそれぞれ表す。

すべてのノードの確率分布を求めてしまったら、次に、ルートと、ルートの子ノードすべてに対して評価値の期待値を求める。ノード n の評価値の期待値を μ_n とすれば、以下のようになる。ただし、INF は十分大きい整数である。

$$\mu_n = \sum_{x=-\mathrm{INF}}^{\mathrm{INF}} x P_n(X = x) \tag{2.4}$$

そして、評価値の期待値が最も大きい子ノードを最善の子ノードとして、ルートと最善の子ノードで評価値の期待値の差 U を求める。

$$U = \mu_{\mathrm{root}} - \max_i \mu_i \tag{2.5}$$

ここで求めた U は U_{before} と呼ぶことにする。図 2.21 にここまでの操作を行った例を示す。評価値の確率は中括弧の中で各評価値の右下に確率を付けることで表現している。また、小括弧の中は各ノードの評価値の期待値を示している。評価値の期待値を求めるのはルートとその子ノードだけであることに注意する。図 2.21 の例において、最善の子ノードは期待値が 8 の S(あるいは T) なので、U_{before} は $\frac{287}{32} - 8 = \frac{31}{32}$ となる。

次に、どのリーフを展開するかを決定するために、それぞれのリーフについて、以下の計算を行う。

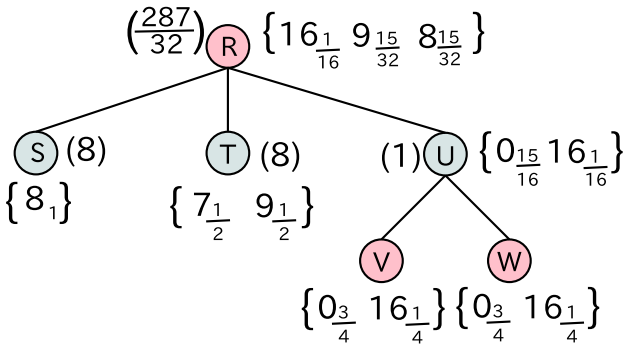

図 2.21: Bayesian search の例

1. ある一つのリーフ L の確率分布を、評価値を一つに固定した分布にする。
2. 残りのリーフの評価値の分布はすべてそのまま用いて U を再計算する。
3. 以上を L の確率分布に含まれるすべての評価値について行い、$|U_{\text{before}} - U|$ の評価値の確率の重み付け平均をリーフ L の $\Delta_L U$ とする。

具体的に図 2.21 の例を用いて計算してみる。まずリーフ T を探索したときを考える。T を調べると評価値が 7 であることがわかるとする。このときルートの評価値の分布は $\{8_{15/16}, 16_{1/16}\}$ となり、期待値は $\frac{17}{2}$ なので、$U = \frac{17}{2} - 8 = \frac{1}{2}$ となる。次に T を調べると評価値が 9 であることがわかるとする。このときルートの評価値の分布は $\{9_{15/16}, 16_{1/16}\}$ となり、期待値は $\frac{151}{16}$ である。また、このときは T の評価値の期待値が大きくなって 9 になったため、最善の子ノードが S から T に変化していることに注意すると、$U = \frac{151}{16} - 9 = \frac{7}{16}$ となる。したがって、T の評価値がそれぞれ 7、9 となる確率で重みをつけた $|U_{\text{before}} - U|$ の平均を計算すると、$\frac{1}{2}|\frac{31}{32} - \frac{1}{2}| + \frac{1}{2}|\frac{31}{32} - \frac{7}{16}| = \frac{1}{2}$ となり、これを $\Delta_T U$ とする。次に、リーフ V を展開したときを考える (V と W は同じなので V だけ考えれば十分である)。まず、評価値が 0 であることがわかったとすると、ルートの分布は $\{8_{1/2}, 9_{1/2}\}$ となり、最善の子ノードは S のままなので、$U = \frac{17}{2} - 8 = \frac{1}{2}$ となる。次に、評価値が 1 であることがわかったとすると、ルートの分布は $\{8_{3/8}, 9_{3/8}, 16_{1/4}\}$ となり、最善の子ノードは S のままなので、$U = \frac{83}{8} - 8 = \frac{19}{8}$ となる。したがって、$\Delta_V U = \frac{3}{4}|\frac{31}{32} - \frac{1}{2}| + \frac{1}{4}|\frac{31}{32} - \frac{19}{8}| = \frac{45}{64}$ となる。最後にリーフ S を展開したときを考えるが、この場合はなにも変化がないので明らかに $\Delta_S U = 0$ である。

各リーフについて ΔU の計算が終了したので、あとは $\Delta_L U$ が最大となるリーフ L を展開するだけである。図 2.21 の例では、$\Delta_V U$ が最も大きかったので V を展開することになる。

Bayesian search が次に展開するリーフの選択基準について、ここまでの例を用いて考えてみる。すでに述べたように、bayesian search ではルートとその子ノードの評価値の期待値の差 U に着目している。この差がルートの評価値の不安定さを表していると考えられるからである。そして、U が大きく変化するリーフこそルートの評価値に大きな影響を与えると考えている。ここで注目するべきなのは、U が小さくなる変化はルートの評価値が安定するという意味でもちろん重要だが、U が大きくなる変化もルートの評価値が安定しなくなるという情報が得られるという意味で重要だと捉

えている点である。図 2.21 では、次に V を展開することになったが、これは、もし V の評価値が 16 だったならば、ゲーム木に大きな影響を与えると考えられるからである。この影響は U を大きくするものだったことに注意すると、bayesian search のリーフ選択基準は納得のいくものであると考えられる。

$\alpha\beta$ アルゴリズムと比較した bayesian search の性能を述べる。ゲームがオセロの場合に、著者らが自分たちで実装した $\alpha\beta$ アルゴリズムを用いたプログラムに、bayesian search を用いたプログラムが持ち時間が同じという条件で勝ち越した。既存のオセロプログラムに対しても、当時 2 番目に強かったプログラムと互角であったが、最も強かったプログラムである Logistello に対しては勝ち越すことができなかったと報告されている。

なお、bayesian search のように評価値の不確かさを分布で表し、それを親ノードへと伝搬させていく手法は、モンテカルロ木探索でも研究が行われている [102]。

2.3.6 ミニマックス探索に対する探索順序付けのまとめ

ミニマックス探索に対して各ノードに優先度を付け、探索順序を決定する手法を紹介した。各手法が何に着目して探索を行っていたかをまとめて以下に示す。

- Best-first minimax search
  現在最善だと考えられる子ノードを中心に探索
- B* search
  最善の子ノードの評価値と 2 番目に良い子ノードの評価値を引き離すための目標値に近づく可能性の高い子ノードを探索
- 共謀数探索
  評価値を大きく変動させることができる子ノードを探索
- Bayesian search
  探索前と探索後で、評価値の不安定さに対する情報が最も得られる子ノードを探索

また、深さ優先でない探索順序を用いたプログラムと $\alpha\beta$ アルゴリズムを用いたプログラムとを比較すると、対戦では時間制限が同じ条件ではよい性能は得られていない。しかし、bayesian search に関しては、$\alpha\beta$ アルゴリズムに匹敵する性能が期待できる。また、これらの評価では、プログラムには、$\alpha\beta$ で使われているようなヒューリスティックなどの手法が組み込まれておらず、このようなヒューリスティックを含めた探索の優先度付けの研究が進めば、$\alpha\beta$ より強くなる可能性は十分に考えられる。

第3章　評価関数とそのパラメータ調整

3.1　評価関数

評価関数は、局面を受け取って、その局面の有利不利を判断し、数値化したものを出力する関数である。ミニマックス探索では、ある深さで探索を打ちきるが、そのときのリーフ局面の評価を行うために評価関数が必要である。コンピュータゲームプレイヤを強くするためには、評価関数の精度が高いことが求められる。

評価関数は、重みベクトル $\boldsymbol{w}$ と局面 s の特徴ベクトル $\boldsymbol{\phi}(s)$ の関数として、$f(\boldsymbol{w}, \boldsymbol{\phi}(s))$ と書ける。このとき、特徴ベクトル $\boldsymbol{\phi}$ は、例えば、将棋において、歩を一枚持っていたら、ある要素が 1 となるようなベクトルである。従来、特徴ベクトルも重みベクトルも人間の手によって決められていた。研究としては、機械学習によってこれらを自動的に決定しようとする試みが行われている。

3.2　評価関数の学習に関する関連研究

コンピュータゲームプレイヤのパラメータを調整する研究が数多く行われている [40, 68, 104] 中でも、評価関数のパラメータの自動調整が主に研究されている [28, 47, 54, 66, 100, 108]。学習手法としては、人間の上級者の棋譜を使う教師あり学習も研究されている [47, 54, 100, 108]。比較学習 (comparison training) は、上級者の指し手のミニマックス値が他の手より良くなるように評価関数のパラメータを調整する手法であり、Deep Blue にも取り入れられている [100]。将棋では、保木が同様の手法を将棋プログラム Bonanza に組み込み、コンピュータ将棋選手権で優勝を収めている [127]。これは将棋において評価関数のパラメータの自動調整がうまくいったはじめての例であった。今日ではトップレベルの将棋プログラムのほぼ全てが比較学習（あるいはその派生手法）を評価関数の調整に用いている。

比較学習の概要を図 3.1 に示す。入力としては、訓練局面と正解の指し手が与えられる。まず、局面の各合法手について、手を進めた先の局面を探索して評価値を得る。このとき、正解の指し手の先の局面の評価値が他の合法手よりも高くなるように評価関数のパラメータを調整する。

比較学習では棋譜の手の質が特に重要となる。つまり、訓練データ中の手は「正しい」手だとみなせるほど信頼できるものでなければならない。金子は棋譜の質が、学習した結果得られるプレイヤの強さにどう影響するのかを、将棋プログラムを用いて調べた [120]。質の異なる三種類の棋譜集合を用いて、評価関数のパラメータをそれぞれ調整している。用いた棋譜集合は、プロの棋譜、アマチュアの棋譜、コンピュータプレイヤの棋譜であり、それぞれ 10,000 局から成る。コンピュータプレイ

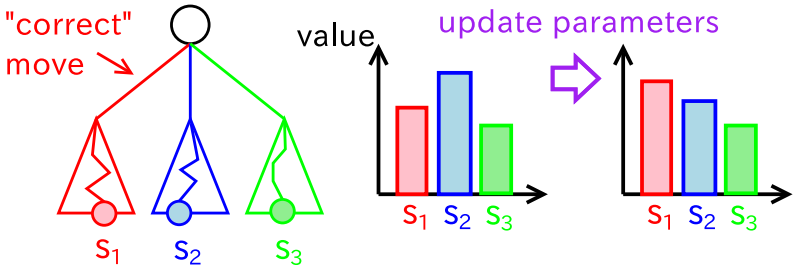

図 3.1: 比較学習の概要

ヤの棋譜は将棋サーバの *floodgate*[1]からレーティングが高い 4 つのプログラムの棋譜を用いており、対局は制限時間 30 分で行われたものである。結果としては、3 種類の棋譜集合の中で、プロの棋譜を用いたプレイヤが最も強くなったことを報告している。コンピュータプレイヤの棋譜を用いて学習したプレイヤはアマチュアの棋譜を用いて学習したプレイヤより強くなったことが報告されている。

比較学習以外でも評価関数の自動調整の手法は数多く存在する。ここではコンピュータプレイヤの探索結果を用いたものに焦点を当てて紹介する。Lee らはオセロにおいて、初期局面から 20 手をランダムに進めることで生成した局面からの自己対戦によって得られたデータから、評価関数の調整を行う手法を提案している [66]。Buro はゲーム木探索で評価される局面を抽出し、その局面に対してコンピュータプレイヤを用いてさらなる探索を行うことで評価値を付けるということを行っている [27]。ゲームが終了すると勝ち負けの情報が得られるため、コンピュータプレイヤは自己対戦を行い、その結果を用いて学習を行うこともできる。自己対戦を用いる手法は強化学習 [99] の文脈で広く用いられており、バックギャモン [101]、チェス [14, 109]、将棋 [15] で適用例がある。評価関数のパラメータ調整に進化計算を用いた手法も提案されており、個体の適応度を評価するために自己対戦が行われている [22, 38]。つまり勝率が高い個体を生き残らせる。Tabibi らは別のコンピュータプレイヤを真似するコンピュータプレイヤを作る手法を提案している [33]。あるコンピュータプレイヤを用いて上級者の局面にミニマックス値を割り当て、評価関数が出力する値がそのミニマックス値になるべく近づくように遺伝的アルゴリズムを用いて、別のコンピュータプレイヤの評価関数のパラメータを調整している。

3.3 激指における評価関数の学習手法

本研究は、将棋プログラムである激指を実験に用いているので、激指が行っている比較学習について述べる。激指は次式で表されるような線形の評価関数を用いている。

$$f(\boldsymbol{w}, s) = \boldsymbol{w}^T \boldsymbol{\phi}(s) \tag{3.1}$$

ただし、$\boldsymbol{w}$ は重みベクトル（調整するパラメータ）、$\boldsymbol{\phi}(s)$ は局面 s に対する特徴ベクトルである. 特徴ベクトルの要素数は全部で約 1,900,000 である。激指の比較学習では平均化パーセプトロン (Averaged

[1] http://wdoor.c.u-tokyo.ac.jp/shogi/floodgate.html

Perceptron) [32] と呼ばれるオンライン学習手法を用いている。まず、重みベクトルの初期値として、各駒の重みなどのいくつかの要素についてはヒューリスティックな値とし、その他の重みは 0 にする。パーセプトロンの更新式は次式で与えられる。

$$\boldsymbol{w}^{(t)} = \boldsymbol{w}^{(t-1)} + \frac{1}{|S^{(t)}|} \sum_{s_i \in S^{(t)}} (\boldsymbol{\phi}(s_1) - \boldsymbol{\phi}(s_i)) \tag{3.2}$$

ただし、t は更新回数、s_1 はプロの手を進めた後の局面から探索した最善応手手順の先のリーフ局面、s_i $(i \geq 2)$ はプロが選ばなかった手を進めた後の局面からの最善応手手順の先のリーフ局面である。$S^{(t)}$ は t 回目の更新において、プロの手より良いと判断されたか、あるいは、あるマージンの値以内しかプロの手との評価値の差がなかった手の後の最善応手手順の先のリーフ局面の集合であり、次式で表される。

$$S^{(t)} = \{s_i | i \geq 2 \wedge f(\boldsymbol{w}^{(t-1)}, s_1) < f(\boldsymbol{w}^{(t-1)}, s_i) + m\} \tag{3.3}$$

$S^{(t)}$ に含まれる局面の数を $|S^{(t)}|$ と表記した。m はマージンであり、激指では局面の進行度が大きく終盤に近いほどマージンも大きくとられている。最終的に出力される重みベクトルは、学習の途中で得られた重みベクトルの平均ベクトルとして、次のように計算される。

$$\boldsymbol{w}^* = \frac{1}{T+1} \sum_{t=0}^{T} \boldsymbol{w}^{(t)} \tag{3.4}$$

ただし、T は学習における更新回数の合計である。

第4章　大規模計算環境を用いた評価関数のパラメータ調整

本章では、まず 4.1 節で、より精度の良い評価関数を得るために、コンピュータプレイヤ自身に訓練データを生成させ、それを学習に用いる手法について述べる。その後 4.2 節で、学習時間を短縮するために、評価関数のパラメータ調整に、学習の並列化手法を適用することを提案し、その実験結果を示す。

4.1　訓練データの自己生成

本節では、比較学習において、訓練データの量が限られているという問題点を解決するために、学習データを追加する手法を提案し、激指を用いてその有効性の評価を行った。評価指標としては、得られた評価関数を用いたプレイヤの強さを評価した。

本節の構成は以下のようになっている。まず、4.1.1 節では、比較学習の問題点を解決するために、コンピュータプレイヤの探索結果を用いて訓練データを自己生成することが有力な方法であることを述べる。次の 4.1.2 節では今回行った局面と手の生成方法と、それらを学習にどのように用いるかについて説明する。激指を用いて得られた実験結果を 4.1.3 節に示し、最後に 4.1.4 節でまとめと今後の課題を述べる。

4.1.1　背景

評価関数のパラメータ調整において、比較学習 [100] は将棋において有効な手法であることが分かっており、研究が進められている [47, 54]。比較学習は教師あり学習の一つであり、人間の上級者が選んだ手を「正しい」とみなし、コンピュータプレイヤがそのような手をなるべく多く選ぶように評価関数のパラメータを調整する。他の教師あり学習の手法と同様に、比較学習の性能は学習データの質と量によって決まり、人間の上級者の棋譜は、質の良い学習データとして用いられる。しかし、その数には通常限りがある。この問題点を解決するために、コンピュータプレイヤ自身によって学習局面数を増やす手法の研究が求められる。

コンピュータゲームプレイヤはすでにいくつかのゲームにおいて人間のトッププレイヤを越える実力を持っている（Backgammon：[101]、Othello：[28]、Chess,[29]）。将棋や囲碁のようなより複雑なゲームについては、コンピュータプレイヤの強さは人間のトップを越えるほどではないが、アマチュアを越える程度にはなっている。これらのゲームにおいては、コンピュータプレイヤが深く探索

した結果選んだ手は人間のプレイヤと同程度に良いと考えられる。もしそうならば、コンピュータプレイヤによって得られた局面と手は比較学習における訓練データに使うことができるはずである。このアプローチの問題点の一つはそのような訓練データを作るのには計算コストがかかるということである。例えば、一回の探索で 100 秒かけるとする。現在将棋ではプロの棋譜が数百万局面存在するため、これに追加して十分意味のある学習を行うために、同程度の数の局面を生成しようとすると、一台の計算機では 1000 日以上かかってしまう。しかし、この手続きは容易に並列化可能であり、計算時間の問題はクラスタ環境や Amazon EC2 といったクラウドサービスを使うことで解決可能である。

4.1.2 手法

コンピュータプレイヤ自身を使って、比較学習のための訓練局面を生成する手法として、Self-play、Leaf、Random の三種類の手法を提案する。

最初の二つは実際のゲームに登場する局面に近い訓練局面を生成することを期待したものである。原則として、生成される局面は学習する価値のある局面である必要がある。評価関数が呼ばれるのは実際のゲームに登場する局面であり、そのような局面に対して特徴の重みを調整する必要があるため、実際のゲームに出てくる局面を用いる必要がある。極端な例として、全ての駒をランダムに置いて得られる局面を考える。このような局面は実際のゲームには登場せず、実際のゲームに登場する局面の特徴を持たないため、学習にはほとんど役に立たない。以下では最初の二つの手法についてさらに具体的に説明する。

Self-play コンピュータプレイヤの自己対戦に登場する局面を学習局面に用いる。これらの局面は実際のゲームにも登場すると考えられる。訓練局面の多様性を確保するために、自己対戦のための初期局面としてプロの棋譜に現れる局面を用いた。

Leaf プロの棋譜に現れる局面についてゲーム木探索を行い、そのリーフノードの一部を訓練局面として用いた。このように探索木のリーフノードを用いるという手法は、比較学習に用いられたわけではないが、Buro によって既に提案されている [27]。この手法は、訓練局面は、実際のゲーム木探索で呼ばれる評価関数が評価する局面と同じ特徴を共有するべきだという考えにも基づいたものである。

三つ目の手法は主にベースラインとして用いるためのものである。最初の二つの手法と違い、この手法で生成される局面のほとんどは実際のゲームに現れる局面に似た性質を持たないと考えられる。

Random プロの棋譜に現れる局面から 2 手ランダムに進めることで得られる局面を訓練データとして用いる。ランダムに手を進めるといっても、実際のゲームに現れる局面から離れすぎないようにしたかったため、ランダムに進める手数は少なくした。2 手としたのは 1 手ではランダムに手を進めた方のプレイヤが極端に不利な局面になると考えられるからである。

局面を生成した後は、その各局面について「正しい」手を、比較的長い時間をかけたコンピュータプレイヤ自身の探索によって計算した。得られた局面と手を比較学習に用いる。

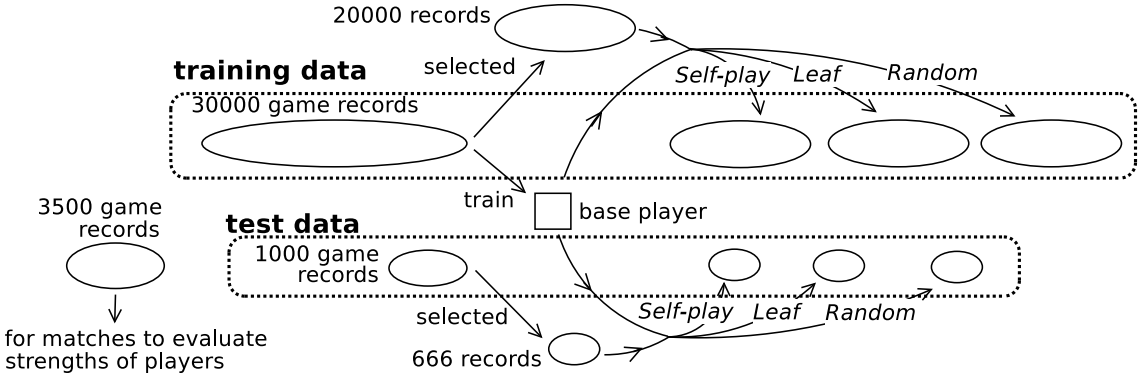

図 4.1: 訓練データ生成の概観

学習時には、訓練データにある「正しい」手をコンピュータプレイヤが着手できるように、評価関数のパラメータ調整を行う。しかし、機械学習では、訓練データに過度に適合してしまうと、未知のデータに対する性能が下がってしまうこと（過学習）が一般に知られている。すなわち、強いコンピュータプレイヤを実現するためには、適切な程度に学習ができたところで打ち切る必要がある。そこで、訓練データとは別にテストデータも準備し、そのテストデータに対する性能を見ながら学習を行うことで、適切なところで学習を打ち切ることにした。激指の学習では、訓練データ中の各局面を全て一度ずつ見て、1 イテレーションとする。各イテレーションの後に、それまでに得られた重みベクトルから式（3.4）により平均重みベクトルを求め、その平均ベクトルを用いてテストデータの局面に対して「正しい」手が選べたかどうかの一致率を調べる。学習をある程度続けると一致率は過学習により下がり始めるが、このとき一致率が最大となるイテレーションの結果を用いた。つまり、最終的に出力される重みベクトルは各イテレーションで求めた平均重みベクトルのうち、「正しい」手を最も多く選べたものとなる。

4.1.2.1　データ生成と訓練

図 4.1 に訓練局面とその正解となる手がどのように生成されるかの概観を示す。はじめに、3.3 節で述べた激指の学習手法を用いて、プロの棋譜から通常どおりコンピュータプレイヤを学習させる。このプレイヤを base player と呼ぶ。次に、base player を用いて、Self-play、Leaf、Random の三種類の訓練データとテストデータを生成する。最後に、各訓練データを二種類の方法で用いて二つのプレイヤを学習させる。一つは生成した訓練データとプロの棋譜を両方用いて学習させたプレイヤである。このプレイヤは生成された訓練データがプレイヤの強さを向上させることができたかどうかを評価するために用いた。もう一つは生成した訓練データだけを用いて学習させたプレイヤである。このプレイヤは生成された訓練データの質をプロの棋譜と比較しながら評価するために用いた。

Self-play の局面生成手法を図 4.2 に示す。プロの各棋譜の最初の 30 手を進めて得られた局面から自己対戦を開始する。同じ局面を重複して訓練データに追加するのを避けるために、該当棋譜にすで

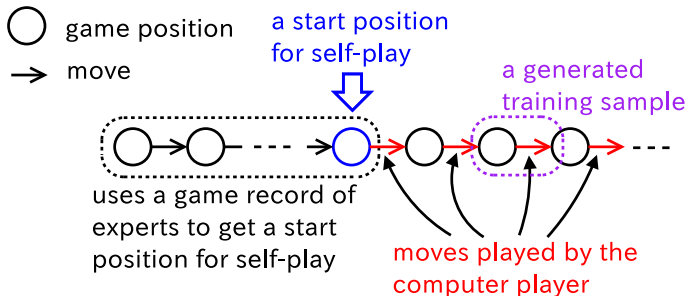

図 4.2: Self-play の局面生成手法

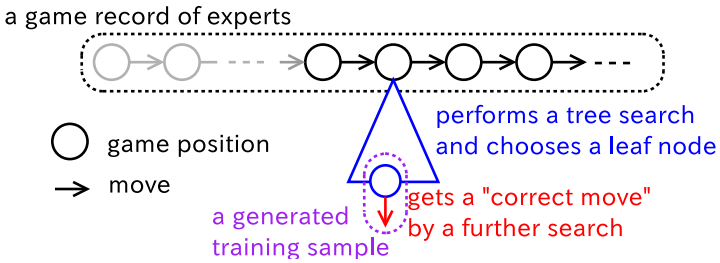

図 4.3: Leaf の局面生成手法

に現れていた局面や、その自己対戦中にすでに現れた局面は、訓練データに追加しないことにした。

Leaf の局面生成手法を図 4.3 に示す。プロの各棋譜の 36 手目以降の各局面について base player を用いて深さ 14 の探索を行う。激指では中盤の局面を深さ 20 で探索するのにおよそ数十秒かかり、これが実戦では望ましい探索深さである。学習時の探索深さが 6 であることから、$20 - 6 = 14$ として、生成のための探索深さを 14 に決定した。探索した後は探索木のリーフ局面からランダムに一つ選び、訓練データとした。

Random の局面生成手法を図 4.4 に示す。プロの各棋譜の 36 手目以降の各局面について、ランダムに 2 手進めて局面を得る。2 手のうち、少なくとも片方が棋譜の手と異なった場合に、訓練データに追加する。

訓練データに追加する局面の多様性を確保するために、プロの棋譜の最初の 36 手目以降の局面を対象に訓練局面の生成を行っている。将棋では 36 手後の局面でも通常は序盤である。Leaf や Random では 36 手目以降の局面を用いているのに対し、Self-play では 30 手目以降の局面を対象にしているのは、自己対戦ではコンピュータプレイヤがプロ棋士と同じ手を指す可能性を考慮すると、プロの棋譜と違う局面を得るためには、36 手より少し少なめに設定する必要があったためである。

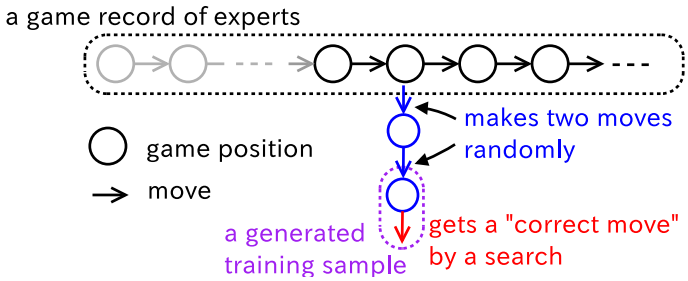

図 4.4: Random の局面生成手法

表 4.1: 用いたクラスタの各ノードの情報

CPU	clock speed	memory / node	core / node	# of nodes
Xeon E5530	2.40GHz	24GB	8	127
Xeon E5620	2.40GHz	24GB	8	16
Xeon X5687	3.60GHz	24GB	8	4
Xeon E5-2665	2.40GHz	128GB	16	32

4.1.3 評価

4.1.3.1 実験設定

用いた実験環境について説明する。本章だけではなく、本論文全体で同じクラスタ環境を用いており、その各ノードの情報を表 4.1 に示す。ノードは 10Gbps のイーサネットで接続されている。実験では表 4.1 で示す計算ノードをすべて用いた。コア数は全部で 1,688 コアである。

プロの棋譜は訓練データとして 30,000 棋譜、テストデータとして 1,000 棋譜用いた（図 4.1）。4.1.2 節で述べた各データ生成手法について、訓練局面をプロの 20,000 棋譜から、テストデータをプロの 666 棋譜から生成した。訓練局面を base player で探索し、比較学習で用いるための正解の手を得た。このときの探索はノード数で制限し、探索ノード数は、30,000,000、3,000,000、300,000 の三種類とした。30,000,000 ノードの探索は激指で Xeon E5530 (2.40 GHz) の 1 コアで 120 秒程度かかり、信頼できる指し手を得ることができ、かつ、十分な量の訓練データを生成することができる程度の探索時間だと判断したため用いた。探索によって詰みが見つかった局面は訓練局面から除いた。表 4.2 に探索ノード数 30,000,000 で指し手を生成したときの生成した訓練局面数を示す。

本研究では、全ての学習において比較学習のために行う探索の深さを 6 としている。つまり、各合法手を進めてから先の局面は深さ 5 で探索した。

学習させた結果のプレイヤの強さを評価するために、プロの 3,500 棋譜を別に用いた。これらの棋譜は二つのプレイヤの対局のための初期局面を決定するために用いた。具体的には、各棋譜の 36 手

表 4.2: 訓練データ数の局面数

training data	# of positions
experts' 30,000 game records	2,859,698
Self-play from 20,000 game records	1,862,518
Leaf from 20,000 game records	1,287,870
Random from 20,000 game records	1,381,263

目後の局面を対局の初期局面に選んだ。各初期局面について対局は先手後手を入れ替えて 2 局行い、全部で 7,000 局の対局を行った。200 手を越えて決着がつかない場合は引き分けとし、勝率の計算からは除いた。対局では手を一つ決定するための探索ノード数を 100,000 に制限して行った。

4.1.3.2　プロの棋譜と生成したデータを両方用いた評価

プロ棋士の 30,000 棋譜と生成した訓練データを両方用いて学習プレイヤの勝率を評価した。訓練データやテストデータを生成するときの探索ノード数は 30,000,000（およそ 120 秒）とした。テストデータはプロ棋士の棋譜 1,000 局面と、訓練データと生成されたテストデータのうち訓練データと同じ方法で生成されたものの両方からなる。例えば、訓練データがプロ棋士の棋譜と Leaf と Random のデータから構成されているなら、テストデータもプロ棋士と Leaf と Random のデータから構成される。評価のための対局はそれぞれ 7,000 局行われる（4.1.3.1 節）。勝率を求める基準となる対戦相手のプレイヤは、2 種類準備した。一つはプロ棋士の 30,000 棋譜だけから学習したプレイヤ（つまり base player である）であり、もう一つはプロ棋士の 40,000 棋譜を用いて学習したプレイヤである。前者のプレイヤは追加した訓練データによって bese player から強さが向上しているかを確かめるために用いた。後者のプレイヤは生成した訓練データを追加して学習したプレイヤと、プロ棋士の 10,000 棋譜を追加して学習したプレイヤの強さを比較するために用いた。つまり、追加した訓練データによる強さの向上をプロ棋士の 10,000 棋譜による強さの向上と比べることを目的とした。

結果を表 4.3 に示す。訓練データを生成するときに「正しい」手を決めるのに行った探索ノード数は 30,000,000 である。表にはプロ棋士の 40,000 棋譜を用いて学習したプレイヤ（表では 10,000 game records と表示）と base player との対局結果も比較のために載せてある。表では、二つのプレイヤの強さが二項検定で統計的に有意に違うときは、勝率に*をつけており、*は有意水準 0.05、**は有意水準 0.01、***は有意水準 0.001 で差があることを示す。同じプレイヤ同士の勝率は理論値である 50.00%を載せている。

表 4.3 によると、Leaf のデータを用いたプレイヤと Leaf のデータと Self-play のデータを両方を用いた 2 つのプレイヤが base player より有意に強い。他のプレイヤの中にはプロ棋士の 40,000 棋譜を用いたプレイヤに有意に負け越しているのもあるのに対し、これらのプレイヤはに有意には負け越していない。さらに、Leaf のデータを用いなかったプレイヤは base player とは有意な強さの差が

表 4.3: プロ棋士の 30,000 棋譜と生成した追加訓練データを両方用いたプレイヤ 2 つのプレイヤに対する勝率

additional training data	vs 30,000 records (%)	vs 40,000 records (%)
10,000 game records (40,000 records in total)	***52.76	50.00
Self-play	49.92	***47.12
Leaf	*51.51	48.85
Random	49.04	**48.33
Self-play + *Leaf*	*51.27	48.91
Self-play + *Random*	50.56	48.89
Leaf + *Random*	50.48	50.16
Self-play + *Leaf* + *Random*	50.50	50.29

表 4.4: プロ棋士の 30,000 棋譜と Leaf のデータを両方用いたプレイヤの 2 つのプレイヤに対する勝率

additional training data	vs 30,000 records (%)	vs 40,000 records (%)
Leaf from 20,000 game records	51.51	48.85
Leaf from 30,000 game records	51.52	49.68
Leaf from 40,000 game records	49.79	49.31
Leaf from 50,000 game records	50.49	49.01

見られなかった。これらの結果は Leaf のデータ生成手法が 3 つの生成手法の中で最も効果的であったことを示している。

Leaf のデータは学習に有効であることが分かったため、追加実験として、Leaf のデータをさらに生成することを行った。生成するときに使うプロ棋士の棋譜は 20,000 棋譜から増やさず、一つの探索木から二つのリーフノードを別に選ぶことで局面を追加している。Leaf のデータの追加量を変えながらプレイヤを学習させた。結果を表 4.4 に示す。表 4.3 の 20,000 棋譜から生成した Leaf のデータの結果も示している。直感に反し、Leaf のデータを増やしても強さの向上にはつながっていない。

4.1.3.3　異なる訓練データの効果についての分析

前節の実験結果の原因を調べるため、訓練データ生成したプレイヤの強さと訓練局面の種類という異なる二つの観点から訓練データの影響をさらに分析した。

まず、プレイヤの強さが、そのプレイヤによって生成された訓練データを用いて学習したプレイヤ

の強さにどのように影響するかについて調べた。すなわち、次の疑問点について考察するために実験を行った。

- プロ棋士の中でも強さに明らかな違いが存在する。強いプロ棋士の棋譜を用いて学習したプレイヤは、それより強くないプロ棋士の棋譜を用いて学習したプレイヤより強いのか？
- コンピュータプレイヤは探索深さが深いほど強くなる。深い探索で生成された訓練データは浅い探索で生成された訓練データよりも学習に有効なのか？
- コンピュータ将棋プレイヤはすでに人間のプロ棋士と同程度に強い可能性がある。コンピュータプレイヤによって生成された訓練データだけを用いたプレイヤは、プロ棋士の棋譜を用いて学習したプレイヤと同程度の強さになるのか？

評価のために、以下の 4 種類の訓練データを追加で準備した。

Experts プロ棋士の棋譜の 36 手目以降の局面を選んだもの。プロ棋士が指した手を「正しい」とみなして学習に用いる。

High-rating experts (High-rating) Experts のデータに含まれる局面のうち、レーティングが高いプロ棋士の局面と指し手を選んだ。Experts の局面のうち、52%の局面を選択した。この割合にしたのは、学習局面数を確保するためである。

Low-rating experts (Low-rating) Experts のデータに含まれる局面のうち、レーティングが低いプロ棋士の局面と指し手を選んだ。選んだ局面は Experts の局面のうち 49%である。

Base player 局面は Experts と同じであるが、プロ棋士の指し手の代わりに、base player の探索によって得られた手を「正しい」とみなして学習に用いる。局面は同じだが指し手が違う場合の学習に与える影響を調べることを目的として用いた。

High-rating と Low-rating のデータセットを準備するときのレーティングとしては、次のような簡易的な方法を用いた。プロの棋譜を一つずつ見ていき、その棋譜で勝った棋士のレーティングを R_{win}、負けた棋士のレーティングを R_{lose} とする。このとき、式（4.1）で計算される ΔR を、勝った棋士のレーティングに加え、負けた棋士のレーティングから引くことでレーティングの更新を行う。一つの棋譜を一回ずつだけ用いて計算した。この方法では、レーティングが棋譜を処理する順序に依存し、かつ、強さが時期によって大きく変わった棋士の強さを正確に反映することができないが、実験に必要な精度のレーティングを計算することはできると考えた。

$$\Delta R = \min\{31, \max\{1, 0.04 \times (R_{\mathrm{lose}} - R_{\mathrm{win}})\}\} \tag{4.1}$$

各種類の訓練データについて 1,200,000 局面をプレイヤの学習に用いた。Base player、Self-play、Leaf、Random のデータについては、「正しい」手を得るための探索ノード数を 4.1.3.1 節で述べた通り、30,000,000、3,000,000、300,000 の三通りを試した。強さを評価するために対戦実験を行ったが、ここでも二つの相手プレイヤを準備した。一つは Experts の訓練データを用いて学習を行ったプレイヤ、もう一つは探索ノード数 30,000,000 で生成した Leaf の訓練データを用いて学習したプレイヤである。後者のプレイヤはプロ棋士の棋譜を直接使っているわけではなく、かつ、表 4.3 の中で学習に効果的であることが分かり、基準として適切だと判断したため用いた。

表 4.5: プロ棋士の棋譜を用いて学習したプレイヤの 2 つのプレイヤに対する勝率

training data	vs *Experts* (%)	vs *Leaf* (30,000,000) (%)
Experts	50.00	***55.40
High-rating	50.91	***56.72
Low-rating	*48.42	***54.87

表 4.6: base player の探索によって得られた「正しい」手を学習に用いたプレイヤの 2 つのプレイヤに対する勝率

training data	vs *Experts* (%)	vs *Leaf* (30,000,000) (%)
Base player (300,000)	40.43	46.29
Base player (3,000,000)	42.52	48.56
Base player (30,000,000)	41.78	47.75
Self-play (300,000)	41.85	47.68
Self-play (3,000,000)	42.82	48.42
Self-play (30,000,000)	43.69	49.58
Leaf (300,000)	41.92	46.66
Leaf (3,000,000)	42.23	49.05
Leaf (30,000,000)	44.60	50.00
Random (300,000)	40.03	45.27
Random (3,000,000)	39.38	45.50
Random (30,000,000)	40.63	46.20

プロ棋士の棋譜を用いて学習したプレイヤの勝率を表 4.5 に示す。表中の*は有意水準 0.05、**は有意水準 0.01、***は有意水準 0.001 で統計的に強さに差があることを示す。また、生成した訓練データを用いて学習したプレイヤの勝率を表 4.6 に示す。この表では、「正しい」手を得るために用いた探索ノード数を括弧の中に示している。これらの結果から以下のことが分かる。

- 表 4.5 を見ると、High-rating の訓練データを用いたプレイヤは、Low-rating の訓練データを用いたプレイヤより、強いことが分かる。この結果は金子によって報告されている結果とも一致している [120]。
- 表 4.6 を見ると、深い探索で生成された訓練データを用いたプレイヤは、多くの場合、浅い探索で生成された訓練データを用いたプレイヤより強いことが分かる。探索ノード数 30,000,000 で生成された Base player が唯一の例外である。
- 表 4.5、表 4.6 の両方ともから、プロ棋士の棋譜を用いたプレイヤの方が、プログラム自身が

表 4.7: 将棋サーバの floodgate 上でレーティングの高い激指以外のコンピュータプレイヤの棋譜を用いたプレイヤの 2 つのプレイヤに対する勝率

training data	vs *Experts* (%)	vs *Leaf* (30,000,000) (%)
floodgate	44.51	50.62
Self-play (3,000,000)	43.69	49.58

生成した訓練データを用いたプレイヤより、明らかに強いことが分かる。考えられる理由としては、30,000,000 ノードの探索では、コンピュータプレイヤはプロ棋士ほど強くないというものが挙げられる。

次に、表 4.6 から訓練データの局面の性質が与える影響を見ていく。まず、Random のデータは残りの 3 種類より学習の効果が低いことが分かる。さらに、Self-play のデータと Leaf のデータは同程度に有効であることも分かる。これは、表 4.3 で、プロ棋士の棋譜に追加する訓練データとして用いたときは、Leaf のデータの方が学習に有効だった結果と異なっていると言える。この違いが生じた理由としては、Leaf のデータに含まれる局面はプロ棋士の棋譜に含まれる局面とは性質が異なっており、そのことが学習に影響を与えているというからという可能性がある。逆に Self-play のデータはプロ棋士の棋譜に出てくる局面に似すぎており、しかも生成した訓練データはプロ棋士の棋譜よりそもそも学習効果が小さいことを考慮すると、プロ棋士の棋譜と合わせて用いると効果がなくなるということが考えられる。

しかし、コンピュータプレイヤの実力はプロ棋士と同程度と言われている割には、生成した訓練データを用いて学習したプレイヤは弱すぎると言わざるを得ない。この弱さの原因として、訓練データが「自己生成」だからということが考えられる。自己対戦の影響を排除するため、将棋サーバである floodgate から、レーティングの高い[1]プレイヤの棋譜を抽出した。ただし激指の棋譜は除いた。訓練データとしては消費時間が 5 秒以上の指し手のみを用いた。その中の平均消費時間は 15 秒であった。訓練局面数は 1,200,000 とした。

対戦実験の勝率を表 4.7 に示す。激指の自己対戦と比較するために、探索ノード数 30,000,000 で生成した Self-play のデータを用いたプレイヤの勝率も再掲している。表 4.7 を見ると、floodgate の棋譜を用いて学習したプレイヤは、探索ノード数 30,000,000 で自己対戦して生成した訓練データと同程度の強さであることが分かる。floodgate の平均消費時間は 15 秒であるが、実際にはコンピュータプレイヤが動いている計算環境はそれぞれ異なっているし、並列探索を行っていることも考えられるため、単純に比較することはできない。しかし、それでも、自己対戦で生成するときの探索時間は約 120 秒であることを考えると、「自己生成」という要因が学習に悪い影響を与えているという可能性がある。

[1] 2013 年 6 月 10 日時点でレーティングが 2550 以上のコンピュータプレイヤを対象とした。

4.1.4 まとめと今後の課題

棋譜の数が限られるという比較学習の問題点を解決するため、コンピュータゲームプレイヤの深い探索によって訓練データを生成し、プロ棋士の棋譜と合わせて用いるというアプローチを提案した。また、訓練局面の生成には、それぞれ性質が異なる局面を生成するために 3 つの手法を提案した。実験結果から、プロ棋士の棋譜に追加する場合には、探索木のリーフ局面を学習局面を生成する方法が効果的であることが分かった。これはおそらく実際の探索に登場する局面の性質は保ちつつ、プロ棋士の棋譜に現れる局面とは異なる性質があるためだと考えられる。

さらに、データを生成するプレイヤの強さと、生成される局面の性質の、大きく二つの観点から、訓練データの効果を調べた。深い探索で生成された訓練データを用いて学習したプレイヤは浅い探索で生成された訓練データを用いたプレイヤより強かった。しかし、学習に用いる「正しい」手を得るための探索が 120 秒程度 (30,000,000 ノード）だったのにもかかわらず、自己生成した訓練データを用いたプレイヤは、プロ棋士の棋譜を用いたプレイヤより明らかに弱かった。プロ棋士が選んだ手と同じくらい信頼できる手を選ぶためには探索時間が足りなかった可能性がある。局面の性質という観点からは、ランダムに手を進めて生成した局面は学習には向かないことが分かった。

実験結果から、訓練局面の選択は、学習結果のプレイヤの強さに影響を与えることが分かった。今後の課題の一つの方向性としては、プレイヤの強さを改善するために、より効果的な選択手法を探求するということが挙げられる。例えば、値が少し変わることで最善応手手順が変わるようなリーフ局面を選ぶといったことが考えられる。そのような局面の正確の評価値は最善手を決定するために必要だからである。他には、勝ち負けの情報を用いるということが考えられる。Sato らはテストデータとの指し手の一致率だけではなく、勝ち負けの情報を目的関数に導入する手法を提案している [90]。勝ち負けの情報が学習に重要な局面もあったことが報告されている。この手法はプロ棋士の棋譜だけでなく、自己生成した局面についても同様に有用であると考えられる。

今後の課題のもう一つの方向性として、激指以外の複数のプログラムを用いて訓練データを生成するということが挙げられる。Baxter らはチェスのプログラムを強化学習によって強くしたが、これは、プログラムが人間も含めた様々なプレイヤと対戦できたからだと報告している [14]。本章でも自己生成という要因が学習に悪い影響を与えている可能性を示した。これらの結果を考慮すると、複数のコンピュータプレイヤによって生成された訓練データ用いて学習を行うことで強いプレイヤが作ることができると考えられる。

4.2 評価関数の学習の並列化

本節では、比較学習による評価関数の学習にかかる時間に着目する。評価関数は探索中で用いられるため、学習の結果を用いるときにかかる時間は短くなければならない。これに比べて、評価関数の学習そのものにかかる時間は、実際に用いるときの性能に直接は影響しないため、学習が実行可能な範囲であれば長くても構わない。しかし、評価関数の学習の時間も短ければ短いほど有利である。

学習時間が短ければ、例えば、学習パラメータの調整に時間をかけられるようになる。実際に評価関数を学習させるときには、学習手法に伴う学習パラメータを適切に調整する必要がある。評価関

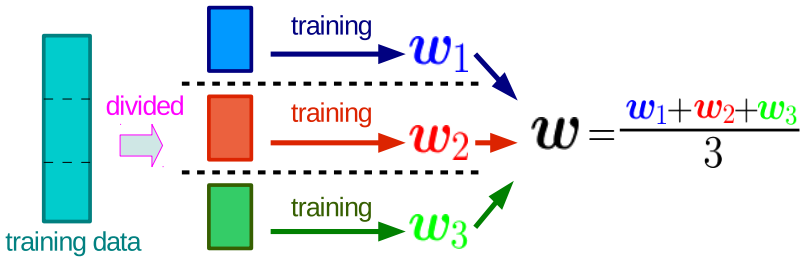

図 4.5: Parameter mixing

数の学習が高速に実行できれば、一回の学習を行い、それに応じて学習パラメータを少し変更し、再び学習を行うというように、学習パラメータの調整が容易になる。さらに、より適切な学習パラメータを選ぶことができるようになるため、学習の精度そのものを高めることができる可能性もある。

学習が高速に実行できるようになると、使える棋譜の数を増やせるという利点も存在する。4.1 節では、訓練データの数を増やす手法を述べてきた。しかし、訓練データの数が増加すると、それに応じて学習時間も長くなってしまう。学習が高速に実行できるようになれば、多くの訓練データを用いて精度のよい学習が可能となる。

ここでは、将棋の評価関数の学習を高速に行うことを目的として、分散計算環境において並列化することを提案する。将棋プログラムとしては激指を用いた。激指の学習手法にはパーセプトロンが用いられている。パーセプトロンについては、ミニバッチを用いた並列化手法が提案されている [117]。そこで、この並列化手法を将棋の評価関数の学習に適用した。学習の精度の評価には、棋譜の指し手との一致率を用いた。64 計算ノードを用いた実験では、1 計算ノードでの学習と比較して、ある基準となる一致率を実現するための学習時間を短縮できることを示した。

4.2.1　学習の並列化に関する先行研究

学習手法は大きくバッチ学習とオンライン学習の二つに分けられる。バッチ学習は、訓練データを全部処理してから重みベクトルを更新する方法である。これに対し、オンライン学習は、訓練例を一つ処理するごとに重みベクトルを更新していく方法であり、大規模データの学習に用いられる [116]。

バッチ学習は訓練データを分割することで容易に並列化が可能である。各プロセスに訓練データの一部分を分配し、プロセスはそれぞれ与えられた訓練データを処理し、最後にその結果を集約すればよい。これに対し、オンライン学習では、訓練例を一つずつ処理するという意味で、本質的に逐次的な処理であるため、バッチ学習より並列化が難しい。

激指では重みパラメータの調整にパーセプトロンを用いていることを 3.3 節で述べた。パーセプトロンはオンライン学習の一つである。ここでは、オンライン学習の並列化に関する先行研究について述べる。オンライン学習を並列化する場合に考慮しなければならないのは、「重みベクトルの更新をどの程度行わなくてもよいか」という点になる。

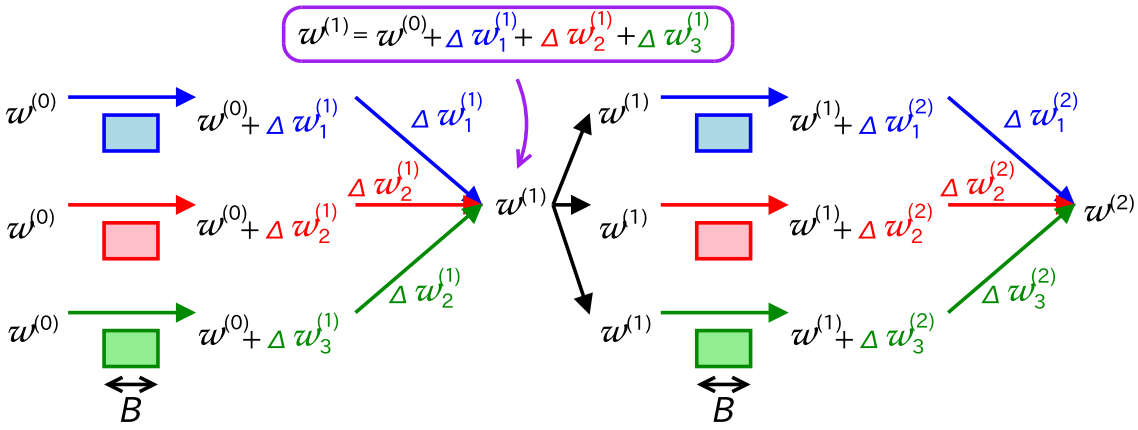

図 4.6: ミニバッチを用いた並列オンライン学習

パーセプトロンの並列化の手法として、parameter mixing [77] がある。Parameter mixing の概要を図 4.5 に示す。各プロセスは訓練データの一部分だけを持つ。そして、自分が持っている訓練データだけを用いて重みベクトルの更新を行う。各プロセスが学習した重みベクトルの平均ベクトルが、最終的なシステム全体の重みベクトルとして出力される。必要ならば、出力される重みベクトルを初期値として、学習を繰り返す (iterative parameter mixing; IPM)。Parameter mixing は、[43] や [52] でも用いられている。

パーセプトロンの並列化を行った他の研究として、[117] がある。これは、訓練データをミニバッチと呼ばれる小さな訓練データに分割し、重みベクトルの更新をこのミニバッチが終わるごとに行う手法である。並列に実行する際には、各ミニバッチをさらに分割して、プロセスに訓練例を分配し、各プロセスが重みベクトルの更新ベクトルを求める。評価には、品詞タグ付けが用いられているが、各プロセスの処理量を均一にするために、各プロセスが処理する文章の長さがなるべく同じになるような工夫をしている。速度向上としては、12 プロセスで 5.6 倍が報告されている。IPM の速度向上は 4 倍だったことから、IPM に対する優位性も報告されている。

パーセプトロン以外にも、確率的勾配法 (stochastic gradient descent; SGD) を非同期に並列化する手法が研究されている [41, 9, 34]。これらは、ワーカは少数の訓練データから独立に勾配ベクトルを求め、マスタがワーカからそれを受け取り重みベクトルを更新していくという手法を採用している。このとき、ワーカは、マスタが持っている本当の重みベクトルではなく、いくらか情報が古い重みベクトルを使って勾配ベクトルを求めているため、この情報がどのくらい古くてもよいのか、といったことが議論されている。

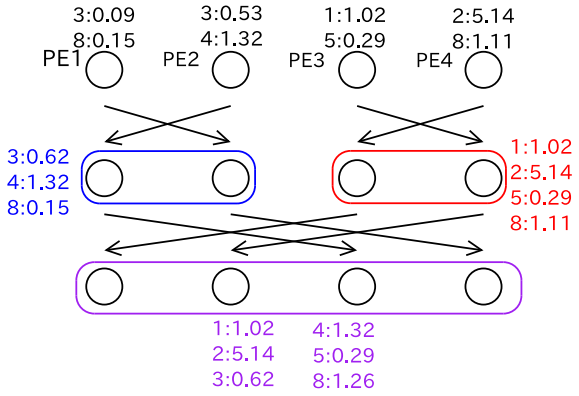

図 4.7: 非零ベクトルの通信の例

4.2.2 提案手法

4.2.2.1 並列パーセプトロンの将棋評価関数への適用

本研究では、コンピュータ将棋の評価関数の学習時間を短縮することを目的として、[117] で提案されているミニバッチを用いた並列パーセプトロンを用いて、学習を分散計算環境で並列化することを提案する。学習の大まかな手続きを図 4.6 に示す。各プロセスは B 個の訓練例を用いて、他のプロセスと独立にパーセプトロンによって重みベクトルを更新した後、更新ベクトルをお互い通信し、そのベクトル和を求め、全体の重みベクトル $w^{(t)}$ を更新する。この全体の重みベクトル $w^{(t)}$ は全てのプロセスが保持する。各プロセスは更新されたベクトルを初期値として、次の B 個の訓練例を処理する。平均化パーセプトロンでは、全体の重みベクトル $w^{(t)}$ の各 t による平均ベクトルを求める。

更新ベクトルの通信の際には、通信量を減らすために非零要素だけを通信する必要がある。ベクトルの重みベクトルの要素数は約 1,900,000 であるが、一つの訓練例の処理で更新される重みベクトルの要素数はずっと少なく、数百から数千程度である。つまり、更新ベクトルはほとんどの要素が 0 のスパースなベクトルである。したがって、重みベクトルをそのまま通信するのではなく、更新ベクトルの非零要素だけを通信することが重要である。非零要素だけを通信するためには、そのインデックスも通信する必要がある。しかし、更新ベクトルが十分にスパースならば、非零要素だけを通信したほうが通信量が少なくて済む。

今回の実装では、非零要素だけを効率よく通信するために、バタフライを用いた集合通信 [85] を実装した。要は、非零要素だけを通信して、MPI_Allreduce() に相当することを行う。図 4.7 に更新ベクトルの和を求める通信の例を示す。更新ベクトルはコロンの左側に重みベクトルのインデックスを、右側に更新量を書いている。二つのプロセスで通信したとき、同じインデックスの要素があれば、その時点で足し合わされる。バタフライにより、全てのプロセスが更新ベクトルの和を得るまでに、一つのプロセスが通信する回数を、プロセス数を P として、$\log P$ 回に抑えることができる。また、通信される要素数は、はじめは少なく、通信のたびに増えていくので、通信量が多いのは最後

の数回の通信のみとなる。

各プロセスでの処理を詳細を述べる。各プロセスは訓練例を一つ処理するたびに、自身の重みベクトルを式 3.2 により、他のプロセスとは独立に更新する。次の訓練例を処理するときには、この更新された重みベクトルを用いる。なお、[117] ではこのローカルな重みベクトルの更新は行われていない代わりに、収束性が保証されている。本研究では、収束性が保証されていないとしても、使うことができる情報は使った方がいいと判断した。重みベクトルの更新の際には、その更新ベクトルも保存しておく。このとき、図 4.7 で示したように、更新を起こした要素のインデックスとその更新量を保持しておく。更新が起こった要素が以前の訓練例でも更新されていたら、その更新量に今の訓練例の分を加えていく。これを繰り返すと、B 個の訓練例を処理した後、更新された重みベクトルの要素のインデックスとその更新量が得られる。これを前述した図 4.7 の方法を用いて、お互いに通信しあう。通信後、すべてのプロセスは各プロセスが求めた重みベクトルの更新量の和を持っていることになるので、これを B 個の訓練例を処理する前のベクトルに加える。この操作により、次の B 個の訓練例を処理する前には、全プロセスは同じ重みベクトルを持っていることになる。

4.2.2.2　ミニバッチの大きさを時間で制限

比較学習による将棋の評価関数の学習では、探索が実行されるため、訓練例によって処理時間のばらつきが大きい。したがって、B 個の訓練例を処理する場合でも、あるプロセスはすぐに終わるのに、別のプロセスは時間がかかり、結果としてアイドル状態が長くなるという問題が起こる。品詞タグ付けで評価している [117] では、文の長さによって処理量が予測できるため、各プロセスの処理量を均一にするという工夫ができたが、ゲーム木探索では探索量の予測が難しいため、このような工夫はできない。

処理量のばらつきに起因するプロセスのアイドル時間を短くするため、ミニバッチの大きさを訓練例の個数にする方法とは別に、時間にする方法も試みた。ミニバッチの大きさを時間 T_B にする方法として次の二つが考えられる。

(A) 訓練例を処理するたびに時間をチェックし、時間 T_B が経過していたら更新ベクトルを通信する。
(B) 時間 T_B が経過したら、訓練例の処理を強制的に打ち切り、更新ベクトルを通信する。打ちきられた訓練例の処理は、更新ベクトルの通信後に最初から改めて実行される。

B の方法では、処理時間が T_B を越える訓練例の扱いが問題になる。これについても二つの方法が考えられる。

(a) そのような訓練例は捨てる。
(b) そのような訓練例には時間制限を適用しない。

本研究では、アイドル時間を極力抑えることを目的として、上記の B と a を組み合わせた方法を用いることにした。前回のミニバッチで、一つも訓練例を処理できなかったプロセスは、その訓練例を処理するのを諦め、次の訓練例を処理する。

4.2.3 評価

プロ棋士の棋譜を用いて逐次の学習と並列の学習を行い比較した。学習精度の指標としては、コンピュータプレイヤが選んだ指し手が、プロ棋士の指し手と同じである一致率を用いた。

まず全体の実験に共通する設定を 4.2.3.1 節で述べた上で、ミニバッチのサイズを訓練例の個数と時間にした場合の評価をそれぞれ 4.2.3.2 節と 4.2.3.3 節で述べる。

4.2.3.1 実験設定

学習にはプロ棋士の 30,000 棋譜を用い、一致率を調べるためのテスト用の棋譜としては、学習に用いていない 1,000 棋譜を用いた。学習を始める前には 30,000 棋譜分の訓練例をランダムに並び替える。訓練例をすべて一回ずつ処理したら、またランダムに並び替えてから次のイテレーションを開始する。並列の学習では、訓練データを分割することはしなかった。つまり、全プロセスが全ての訓練データを持つ。ただし、各プロセスが訓練例をランダムに並び替えるため、訓練例を処理する順序は異なる。

比較学習で、1 手進めたあとの局面を探索する深さは 5 とした。つまり、全体の探索深さは 6 である。一致率を調べるときの探索深さも 6 とした。

実験には表 4.1 の E5530 の 64 計算ノードを用いた。実装には MPI を用い、処理系には MPICH2（バージョンは 1.4.1p1）を用いた。

各計算ノードのコア数は 8 であるが、1 プロセスずつ実行するようにした。これは、激指の学習はマルチスレッドを用いて並列化されているためである。激指の比較学習では、各スレッドが、訓練局面から一手進めた子ノードの探索を並列に実行する。以降、逐次の学習という用語は、このスレッド並列の学習を指し、並列の学習という用語は分散並列の学習を指すことにする。つまり、並列の学習では、一つの計算ノードが図 4.6 の 1 プロセスに相当する。提案手法では、プロセス数が多くなるほど通信量も多くなるため、通信量を抑えるためにも、スレッド並列が可能なところはスレッド並列にするべきだと判断した。

4.2.3.2 ミニバッチの大きさを訓練例の個数にした場合

まず、並列パーセプトロンの将棋評価関数に適用したときの評価を、ミニバッチのサイズを訓練例の個数として行った。ミニバッチのサイズとなる訓練例の数 B は 5、10、50 で実験した。B は大きいほど情報の更新が遅れてしまうが、小さすぎると計算に比較して通信時間が長くなってしまう。今回は、計算時間と通信時間のバランスがよいと考えられる $B = 10$ の場合を中心に実験を行った。

逐次の学習は訓練例をランダムに並び替えるための乱数の種を変更することにより、10 回の試行を行った。並列の学習の試行回数ついては、中心に調べる $B = 10$ の場合を 10 回とし、$B = 5$ の場合は 3 回、$B = 50$ の場合は 2 回とした。

逐次の学習では訓練例を 128,000 個処理するたびに、それまでの重みベクトルの平均ベクトルを求めてデータ点とした。並列の学習では、各プロセスが訓練例を 2,000 個処理するたびに、同様に平均ベクトルを求めた。つまり、一つのデータ点をプロットするための訓練例の総数を同じにした。

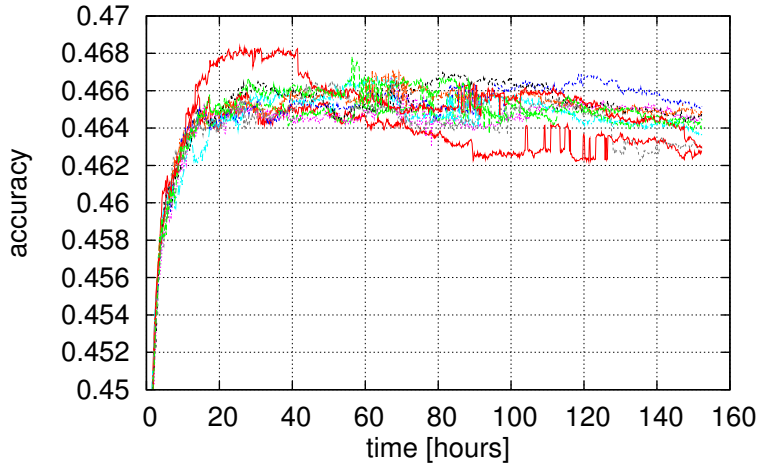

図 4.8: 逐次の学習での学習時間と一致率

データ点の間隔は、時間にすると、逐次の学習で 10 分程度、$B = 10$ のときの並列の学習で 25 秒程度である。

学習の進行度の指標としては、1 プロセスあたりの学習局面数と、学習時間の 2 つを用いる。ただし、学習時間には、学習開始時の棋譜の読み込み時間は含まない。また、それまでの重みベクトルの平均ベクトルを求める時間も含まない[2]。

はじめに、逐次の学習の 10 回の試行結果を図 4.8 に示す。また並列の学習の試行結果を $B = 5$、$B = 10$、$B = 50$ の場合についてそれぞれ図 4.9、図 4.10、図 4.11 に示す。横軸は時間（単位は時間）、縦軸は棋譜の指し手との一致率である。図中の一つの線は一回の試行の学習の進行を示す。これらのグラフから以下のことが読み取れる。

- $B = 5$ と $B = 10$ の並列の学習では時間をかけすぎると、一致率が大きく下がっていく。逐次の学習でもその傾向は見られるが、並列の学習よりはその傾向は弱い。
- 逐次の学習の方が、並列の学習より、一致率が高い傾向がある。特に、$B = 50$ の並列の学習では明らかに一致率が低い。
- 一致率のピークに到達する時間は並列の学習の方が短い。

並列の学習では学習が進むと一致率が下がる傾向があった。逐次でもその傾向は見られたが、程度は小さい。一致率が下がるのは過学習のためであると考えられるが、並列の学習では更新ベクトルを足し合わせている影響が過学習を助長していた可能性がある。しかし、実際の学習では、一致率が下がりはじめた時点で学習を打ちきることができるため、あまり問題にはならない[3]。

[2]平均ベクトルを求める操作は 1 プロセスしか実行しないため、他のプロセスは次の集合通信まで（つまり 1 バッチ）の処理を先に実行してしまうことになり、不当な評価になってしまっている。ただし、他のプロセスの処理も次のバッチまでしか進まず、平均ベクトルを求める操作も、例えば $B = 10$ の場合は 200 バッチごとに一回しか起こらないため、この影響は小さいと考えられる。

[3]平均ベクトルを求めてファイルに書き出す操作にかかる時間は数秒である。今回の実験ではデータ点を多くとったが、図 4.10 などを見る限り、実用上はここまで多くのデータ点をとる必要はない。平均ベクトルがあれば、一致率を求める操作は、他の計算ノードで実行することができるため、この時間は問題にならない。

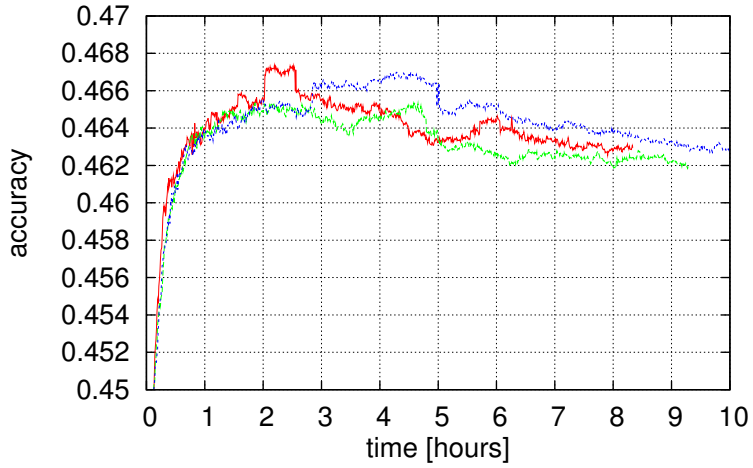

図 4.9: 並列の学習での学習時間と一致率（64 プロセス、$B = 5$）

表 4.8: 最大の一致率とそれを実現するための 1 プロセスあたりの学習局面数と学習時間

	逐次	$B = 5$	$B = 10$	$B = 50$
最大一致率 [%]	46.694	46.658	46.608	44.138
学習局面数 [$\times 10^6$]	51.302	0.813	0.733	2.244
学習時間 [hours]	63.376	3.692	2.461	3.542

各学習において、どの程度の一致率を実現できたかを比較する。各学習において、一致率の最大値と、それを実現したときの 1 プロセスあたりの学習局面数と学習時間を調べた。逐次の学習と並列の学習のそれぞれで、これらの相加平均を求めたものを表 4.8 に示す。並列の学習は逐次の学習より一致率が低く、B が大きいほどその程度も大きいことが分かる。一致率が下がるのは、重みベクトルの更新を 1 回ずつではなく、まとめて行っていることに起因していると推測される。

今回の実装では、ミニバッチの処理のときに、訓練例を処理するごとにローカルな重みベクトルも更新しており、これが一致率を下げている原因という可能性もある。なぜなら、ローカルな重みベクトルを更新することにより、学習の収束の保証がなくなるためである。この影響は B が大きいほど大きいと考えられる。そこで、$B = 50$ の場合については、ローカルな重みベクトルの更新を行わない場合についても実験を行った。試行回数は 2 回とした。結果を図 4.12 に示す。一致率は図 4.11 と同様に低く、ローカルな重みベクトルの更新は学習性能に影響がないことが分かった。

次に、並列化による学習の高速化の程度を見ていく。表 4.8 では、最大の一致率を実現するための学習局面数は少なくなり、学習時間は短縮されていることが分かる。しかし、そもそもの一致率が異なるため、このまま単純に比較することはできない。そこで、基準となる一致率を定め、この一致率

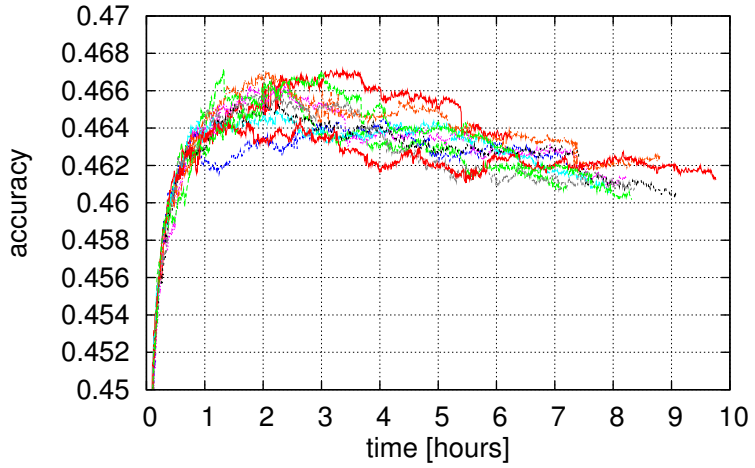

図 4.10: 並列の学習での学習時間と一致率（64 プロセス、$B = 10$）

表 4.9: 一致率 46.4496%を実現するための 1 プロセスあたりの学習局面数と学習時間

	逐次	$B = 5$	$B = 10$
学習局面数 [$\times 10^6$]	11.059	0.251	0.472
（台数効果）		44.1	23.4
学習時間 [hours]	14.284	1.235	1.607
（台数効果）		11.6	8.89

を実現するための学習局面数や学習時間で比較を行う。基準となる一致率としては、$B = 10$ の学習の試行のうち、最小のピークの一致率を用いた。具体的には、$B = 10$ の並列の学習では、一致率が最大でも 46.4496%までしか到達しない試行があったため、これを基準となる一致率とした。

基準となる一致率を用いて並列効果を見る。各試行で一致率 46.4496%を最初に実現したときの、1 プロセスあたりの学習局面数と学習時間を求めた。逐次の学習と並列の学習それぞれで、これらの相加平均を求めたものを表 4.9 に示す。$B = 50$ では基準の一致率を達成できなかったため、表 4.9 には含めていない。台数効果は、1 プロセスあたりの学習局面数と学習時間が、並列化により逐次の学習の何分の 1 になったかを示す。B が小さいほど、学習局面数や学習時間が少なくて済んでいることが分かる。特に学習局面数は B による差が顕著である。なお、$B = 5$ の場合は試行回数が少ないが、学習局面数については 3 回全ての試行において、$B = 10$ の全ての試行の中で最小の学習局面数以下だった。学習時間については、B が小さいほど以下で述べるアイドル時間や通信時間の影響が大きくなるため、学習局面に比べて台数効果の差は小さくなるが、$B = 5$ の場合で、11.6 倍の台数効果が得られた。

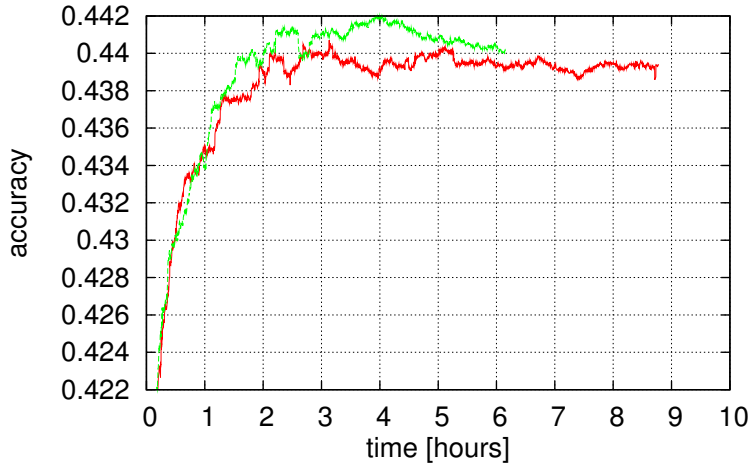

図 4.11: 並列の学習での学習時間と一致率（64 プロセス、$B = 50$）

表 4.10: 64 プロセスが B 個の訓練例を処理する時間と、更新ベクトルの通信時間と非零要素数

	T_{mean} [ミリ秒]	T_{max} [ミリ秒]	T_{comm} [ミリ秒]	N_{comm}
$B = 5$	24.4	74.5	16.9	80,179
$B = 10$	47.0	108.9	25.2	118,068
$B = 50$	265.6	184.7	47.8	183,806

線形な台数効果が得られていない理由を述べる。まず、学習局面数が 1/64 になっていないのは、B が少ないほど基準となる一致率を実現する学習局面数が少なくて済んでいることから、重みベクトルの更新が訓練例 1 つを処理するたびに行っていないことに起因すると推測される。学習時間の台数効果が学習局面数の台数効果より小さいのは、計算時間以外に、各プロセスの重みベクトルの平均を求めるために行う通信にかかる時間と、B 個の訓練例を処理し終えた後に、他のプロセスが終わるのを待っているアイドル時間が存在するからである。

重みベクトルの通信時間とアイドル時間の大きさを定量的に議論する。並列の学習では、図 4.13 のように B 個の訓練例を処理するためにかかる時間がばらつくことによるアイドル時間と、通信時間が問題となる。これらの時間を調べるための実験を、これまでに述べた実験とは別に実行した。学習が進むほど重みベクトルが更新される回数が減るため、訓練例の処理時間や更新ベクトルの通信時間は短くなる。ここでは、一致率がまだ安定しない学習前半に着目し、1 プロセスあたり 128,000 局面を処理するまでの各ミニバッチにかかる平均処理時間を求めた。結果を表 4.10 に示す。T_{mean} は B 個の訓練例を処理する時間、T_{max} は 64 プロセスが B 個の訓練例を処理したときの最大処理時

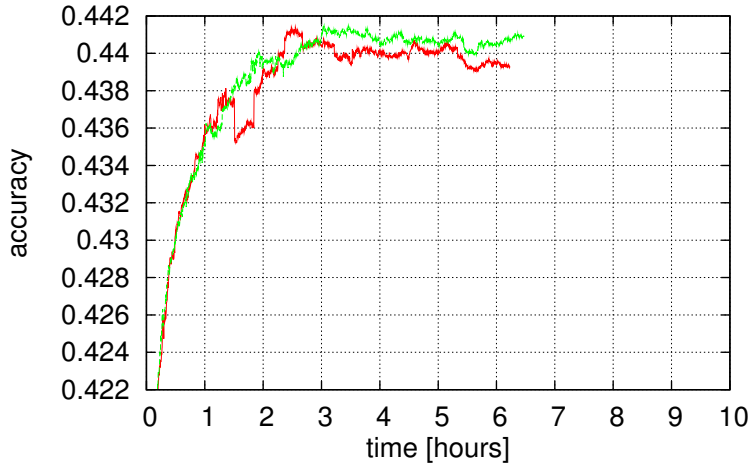

図 4.12: 並列の学習での学習時間と一致率（64 プロセス、$B = 50$、ローカルな重みベクトルの更新なし）

間、T_{comm} は更新ベクトルの通信時間とそれを用いて重みベクトルを更新する時間を合わせたものである。また、通信後の更新ベクトルの非零要素数 N_{comm} も参考のために載せた。

B 個の訓練例そのものの平均処理時間に対する、通信時間も含めた実際の処理時間の割合に注目する。例えば、$B = 5$ の場合では、訓練例そのものの平均処理時間は 24.4 ミリ秒であるが、通信時間も含めた実際の処理時間は 91.4 ミリ秒であり、平均処理時間の 3.75 倍である。表 4.9 では学習局面数に対する台数効果が 44.1 倍だったので、学習時間に対する台数効果は $44.1/3.75 = 11.8$ 倍と予測される。これは実験結果の 11.6 倍とほぼ一致している。$B = 10$ の場合も、同様にして求めた値と実験結果がおおむね一致する。このことから、速度向上を抑制しているのは、アイドル時間と通信時間が主な要因だと結論づけることができる。また、通信時間より、アイドル時間の方がより大きな問題であることも分かる。

4.2.3.3　ミニバッチの大きさを時間にした場合

ミニバッチの大きさを時間にした場合の評価も行った。ミニバッチ一つの時間 T_B は 20 ミリ秒と 50 ミリ秒に設定し、学習の試行回数は 3 回ずつとした。プロセスが処理する訓練例の数は一様ではないため、データ点を得るために平均ベクトルを求めるまでの間隔を時間にし、30 秒とした。前節と同様に、学習時間には棋譜の読み込み時間と、データ点を得るための時間は含めない。

学習の進行の様子を $T_B = 20$ と $T_B = 50$ について、図 4.14 と図 4.15 にそれぞれ示す。また、各試行で最大の一致率とそれを実現した学習時間の相加平均を表 4.11 に示す。ミニバッチの大きさを訓練例の個数にした場合より、一致率が低い傾向にあることが分かる。$T_B = 20$ と $T_B = 50$ のそれぞれの試行のうち、2 回は一致率が基準とした 46.4496%に到達していなかった。また、図 4.14 や図 4.15 を図 4.9 や図 4.10 と比較しても、ミニバッチの大きさを時間にすることによって、学習が速く

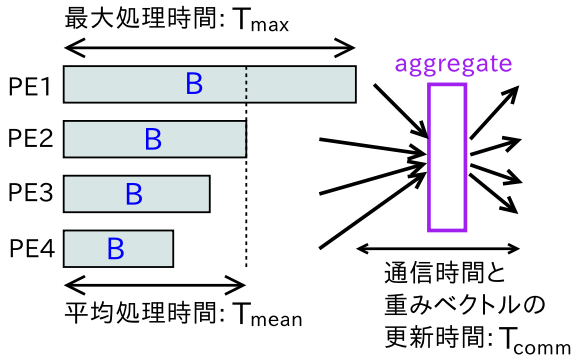

図 4.13: B 個の訓練例を処理する時間と重みベクトルの通信時間

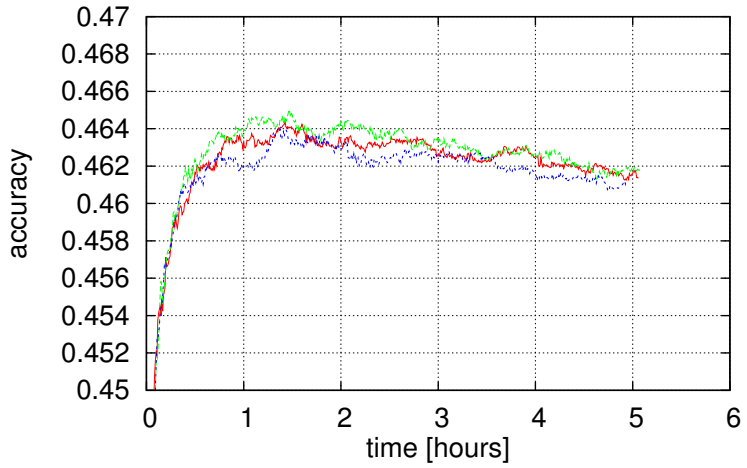

図 4.14: 並列の学習での学習時間と一致率（64 プロセス、$T_B = 20$）

なったわけではないことも分かる。

このような実験結果になった理由の一つとしては、処理時間が T_B 以上かかる訓練例を捨てていることが挙げられる。調べたところ、1 プロセスあたり平均 128,000 局面を処理した時点で、学習局面のうち、$T_B = 20$ の場合には 2.41%、$T_B = 50$ の場合には 0.112%の局面が捨てられていた。この割合は大きくはないかもしれないが、捨てられた局面が学習には重要な局面だった可能性がある。

学習速度に関しては、同じ学習局面数を処理する時間は短くなっていることが分かった。具体的には、1 プロセスあたり平均 128,000 局面を学習するための時間をおおまかに比較したところ、ミニバッチの大きさを訓練例の個数にした場合では、$B = 5$ の場合で 2,300 秒、$B = 10$ の場合で 1,700 秒だったのに対し、ミニバッチの大きさを時間にした場合では、$T_B = 20$ の場合で 1,700 秒、$T_B = 50$ の場合で 1,100 秒であった[4]。実際には打ち切られて無駄になった処理もあることから、この時間も

[4] これらの値は細かい時間を測定するための処理を行った場合であり、実際にはもう少し速い。

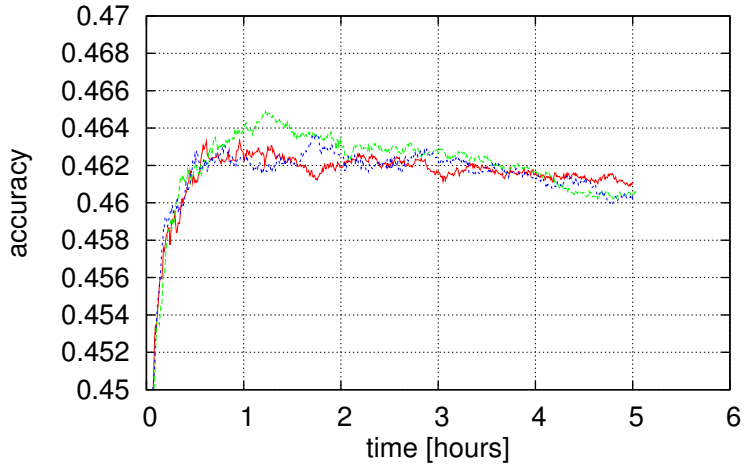

図 4.15: 並列の学習での学習時間と一致率（64 プロセス、$T_B = 50$）

表 4.11: 最大の一致率とそれを実現するための学習時間（ミニバッチの大きさが時間の場合）

	$T_B = 20$	$T_B = 50$
最大一致率 [%]	46.4459	46.3983
学習時間 [hours]	1.414	1.311

理想的には短くなっていない。また、一致率が上昇する速度自体は改善されなかったことから、多くの学習局面を処理できるようになっても、訓練例を捨てている悪影響で相殺されたと考えられる。

4.2.4 まとめと今後の課題

将棋プログラムの評価関数のパラメータ調整に、ミニバッチを用いたパーセプトロンの並列化手法を適用した。評価では、並列化により学習時間を 1/11.6 に短縮できることを示した。速度向上を妨げている要因としては、プロセスで処理時間がばらつくことに起因するアイドル時間と通信時間があり、これらの大きさを定量的に議論し、通信時間よりアイドル時間の方がより大きな問題であることを示した。

アイドル時間を減らすために、ミニバッチの大きさを時間にすることも提案した。具体的には、時間が来たら訓練例の処理を打ち切り、また、処理に時間がかかる訓練例は無視するようにした。しかし、一致率はミニバッチの大きさを訓練例の個数にしたときより下がる傾向にあることが分かった。これにより多くの学習局面を処理できるようにはなったが、一致率が上昇する速度は大きくなったわけではなかった。

今後の課題としては、通信時間やアイドル時間を小さくすることができるような適切な手法を考えることが挙げられる。例えば、重みベクトルの更新を非同期にすることが考えられる。しかし、この場合、通信効率の良い集合通信をそのまま使うことができず、分散環境では速度向上が得られないと考えられるため、よりよい方法を模索する必要がある。

第5章　並列探索の先行研究

本章ではまず、5.1 節で、状態空間の探索を並列化するときに、考慮しなければならない点や、実際に並列化する手法をいくつか紹介する。次に 5.2 節で、$\alpha\beta$ 探索に焦点を当てて、並列 $\alpha\beta$ 探索の先行研究について述べる。

5.1　状態空間の並列探索

状態空間探索は (1) 初期状態、(2) ある状態から次状態を生成する関数、(3) ある状態が解状態かどうかを判定する関数、の 3 つが与えられたときに、初期状態から解状態までのパスを求めるものである。探索空間は一般的にはグラフとなるが、実際の探索ではこれを木とみなして探索することも多い。この場合、異なるパスを経由して同じ状態に到達することがあるため、木の中の異なるノードが同じ状態を表すことがある。ただし本論文では、区別する必要があるときを除いて、「ノード」と「状態」を同じ意味で用いることにする。

5.1.1　問題の分類と具体例

状態空間探索問題を、[98] にならって、どのような解が必要かに応じて 3 つに分類する。

- **全解探索 (All solutions search)**
  探索空間中の全ての解状態を見つける
- **一解探索 (First solution search)**
  探索空間中の解状態のうちどれでもよいので一つを見つける
- **最善解探索 (Best solution search)**
  探索空間中の解状態のうち、最善 (コストが最小) のものを見つける

全解探索は状態空間のうち、解が存在する可能性のある部分を全てを探索しなければならないのに対し、一解探索は解の存在が有望と考えられる部分空間から優先して探索し、比較的狭い範囲の探索で目的を達することができる場合も多い。しかし、一解探索はしばしば、一つの解を見つけるのに加え同時に探索空間中に解状態が一つもない場合はそれを示す問題にもなる (例えば、命題論理式の充足可能性問題において論理式が真にならない場合)。その場合は全解探索に近い問題になる。全解探索は、結局は解の存在する可能性のある状態空間をすべて探索する必要があるので、並列化の際には探索木を部分木に分割して各プロセスが各部分木を深さ優先探索するという方法がとられることが多い。

一解探索でも同様に部分木を分割して並列に探索するという方法をとることも多いが、この場合は、解がありそうな空間を多くのプロセスで重点的に探索するといった方法も考えられる。また、一解探索の並列化では、逐次探索と異なる空間を先に探索することもあり、解が偶然早く見つかるということが起こるため、例えば 4 プロセスで並列化して実行時間が逐次探索の 1/4 より短くなることがある。

最善解探索は全解探索と同じく探索空間全てを探索する必要があるが、「現在得られている解よりコストが大きい解は最善解にはなりえない」という性質を用いて枝刈りする方法がとられることも多い。

次に、並列探索が適用されている問題の具体例を列挙する。

- N-Queen 問題 [98, 21, 126]
- モデル検査 [36, 78, 64]
- 充足可能性問題 [21, 30, 45, 84, 122]
- And-or 木探索 [53, 128]
- 巡回セールスマン問題 [98]
- 混合整数線形計画問題 [113]
- シーケンスアラインメント [60]
- プランニング [56]
- 15 パズル [89, 106]
- ゲーム木探索 [37, 29, 42, 124, 58, 95]

一般的に N-Queen は全解探索の問題として、モデル検査、充足可能性問題、and-or 木探索は一解探索の問題として扱われる。モデル検査と充足可能性問題は解状態がない場合はそれを示す必要があることも多い。And-or 木探索は将棋などで現在局面から相手を詰ませる指し手を素早く調べるために用いられる。残りの問題は全て最善探索の問題である。

並列探索において高い台数効果を実現するために本節で議論する事項を列挙する。それぞれについては以降で詳しく見ていく。

- 探索処理の分割方法。つまり、どのようにして各プロセスに異なる処理を実行させるか。
- プロセスの役割分担。
- 負荷分散手法。つまり、どのようにして各プロセスにアイドル状態にならないように意味のある処理を実行させ続けるか。
- プロセス間の情報共有。つまり、他のプロセスが探索中に得られた情報 (特に探索空間の削減に関する情報) の共有をどのように行うか。

5.1.2 探索処理の分割方法

並列処理を行うには逐次で行っていた処理を同時に実行可能な複数の処理に分割しなければならない。ここでは処理の分割方法を以下のように分けて説明する。

- 異なる方略を用いた探索に分割
- 異なる子状態の探索に分割

異なる方略を用いた探索に分割する方法では、各プロセスは最初から異なる方略を与えられているため、探索中に通信を行わなくても並列に異なる探索を実行することが可能となる。これは実装の容易さにつながる。もちろん、通信を行い、プロセス間で情報を共有することで探索を効率よくすることも可能である。異なる子状態の探索に分割する方法は、並列探索において一般的な処理の分割方法である。この方法では、タスクを割り当てるときにも、結果を受け取るときにも通信が発生する上、後で述べるように負荷分散のためにも通信が必要となる。このため異なる方略を用いた探索に分割する方法よりも実装が難しくなる。

5.1.2.1 異なる方略を用いた探索に分割

各プロセスは異なる方略を用いて木を探索する。

並列 SAT ソルバにおいて portfolio というアプローチを利用したものがある [21, 45]。SAT ソルバには探索パラメータが複数存在する。そこで、パラメータを異なる設定にした SAT ソルバを各プロセス上で同時に実行する。SAT ソルバでは探索途中で得られた情報を利用して探索空間の削減を行うが、これらの情報は各プロセスで共有される。[21] は、このアプローチと、異なる子状態の探索に分割するアプローチを 128 コアを用いて比較し、計算環境が大規模になった場合には、このアプローチの方が並列処理効果を得やすいと主張している。また、このアプローチは SAT ソルバ以外でも、weighted A*アルゴリズムで 15 パズルを解く問題で、実装の容易さにも関わらず 8 プロセッサで 2 倍の速度向上と、その有効性が示されている [106]。

並列ゲーム木探索でも複数のプログラムを同時に実行するという手法が存在する。ゲーム木探索は、そもそも局面の「良さ」を完全には正しく評価できないという点が他の問題と大きく異なる。合議アルゴリズム [124] は、この局面の「良さ」の判定が異なるプログラムを各プロセスで同時に実行し、時間制限の後に各プロセスが得た合法手のうち多数決によって指し手を決める手法である。通信が必要ないため実装が容易であり、並列に実行することで強いゲームプレイヤができることが知られている。合議アルゴリズムは $\alpha\beta$ 探索 [59] で用いられるが、同じくゲーム木探索である UCT 探索 [61] にも root 並列化 [95] という、乱数を変更したプログラムを複数同時に実行して、それらの結果を集計する手法が存在する。

Dwyer らは、モデル検査プログラムの並列化において、parallel randomized state-space search という手法を提案している [36]。これは各プロセスは同じ木を深さ優先探索していくが、子ノードを探索する順番がランダムで決まるというものである。例を図 5.1 に示す。A から D は部分木であり、プロセス 1 は ABCD の順に、プロセス 2 は BADC の順に、プロセス 3 は CDAB の順に探索することを示している。このとき A から D のどこに解状態が存在しても、多くても二つ目の部分木を探索すれば解が見つかることになる。すなわち、この手法は、探索する順序が異なると解が早く見つかることがあるという一解探索の性質を利用している。性能としては 10 から 15 プロセスぐらいまで台数効果が得られることが報告されている。この手法は通信を行わないため、実装も容易であるという半面、状態空間内に解状態が存在しない場合に、それを確かめるのには逐次探索と同じだけ時間

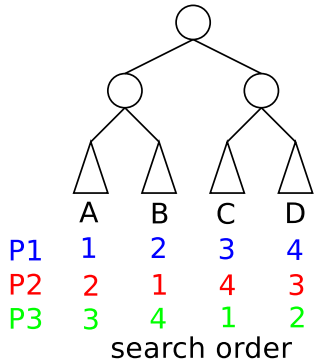

図 5.1: Parallel randomized state-space search の例

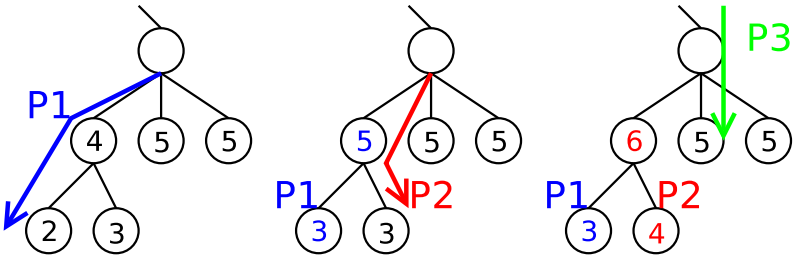

図 5.2: Kaneko による並列 df-pn

がかかってしまうという欠点がある。これは、各プロセスがお互いに探索空間中のどこを探索しているのかを知らないためである。

And-or 木を探索するアルゴリズムの一つである df-pn [80] の並列化でも、各プロセスが異なる探索順序を探索するようにしている手法が提案されている [53, 128]。Df-pn は現在状態から解状態を発見するまでにかかるコストが小さい順に子ノードを探索していく深さ優先探索の手法である。Kaneko [53] は共有メモリ環境において、各プロセスに別々の子ノードを探索させるために、プロセスがノードを訪問したときにコストを増やすという手法をとっている。手順を図 5.2 に示す。まずプロセス 1 がこの部分木を訪問したときは、最初から与えられていたコストを使って子ノードを選んでいく (左図)。次にプロセス 2 がこの部分木を訪問したときは、プロセス 1 が増やしたコストを使って子ノードを選んでいく (中央の図)。さらにプロセス 3 がこの部分木を訪問すると、プロセス 1 とプロセス 2 が増やしたコストを使って子ノードを選ぶ (右図)。このようにして各プロセスは異なりながらもコストが小さい状態空間を探索することができる。結果として共有メモリ環境を用いて 8 コアで 3.6 倍の高速化が報告されているが、この手法はプロセス間でコストを共有する必要があるため、分散メモリ環境には適さない。そこで、保木らは分散メモリ環境で df-pn を並列化するために、乱数によってコストを増やし、各プロセスで異なるコストを用いることにより異なる順序で探索する手法を提案している [128]。

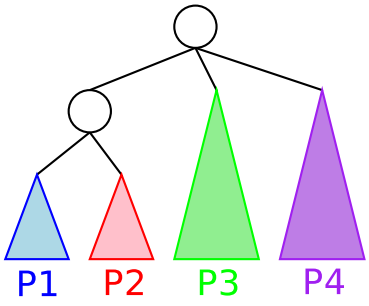

図 5.3: 異なる子状態を並列に探索

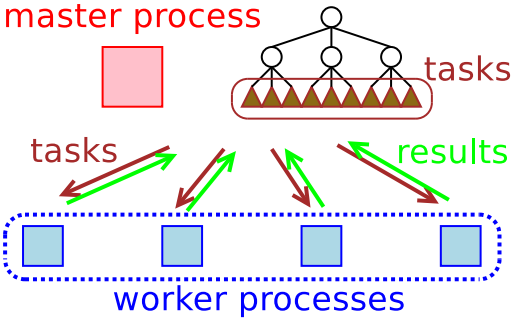

図 5.4: マスタ・ワーカ方式

5.1.2.2 異なる子状態の探索に分割

親状態を展開して生成される子状態の探索を別々のプロセスに割り当てる手法は、並列探索において一般的な方法である。例えば、4 プロセスを用いて並列化をする場合は、図 5.3 のように子状態の探索をタスクとして割り当てればよい。また、負荷分散も考慮して、はじめから図 5.3 よりももっと細かく探索空間を分割し、各プロセスに複数の細かいタスクを割り当てることも多い。いずれにせよ、後の章で述べるように動的な負荷分散が必要となることが多い。以降の議論は、特に明記しない限り、処理を子状態の探索に分割していくことを前提として進める。

5.1.3 プロセスの役割分担

実装方式として、各プロセスの役割分担について述べていく。

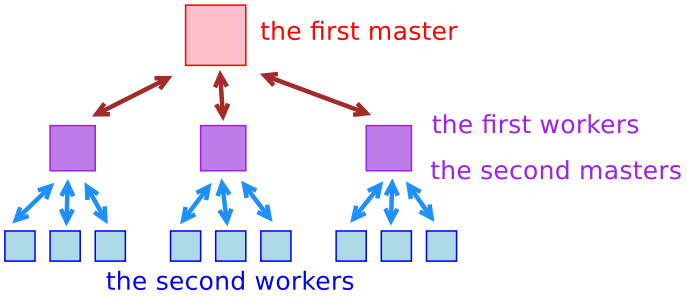

図 5.5: 階層的マスタ・ワーカ方式

5.1.3.1 マスタ・ワーカ方式

マスタ・ワーカ方式は並列処理で広く用いられている実装方式である。例を図 5.4 に示す。ある一つのプロセスがマスタとなって探索空間を分割し、生じたタスクを適切な方法でワーカに割り当てる。さらに、マスタはワーカから得られた結果を元に、必要ならばタスクの再割り当てを行ったり、不必要と判明したタスクの終了をワーカに指示したりするなど、適切な処理を行う。マスタ・ワーカ方式はワーカの数が少ないときはうまく動作する。しかし、ワーカの数が非常に多くなると、マスタの処理が追いつかず、ワーカがマスタの処理待ちという状態になり、マスタがボトルネックとなりうる。この問題の解決策としては、タスクや結果など、情報をまとめて送受信することで、オーバーヘッドを減らす方法がある。また、次で紹介する階層的マスタ・ワーカ方式も有効な手法である。

5.1.3.2 階層的マスタ・ワーカ方式

階層的マスタ・ワーカはマスタがボトルネックとならないようにマスタを増やし、マスタ・ワーカの関係を階層的にしたものでる。二段階のマスタ・ワーカ方式の例を図 5.5 に示す。ある一つのプロセスが全体を取り仕切る一段階目のマスタとなる。一段階目のマスタは探索空間をある程度の大きさのタスクに分割して、一段階目のワーカに送信し、これらの一段階目のワーカが二段階目のマスタとなり、受け取ったタスクを必要ならばさらに細かく分割して二段階目のワーカに送信する。二段階目のワーカが実際にタスクを実行して部分木の探索を行うが、一段階目のワーカもタスクの実行を行うこともある。プロセス間の通信は、直接マスタ・ワーカ関係となっているプロセス間でしか、基本的には行われない。このため、異なるマスタの下にあるワーカの間の情報共有は、階層を持たないマスタ・ワーカ方式の場合より難しくなる。

階層的マスタ・ワーカを採用している研究としては、[84] や [10]、[113] がある。[84] は SAT ソルバの並列化、[10] は分枝限定法を用いて BMI 固有値問題を解く処理を並列化しており、[113] は一般的な混合整数線形計画問題を解く並列ライブラリの研究である。すべて二段階のマスタ・ワーカの有効性を示しているが、これに加えて、[10] と [113] では、一段階目のワーカ (二段階目のマスタ) の数が多すぎても性能が劣化することも報告している。

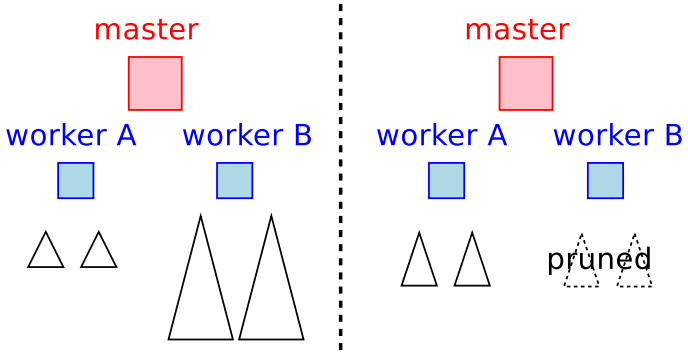

図 5.6: 静的負荷分散の問題点

5.1.3.3 対等関係

マスタ・ワーカのような明示的な関係を持たず、各プロセスが対等な関係でタスクやその結果をやりとりする実装方式もある。この方式はいわば、すべてのプロセスがマスタであり、かつワーカである方式と言える。後で説明する TDS [89] はこの方式である。後で説明する負荷分散の手法のうち、worker-initiated な手法の場合にはこの実装方式を用いることも可能である。

5.1.4 負荷分散手法

5.1.4.1 動的負荷分散の必要性

探索開始時に探索空間は部分木に分割され、少なくともプロセス数以上のタスクが生成される。このとき、各プロセスに割り当てられるタスクの実行時間がプロセス間で同じであればよいが、実際には、前もって各部分木の探索に要する計算量を正確に知ることはしばしば困難であり、タスクの大きさはばらつくことが多い。したがって、各プロセスの処理量も大きくばらつくことになる (図 5.6 の左図)。これでは小さいタスクが割り当てられたプロセスはすぐにアイドル状態になってしまい、有効に活用できない。各プロセスの処理量が均一になるように、タスクをできるだけ細かく分割し、最初からすべてのタスクを割り当てることはせず、アイドル状態になった、あるいはなりそうなプロセスに順次割り当てていく方法もあるが、通信のオーバーヘッドなどを考慮すると限界がある。

また、各プロセスに割り当てられたタスクの大きさが最初は同じだったとしても、探索が進むことで、枝刈りによる探索空間の削減が起こるため、タスクの大きさが変わったり、タスクそのものが不要になったりすることがある。この場合も各プロセスの処理量がばらつくことになる (図 5.6 の右図)。タスクを細かくして、順次割り当てていくようにすれば、この問題も軽減できるが、やはり限界がある。

タスクの量だけではなくて、タスクの質も重要となることがある。あるプロセスは解状態を含む可能性が高い有望な部分木ばかりを探索しているのに対し、別のプロセスは解状態を含む可能性が

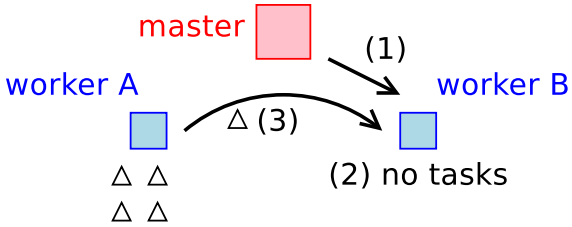

図 5.7: Master-initiated な動的負荷分散

低い部分木ばかりを探索している状況は、プロセスによってタスクの質が偏っている。解を見つけるために意味のある計算は、有望な部分木の方が多いとみなせるため、タスクの質が偏っていることは、結局はタスクの量が偏っているとみなすことができる。どの部分木が有望かという情報も探索が進むことで変化しうるので、タスクの質もプロセス間でばらついていくことになる。

このように探索開始時に静的に負荷分散するだけではプロセス間で負荷が偏るので、探索の進行と共に動的に負荷分散をする必要がある。[113] の手法を参考に、master-initiated な負荷分散と worker-initiated な負荷分散に分けて説明する。

5.1.4.2 Master-initiated な動的負荷分散

図 5.7 を用いて説明する。Master-initiated な動的負荷分散では、マスタがワーカの負荷を定期的に調べる (図 5.7 の (1))。ワーカがアイドル状態である場合 (図 5.7 の (2)) や、あるワーカのタスクの数が他のワーカとの平均より少ない場合には、マスタは他のワーカにメッセージを送信し、そのワーカのタスクをタスクを必要としているワーカに再び割り当てる (図 5.7 の (3))。[113] では平均よりタスクが少ないワーカとタスクが多いワーカでそれぞれペアを作り、そのペアの間で質が最も良いタスクを一つ移動させる手法をとっている。もし、送信元のワーカにタスクがひとつしかなければ、タスクをさらに小さく分割してから送信する。

Master-initiated な負荷分散はマスタは各ワーカが所有しているタスクの量と質をすべて把握できるため、適切な負荷分散が行いやすい。これは長所であるが、ワーカの負荷を調べる周期の長さが問題となる。この周期が長いと負荷が分散されにくくなるが、短いとマスタに大きな負荷がかかることになる。

マスタを通さないが、定期的に各プロセスが他のプロセスの負荷を調べて負荷分散をする方法として、[64] で用いられている手法がある。この手法では、各プロセスはあらかじめ決められた少数のプロセスのそれぞれと、タスクの数を教え合うために定期的に通信を行う。このとき、タスクの数が自分と相手で均等になるようにタスクを送受信する。

5.1.4.3 Worker-initiated な動的負荷分散

Worker-initiated な負荷分散では、タスクがなくなった worker が自分からマスタにタスクを要求するメッセージを送信する。マスタはこのメッセージを受け取ったら他のワーカにメッセージを送り、master-initiated の場合と同様にタスクの再割り当てを行う。[113] の手法では worker-initiated な場合でもマスタはワーカのタスクの状況を定期的に調べているので、タスクの送信元となるワーカとしては最もタスクを持っているワーカを選ぶことができる。もし、マスタがワーカのタスクの状況を知らなかった場合は、マスタはワーカから一つをランダムに選ぶことになる。

Worker-initiated な負荷分散ではタスクがなくなることが負荷分散操作のトリガーとなる。しかし、あるワーカは自分が持っているタスクの質と他のワーカのタスクの質を比較しづらいため、タスクの質による負荷分散が難しいという欠点がある。しかし、これはマスタがいない場合、つまり、プロセスにマスタ・ワーカの関係がない対等関係の場合でも使える手法である。タスクがなくなったプロセスは、マスタを介さずに直接他のプロセスにタスクを要求するメッセージを送信すればいいからである。これは work stealing として知られている手法 [44] であり、ゲーム木探索の実装で使われている例 [37] がある他、work stealing そのものについても広く研究されている [126, 35, 51]。

5.1.4.4 タスクの粒度に関する議論

タスクを十分に細かく分割した上で、合計の処理量がほぼ等しくなるように、各プロセスに割り当てていけば、負荷分散は実現可能である。ただし、非常に細かいタスクに分割してしまうと、タスク生成にかかる時間や、タスク割り当てにかかる通信時間などのオーバーヘッドがタスクの実行時間そのものに対して相対的に大きくなってしまう。したがって、性能向上のためにはタスクは細かく分割しすぎてもいけない。負荷分散のことを考えるならば、タスクがなくなったときにタスクを分割すればいいという考えもある。すでに説明した動的負荷分散では他のプロセスにタスクを送信するプロセスがタスクを一つしか持っていなかった場合は、タスクを分割していた。ゲーム木探索を並列化している [29] では、ワーカに割り当てられたタスクの実行時間があるしきい値を越えたら、いったんタスクをマスタに返して、細かく分割するという方法がとられている。また、一般的な探索フレームワークを実装した [98] では部分木の実行時間をあらかじめ予測し、それがしきい値を越えないようにタスクを細かく分割する。このときのしきい値は一つのタスクにかかるオーバーヘッドを用いて計算される。このように、タスクの粒度を動的に変更できるのは、状態空間探索には探索空間を細かい空間に分割するのが容易であるという特徴があるからである。

5.1.5 プロセス間の情報共有

探索が進むことで、各プロセスは探索している状態空間に関する情報を得て、それを用いることで探索を効率よく勧めていく。したがって、これらの得られた情報をプロセス間で共有することは重要である。そこで、グラフを木で表現していることによって無駄な探索が起こってしまうのを避けるための transposition table driven work scheduling(TDS) という手法について紹介した後で、探索空

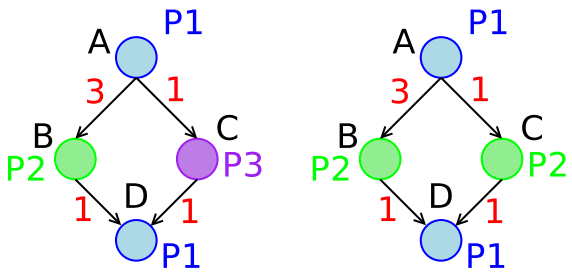

図 5.8: エッジのコストが一様ではない問題の例

間を削減するための情報など、プロセス間で情報を共有するときに考慮するべき事項について議論する。

5.1.5.1 Transposition table driven work scheduling

状態空間は実際にはグラフであるが、これを木として探索してしまうと、異なるノードが同じ状態を表すということが起こる。つまり、異なる状態に、異なるパスを経由して到達することがある (図 2.6)。このとき、そのノードが他のノードと同じ状態を表しているということがわかっていないと、すでに探索した状態を重複して探索してしまうことになる。そのため木探索では、訪問したノードとその探索結果を格納しておく。一例としては状態をキーとしたハッシュテーブルが用いられる。並列探索ではこのハッシュテーブルの情報をプロセス間でどうやって共有するかが問題となる。

同一状態を複数回訪問したときに無駄な探索をしないための手法として、transposition table driven work scheduling(TDS) [89] がある。TDS では、状態が同じであることを判別するためにハッシュ値を用いる。各プロセスには、処理を担当する状態のハッシュ値が割り当てられてており、同じハッシュ値の状態は同じプロセスが処理する。各プロセスは子ノードを生成すると、その子ノードのハッシュ値を計算し、それを担当することになっているプロセスに送信する。これにより異なるノードが同じ状態を表していても、必ず同じプロセスに割り当てられるので、すでに得られた結果を再利用できる。

TDS はハッシュ値を用いてタスクをプロセスに割り当てるため、探索開始時にタスクの割り当てが静的に決まっていると言える。しかし、ハッシュ関数が十分に良いものであれば、各プロセスに割り当てられるタスクの数はほぼ同じになると考えられる。ただ、すでに述べたようにタスクの大きさのばらつきや枝刈りのばらつきの程度が大きいと負荷が偏るという可能性がある。

TDS はもともと深さ優先探索である Iterative deepening A* (IDA*) を並列化したものであるが、同様にハッシュ値を用いて最良優先探索である A*を並列化した Hash Distributed A* (HDA*) [56] という手法も存在する。また TDS は 15 パズルなどの一人ゲームを解くのに用いられるが、これを二人ゲームの探索に応用した研究として [42, 115] や [58] がある。

また、Kobayashi ら [60] は、グラフのエッジ (状態の遷移) にコストが付加されている場合、HDA* では次のような問題点があることを指摘した。図 5.8 の左図にその問題点を示す。ノード A とノー

ド D はプロセス 1 に、ノード B はプロセス 2 に、ノード C はプロセス 3 に割り当てられるとする。図中の赤い数字はコストを示す。このとき、プロセス 1 に $A \to B \to D$ と経由してきたタスク (コスト 4) が $A \to C \to D$ と経由してきたタスク (コスト 2) よりも先に到達すると問題が発生する。つまり、後からきたタスクの方がコストが小さいため、先に同じ状態でコストが大きいタスクを実行していたとしても、もう一回後から来たタスクを実行する必要が発生してしまう。これを [60] ではハッシュ関数を修正して、ある意味でハッシュ関数を「偏らせる」ことにより、B と C が同じプロセス 2 に割り当てられる確率が高くなるようにしている (図 5.8 の右図)。こうすれば、B と C では C の方がコストが小さいため、C が先にプロセス 1 に送信される。ハッシュ関数を偏らせるということは、プロセス間で負荷が偏ることを意味し、トレードオフが発生するが、その程度は手法が用いるパラメータによって調整できる。

5.1.5.2 情報共有で考慮するべき事項

探索空間を削減するための情報をプロセス間で共有することは重要であり、特に SAT ソルバの並列化ではよく議論されている [30, 45, 84, 122]。探索空間の削減以外でも、前節で説明したような、異なる状態を重複して探索しないようにするための情報の共有も TDS を用いない場合は重要である。さらには、タスクが解状態を含むものである可能性などの、タスクの質も探索の進行とともに変化するため、これもプロセス間で情報をやりとりして更新していかなければならない。これらは異なる子状態の探索に分割するときだけではなく、各プロセスが異なる方略で探索するときにも必要である。ここでは、このような情報をプロセス間で共有するときに考慮するべき事項として次の 2 つを挙げる。

- 共有する情報の内容
- 情報を共有するための通信の頻度

一つ目は、どのような情報を共有するべきかというものである。つまり、共有したところで探索空間の削減がほとんど起きないような情報まで共有しようとしてしまうと、そのための処理時間や通信時間が大きくなりすぎてしまう。二つ目は、どのくらい頻繁に各プロセスで情報をやりとりするかであり、頻繁なほど他のプロセスで得られた最新の情報を用いることができるが、通信のオーバーヘッドが問題となる可能性がある。これを避けるためにある程度の量の情報をまとめて送受信することで通信の頻度を減らせば、通信のオーバーヘッドは小さくなるが、他のプロセスでの探索空間の削減が遅れることになり、必要のない探索が行われるというトレードオフがある。このトレードオフの最適点は並列化する対象の問題によって異なってくるため、チューニングが必要となる。

5.2 並列 $\alpha\beta$ 探索

本節では、まず 5.2.1 節において並列 $\alpha\beta$ の難しさについて述べる。次に、並列 $\alpha\beta$ 探索の先行研究を、他のプロセスの結果をどの程度待つのかという軸で大きく二つに分け、他のプロセスの結果

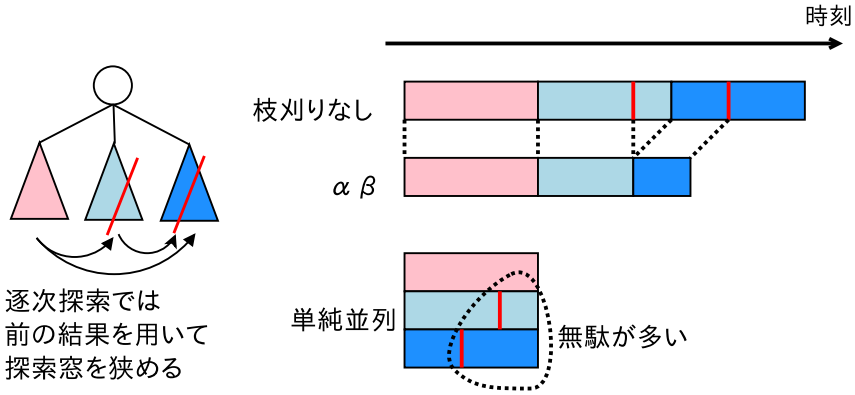

図 5.9: 単純に $\alpha\beta$ を並列化した例

を明示的に待つアルゴリズムを 5.2.2 節に、そうでないものを 5.2.3 節で紹介する。最後に 5.2.4 節で、並列ゲーム木探索で重要と考えられるトランスポジションテーブルの共有について述べる。

5.2.1 並列 $\alpha\beta$ 探索の難しさ

並列 $\alpha\beta$ 探索の難しさは、逐次探索では枝刈りできたはずの部分木を探索してしまい、探索ノード数が増えてしまうというところにある。これは、並列探索では、逐次探索ならば用いることができた結果を得る前に、部分木の探索を開始しないと、十分な並列度が得られないために生じる問題である。図 5.9 を用いて説明する。図 5.9 の 3 つの子ノードの探索時間は、探索窓が同じならばすべて同程度とする。図 5.9 の右側の図は各時刻にどのノードを探索しているかを表現しており、ミニマックス探索から $\alpha\beta$ 探索で探索時間が図のように短縮できていたとする。このとき $\alpha\beta$ 探索において子ノードを単純にすべて並列に探索すると、並列に実行できる部分は大きくなるが、探索窓が広いまま探索を開始してしまい、逐次探索で発生した枝刈りの大部分が発生しなくなってしまうため、逐次探索では探索する必要がなかった探索を大量に行ってしまう。逆に、逐次探索と同じくらい枝刈りを発生させようとすると、先行する探索の終了を基本的には待たなければいけないため、並列に実行できる部分がほとんどなくなってしまう。

したがって、並列 $\alpha\beta$ 探索では、ある時刻に並列に実行可能な探索の数と、枝刈り効率のトレードオフを考慮しなければならないことになる。プロセッサの数が少なければ、並列に実行可能な探索の量を増やすよりも、枝刈り効率を重視するべきであるし、プロセッサの数が非常に多いのであれば、並列に実行可能な探索の量を増やすことを考えなければならない。

探索の進行によって枝刈りされてしまうような部分木の探索は、全体の探索結果に関係ないため、無駄となる。しかし、並列に実行可能な計算を増やすためには、たとえ枝刈りされる可能性がある部分木でも探索しなければならない。このとき、枝刈りされない可能性の高い部分木の探索を優先することによって、無駄になってしまう探索を減らす必要がある。また、枝刈りされない可能性の高

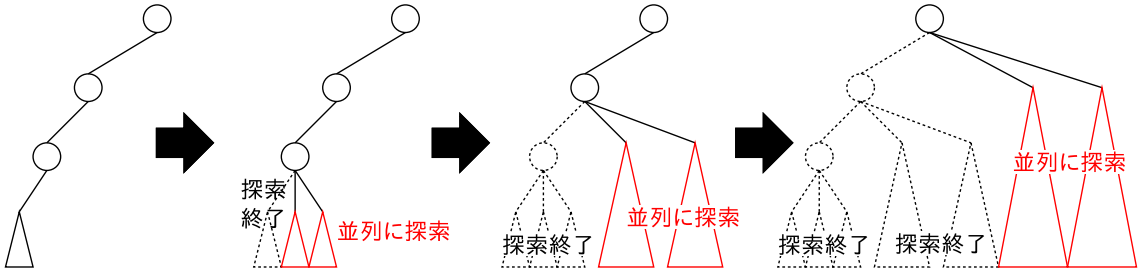

図 5.10: PV Split での探索の進行

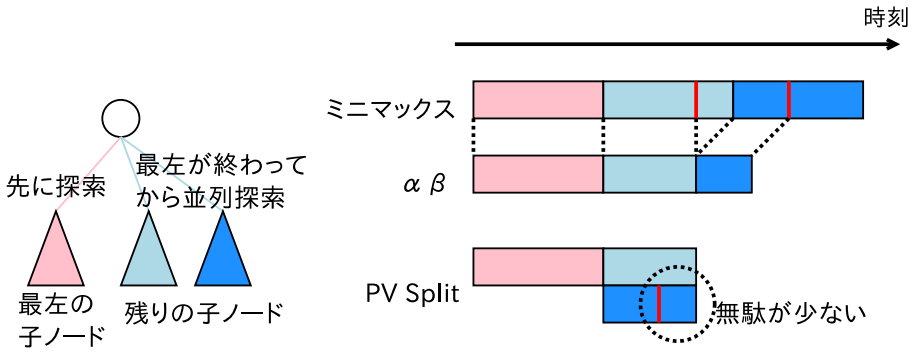

図 5.11: PV Split による無駄な探索の抑制

い部分木だけではなく、枝刈りに必要な情報を得るための部分木も、優先的に探索されるべきである。つまり、並列 $\alpha\beta$ 探索では、ノードを探索する優先度を、これらを考慮して付けることが重要となる。並列 $\alpha\beta$ 探索の先行研究では、枝刈りに必要な情報を得るためのノードの探索の終了を明示的に待ってから、残りのノードを探索することが行われてきた。この「待つ」という操作は、「同期」と呼ばれている。ここでは、並列 $\alpha\beta$ 探索を、同期をどれだけ行うか、すなわち、先行する探索の結果をどれだけ待つか、という軸で整理して述べていく。

また、ゲーム木探索では、トランスポジションテーブルによる重複探索を避けることが重要であることはすでに述べた。したがって、並列ゲーム木探索では、トランスポジションテーブルの情報をどのように共有するかというのも重要な問題である。この点についても議論する。

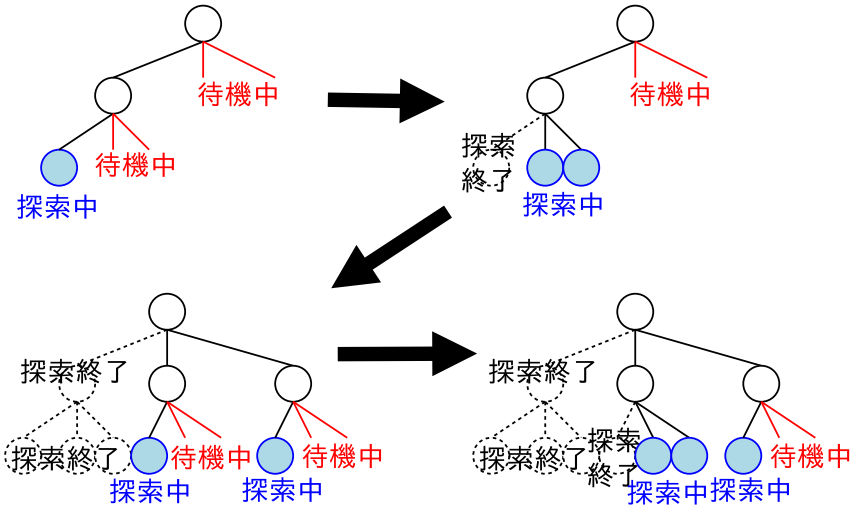

図 5.12: YBWC の探索の進行

5.2.2　同期型アルゴリズム

5.2.2.1　PV Split

PV Split [72] は基本的な同期型アルゴリズムである。PV ノードと予測されるノード[1]において、最も有望な子ノードの探索を先に開始し、その子ノードの探索が終了してから、残りの子ノードを並列に探索する。PV Split における探索の進み方を図 5.10 に示す。PV ノードは、ゲーム木の中で最初に探索されるため、探索窓がまだ狭まっておらず、$(-\infty, \infty)$ となっている。したがって、探索の無駄を減らすためには、図 5.11 のように最も有望な子ノードの探索を先に実行してから、その結果を用いて探索窓を狭め、残りの子ノードを並列に探索すればよい。もし最も有望な子ノードが実際に最善の子ノードであるならば、最も有望な子ノードの結果で探索窓を狭めた後は、残りの子ノードの探索結果で探索窓が狭まることはないため、PV Split は逐次探索と同じ訪問ノード数で探索を完了させることができる。

PV Split の最大の問題点は、並列性が合法手の数に制限されてしまうということである。各 PV ノードについて同時に実行可能な部分木は合法手の数しか存在しない。これは大規模計算環境を活用するためには明らかな問題である。他にも、部分木の大きさは著しくばらつくため、最も大きい部分木の探索が終了するまで、他の部分木がアイドル状態になってしまうという問題点も存在する。

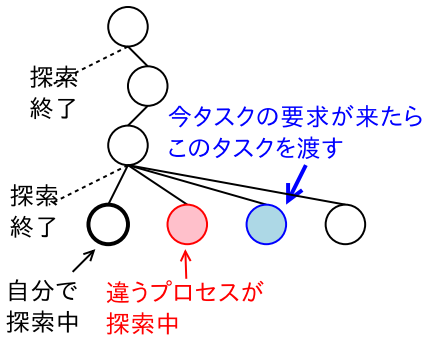

図 5.13: YBWC の負荷分散

5.2.2.2 Young Brothers Wait Concept

Young Brothers Wait Concept (YBWC) [37] は、PV Split を改良して、並列性が小さく、プロセスがアイドル状態になってしまうという問題点の解消を目指したアルゴリズムであり、$\alpha\beta$ 探索の並列化において広く用いられている手法である。YBWC のアルゴリズムを簡単に記述すると、「全てのノードにおいて、最左の子ノードだけを先に探索し、残りの子ノードは最左の子ノードが終了した後で探索窓を狭めてから、その探索窓を用いて並列に探索する」となる。YBWC の探索の進行の例を図 5.12 に示す。PV Split と同様に、全てのノードで常に最善の子ノードを最初に探索する理想的な場合には、他の子ノードが探索窓を狭めることはないので、逐次探索と同じ訪問ノード数で探索を完了させることができる。YBWC も指し手の並び替えの精度が高いほど性能が良くなる。

YBWC は PV ノード以外のノードでも、子ノードを並列に実行するため、並列に探索可能なタスクを数多く生成している。タスクが十分にあれば、動的負荷分散を行うことができる。YBWC では、アイドル状態になったプロセスは、ランダムにプロセスを選び、タスクを要求するという実装になっている。このとき、奪うタスクの選択方針を図 5.13 に示す。タスクを要求されたプロセスは、ゲーム木の左下にあり、どのプロセスも実行中でないタスクを渡す。つまり、逐次探索において、次に探索されるノードをタスクとして渡す。動的負荷分散では、ルートに近いノードを渡すことが多いが、ここではルートから遠いノードを渡している。これは、探索で得られた結果をなるべく反映して、効率的に枝刈りが行えるタスクを選んでいるためである。

実装に YBWC を用いたものを紹介する。

Feldmann [37] は、チェスプログラム ZugZwang を work stealing を用いて実装を行った。30MHz、20Mbps の T805 トランスピュータ 1024 台を用いて、逐次探索の時間が 10 万秒程度の規模の問題で、344 倍の速度向上比を実現している。

Kuszmaul は YBWC の改良であり、NegaScout を並列化した Jamboree アルゴリズム [65] を提案し、チェスプログラム StarTech を実装した。512 プロセッサのコネクションマシン CM-5 を用い、逐次探索時間が平均 1500 秒程度の問題規模で、およそ 50 倍の速度向上比を報告している。

[1]探索中に最小探索木は分からないため、以下単に PV ノード、CUT ノード、ALL ノードと記した場合、文脈によって自明な場合については、予測されたものを指すこともある。

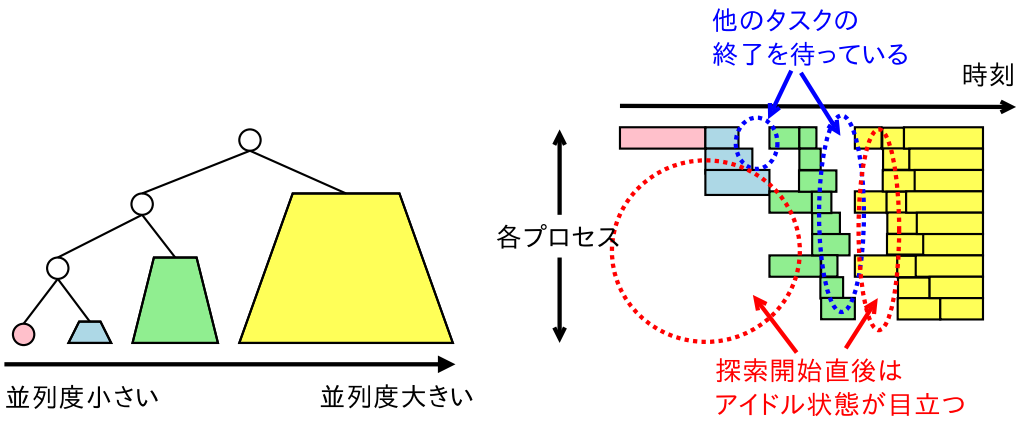

図 5.14: YBWC の問題点

Campbell ら [29] はチェスプログラム DeepBlue を、200 万ノード/秒で探索できる専用のチェスチップをワーカに用いたマスタ・ワーカ方式で実装した。NegaScout の並列化で 480 台で 38 倍から 58 倍程度の速度向上比を達成している。

5.2.2.3　YBWC のアイドル時間を減らすための改良手法

YBWC は無駄な探索を最大限に抑えつつ、並列に実行可能な部分も大きい手法であるが、それでもプロセッサ数が多い場合は、タスクが不足することがある。具体的な例を図 5.14 に示す。図 5.14 の左の図は探索が進むごとに並列に実行可能なタスクが段階的に増えていることを表現している。それぞれの色の部分木に含まれるタスクの各時刻でのプロセスの割り当て例を右の図に示している。YBWC は最左の子ノードの探索が終了してから他の子ノードの探索を開始する。探索を開始した瞬間は同時に実行可能なタスクは 1 つだけである。その後、徐々に並列に実行可能なタスクの数は増えていくが、プロセス数が多いと、探索開始後しばらくはアイドル状態のプロセスが目立つこととなる。これに加え、各ノードですべての子ノードの探索が終了するまで待つことになるので、そこでもプロセスがアイドル状態になってしまう。ゲーム木探索では探索深さや探索窓が同じでも、実行時間が 100 倍程度異なることもあり、子ノードのどれか一つに時間がかかるときはますます深刻な問題となる。さらに、この問題はプロセス数が増えるほど深刻になる。図 5.14 の右側の図の縦方向がプロセス数を示しているが、プロセス数が多いときはアイドル状態が長くなることが直感的に理解できる。さらに、この図には含まれていないが、反復深化を行う場合、ある深さの探索が終わらないと次の深さの探索に進めないため、アイドル時間はさらに長くなる [123]。

これらの問題点を解決するもっとも有力な手段はタスクの粒度を小さくすることである。タスクの粒度が小さければ、探索開始から十分なタスクの数になるまでの時間が短くなる。また、タスクが増えることで負荷分散が行えるようになる。つまり、実行時間がかかるタスクがあったとしても、他のプロセスは実行時間の小さいタスクを数多く実行するようにスケジューリングされるため、ア

イドル時間を少なくできる。

しかし粒度を小さくするのにも限界がある。なぜならば、並列計算のタスクとして実行すると、逐次に実行した場合と比べて、オーバーヘッドが必ず存在するからである。主なオーバーヘッドとしてはタスクを送信するための通信時間などが挙げられる。一般的に並列計算では、このようなオーバーヘッドと比べて実行時間が十分大きくなるようにタスクの粒度を設定しなければならない。

粒度を小さくするのに限界があるのならば、YBWC の問題点を解決するためには他の方法で同時に実行可能なタスクを増やす必要がある。YBWC と同時に提案された改良アルゴリズムである YBWC* [37] では、ALL ノードでは最も有望な子ノードの結果を待たない。もちろん最も有望な子ノードの結果を用いて探索窓を狭めることを行わなくなってしまうので、最も有望な子ノードの結果を待たないと無駄に探索するノード数は増加する。しかし、ALL ノードでは子ノードを最終的にはすべて探索しなければならず、無駄な探索の増加は多くなく、それに比べると同時並列性を大きくする利点の方が大きい。また、ルートノードでは最も有望な子ノードの結果を待たないということも試みられている [55]。

YBWC において、結果を待っていたところを待たずにプロセッサが存在する限り次々と探索させる手法として ABDADA [110, 111] がある。ABDADA では、全プロセスがトランスポジションテーブルを通してあるノードが探索中かそうでないかを判断し、探索中ではないと分かったらそのノード以下の部分木を探索し始める。もし他のプロセッサが探索中であることがわかっても、他の兄弟ノードもすべて探索中ならば、その部分木の探索を手伝うようにして探索を開始する。ABDADA は逐次探索のコードに少し変更を加えるだけでよいのが特徴であるが、枝刈りや探索窓の更新は実装されていない。さらに、ABDADA は全プロセスがグローバルなトランスポジションテーブルへのアクセスが要求されるため、通信遅延が大きい環境には向かない。性能としては 32 プロセッサの CM-5 を用い、NegaScout アルゴリズムを用いた平均実行時間が 2200 秒程度の問題規模のチェスの探索で、15.8 倍の SpeedUp を報告している。

Himstedt ら [46] はチェスを対象に YBWC で 16 コアを用いて並列化したものを 1 ワーカとして、5 ワーカを集め、自分の手番の探索のときに、パイプライン的に相手の手を予測してその探索にも用いることを提案している。これは、YBWC が大規模計算環境ではアイドル状態になりやすいという問題点の解決を図った一つの方法であると言える。2 つのクアッドコアの Xeon CPU(2.66GHz) を 2 ノードと、2 コアの AMD Opteron CPU(2.4GHz)32 ノードの計 80 コアを用い、単純に 80 コアを用いて並列化を行うよりも、相手の手を予測するのに計算資源を割いた方が実質的な速度向上になったと主張しており、その速度向上比は 11.7 倍である。

5.2.2.4 Dynamic Tree Splitting

YBWC は最小探索木に予測に基づいた探索手法であると言える。YBWC では最も有望な子ノードを探索してから残りの子ノードの探索をしていたが、これは CUT ノードに着目すると、予測される最小探索木に含まれないノードの探索は、それが必要だと明らかになるまでは探索を後回しにしているとみなせる。また、前節で述べた改良手法も、ALL ノードなら最も有望な子ノードの結果を待たない、というように、最小探索木の予測に基づいたものであった。

しかし、YBWC では最小探索木の予測は基本的に固定され、明示的には修正されない。CUT ノードにおいて、最も有望な子ノードの探索終了後に枝刈りが起こらなかった場合、残りの子ノードを探索することは、局所的に予測の修正を行っているとみなすこともできるが、ゲーム木全体としては最小探索木の予測はそのままである。

Dynamic Tree Splitting (DTS) [49] は、最小探索木の予測の修正を行っているとみなせるアルゴリズムである。DTS では各ノードが、confidence という値を持つ。これは各ノードがどのくらい ALL ノードだと考えられるか、あるいは CUT ノードだと考えられるかという予測を数値化したものである。ALL ノードだと予測されるノードでは、子ノードは並列に実行されるが、CUT ノードだと予測されるノードでは、子ノードは一つずつしか実行されない。CUT ノードの子ノードの探索がひとつ終了しても枝刈りが起こらなかった場合、そのノードの CUT ノードらしさが減少し、ALL ノードらしさが増加する。DTS では、負荷分散を行うときにどのノードの子ノードの探索を並列に行うかの判断を行うときに、ALL ノードらしさが高いノードを優先的に選ぶようにしている。ただし、この予測が修正されるのは子ノードの探索が終了したときのみであり、途中の結果は用いられない。さらに、DTS は共有メモリ用に設計されたアルゴリズムであるため、分散計算環境で有効かどうかは疑問が残る。

DTS の評価としては、16 プロセッサの Cray C916/1024 コンピュータを用い、11.1 倍の速度向上が報告されている。

5.2.2.5 ニューラルネットを用いた並列に実行する部分の予測

YBWC や DTS ではどこを並列に実行するかに焦点を当てていたが、これをニューラルネットを用いて予測する研究が存在する [70]。ニューラルネットの入力としては、親ノードが PV ノードかどうか、親ノードのタイプ、子ノードの中で何番目か、という情報であり、出力は、そのノードのタイプと、並列に実行するまでにいくつの子ノードを逐次的に探索するか、という値である。ノードのタイプは 0 以上 100 以下の整数で表され、それぞれ、PV ノードらしい、CUT ノードらしい、ALL ノードらしいという意味を持っている。

この研究では、48 コアを模したシミュレータを実装し、人工木を用いて評価している。YBWC や DTS の分割方策と比較して、優位性を示している。

5.2.2.6 激指のスレッド並列化手法

7.5 節でスレッド並列の激指との対戦実験を行っているため、激指のスレッド並列化手法 [119] についても述べる。

激指では YBWC に近い探索手法を用いている。激指では以下のような手順で指し手を生成する。

1. トランスポジションテーブルのチェック、手損チェックなど
2. null-move forward pruning
3. ヒューリスティックな手
4. 浅い読みでの最善手

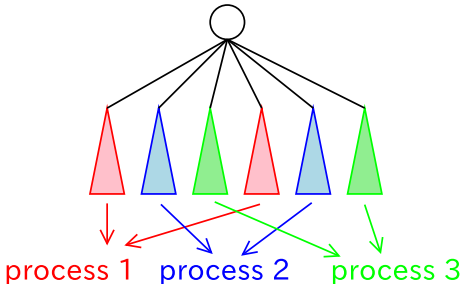

図 5.15: UIDPABS のタスク割り当て

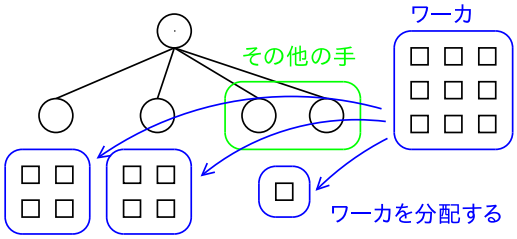

図 5.16: GPS 将棋の分散並列化手法でのタスク割り当て

5. killer-move（存在すれば）
6. パス
7. その他の手（実現確率に従った重みで生成）
8. 詰みチェック（専用アルゴリズムによる）

スレッド並列の激指では、上記の 1 から 6 までは逐次的に実行され、7 のその他の手以降を並列に実行する。これは、訪問ノード数を増やさないようにするために、有望な手については探索の終了を待ちたいからである。なぜなら、有望な手はその探索終了後に枝刈りを起こすことが期待されるからである。

激指におけるタスクスケジューリングは、スレッドがアイドル状態になった場合、まだタスクが存在する任意の並列実行可能なポイントの処理を実行するというものになっている。

5.2.3 非同期型アルゴリズム

5.2.3.1 Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search

基本的な非同期アルゴリズムとして、UIDPABS(Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search) [82] がある。UIDPABS は図 5.15 のように、ルート局面の M 個の指し手を均等に K プロセッサに分配して、反復深化を用いて個別に探索させる。各プロセスの探索途中の結果は、他のプロセスでは利用されない。

5.2.3.2 GPS 将棋の並列化手法

トップレベルの将棋プログラムである GPS 将棋 [8] の分散環境での並列化がマスタ・ワーカ方式で行われている。この手法におけるプロセスへのタスク割り当ての手続きを図 5.16 に示す。ルートノードから左側の子ノードほど多くのワーカを割り当てていきながら、ワーカが 1 つになるまでゲーム木を展開していき、各ワーカが探索を担当することになるノードを指定する。ワーカは時間いっぱいに探索を実行し、最後にマスタが結果をまとめる。性能評価としては、634 コアで並列化して、8 コアマシンでのスレッド並列化に、バグを除いて 13 戦全勝であった [125]。

また、ワーカの割り当てアルゴリズムを子ノードのうちもっとも有望なものから 2 つに重点的に割り当て、それ以外のノードに 1 プロセッサを割り当てた場合も評価しており、1 コア 1 ワーカの 8 ワーカの分散メモリ並列化で、4 コアの共有メモリ並列に 43 勝 50 敗 1 分であった [121]。

5.2.3.3 合議アルゴリズム

強いプレイヤを作ることを考えると、ゲーム木探索を高速化させるのとは異なった合議アルゴリズム [83, 97] という新しい並列化手法も存在する。合議アルゴリズムは、同一のプログラムに乱数を加えたもの、あるいは、異なるプログラムを複数実行させ、複数の指し手を求め、その中から投票によって最終的な指し手を求めるというものである。合議アルゴリズムは簡単に分散並列化が可能であるが、計算資源を大量に投入したときに性能が伸びるかどうかは疑問が残る。

合議アルゴリズムは、異なる乱数で探索した結果をまとめるという点でモンテカルロ木探索におけるルート並列化 [95] とよく似た手法である。また、リーフに与える乱数をパラメータとみなせば、SSAT ソルバの並列化において、異なるパラメータで同時に探索を行わせる portfolio [45] アプローチとも似ており、複数のアルゴリズムで似た手法が提案されているのは興味深い点である。

5.2.3.4 Asynchronous Parallel Hierarchical Iterative Deepening

UIDPABS や、GPS 将棋の並列化手法、合議アルゴリズムとも、並列化に際に最小探索木の情報を用いておらず、効率の良い探索とはなっていない可能性がある。

Asynchronous Parallel Hierarchical Iterative Deepening (APHID) [24, 25] は最小探索木の情報を用いた明示的に子ノードの終了を待つということを行わないアルゴリズムである。APHID は様々なの逐次探索プログラムに適用できるようなフレームワークとして実装されている。APHID はマスタ・ワーカ方式をとる。全体の探索深さを d とする。マスタは深さ d' の探索を繰り返し行う。一回の探索を pass と呼ぶ。マスタが探索しているゲーム木のリーフノードをタスクとみなしてワーカに割り当てるが、マスタはワーカからの結果を待たずに静的評価値を用いて探索を続ける。ワーカは送られてきたリーフノード以下の深さ $d - d'$ の部分木を、反復深化を用いて探索する。ある反復（浅い探索）が終了すると、その都度ワーカは結果をマスタに報告し、少しだけ深い探索を次に実行する。マスタは結果を受け取ると、静的評価値の代わりにその評価値を以降の pass で用いる。これにより、より良い最小探索木の予測が可能になる。その結果、マスタの探索木の形が代わり、ワーカに新しいタスクが送信されることもある。逆に、最小探索木の予測から外れて、枝刈りされそうだと判明したタスクの実行は中止される。マスタが一回の pass を実行した結果、訪問した全てのリーフノード以下の部分木の探索が $d - d'$ で探索が終了していた場合に、全体の探索を終了する。

APHID の重要な性質の一つが、ワーカから得られた結果を用いて、最小探索木の予測がすぐに修正されるということである。つまり、探索途中の結果から、最も有望だった子ノードがあまり有望ではなくなり、他の子ノードが最も有望になるということが起こるということである。図 5.17 は最小探索木の予測の修正の例を示す。図中の数値は評価値であり、ネガマックス形式で記述されている。左側の図はルートの左側の子ノードが右側の子ノードより有望である状況を表している。部分木 D

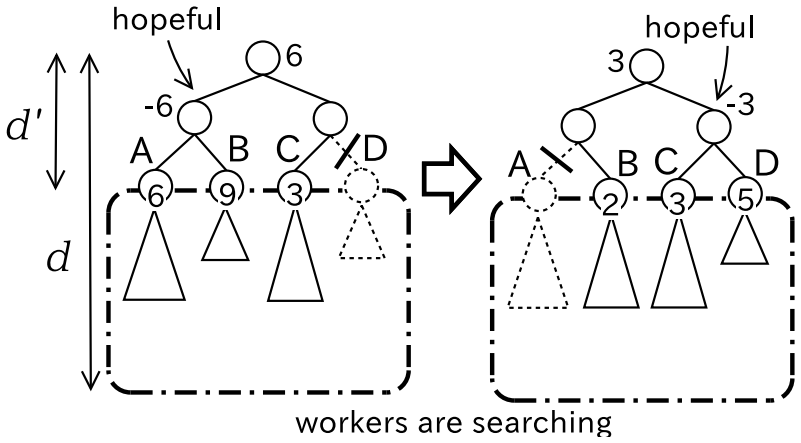

図 5.17: APHID における最小探索木の予測の修正の例

表 5.1: 64 プロセッサでの APHID の速度向上比

Keyano(Othello)	Chinook(Checker)	Crafty(Chess)	TheTurk(Chess)
37.44	14.35	18.00	15.96

はこの時点では枝刈りされると予測されているため探索されていない。右側の図は部分木 B の探索が進み、評価値が 9 から 2 に下がったときの状況を表している。この場合、ルートの右側の子ノードの方が有望になる。結果として、マスタの木の形が代わり、部分木 A の探索は中止され、部分木 D が代わりに探索されるようになる。

APHID はフレームワークとして実装されているため、既存のプログラムに適用するのが容易であり、実際に 4 つの既存のプログラムに適用して評価が行われている。64 プロセッサでの APHID の性能を表 5.1 に示す。

5.2.4 トランスポジションテーブルの共有

並列ゲーム木探索において問題になることとして、分散メモリ環境でのトランスポジションテーブルの共有の問題もある。探索を効率よく行うためにはトランスポジションテーブルを用いて重複した探索を避けることが必要である共有メモリ環境ではすべてのプロセッサが同一のテーブルに高速にアクセスできるので問題はないのだが、分散メモリ環境では他のプロセッサが所有しているテーブルにアクセスするためには通信が必要となり、通信のオーバーヘッドが問題となってしまう。トランスポジションテーブルの共有の方法について、大きく分けて次の 4 パターンがある。

1. トランスポジションテーブルをローカルに所有する

2. トランスポジションテーブルを分割する
3. トランスポジションテーブルの複製を所有する
4. TDS を用いる

一つ目は、そもそもトランスポジションテーブルを共有しないというものであり、エントリの参照も更新もローカルだけで行う。エントリの情報に関する通信は行わない。実装は楽ではあるが、他のプロセッサの計算結果を用いることができないので無駄に探索してしまうノード数が増えてしまう。

二つ目は、トランスポジションテーブル全体を分割し、各プロセッサにその一部ずつを持たせるというものである。プロセッサは自分が担当するテーブル以外のエントリにアクセスするときは該当するプロセッサにメッセージを送る必要がある。エントリの更新は非同期に行うことが可能だが、エントリの参照は、たとえエントリに情報がなかったとしても、通信して結果が返ってくるまで待たなければいけないので、通信にかかる時間が大きくなる。この方法では、プロセッサ全体としてはプロセッサ数だけの大きなトランスポジションテーブルを扱うことができるが利点である。YBWC の実装ではこの方法を用いている [37]。

三つ目は、それぞれのプロセッサはトランスポジションテーブル全体の複製を所有している。エントリへの参照はローカル参照となるので通信が必要ない。エントリの更新は全プロセスにブロードキャストする。プロセッサの数が増加するにつれて通信が多くなってしまう欠点がある。APHID では、エントリの更新の度にブロードキャストする代わりに、重要そうなエントリの内容だけを定期的にプロセス間で循環させることで、情報の共有を行っている。

四つ目は、A*で用いられている手法を $\alpha\beta$ 探索でも用いるというものでもある。TDS では、テーブルへのアクセスをすべてローカルで行うことができるため、テーブルアクセスのための通信コストが存在しないことが最大の利点である。ただし、タスクそのものを次から次に通信する必要があるため、その通信量は多くなり、通信回数も多くなるので通信遅延も大きくなる。しかし、タスクは次から次へ通信されるため、通信遅延が隠蔽されやすく、通信のオーバーヘッドは小さいとされている。また、タスクが十分に多ければ、ハッシュ関数によって自動的に負荷が分散されるのも利点である。

$\alpha\beta$ 探索に TDS を用いた研究を紹介する。

Kishimoto ら [58] は YBWC と TDS を用いて、MTD(f) を並列化している。ゲームとしてはアワリとアマゾンを対象に、CPU が 933MHz の Pentium III を 64 台、ネットワークが 100Mbps の環境で実験を行ったところ、逐次探索時間が 2100 秒程度のアワリで 21.8 倍、逐次探索時間が 1600 秒程度のアマゾンで 23.5 倍の速度向上比が得られている。

Steenhuisen [96] は YBWC と TDS を用いて、PVS を並列化し、チェスプログラム DarkSight の実装を行っている。CPU が 1GHz の Pentium III、ネットワークが 1.2Gbps の環境で、32 台を用いて 5.7 倍の速度向上比が得られている。

第6章　実行が必要となる可能性に着目したタスクの優先度付け

6.1　投機的実行とその制御の必要性

YBWC では、他の部分木の探索の終了を待つためにタスクが足りなくなり、タスクを増やすための改良が提案されていることはすでに述べた。アイドル状態のプロセスが存在する場合は、他の探索の終了を待たずに、YBWC で実行することになっているタスク以外のタスクも実行する手法を、我々も提案している [105]。

この手法では、タスクを「高優先」と「低優先」の 2 種類に分類し、高優先のタスクを実行するだけでは、アイドル状態のプロセスが生じた場合に、低優先のタスクを割り当てる。このとき、低優先のタスクの実行が高優先のタスクの実行を妨げないようにスケジューリングする。つまり、新しく高優先のタスクが発生したり、低優先のタスクが高優先に変化したりしたときに、低優先のタスクしか実行していないプロセスがいた場合、そのプロセスに新しい高優先のタスクが割り当てあられる。このとき、そのプロセスは低優先のタスクの実行を途中で中断し、新しく割り当てられた高優先のタスクを実行してから、低優先のタスクの実行に戻る。実験では、YBWC のタスクだけを実行するよりも PV ノードや ALL ノードでは、もっとも有望な子ノードの探索の終了を待たない方が性能がよいことを示した。さらに、PV ノードや ALL ノードでもっとも有望な子ノードの終了を待たずに生成されるタスク、つまり、予測される最小探索木に含まれるタスクを高優先と分類し、CUT ノードでもっとも有望な子ノードの終了を待たずに生成されるタスクを低優先と分類した場合、タスクを分類しないと性能が著しく劣化するが、タスクを分類して高優先のタスクの実行が妨げられないようにスケジューリングすることで、その性能劣化を抑えることができることを示した。しかし、低優先のタスクを投機的に実行することで得られた性能向上は、タスクの粒度が大きい場合に少し見られた程度であった。

このとき低優先のタスクの実行は、深さ優先探索順で先に探索されるものを先にやるという簡単なものであった。しかし、実際には低優先のタスクの中でも、実行が必要になる可能性が高いタスクと低いタスクが存在するはずである。したがって投機的実行を行う場合は、実行が必要になる可能性の高いタスクを先に実行するように制御するべきである。本章では、ゲーム木中の各ノードの探索が必要になる可能性を確率として見積り、必要になる確率が大きいノードから先に最良優先探索する手法を提案する。つまり、図 6.1 に示すように、未展開ノードのうち、最も探索が必要となる確率の高いノードを展開し、子ノードをゲーム木に付け加えることによって探索を進める。

本章では、6.2 節において、親ノードの探索が必要となる確率が計算されているときに、各子ノー

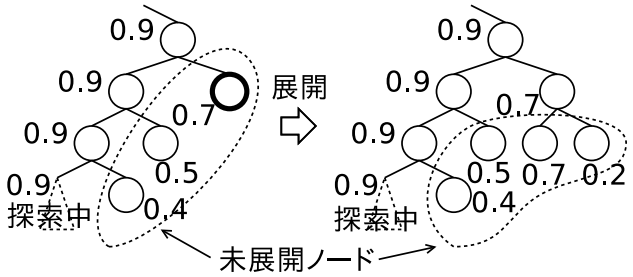

図 6.1: 探索の進行例

ドが枝刈りされない確率をそれぞれ見積り、それを親ノードの探索に乗じることによって、子ノードの探索が必要となる確率を計算する手法を述べる。次の 6.3 節では、各部分木の探索結果を確率分布で予測することにより、各ノードが探索されるときの探索窓の α と β の値を確率分布で計算し、そこから、そのノードの探索が必要となる確率を計算する手法を述べる。

6.2 子ノードの確率を用いる手法

ここでは、ノード n の探索が必要となる確率 P_n を n の訪問確率と呼ぶ。全てのノードの P_n を推定し、並列探索の優先度に用いることで、P_n の大きいノードから順に最良優先探索する手法を提案する。探索の必要性が判明しているノードでは $P_n = 1$、投機的に探索されるノードでは $P_n < 1$ となり、提案手法は投機的な探索の実行順序を制御していることになる。

P_n を計算する方法を大まかに述べる。ノード n の $i(= 1, 2, ...)$ 番目の子ノードを n_i と書く。n_i は有望な順に $n_1, n_2, ...$ となるように常に並び替えられているものとし、特に n_1 を n の最左の子ノードと呼ぶ。このとき、n の子ノードの中で n_i の評価値が最大となる確率 b_{n_i} と、$n_1...n_i$ を順に探索したとき、n_i を探索して n で初めて枝刈りが発生する確率 c_{n_i} を推定する (その推定法は 6.2.3 節で述べる)。b_{n_i} を n_i の最善確率、c_{n_i} を n_i の枝刈り確率と呼び、それぞれ i について和をとると 1 になる。最善確率と枝刈り確率は、ノード n における指し手の並び替えの尺度の一種と考えられる。この b_{n_i} と c_{n_i} を用いて、親ノード n の P_n から子ノード n_i の P_{n_i} を求めていき、全てのノードの P_n を計算する。

本節の構成は以下の通りである。訪問確率 P_n の計算には、最小探索木の概念を用いているが、それに加えて、あるノードがその時点で YBWC で探索中の木に含まれるかどうかという考え方も用いている。そこでまず、6.2.1 節で、YBWC で探索中の木について述べる。その後、6.2.2 節で訪問確率 P_n の詳細な計算方法を説明し、6.2.3 節で最善確率 b_{n_i} と枝刈り確率 c_{n_i} の推定方法を述べる。続く 6.2.4 節で行った実験とその結果について述べ、6.2.5 節でその結果に対する考察を述べる。

6.2.1 YBWC で探索中の木

YBWC は多くの並列 $\alpha\beta$ 探索に用いられている手法であり、逐次探索では探索する必要がなかったノードを無駄に探索することの少ない手法である。そこで、その時点で YBWC で探索中の木に含まれるノードでは、優先度が最大となるように設計する。

YBWC は、全てのノード n において、n_1 の探索の終了を待った後で、残りの子ノードを並列に探索する手法である。そこで、その時点で YBWC で探索中の木 T_{YBWC} を次のように定義する。YBWC で探索中の木は探索の各時点ごとに定義され、探索の進行と共に変化する。以下ではノード n が YBWC で探索中の木に含まれることを、$n \in T_{\mathrm{YBWC}}$ と表記する。

- ルートノード root は root $\in T_{\mathrm{YBWC}}$
- $n \in T_{\mathrm{YBWC}}$ のとき、
  - n_1 の探索が終了していないならば、
    $n_1 \in T_{\mathrm{YBWC}}$ であり、かつ、
    $i \geq 2$ について $n_i \notin T_{\mathrm{YBWC}}$
  - n_1 の探索が終了しているならば、探索が終了していない $n_i (i \geq 2)$ について $n_i \in T_{\mathrm{YBWC}}$
- $n \notin T_{\mathrm{YBWC}}$ ならば全ての i について $n_i \notin T_{\mathrm{YBWC}}$
- n の探索が終了しているならば、$n \notin T_{\mathrm{YBWC}}$

6.2.2 訪問確率 P_n の計算方法

本研究では、最小探索木に着目し、P_n を式 (6.1) のように 3 つの成分の和で書き表す。

$$P_n = P_n^{\mathrm{PV}} + P_n^{\mathrm{CUT}} + P_n^{\mathrm{ALL}} \tag{6.1}$$

おおまかに述べれば、P_n^{PV}、P_n^{CUT}、P_n^{ALL} は n がそれぞれ PV ノード、CUT ノード、ALL ノードのように探索される確率である。ルートノードでは、$P_{\mathrm{root}}^{\mathrm{PV}} = 1$、$P_{\mathrm{root}}^{\mathrm{CUT}} = P_{\mathrm{root}}^{\mathrm{ALL}} = 0$ であり、子ノードの値は親ノードから式 (6.2)、(6.3)、(6.4) で算出される。

$$P_{n_i}^{\mathrm{PV}} = q_{n_i} P_n^{\mathrm{PV}} \tag{6.2}$$

$$P_{n_i}^{\mathrm{CUT}} = (1 - q_{n_i}) P_n^{\mathrm{PV}} + P_n^{\mathrm{ALL}} \tag{6.3}$$

$$P_{n_i}^{\mathrm{ALL}} = r_{n_i} P_n^{\mathrm{CUT}} \tag{6.4}$$

q_{n_i} は n が PV ノードとして探索されると仮定した場合に、YBWC において n_i が PV ノードと予測されるか、あるいは実際に PV ノードとなる確率である。$i = 1$ のとき $q_{n_1} = 1$ で、$i \geq 2$ のとき $q_{n_i} = b_{n_i}$ である。ここで $q_{n_1} = 1$ としたのは、PV ノードの最左の子ノードは PV ノードと予測して探索することを考慮したためである。

r_{n_i} は n が CUT ノードとして探索されると仮定した場合に、YBWC において n_i を探索することになる確率であり、式 (6.5)、(6.6)、(6.7) を使い分ける。ただし、まだ探索が終了していない n の子ノードの集合を S_n とする。

- $n \in T_{\mathrm{YBWC}}$ かつ n_1 の探索が終了していないとき

$$r_{n_i} = 1 - \frac{\sum_{(n_j \in S_n) \wedge (j<i)} c_{n_j}}{\sum_{n_j \in S_n} c_{n_j}} \tag{6.5}$$

- $n \in T_{\mathrm{YBWC}}$ かつ n_1 の探索が終了しているとき

$$r_{n_i} = 1 \tag{6.6}$$

- $n \notin T_{\mathrm{YBWC}}$ のとき

$$r_{n_i} = 1 - \sum_{j<i} c_{n_j} \tag{6.7}$$

CUT ノードでは枝刈りを引き起こす子ノードを一つ探索すればよい。YBWC で探索中の木に含まれる CUT ノードでは、子ノードの探索が終了しても枝刈りが発生しなかった場合、枝刈りを引き起こす可能性があった候補が一つ減ると考えられるため、残った子ノードが枝刈りを起こす確率は大きくなる。また、兄ノードの探索が全て終了しても枝刈りが発生しない場合は、次の子ノードを探索する必要がある。これらを表現したのが式 (6.5) である。YBWC で探索中のノードの最左の子ノードが終了したときは、残りの子ノードは YBWC で探索中の木に含まれる。よって、探索する必要があると判断されるため式 (6.6) を用いる。YBWC で探索中の木に含まれない CUT ノードでは、探索が進むことで後から探索窓が狭まる可能性が高い。すなわち、子ノードの探索が終了してその時点で枝刈りが発生しなかったとしても、その子ノードの探索結果が枝刈りの原因にならないとは断定できない。よって、他の子ノードが枝刈りを引き起こす確率は変わらないと考え、式 (6.7) を用いる。なお、r_{n_i} が探索の進行によって変化した場合は、全ての子孫ノードの P_n を直ちに更新する必要がある。

以上で計算される訪問確率 P_n は、n が予測される最小探索木に含まれるか、あるいは、YBWC で探索中の木に含まれる場合は、$P_n = 1$ となるように設計されている。さらに、子ノードの訪問確率は必ず親ノードの訪問確率以下である。つまり、全ての n と i について、$P_{n_i} \leq P_n$ が成り立つ。

6.2.3 最善確率と枝刈り確率の推定方法

ここまでは最善確率 b_{n_i} と枝刈り確率 c_{n_i} が既知のものとして話を進めてきたが、実際に利用するにはこれらの確率を推定する必要がある。推定に使う量としては、実現確率 [104] を用いたプログラムに対して次の二つを用いる。一つは残り探索深さ (実現確率の対数から負号を取り去ることで探索深さとなる) であり、もう一つは指し手の遷移確率の対数から負号を取り去ったもの (以下、指し手の遷移評価値と呼ぶ) である。指し手の遷移確率は、指し手の並び替えにも用いられる情報であり、並び替えの尺度である b_{n_i} や c_{n_i} の推定に用いるのは妥当と判断した。

確率を推定するために、逐次の $\alpha\beta$ 探索を繰り返し行いヒストグラムを作成する。つまり、推定に使う量がとりうる値を K 個の領域 $R_1, ..., R_K$ に区切り、各領域 R_k に入るノードの数を数える。ヒストグラムは複数作成するが、各ヒストグラムでは数える対象として着目するノードが異なる。

b_{n_i} は n を PV ノードと仮定したときに、c_{n_i} は n を CUT ノードと仮定したときに用いられる。よって、b_{n_i} を推定するためのヒストグラムを作成するときは、α 値が少なくとも 1 回は更新され、かつ枝刈りが発生しなかったノード n だけに着目し、c_{n_i} を推定するためのヒストグラムを作成するときは、枝刈りが発生したノード n だけに着目する。

今回は $i = 1$ のときの確率と $i \geq 2$ のときの確率を別に求める。これは最左の子ノードは重要であるため、分けて確率を推定するべきだと考えたからである。ここで、遷移評価値の用い方を $i = 1$ のときと $i \geq 2$ のときで変更する。$i = 1$ のときは最左の子ノードの遷移評価値 v_{n_1} と、n_1 以外の子ノードの遷移評価値 v_{n_i} の最小値との差 v_n^{diff} を用いる。v_n^{diff} が大きいほど n_1 が有望であることが期待される。

$$v_n^{\mathrm{diff}} = \min_{i \geq 2} v_{n_i} - v_{n_1} \tag{6.8}$$

これに対し、$i \geq 2$ のときは n_i の遷移評価値の逆数と、n の最左以外の子ノードの遷移評価値の逆数の和との比を用いる。

$$f_{n_i} = \frac{1/v_{n_i}}{\sum_{j \geq 2}(1/v_{n_j})} \tag{6.9}$$

逆数を用いたのは、f_{n_i} が大きいほど c_{n_i} も大きくなるようにするためである。ここで、v_n^{diff}、f_{n_i} を用いたときの R_k をそれぞれ R_k^1、R_k^2 とする。

$i = 1$ については、着目したノードが R_k^1 に属するときに、そのノードの子ノードのうち n_1 が最善だった、あるいは枝刈りを引き起こした割合を調べ、b_{n_1} や c_{n_1} とする。$i \geq 2$ については、最左が最善でもなく枝刈りも引き起こさないと分かっているときに、n_i が最善となる確率と、n_i で初めて枝刈りを引き起こす確率を、それぞれ b'_{n_i} と c'_{n_i} とし、これらを求める。そこで、n に着目し、かつ、n_1 が最善でもなく枝刈りも引き起こさなかったときに、2 番目以降の各子ノードが R_k^2 に属する回数をそれぞれ数える。また、このときに、最善の子ノードあるいは枝刈りを引き起こした子ノードが領域 R_k^2 に属する回数を数える。この二つの割合を b'_{n_i} や c'_{n_i} とする。

最後に b_{n_i} や c_{n_i} の i に関する和が 1 となるように正規化を行う。ただし、n_1 を別に扱っていることに注意する。

$$b_{n_i} = (1 - b_{n_1}) \times \frac{b'_{n_i}}{\sum_{j \geq 2} b'_{n_j}} \tag{6.10}$$

$$c_{n_i} = (1 - c_{n_1}) \times \frac{c'_{n_i}}{\sum_{j \geq 2} c'_{n_j}} \tag{6.11}$$

6.2.4 評価

まず、将棋のプログラムを用いて、300 局の自己対戦を行うことで $\alpha\beta$ 探索を繰り返し実行し、ヒストグラムを作成することで b_{n_i} と c_{n_i} を推定した。

その後、人工ゲーム木を用いたシミュレーションを行い、確率が既知の場合での評価を行った。

次に、将棋のゲーム木を用いて並列探索のシミュレーションを行い、実行時間を調べた。これにより、提案手法で必要な確率の計算や更新などの処理にかかる時間を無視した場合の性能を評価した。

最後に、実際に並列探索を行ったときの探索時間を測定し、提案手法の処理にかかる時間も含めた性能の評価を行った。

6.2.4.1 評価に用いたプログラム

本研究では並列プログラムとして、マスタ・ワーカ方式で実装された [118] と同じ実装のものを用いた。マスタは残り探索深さがある決められた値 (ワーカの探索深さ) 以下になるまでゲーム木を展開し、そのリーフノードをタスクとして一つずつワーカに送信し、ワーカから返ってきた探索結果を用いて探索を進めていく。マスタとワーカは既存の将棋プログラムである激指 [7] を用いて C++ 言語で実装した。

マスタでは、多重反復深化と激指の実現確率を用いた $\alpha\beta$ 探索を行う。反復深化を行うときは浅い探索の終了を待った後で、全ての子ノードを展開する。次に子ノードを展開するノードを選ぶときは、提案手法を用いて訪問確率 P_n が最も高いノードを選ぶ。リーフノードが選ばれたときは、タスクとしてワーカに送信する。P_n が同じノードが複数あるときは、深さ優先探索で先に探索するノードを選んだ [118][96]。なお、実現確率を用いるときには、評価値の更新を起こした子ノードは再探索を行うのが一般的であるが、この再探索は未実装である。

ワーカでは、タスクを受け取って激指の探索関数を実行し、結果をマスタに返すことを繰り返す。

b_{n_i} や c_{n_i} を推定するときに用いるプログラムには、マスタに相当する処理を再帰呼び出しを用いた $\alpha\beta$ 探索で実装し、残り探索深さがワーカの探索深さ以下になると激指の探索関数を呼び出すようにしたものを用いた。ワーカが 1 台のときの探索時間の測定にもこのプログラムを用いた。

N 台のワーカを用いた並列探索のシミュレーションは、全てのリーフの評価にかかる時間は等しく 1 ステップで、リーフの評価以外の操作には全く時間がかからないものとして行った。これは、リーフノードの評価としてある程度大きな部分木の探索を行う場合を想定したものである。ゲーム木を展開していき、未展開のどの中間ノードよりも優先度が大きい未評価のリーフノードが N 個になったときに、それらを全て評価して結果をゲーム木に反映させるという操作を 1 ステップとしてこれを繰り返し、探索終了までにかかったステップ数を実行時間とした。

リーフは実際の並列探索では部分木であり、この探索時間は実際には大きくばらつく上、探索窓によっても大きく異なる。これを全て同じ時間しかかからないとみなすのは非常に乱暴である。したがって、このシミュレーションは実際の並列探索をシミュレーションしているわけではない。つまり、このシミュレーション方法で性能が良かったからといって実際の並列探索で性能が良いとは限らない。しかし、各ワーカが評価関数を呼んだ回数を数えていくというシミュレーションは非常に時間がかかると思われるため、今回のシミュレーション方法をとった。理想的な場合でも性能が良くなければ、実探索で性能が良いことは期待できないため、理想的な場合の性能を確認する意味はあると考えた。

提案手法の比較対象としては、提案手法において常に $b_{n_1} = c_{n_1} = 1.0$ としたもの (深さ優先) を用いた。これは、最小探索木と YBWC で探索中の木に含まれるノードを優先的に扱い、それ以外の

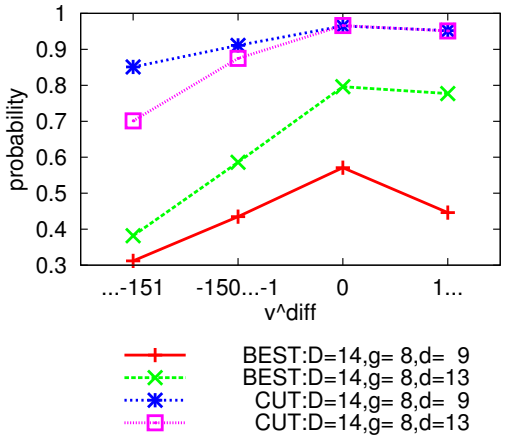

図 6.2: $D = 14, g = 8$ のときの b_{n_1} と c_{n_1}

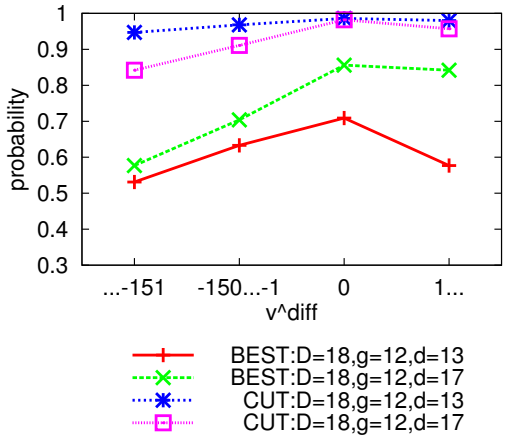

図 6.3: $D = 18, g = 12$ のときの b_{n_1} と c_{n_1}

ノードを深さ優先探索順に投機的実行する手法 [105] と同一となる。

6.2.4.2 訪問確率と枝刈り確率の推定結果

全体の探索深さ D と、ワーカの探索深さ g の組み合わせを $D = 14, g = 8$ と $D = 18, g = 12$ に設定し、b_{n_i} と c_{n_i} を推定した。v_n^{diff} は -150 未満、-150 以上 0 未満、0、1 以上の 4 区間に分け、f_{n_i} は 0.01 未満、0.01 以上は 0.01 刻みで b'_{n_i} は 0.08 未満まで、c'_{n_i} は 0.07 未満までと、それ以上の区間に分けてヒストグラムを作成した。ちなみに激指では遷移評価値の最小値は 100 であり、これは遷移確率が 1/4 のときである。残り探索深さ d_n は 0.5 刻み (実現確率を用いているので深さも実数である) でヒストグラムを作成しているが、そのうち、$D = 14, g = 8$ の場合に、d_n が 9 以上 9.5 未満のとき ($d = 9$) と 13 以上 13.5 未満のとき ($d = 13$) の結果を、$D = 18, g = 12$ の場合に、d_n が 13 以上 13.5 未満のとき ($d = 13$) と 17 以上 17.5 未満のとき ($d = 17$) の結果のみを示す。

まず、最左の子ノードについて b_{n_1} と c_{n_1} の推定結果を示す。図 6.2 と図 6.3 はそれぞれ $D = 14, g = 8$ と $D = 18, g = 12$ の結果である。各図の横軸は v^{diff} の範囲を示しており、縦軸が推定した確率である。また、BEST で示されているのが b_{n_1} であり、CUT で示されているのが c_{n_1} である。全体的な傾向として、v^{diff} が 0 より小さくなるほど、b_{n_1} も c_{n_1} も小さくなっている。また、図 6.2 と図 6.3 を比較すると、全体の探索深さが大きいほど、b_{n_1} も c_{n_1} も大きく、さらに v^{diff} の変化の影響を受けにくくなっている。特に c_{n_1} に関しては、$D = 18, d = 13$ のときは小さくても 0.9 を越えている。提案手法はノード間で b_{n_1} や c_{n_1} の差が激しいほど効力が発揮されると考えられるが、遷移確率だけを用いた本推定手法ではうまくノードを峻別できていないことが分かる。

次に、最左以外の子ノードについて $b'_{n_{i\geq 2}}$ と $c'_{n_{i\geq 2}}$ の推定結果を示す。図 6.4 が $D = 14, g = 8$ の結果であり、図 6.5 が $D = 18, g = 12$ の結果である。縦軸が推定した確率であり、横軸は f_{n_i} がどの値未満かを示している。ただし、図中の最も右側の点は、それより左側の点よりも f_{n_i} が大きいも

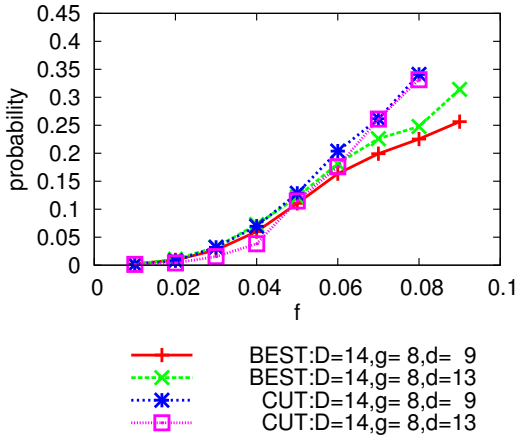

図 6.4: $D = 14, g = 8$ のときの b'_{n_i} と c'_{n_i}

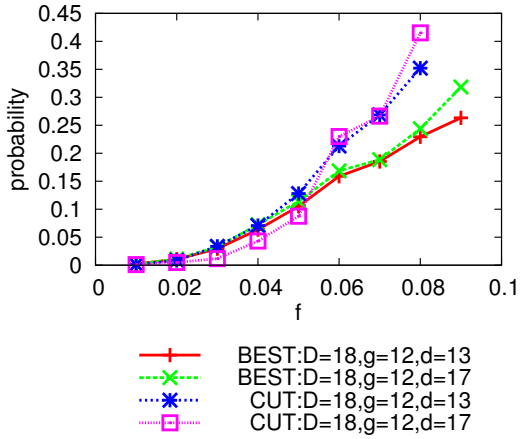

図 6.5: $D = 18, g = 12$ のときの b'_{n_i} と c'_{n_i}

の全てをまとめて表している。図 6.4 と図 6.5 によると、f_{n_i} が大きくなるほど、$b'_{n_{i\geq 2}}$ と $c'_{n_{i\geq 2}}$ も大きくなる傾向にある。また、D や g、d の変化による影響は小さかった。

6.2.4.3 人工ゲーム木を用いたシミュレーション

b_{n_i} や c_{n_i} の推定に誤差がない状況における提案手法の有効性を評価する。ここでは簡単のため $b_{n_i} = c_{n_i}$ とし、b_{n_i} の確率分布を前もって与えた深さ 16 の三分木の人工ゲーム木を探索させるシミュレーションを行った。人工ゲーム木の作成方法であるが、まず全てのリーフノードにランダムに評価値を与えた後に、全てのノードの minimax 値を求めて子ノードを正しく並び替える。次に、全ての中間ノードについて b_{n_i} を確率分布に従ってランダムに選ぶ。例えば表 6.1 の確率分布を用いる場合は、確率 0.95 で $b_{n_1} = 1$、$b_{n_2} = b_{n_3} = 0$、確率 0.05 で $b_{n_1} = 0.73$、$b_{n_2} = 0.2$、$b_{n_3} = 0.07$ とする。そして b_{n_i} に従って並び替えを間違えさせる。例えば $b_{n_1} = 0.73$、$b_{n_2} = 0.2$、$b_{n_3} = 0.07$ では、確率 0.2 で 2 番目に minimax 値が大きい子ノードが n_1 になり、そのときの n_2 は確率 $\frac{0.73}{0.73+0.07}$ で最も minimax 値が大きい子ノード、確率 $\frac{0.07}{0.73+0.07}$ で 3 番目に minimax 値が大きい子ノードとなる。以上の操作で 100 個の人工ゲーム木を作成して探索させた。

確率分布が表 6.1 と表 6.2 のときのそれぞれについて、表 6.3 と表 6.4 に実行ステップ数の相乗平均と速度向上比を示す。表 6.1 の分布は理想的な場合を、表 6.2 の分布はより現実的な場合を表している。両方の確率分布で、ワーカ数が多いほど二つの手法の差は大きくなっている。ワーカ数が 4096 のときの結果に注目すると、表 6.3 では提案手法である提案手法の実行時間は深さ優先の 19%であり、大きな性能改善となっている。表 6.4 でも提案手法の実行時間は深さ優先の 48%と性能改善になっているが、表 6.3 と比較すると二つの手法の性能差は小さくなっている。

表 6.1: b_{n_i} の確率分布 1

選ばれる確率	b_{n_1}	b_{n_2}	b_{n_3}
0.95	1.00	0.00	0.00
0.05	0.73	0.20	0.07

表 6.2: b_{n_i} の確率分布 2

選ばれる確率	b_{n_1}	b_{n_2}	b_{n_3}
0.95	0.93	0.05	0.02
0.05	0.73	0.20	0.07

表 6.3: b_{n_i} の確率分布が表 6.1 のときの実行ステップ数

	ワーカ数	1	16	256	4096
提案手法	平均ステップ数	14865	1032	69.1	6.3
	速度向上比	1	14	215	2359
深さ優先	平均ステップ数	14865	1032	80.8	33.2
	速度向上比	1	14	183	447

6.2.4.4 将棋のゲーム木を用いたシミュレーション

深さ 14 の激指を 30 局自己対戦させ、各対局から局面を 1 つずつ抽出し、序盤、中盤、終盤が 10 局面ずつの合計 30 局面を用いて、将棋のゲーム木の並列探索をシミュレートした。全体の探索深さ D は 14、ワーカの探索深さ g は 8 に設定した。b_{n_i} や c_{n_i} も $D = 14, g = 8$ で推定したものを用いた。

まず、表 6.5 にワーカ数を変化させたときの、30 局面の実行ステップ数の相乗平均をとったものと速度向上比を示す。表 6.5 によれば、提案手法の実行ステップ数は witout Prob の 92%になっている。

次に、ワーカが 4096 台のシミュレーションにおける、30 局面それぞれの実行ステップ数を図 6.6 に示す。横軸が各局面を表し、縦軸がその局面を探索したときの実行ステップ数である。図 6.6 では、提案手法の方が実行ステップ数が小さい局面が 18 局面、深さ優先の方が実行ステップ数が小さい局面が 10 局面、両者の実行ステップ数が等しい局面が 2 局面である。提案手法の優先度は深さ優先探索順の優先度と比較して、常に良い性能になるとは限らない。ある特定の局面に着目すれば、深さ優先探索順の優先度を用いた方が結果的に早く探索が終了することはありえるためである。そのため、図 6.6 の結果は妥当なものである。

6.2.4.5 並列探索の実行時間による評価

6.2.4.4 節で用いた 30 局面を実際に並列探索させた。全体の探索深さ D は 18、ワーカの探索深さ g は 12 に設定した。b_{n_i} や c_{n_i} も $D = 18, g = 12$ で推定したものを用いた。各局面は 30 回ずつ探索した。実験に用いたクラスタ環境は表 4.1 の Xeon E5530 のノードである。1 ノードには 8 コアあり、ワーカ数は常にコア数と等しい 8 に設定して実験した。

表 6.4: b_{n_i} の確率分布が表 6.2 のときの実行ステップ数

	ワーカ数	1	16	256	4096
提案手法	平均ステップ数	23725	1834	312	77.0
	速度向上比	1	13	76	308
深さ優先	平均ステップ数	23725	1837	431	160.1
	速度向上比	1	13	55	148

表 6.5: 将棋のゲーム木を用いたシミュレーションにおけるワーカ数と実行ステップ数の関係

	ワーカ数	1	16	256	4096
提案手法	平均ステップ数	14338	984	75.5	20.3
	速度向上比	1	14	189	706
深さ優先	平均ステップ数	14338	1030	87.2	22.0
	速度向上比	1	13	164	651

まず、ワーカ数を変化させていったときの並列探索の実行時間とその標準偏差を示す。速度向上比も合わせて示す。表 6.6 に示す。各局面の 30 回の試行で相加平均と標準偏差を求め、それらの 30 局面での相乗平均を求めた。ただし、ワーカが 1 台のときは 6.2.4.1 節で述べたように、再帰呼び出しを用いて実装したプログラムを用い、各局面の探索は 1 回だけ行った。表 6.6 を見ると、提案手法と深さ優先の両方とも 976 台まで実行時間は短くなっているが、二つの手法で性能の差は見られなかった。

次に、ワーカが 976 台の実際の並列探索における、30 局面それぞれの実行時間を図 6.7 に示す。横軸が各局面を表し、縦軸は実行時間 [秒] である。図中の各点は各局面を 30 回探索したときの相加平均を示す。図 6.7 には、分散を表すエラーバーを入れていないが、全ての局面で片方の点はもう片方のエラーバーに含まれており、提案手法と深さ優先はほぼ同じ性能であると判断された。

6.2.5 考察

人工木のシミュレーションでは提案手法と従来手法で性能差があったが、将棋のゲーム木を用いると、シミュレーションでは性能差が小さくなり、実探索では性能差がなくなった。この理由を考察する。

まず、$P_n = 1$ となるノードが数多く存在し、そもそも投機的実行をしていないという可能性がある。しかし、実行したタスクの中で $P_n = 1$ であった割合を調べた結果、4096 台のワーカを用いた

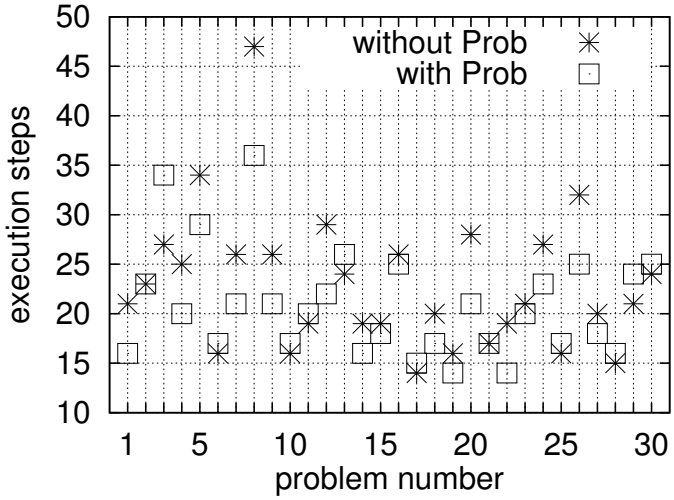

図 6.6: 将棋のゲーム木を用いたシミュレーションにおけるワーカ数 4096 のときの実行ステップ数

シミュレーションで 10%から 60%、976 台のワーカを用いた実探索で 10%から 35%であり、多くの投機的実行が行われていることを確認した。

次に、提案手法では確率の更新が多く、その処理に時間がかかっていた可能性がある。そこで、マスタでのゲーム木の訪問確率の更新にかかった時間の、探索時間に対する割合を調べたところ、ワーカが 976 台のとき、$P_n = 0$ から $P_n = 1$ へ変更するだけの深さ優先で 0.07%、提案手法で 2%であり、提案手法の処理に確かに時間がかかっているが、表 6.6 の標準偏差に比べれば小さく、大きな問題ではなかった。

最後に、最善確率 b_{n_i} や枝刈り確率 c_{n_i} の推定に問題があった可能性がある。人工木では大きかった性能差が将棋のゲーム木では小さくなったことを考慮すると、この可能性は大きい。6.2.4.2 節で述べたように、残り探索深さと指し手の遷移確率だけを用いた推定では、ノード間の違いをうまく表現出来なかったために、提案手法を用いた場合も従来手法に近い順序で探索を行っていたと考えられる。

6.3 確率分布を用いる手法

6.2 節では、各子ノードが探索される確率を予測する手法を提案した。しかし、この手法は、各子ノードが最善の子ノードになる確率と、枝刈りを起こす確率というものを別々に推定する必要があり、この推定の手間が問題となる。別々に推定する必要が発生するのは、枝刈りに重要であるは探索窓の情報を用いていないのが理由だと考えられる。

そこで本節では、ノードが枝刈りされない確率を求めるために、探索窓を確率分布で予測し、そこから枝刈りされない確率を計算する手法を提案する。

表 6.6: 並列探索におけるワーカ数と実探索時間の関係 [単位:秒]

	ワーカ数	1	16	64	256	976
提案手法	平均ステップ数	84.89	19.22	11.61	7.18	5.78
	(標準偏差)		2.92	1.65	0.78	0.56
	速度向上比	1	4.17	7.31	11.8	14.7
深さ優先	平均ステップ数	84.89	20.33	11.13	7.10	5.76
	(標準偏差)		3.44	1.57	0.80	0.52
	速度向上比	1	4.18	7.63	12.0	14.7

6.3.1 探索窓の推定

本研究では探索窓の α と β を確率分布を用いて予測することを提案する。探索窓は他のノードの評価値によって決まるため、探索窓を予測するためには他のノードの評価値を予測しなければならない。本研究では、ノードを探索して得られる評価値は、それより浅い探索で得られる評価値の周りに正規分布すると仮定する。さらに、この仮定を多重反復深化の中で利用する。つまり、各ノードの探索では、まず浅めの探索を実行して得られた指し手が最も有望だとして子ノードを有望な順に並び替える。このとき、浅めの探索の評価値を用いて最も有望な子ノードの評価値を正規分布で予測する。これにより他の子ノードの探索窓の α と β を正規分布で推定する。

探索窓の推定の詳細を述べる。提案手法では、各ノード n は 8 個の値からなる窓情報を持つ。それぞれ、並列探索窓 (α_n, β_n)、逐次探索窓 (α'_n, β'_n)、推定探索窓の正規分布の平均 $(\mu^\alpha_n, \mu^\beta_n)$ と標準偏差 $(\sigma^\alpha_n, \sigma^\beta_n)$ である。並列探索窓は、並列探索において、既に得られた結果から枝刈りの有無を判定するために用いる探索窓である。これは、子ノードが終了して得られた評価値を兄ノードも含めた残りの子ノード全ての探索窓に反映 (図 6.8) して得られる探索窓である。今回、この並列探索窓は提案手法の確率の計算には用いないこととした。逐次探索窓は、ゲーム木を逐次探索と同じ順序でノード n まで辿ったときに通過するノードのうち、探索が終了したノードの結果を用いて求めた探索窓である。つまり、各子ノードの探索が終了したときに、得られた評価値を弟ノードのみの探索窓に反映 (図 6.9) して得られる探索窓である。並列探索窓と逐次探索窓の関係として、$\alpha'_n \leq \alpha_n$ かつ $\beta_n \leq \beta'_n$ が満たされる。逐次探索窓を提案手法の確率の計算に用いる。

次に、最左以外の子ノードについて ゲーム木中のノードの窓情報の求め方を述べる。基本的にはネガマックス形式の $\alpha\beta$ 探索の探索窓の計算と同様である。今ノード n の窓情報が分かっているとする。n の浅めの探索では元々の窓情報と同じ窓情報を用いる。浅めの探索の終了後は、n の子ノードを有望な順に $c_1, c_2, \ldots, c_k$ とすると、全ての子ノード c_i について、$\mu^\beta_{c_i}$ と $\sigma^\beta_{c_i}$ 以外の窓情報は式 (6.12) から式 (6.17) となる。v_i は c_i の探索が既に終了している場合はその評価値、そうでない場合は $-\infty$ とする。図 6.8 と図 6.9 に、式 (6.13) と式 (6.15) による窓情報の更新をそれぞれ示す。

$$\alpha_{c_i} = -\beta_n \tag{6.12}$$

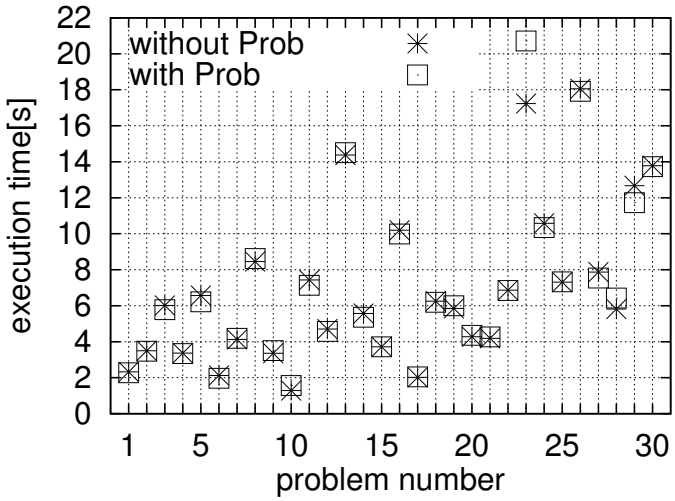

図 6.7: ワーカ 976 台での実探索時間

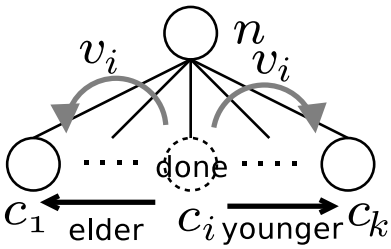

図 6.8: 並列探索窓の更新方向

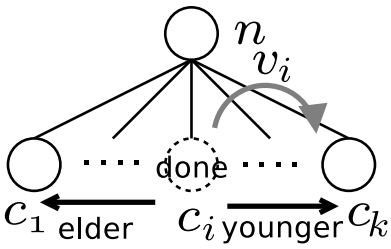

図 6.9: 逐次探索窓の更新方向

$$\beta_{c_i} = -\max\{\alpha_n, \max_{j \neq i}\{v_j\}\} \tag{6.13}$$

$$\alpha'_{c_i} = -\beta'_n \tag{6.14}$$

$$\beta'_{c_i} = -\max\{\alpha'_n, \max_{j < i}\{v_j\}\} \tag{6.15}$$

$$\mu^{\alpha}_{c_i} = -\mu^{\beta}_n \tag{6.16}$$

$$\sigma^{\alpha}_{c_i} = \sigma^{\beta}_n \tag{6.17}$$

$\mu^{\beta}_{c_i}$ と $\sigma^{\beta}_{c_i}$ については、最左の子ノード c_1 では、$\mu^{\beta}_{c_1} = -\mu^{\alpha}_n$ かつ $\sigma^{\beta}_{c_1} = \sigma^{\alpha}_n$ とする。次に c_1 の評価値は、n の浅めの探索の評価値 r_n(c_1 の浅めの探索の評価値に等しい) から ε_n だけずれた周りに標準偏差 σ^r_n で正規分布すると予測する。今回は簡単に c_1 の探索終了までは $r_n + \varepsilon_n$ を c_1 の予測評価値とみなし (図 6.10)、最左以外の子ノード c_i では、$\mu^{\alpha}_n < r_n + \varepsilon_n$ ならば、最左の評価値の予測

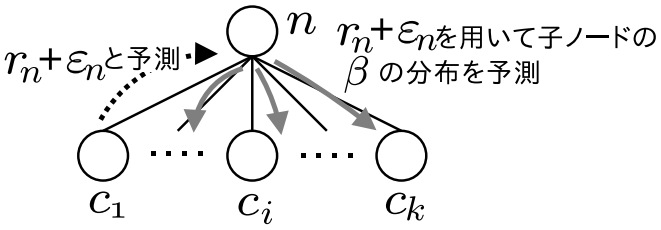

図 6.10: c_1 の予測を用いた β の分布の予測

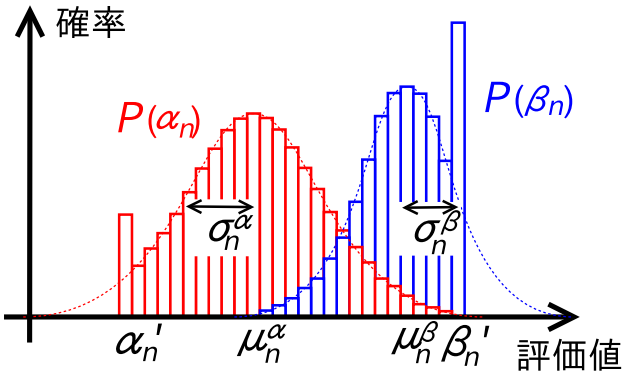

図 6.11: 探索窓の確率分布の例

分布の方を採用し、$\mu_{c_i}^\beta = -(r_n + \varepsilon_n)$ かつ $\sigma_{c_i}^\beta = \sigma_n^r$ とする。逆に $\mu_n^\alpha > r_n + \varepsilon_n$ ならば、もともとの β の分布の方を採用し、$\mu_{c_i}^\beta = -\mu_n^\alpha$ かつ $\sigma_{c_i}^\beta = \sigma_n^\alpha$ とする。c_1 の探索終了後は c_1 の予測はもう必要ないので、$\mu_{c_i}^\beta = -\mu_n^\alpha$ かつ $\sigma_{c_i}^\beta = \sigma_n^\alpha$ とする。以上から、ルートノードで $\alpha_n = \alpha_n' = \mu_n^\alpha = -\infty$、$\beta_n = \beta_n' = \mu_n^\beta = \infty$ とすれば、全てのノードの窓情報が再帰的に求まる。

今回は簡単に、正規分布に従う二つの確率変数の最大値の分布を平均が大きい方の分布としたが、それぞれの平均と標準偏差から最大値の分布の平均と標準偏差を計算する方法 [31] も試した。しかし、実験結果に有意な差は見られなかった。

6.3.2 枝刈りされない確率の計算

ノード n の探索窓 α と β を確率分布で予測できれば、ノード n の探索が必要となる確率 P_n を計算することができる。これは n で $\alpha < \beta$ となる確率なので、α と β の予測分布が独立だと仮定する

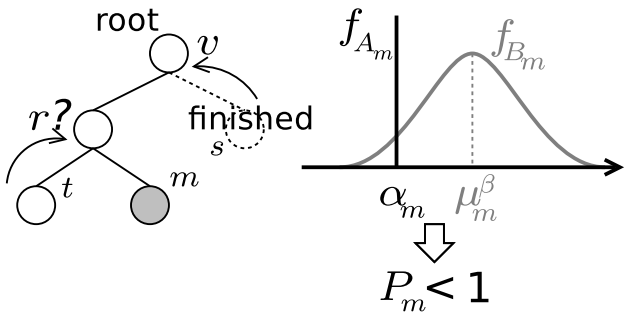

図 6.12: 枝刈りされない確率の計算に並列探索窓ではなく逐次探索窓を用いる理由

と次式となる。

$$
\begin{aligned}
P_n &= P(A_n < B_n) \\
&= \sum_{b=-\mathrm{INF}}^{\mathrm{INF}} \left(f_{B_n}(b) \sum_{a=-\mathrm{INF}}^{b-1} f_{A_n}(a) \right)
\end{aligned}
\tag{6.18}
$$

ただし、A_n と B_n は n の α と β を表す確率変数、INF は十分大きい整数とする。f_{A_n} と f_{B_n} は A_n と B_n の確率質量関数[1]である。評価値や探索窓の値は整数なので密度関数ではなく質量関数で考える。α や β は既に分かっている逐次探索窓 (α'_n, β'_n) の外側の値になることはないと考えられるため、$f_{A_n}(a)$ と $f_{B_n}(b)$ は以下のように正規分布が切れた分布で表現できる。図 6.11 に分布の例を示す。

$$
f_{A_n}(a) = \begin{cases} 0 & (a < \alpha'_n) \\ \sum_{a=-\mathrm{INF}}^{\alpha'_n} \frac{1}{\sqrt{2\pi}\sigma_n^\alpha} \exp\left(\frac{-(a-\mu_n^\alpha)^2}{2(\sigma_n^\alpha)^2} \right) & (a = \alpha'_n) \\ \frac{1}{\sqrt{2\pi}\sigma_n^\alpha} \exp\left(\frac{-(a-\mu_n^\alpha)^2}{2(\sigma_n^\alpha)^2} \right) & (a > \alpha'_n) \end{cases}
\tag{6.19}
$$

$$
f_{B_n}(b) = \begin{cases} 0 & (b > \beta'_n) \\ \sum_{b=\beta'}^{\mathrm{INF}} \frac{1}{\sqrt{2\pi}\sigma_n^\beta} \exp\left(\frac{-\left(b-\mu_n^\beta\right)^2}{2\left(\sigma_n^\beta\right)^2} \right) & (b = \beta'_n) \\ \frac{1}{\sqrt{2\pi}\sigma_n^\beta} \exp\left(\frac{-\left(b-\mu_n^\beta\right)^2}{2\left(\sigma_n^\beta\right)^2} \right) & (b < \beta'_n) \end{cases}
\tag{6.20}
$$

逐次探索で推定される最小探索木に含まれるノード m では、$\mu_m^\alpha = \alpha'_m = -\infty$ または $\mu_m^\beta = \beta'_m = \infty$ が成り立つので、$P_m = 1$ となる。

ここで、枝刈りされない確率の計算に並列探索窓 (α_n, β_n) ではなく、逐次探索窓 (α'_n, β'_n) を用いた理由を述べる。図 6.12 において、ノード m は予測最小探索木に含まれる。しかし、並列探索窓を用いて提案手法の枝刈りされない確率を計算すると、図中のノード s が終了し、かつノード t がまだ終了していない場合、$\mu_m^\beta = -r$、$\alpha_m = v$ から $P_m < 1$ となり、ノード m が枝刈りされることがあ

[1]例えば、$f_{A_n}(a) = P(A_n = a)$ である。これはノード n の α の値が a である確率である。

ると判断する。ここでは今回は最小探索木の予測を優先し、ノード m が枝刈りされると予測されることがないように、並列探索窓の代わりに逐次探索窓を用いた。これはヒューリスティックではあるが、実際に評価では、並列探索窓を用いるより逐次探索窓を用いた方がよい性能が得られた。

6.3.3 表引きによる高速化

式 (6.18) の計算量は評価値が取りうる値の数を N とすると、$O(N^2)$ となり、計算コストが大きい。そこで、実装では各パラメータに応じて予め式 (6.18)(6.19)(6.20) を用いて確率表を作り、探索時には表引きで確率を求めた。用いるパラメータの数を減らすため、$\mu_n^\alpha \to 0$、$\sigma_n^\alpha \to 100$ に正規化し、以下の 4 つのパラメータを用いて表引きを行った。

$$\eta_n^\beta = (100/\sigma_n^\alpha)(\mu_n^\beta - \mu^\alpha) \tag{6.21}$$

$$\rho_n^\beta = (100/\sigma_n^\alpha)\sigma_n^\beta \tag{6.22}$$

$$\delta_n^\alpha = (100/\sigma_n^\alpha)(\alpha'_n - \mu_n^\beta) \tag{6.23}$$

$$\delta_n^\beta = (100/\sigma_n^\alpha)(\beta'_n - \mu_n^\alpha) \tag{6.24}$$

式 (6.23) や式 (6.24) を用いたのは、α'_n は μ_n^β に近いとき、β'_n は μ_n^α に近いときが、パラメータが変化したときに確率が大きく変動するため重要だからである。なお、各パラメータの値は少しずつ集約して表が大きくなりすぎないようにした。具体的には η_n^β と δ_n^α と δ_n^β が -250 から 250 までの 5 刻みごと、δ_n^β が 50 から 200 までの 5 刻みごとに集約した。

6.3.4 異なる深さでの評価値の差の分布の調査

提案手法では各局面について、浅い探索の評価値から実際に行う探索の評価値を正規分布で予測する。具体的には節 6.3.1 で述べた浅い探索の評価値と予測分布の平均とのずれ ε_n と分布の標準偏差 σ_n^r を推定しなければならない。これを実現するために、プロの棋譜に現れる各局面を将棋プログラムである激指[2]を用いて深さ 8、10、12、14、16 で探索し、ある深さの評価値からそれより 2 浅い探索の評価値を引いた差の分布を調べた。つまり、16-14、14-12、12-10、10-8 の 4 通りである。後手番の評価値は符号を反転して、それぞれの評価値は自分の手番から見て有利なほど大きいものとした。用いた棋譜数は 3 万である。初期局面など同じ局面の重複は除いた。さらに詰みが見つかったときの評価値の絶対値は非常に大きく、平均や標準偏差の計算に大きな影響を与えるため、詰みが見つかった局面も除いた。

局面により、ε_n や σ_n^r は異なると考えられる。そこで、激指の進行度と、浅めの探索の評価値の絶対値の 2 つを局面の特徴とした。進行度と評価値は明らかに独立ではないが、簡単に用いることができる量として今回はこれらの 2 つを用いた。激指の進行度は 0 から 127 の整数であり、値が大きいほど終盤であることを示す。これを進行度 16 ごとに 8 つの区分に分けて分布を調べた。同様に、

[2] http://www.logos.t.u-tokyo.ac.jp/~gekisashi

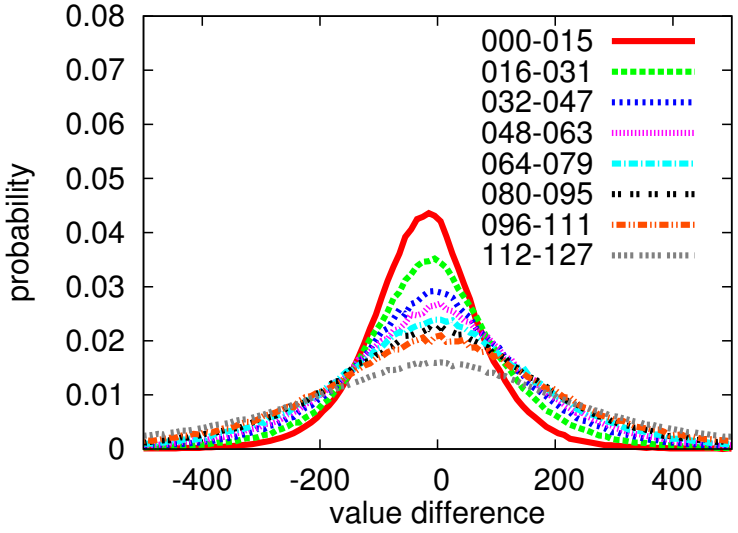

図 6.13: (深さ 10 の評価値)-(深さ 8 の評価値) の進行度ごとの分布

評価値の絶対値も 7 つの区分に分けて分布を調べた。評価値の絶対値は進行度が大きいほど大きくなりやすいため、進行度によって、この 7 つの区分の分け方は変更した。合計で $4 \times 8 \times 7 = 224$ 組の平均と標準偏差が得られた。

得られた分布に関して、以下の傾向があった。

- 評価値の差は 0 が例外的に出現頻度が大きい。周りの点の 3 倍から 10 倍程度の多さになっていた。
- 進行度が小さいほど標準偏差が小さい。
- 評価値の絶対値が小さいほど標準偏差が小さい。
- 深さが大きいほど標準偏差が小さい。ただし例外的に、進行度も評価値の絶対値も大きいときは、深さが大きいほど標準偏差が大きい。
- 進行度が小さく、深さが小さいほど、浅い探索が楽観的評価 ($\varepsilon_n = -10$ 程度) をする傾向にあるが、多くの場合は ε_n はほぼ 0 である。

深さ 10 と深さ 8 の評価値の差の分布を図 6.13 に、深さ 16 と深さ 14 の評価値の差の分布を図 6.14 に示す。横軸は評価値の差、縦軸は各出現回数を総数で割って求めた確率である。横軸は 10 ごとに和をとってまとめてプロットしている (例えば横軸が 5 の点は評価値差が 1 から 10 までとなる確率の和)。また差が 0 になる点は図の見やすさのために除外している。これらの図から、上で述べた分布の傾向の一部を確認できる。

また、求めた ε_n と σ_n^r の正規分布で実際の分布がどのくらい予測できているのかを確認する。各パラメータがおよそ中間の値をとる例として、進行度が 48 から 63 の間で、かつ深さ 10 の評価値の絶対値が 200 から 299 の間の局面を、深さ 12 と深さ 10 で探索したときの評価値の差の分布と、そ

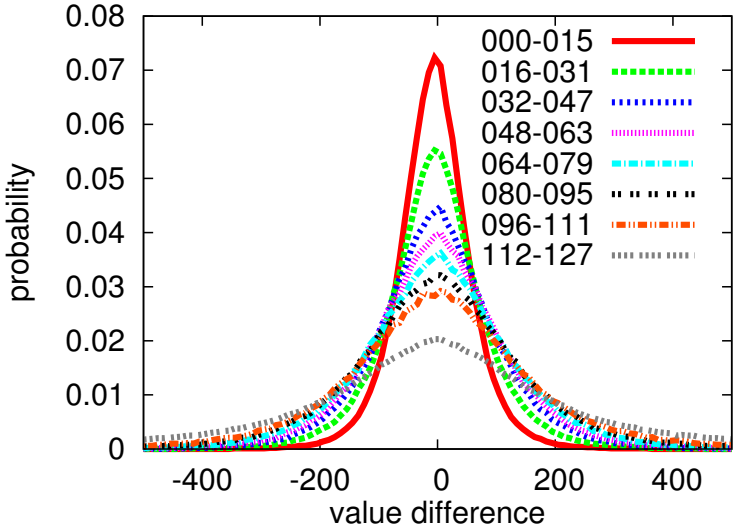

図 6.14: (深さ 16 の評価値)-(深さ 14 の評価値) の進行度ごとの分布

のときの予測正規分布を同時に図 6.15 に示す。この場合は差が 0 付近の部分を除いて正規分布で近似できていることが分かる。

6.3.5 評価

提案手法を評価するために、次の 3 つの探索順序を比較した。なお、全ての手法において、P_n が同じノードはゲーム木の左側にあるノードから順に探索 [96] した。

- **提案手法**
  P_n が大きいものから順に実行
- **分布なし**
  提案手法において、
  $$\max\{\mu_n^\alpha, \alpha_n'\} < \min\{\mu_n^\beta, \beta_n'\} \tag{6.25}$$
  となるノード n は $P_n = 1$、それ以外のノードは $P_n = 0$ として実行
- **深さ優先**
  提案手法の計算で $P_n = 1$ となるノードは優先して実行[3]し、それ以外のノードはゲーム木の左側にあるノード、つまり深さ優先探索で先に探索されるノードから順に実行

深さ優先の手法でも提案手法による確率の計算は行っている。これは実装を統一して実験を簡単に行うためである。したがって、不必要な計算を行っていることになるが、探索順序における性能差を

[3]実際の実装では確率が 1 に十分近ければ 1 とみなす

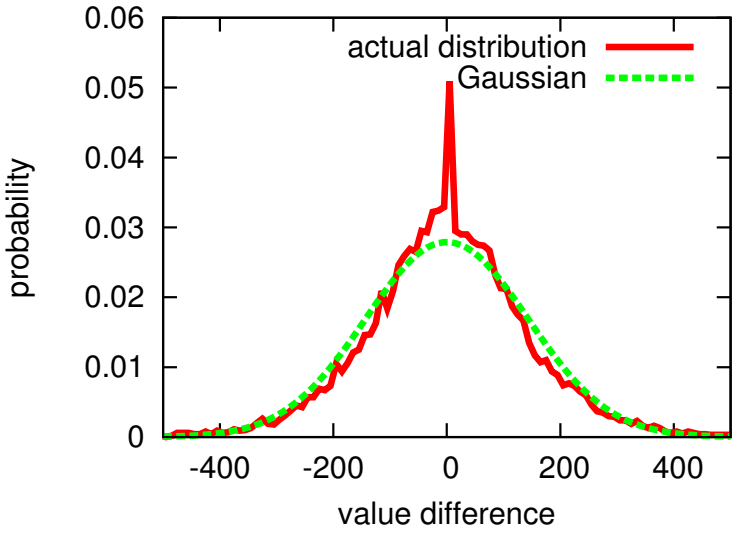

図 6.15: 進行度が 48 から 63、深さ 10 の評価値の絶対値が 200 から 299 の局面について、深さ 12 と深さ 10 の評価値の差の分布

比較するためにはこれでよいと考えた。また、比較として分布を用いない手法（分布なし）も評価する。これは、確率分布を用いずに、単純に予測評価値を用いて探索を進めた場合に、枝刈りされると予測されるノードは単に後回しにすればよいという手法であり、APHID において実行するべきタスクの選択を真似たものである。ただし、浅い探索の評価値を用いて深い探索の評価値を予測するため、今回は最左の子ノードの評価値の予測しか用いていない。

6.3.5.1 将棋のゲーム木の実探索による評価

並列ゲーム木探索のプログラムに提案手法を組み込んで、探索時間を測定した。並列探索の実装は 6.2 節と同じ [118] の実装を用いた。マスタ・ワーカ方式で、マスタが $\alpha\beta$ 探索を行い、残り深さがある閾値 g 以下になるとタスクとして部分木をワーカに送信し、ワーカが探索して結果を返す。なお、確率が $P_n = 1$ であるノードは、$Pn < 1$ のノードの探索を中断してでも探索される [118]。ただし、実装の簡略化のため、$P_n = 1$ だったノードが $P_n < 1$ に変化したとしても、タスクとしてワーカに既に送ってしまった場合は、ワーカに $P_n < 1$ になったことは伝えないものとした。

探索中における確率の更新では、確率が変化しないと分かっている部分木のノードの更新を行う必要がない。しかし、そのような最適化はまだ行っておらず、部分木をすべて辿っているため、実装にはまだ無駄が残っている。探索窓の分布予測には節 6.3.4 の結果の ε_n と σ_n^r 用いた。ただし、深さ 14 以上の部分木については深さ 14 だとして確率を計算している。

実験に用いた局面は、プロ棋士の 100 棋譜から 1 棋譜につき 1 局面ずつ重複しないようにランダムに計 100 局面選び、激指の進行度を用いて序盤と終盤の局面は除いた。各局面 5 回ずつ探索し、そ

表 6.7: 将棋を用いた深さ 18 の実探索の実行時間 [秒]

探索順序	平均実行時間	標準偏差
提案手法	7.69	0.54
分布なし	7.46	0.54
深さ優先	7.64	0.51

表 6.8: 将棋を用いた深さ 22 の実探索の実行時間 [秒]

探索順序	平均実行時間	標準偏差
提案手法	143	16.0
分布なし	142	14.9
深さ優先	137	15.1

の 5 局面の試行で相加平均と標準偏差を求め、それらの 100 局面間での相乗平均を計算した。実験環境は表 4.1 の Xeon E5530 のノードである。各ノードに 8 コアあるため 8 ワーカずつ実行し、全部で 512 ワーカで実験した。

探索時間が短くなるようなパラメータ設定として、探索深さが 18 で閾値 g が 12 のときと、探索深さが 22 で g が 13 のときで実験を行った。ただし、探索深さが 22 のときは実験時間の関係で 30 局面だけを用いた。結果を表 6.7 と表 6.8 にそれぞれ示す。探索順序によって明確な差がついていないことが分かる。

6.3.5.2 将棋のゲーム木を用いたシミュレーション

提案手法をより理想的な場合について評価するため、6.2 節と同じシミュレーション方法を用いた。つまり、リーフの評価にかかる時間、つまり激指の探索関数の計算時間が 1 ステップで全て等しく、それ以外の処理にかかる時間を全て無視したものである。p ワーカでの並列探索のシミュレーションでは、p 個のリーフを選択するまで木を展開し、それらのリーフを評価するという操作を 1 ステップとする。このステップを繰り返し、探索終了までのステップ数を実行時間とみなす。

実験に用いた局面は節 6.3.5.1 で用いた 100 局面を用いた。この 100 局面での実行ステップ数の相乗平均と速度向上比を求めた。本研究では大規模環境を想定しているため、プロセス数が多い場合の結果を示していく。

シミュレーションの結果を表 6.9 に示す。深さ優先を比較すると提案手法の方が実行ステップ数が少なく、台数が多いと 10%から 15%程度の差がついたが、提案手法と分布なしを比較すると性能差は小さかった。

以下、将棋のゲーム木を用いて、ゲーム木の特徴を変更することにより、その特徴が与える影響を調べていく。まず、実現確率を使わない場合のシミュレーションを行った。実現確率は、良くなさそうな指し手は浅くしか探索されない。よって、実現確率を用いると子ノードの部分木の大きさの差が大きくなる。実現確率を用いないことにより、より簡単なゲーム木について調べる。実現確率を用いないときの結果を表 6.10 に示す。手法間の差は、10%程度であり、全体的な傾向は実現確率を用いたときと似ている。

表 6.9: 将棋を用いたシミュレーションの実行ステップ数 (実現確率あり、探索深さ 16、ワーカの探索深さ 8)

	ワーカ数	1	256	1024	4096	16384
提案手法	実行ステップ数	114536	619	191	75.3	42.7
	速度向上比	1	185	600	1521	2682
分布なし	実行ステップ数	114536	649	196	80.0	47.3
	速度向上比	1	176	584	1432	2421
深さ優先	実行ステップ数	114536	642	196	84.6	51.2
	速度向上比	1	178	584	1354	2237

表 6.10: 将棋を用いたシミュレーションの実行ステップ数 (実現確率なし、探索深さ 6、ワーカの探索深さ 2)

	ワーカ数	1	256	1024	4096	16384
提案手法	平均ステップ数	16571	121	42.1	18.7	11.0
	速度向上率	1	137	393	886	1506
分布なし	平均ステップ数	16571	121	42.9	19.7	12.1
	速度向上率	1	137	386	841	1369
深さ優先	平均ステップ数	16571	125	46.5	20.3	12.8
	速度向上率	1	133	356	816	1294

次に、ゲーム木の分枝数が並列性能に影響を与える影響を調べるため、実現確率を使わない場合に、さらに各ノードの分枝数を 5 に制限することを行った。制限の方法としては、激指の指し手の並び替えの後、もっとも有望な 5 つの子ノードだけを探索するようにした。結果を表 6.11 に示す。全体としての速度向上は小さくなっているが、手法間の差は大きくなっていることが分かる。また、1024 ワーカのときに、手法間の性能差がもっとも大きくなっており、提案手法は深さ優先の 53%の実行ステップ数になっている。

浅い探索の修了を待つことは、並列に実行可能なタスクの数を減らすことになるため、さらに、浅い探索の影響を無視した場合のシミュレーションを行った。このシミュレーションでは実際には浅い探索を行うが、すべて逐次的に中断されることなく実行され、その浅い探索中に訪問されるノード数は数えないことにした。結果を表 6.12 に示す。浅い探索を待つ場合に比べ、手法間の性能差が大きくなっており、1024 ワーカでも深さ優先は他の二つに比べて倍以上の実行ステップ数となっている。しかし、1024 ワーカまでは、提案手法より、分布なしの方が性能が少なくなっていることも分かる。

表 6.11: 分枝数を 5 に制限した場合の将棋を用いたシミュレーションの実行ステップ数 (実現確率なし、探索深さ 14、ワーカの探索深さ 2)

	ワーカ数	1	256	1024	4096	16384
提案手法	平均ステップ数	62522	581	302	219	180
	速度向上率	1	108	207	285	347
分布なし	平均ステップ数	62522	723	453	309	211
	速度向上率	1	86	138	202	296
深さ優先	平均ステップ数	62522	924	570	351	218
	速度向上率	1	68	110	178	287

表 6.12: 分枝数を 5 に制限した場合の将棋を用いたシミュレーションの実行ステップ数 (実現確率なし、浅い探索なし、浅い探索探索深さ 14、ワーカの探索深さ 2)

	ワーカ数	1	256	1024	4096	16384
提案手法	平均ステップ数	25072	141	48.4	19.3	10.9
	速度向上率	1	177	518	1299	2300
分布なし	平均ステップ数	25072	132	46.1	28.2	20.7
	速度向上率	1	190	544	889	1211
深さ優先	平均ステップ数	25072	169	111	73.9	47.8
	速度向上率	1	148	226	339	525

6.3.5.3 人工木を用いたシミュレーション

将棋のゲーム木での結果を考察するため、提案手法を人工木を用いてさらに理想的な状況で評価した。人工木は全てのノードでミニマックス値が浅めの探索なしで正規分布で予測でき、その標準偏差は全てのノードで等しいと近似的にみなせる木を作成して用いた。各ノードでの子ノードの並び替えは、予測正規分布の平均が大きいものをより有望な子ノードとして扱うことで行った。

用いた人工木の作り方を述べる。木のエッジに乱数を与えて、ルートからそのノードまでのエッジの値の和がそのノードの評価値となるような人工木を作る。この人工木は先行研究でも評価に用いられている [62, 93]。この人工木の各ノードの評価値の分布を予測するというのが基本的な考え方である。まず、親ノードと子ノード b 個からなる小さな部分木を考える。親ノードの暫定評価値を 0 とする。i 番目の子ノードの評価値は、独立同分布な確率分布 P_S から得られる乱数を、親ノードの暫定評価値に加えたものであるという木を考える。この乱数の確率関数 $f_S(s)$ は既知とし、i 番目の子ノードに加えられた値を表す確率変数を S_i とする。子ノード間の評価値は実際には独立ではな

いと考えられる。しかし、このモデルは評価値の確率分布が与えられている理想的な場合における本提案手法の評価には有効だと考えた。今、V_i は i 番目の子ノードの評価値を表す確率変数とする。$S_i = s_i$ で条件付けられた各子ノードの確率関数 $f_{V_i|S_i}(v_i|s_i)$ は $f_{V_i|S_i}(v_i|0)$ を s_i だけシフトしたものだとする。

$$f_{V_i|S_i}(v_i|s_i) = f_{V_i|S_i}(v_i - s_i|0) \tag{6.26}$$

さらに $f_{V_i|S_i}(v_i|0)$ は全ての子ノードで共通で $f_{V|S}(v|0)$ とし、これは既知だとする。

このとき、各子ノードの s_i が未知のときの親ノードの評価値、すなわち、子ノードの評価値の最大値 X の確率関数 $f_X(x)$ が知りたい。これは以下のように、$f_X(x, s_1, \ldots, s_b)$ を全ての s_i で周辺化することによって得られる。

$$\begin{aligned} f_X(x) &= \sum_{s_1=-\mathrm{INF}}^{\mathrm{INF}} \cdots \sum_{s_b=-\mathrm{INF}}^{\mathrm{INF}} f_{X,S_1,\ldots,S_b}(x, s_1, \ldots, s_b) \\ &= \sum_{s_1=-\mathrm{INF}}^{\mathrm{INF}} \cdots \sum_{s_b=-\mathrm{INF}}^{\mathrm{INF}} \left(f_{X|S_1,\ldots,S_b}(x|s_1, \ldots, s_b) \prod_{i=1}^{b} f_{S_i}(s_i) \right) \end{aligned} \tag{6.27}$$

二つ目の式変形では s_i が他の子ノードとは独立に決まることを利用している。$f_{X|S_1,\ldots,S_b}(x|s_1, \ldots, s_b)$ は、子ノードの s_i が既知のときに、評価値の最大値が x になる確率である。これは、全ての子ノードの評価値が x 以下である確率から $x-1$ 以下である確率を引いたものなので、

$$f_{X|S_1,\ldots,S_b}(x|s_1, \ldots, s_b) = \prod_{i=1}^{b} \sum_{y=-\mathrm{INF}}^{x} f_{V_i|S_i}(y|s_i) - \prod_{i=1}^{b} \sum_{y=-\mathrm{INF}}^{x-1} f_{V_i|S_i}(y|s_i) \tag{6.28}$$

で表せる。このとき、

$$f_{V_i}(v_i) = \sum_{s_i=-\mathrm{INF}}^{\mathrm{INF}} f_{V_i|S_i}(y - s_i|0) f_{S_i}(s_i) \tag{6.29}$$

であるので、式 (6.27)(6.28)(6.29) から以下となる。

$$f_X(x) = \prod_{i=1}^{b} \sum_{y=-\mathrm{INF}}^{x} f_{V_i}(y) - \prod_{i=1}^{b} \sum_{y=-\mathrm{INF}}^{x-1} f_{V_i}(y) \tag{6.30}$$

つまり、式 (6.29) で $f_S(s)$ と $f_{V|S}(v|0)$ から $f_V(v)$ を求めた後で、式 (6.30) から $f_X(x)$ を求めればよい。

次にこの計算を繰り返す。$f_S(s)$ は平均 0、標準偏差 100 の正規分布とする。$f_{V|S}(v|0)$ の初期値 $f_{V|S}^{(0)}(v|0)$ は $v = 0$ で 1、それ以外で 0 となる分布とする。このとき、$f_{V|S}^{(k)}(v|0)$ を既知として、親ノードの $f_X(x)$ を求め、$f_{V|S}^{(k+1)}(v|0) = -f_X(v)$ とする。以降はこの計算を繰り返す。$f_{V|S}^{(k)}(v|0)$ はあるノードから深さ k だけ探索したときの評価値の分布を表している。このとき各 k について、$f_{V|S}^{(k)}(v|0)$ の平均と標準偏差を調べていくと、k が増えるにつれてある値に収束しているように見えた (平均については k の偶奇で異なる) ため、$f_{V|S}^{(50)}(v|0)$ の平均と標準偏差を ε_n と σ_n^r として用いることとした。$b = 5$ のときの ε_n は k が偶数と奇数のときにそれぞれ 26.2 と 125.5、σ_n^r は k の偶奇に依らず 85.6、

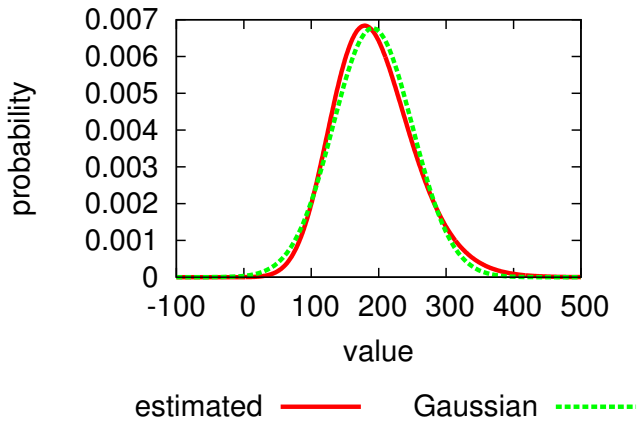

図 6.16: 分枝数 20、残り深さ 1 のときの人工木の評価値の分布

$b = 20$ のときの ε_n が 22.6 と 191.1、σ_n^r が 58.9 であった。実際に人工木を構築するときは、エッジに乱数を与える方法で人工木を作り、各ノードの評価値を予測評価値とする。さらに、リーフノードの評価値に $f_{V|S}^{(50)}(v|0)$ から得られる乱数を与え、これを実際の評価値とする。$b = 20$ のときの残り深さが 1 のノードのミニマックス値の分布と予測正規分布と合わせて図 6.16 に示す。実際の分布は正規分布ではないが、正規分布で近似できていることが分かる。

提案手法では、(1) 最左の子ノードの評価値の予測しか用いていない点 (2) 正規分布に従う二つの確率変数の最大値の分布は正規分布にならないが、それを平均が大きい方の正規分布で近似している点 (3) 粒度の粗い表引き、の 3 つの高速化により確率推定の精度が落ちている可能性がある。そこで、これらの高速化を行わずに確率を精度良く求める手法として、α と β の予測分布を平均と標準偏差で管理するのではなく、各値ごとの確率 (ヒストグラム) で管理し、さらに最左の子ノードだけではなく全ての子ノードの評価値の予測分布を用いる手法 (高速化なし) を準備した。

まず、多重反復深化の計算コストも考慮した、浅めの探索も行うシミュレーションを行った。この浅めの探索は各ノードのミニマックス値や子ノードの並び替えは全て同じで探索する深さだけが違う擬似的なものであり、浅めの探索の結果は捨てるものとした。さらに、トランスポジションテーブルを用いて、既に探索したノードに関しては重複して探索しないようにしている。トランスポジションテーブルに用いるノードのキーとしては、並列探索を深さ優先探索で行うためにも利用している signature [58, 96] を用いた。

分枝数と探索深さがそれぞれ 5 と 10 の結果を表 6.13 に、20 と 6 の結果を表 6.14 に示す。100 個の人工木での実行ステップ数の相乗平均と速度向上比を求めた。表 6.13 で提案手法を深さ優先と比べると、1024 台で性能差が最大となりステップ数が 56%になっている。提案手法と分布なしを比較しても最大でステップ数が 82%に抑えられており、提案手法の有効性が見られる。次に表 6.14 を見ると、表 6.13 と比べて各手法間で性能差が小さくなっている。これは分枝数が多いと確率 1 のタスクが多くなるためだと考えられる。最後に、提案手法でヒストグラムを保持して、より正確に確率を

表 6.13: 人工木でのシミュレーションの実行ステップ数 (分枝数 5、探索深さ 10)

	ワーカ数	1	256	1024	4096	16384
高速化なし	平均ステップ数	22855	142	61.8	39.4	30.6
	速度向上比	1	161	370	580	747
提案手法	平均ステップ数	22855	153	65.0	41.4	32.4
	速度向上比	1	149	352	552	705
分布なし	平均ステップ数	22855	149	74.0	50.3	36.1
	速度向上比	1	153	309	454	633
深さ優先	平均ステップ数	22855	226	116	63.9	39.3
	速度向上比	1	101	197	358	582

表 6.14: 人工木でのシミュレーションの実行ステップ数 (分枝数 20、探索深さ 6)

	ワーカ数	1	256	1024	4096	16384
高速化なし	平均ステップ数	34318	150	47,6	20.2	14.2
	速度向上比	1	229	721	1699	2417
提案手法	平均ステップ数	34318	170	52.6	21.9	15.0
	速度向上比	1	201	652	1567	2288
分布なし	平均ステップ数	34318	162	51.9	22.7	16.3
	速度向上比	1	211	661	1511	2105
深さ優先	平均ステップ数	34318	169	57.8	30.5	22.2
	速度向上比	1	203	594	1125	1546

計算しても、実行ステップ数の減少は 5%から 10%程度であるが、実際の計算時間を調べたところほとんどの場合で 10 倍以上であり、100 倍以上であることもあった。提案手法の高速化は計算時間も考慮すると、十分精度の良い確率の計算ができていると言える。これに対し、提案手法と分布なしの実際の計算時間は同程度であった。

次に、多重反復深化による浅い探索を行わないときのシミュレーションも行った。分枝数が 5 の場合の結果をそれぞれ表 6.15 に示す。将棋のゲーム木の場合と同様に、浅い探索ありの場合に比べて、深さ優先に対する提案手法と分布なしのステップ数の差は大きい。しかし、提案手法と分布なしの差は小さく、ワーカ数が少ない場合は、提案手法の方が分布なしよりステップ数が多くなっているところも将棋のゲーム木を用いた場合の結果と似ている。

表 6.15: 人工木でのシミュレーションの実行ステップ数 (浅い探索なし、分枝数 5、探索深さ 10)

	ワーカ数	1	256	1024	4096	16384
高速化なし	平均ステップ数	12119	72.1	26.2	10.7	4.99
	速度向上比	1	168	463	1133	2429
提案手法	平均ステップ数	12119	78.9	28.0	11.3	5.71
	速度向上比	1	154	433	1072	2122
分布なし	平均ステップ数	12119	75.4	26.2	12.3	9.06
	速度向上比	1	161	463	985	1338
深さ優先	平均ステップ数	12119	118	72.3	42.8	25.3
	速度向上比	1	103	168	283	479

6.3.6 考察

まず、提案手法が実探索で有効でなかった理由を考える。この理由の一つして、実探索の実験でのワーカ数が少なかったことが挙げられる。シミュレーションでも 1024 台で 3%しか差がなかったので、それより少ない 512 台の実探索でその差が見えなくても不思議ではない。次に、表 6.9、表 6.13、表 6.14 を見比べると、将棋でのシミュレーションの結果は分枝数が 5 のときよりも 20 のときの結果に近いと言え、さらに人工木を用いた場合の各探索順序の性能差も分枝数が大きくなると小さくなったことから、分枝数が大きいという将棋の特徴が影響している可能性が考えられる。

次に、シミュレーションの結果から、実現確率、分枝数、浅い探索が各手法に与える影響について議論する。まず、実現確率は、表 6.9 と表 6.10 を比較することにより、各手法の性能差の傾向は近いと言える。つまり、シミュレーションによる評価では、木の深さのばらつきは、各手法の性能差にはあまり影響を与えないことが分かった。次に、分枝数が与える影響を見る。表 6.10 と表 6.11 の比較、あるいは、表 6.13 と表 6.14 の比較から、分枝数が少ないほど、手法間の性能差が大きくなることが分かる。これは、分枝数が少ないと、実行が必要だと予測される $P_n = 1$ のタスクだけでは、タスクが不足しやすいため、$P_n < 1$ となる投機的なタスクの選択が重要であるためであると考えられる。最後に、浅い探索の影響を見る。表 6.11 と表 6.12 の比較、表 6.13 と表 6.15 の比較から、浅い探索により、ワーカ数が大きいときに、手法間で性能の差が大きくなることが分かる。浅めの探索の終了を待つと、各時点で実行可能なタスクの数が限られ、各ステップでそれらが全て実行できることが多くなるため、ワーカ数が増えると差がつきにくくなる。これに対し、浅い探索を待たないと、探索終了直前を除き、タスクが足りなくなることがないため、タスクの選択がより重要になるためだと考えられる。また、浅い探索を行わない場合は、提案手法と分布なしの性能差が小さいことも分かる。ワーカ数が少ない場合では、分布なしの方がステップ数が少ない傾向もあった。探索窓を早く狭めるためには、深さ優先探索順に実行するのが良く、分布なしの手法は提案手法より、深さ優先探索順に近いことが理由だと考えられる。逆に、ワーカ数が多い場合は、探索窓を狭めようとするよ

り、当初の予測を用いて無駄となる探索を少なくすることの方が重要になると言える。

第7章　探索が必要となる部分木の予測の動的な修正

7.1　序論

$\alpha\beta$ 探索では、訪問ノード数を減らすために、最小探索木に近い木を探索することが重要である。そのためには、各ノードにおいて、有望な子ノードから探索する必要がある。有望な子ノードを調べるための方法としては、反復深化が有力な手法であり、広く用いられている。反復深化では、浅い探索の結果に基づいて、探索深さを少しずつ深くしていく。このとき、並列探索において、並列に探索している部分木の探索が独立に行われ、最終的な結果しか用いられないとすれば、その浅い探索で得られた結果を、探索している木全体の最小探索木の予測に用いていないことになる。さらに、並列探索では、複数の部分木の探索が同時に行われるが、探索を開始するときには、逐次探索ならば得られていたような部分木の結果は用いることができない。したがって、並列探索では、他の部分木の浅い探索結果も有効に活用し、最小探索木の予測の修正に用いることが重要となる。

6 章で述べた手法では、子ノードの探索結果を用いて枝刈りを行ったり、子ノードを並列に探索するかどうかを決定したりはしているので、最小探索木の中での局所的な修正は行っていたことになる。しかし、指し手の並び替えは最初に与えられて固定であり、最小探索木全体の構造の修正は行っていなかった。そこで本章では、探索途中の結果も用いて、各ノードにおける指し手の並び替えの修正も含めて、最小探索木の予測の修正を行う手法を実装し評価した結果を報告する。

浅い探索の結果も用いて最小探索木を予測するという考えは、すでに APHID で実現されている [24]。しかし、APHID の論文ではプログラム全体の性能しか評価されておらず、最小探索木の予測を浅い探索の結果を用いて動的に修正することがどの程度性能に与えるかといったことについては直接評価されていない。また、APHID 自体の評価も 64 プロセスを用いたものにとどまっており、大規模環境における性能や問題点も明らかにされていない。本章では、APHID の実装を参考にした並列探索プログラムを実装し、最小探索木の動的な予測の修正が性能に与える影響を、最大で 1,536 コアを用いて評価した。

本章では 2 種類のゲーム木を用いて実験を行った。一つは人工木であり、もう一つは激指を用いて生成した将棋のゲーム木である。実験の結果、最小探索木の動的な予測の修正はどちらのゲーム木においても有効であったが、激指を用いて生成したゲーム木の方がその効果は大きいことが分かった。しかし、それでも速度向上は線形とは言えなかったため、その原因を詳細に解析することも行った。

また、将棋プログラムとして対局することができるようにするために、並列プログラムの実装にさらなる変更を加え、勝率による評価も行った。

本章の構成は以下のとおりである。まず、7.2 節において、本章で行った実装について述べる。次に 7.3 節において、実装した並列プログラムを用いて行った評価実験の結果を載せる。7.4 節では、実装した並列プログラムの性能を抑制している原因について解析した結果を述べる。7.5 節では、将棋プログラムとして対局するために追加した実装について述べ、実際に激指と対局させたときの勝率を示す。最後に 7.6 節で本章のまとめと今後の課題を述べる。

7.2 実装

探索途中の結果を用いて最小探索木の予測が動的に修正されるべきだという考えに基づいて、6 章で用いたプログラムとは別のプログラムを実装した。大まかな実装の方向性を決めるにあたっては、APHID の実装を参考にした。たとえば、マスタ・ワーカ方式や、マスタが探索木のルートに近いの探索（pass と呼ぶ）を何度も実行するといった点は、本章の実装でも用いている。しかし、実装の詳細については変更を加えた。これらの違いは次の二つの点を意識したものである。一つ目は、マスタは自身の探索木の形が変わったときに、それをなるべく早くワーカに伝える点であり、二つ目はワーカはマスタからの情報を素早く取得し、探索に反映させるという点である。

APHID のマスタは uncertain values と decided values[1]を区別する。マスタは pass 中にマスタの探索木上の未訪問のリーフノードに到達すると、その静的評価値を uncertain value として用いる。それ以降の pass ではワーカからの結果がもし使えるならばそれを使う。ただし、その場合でも、d と d' を 5.2.3.4 節で述べたように、全体の探索深さとマスタが探索する深さとして、終了した探索の深さが $d - d'$ より小さければ、その値は依然として uncertain である。逆に、その結果が深さ $d - d'$ の探索のものであるならば、その値は decided である。一回の pass 中で、uncertain の値を一回も使うことがなくなるまで、マスタは探索を続ける。

次に APHID とは独立に行った実装について述べる。本章の実装ではマスタはその探索木をメモリに保存する。APHID は元のゲーム木探索プログラムに依存しないフレームワークとして設計されており、探索木をメモリに保存するためには、元のプログラムの情報が必要になるため、木をメモリに保存することは行っていない。探索木をメモリに保存することで、合法手の生成はその局面を最初に訪問したときだけになる。これにより、もし合法手の生成が時間のかかるものだった場合、一回のマスタの探索木の pass にかかる時間は短くなる。結果としてワーカに情報を頻繁に伝えることが可能になる。一回の pass が短いことは特に大規模計算環境では重要な性質である。大規模計算環境を活用するためには、ワーカの数に見合った数のタスクを生成するために、マスタの探索木は十分に大きくなければならないからである。APHID ではマスタの一回の pass にかかる時間は十分短いことを仮定して設計されているため、この変更は大規模計算環境を活用するためには重要であると考えられる。

本章の実装では、マスタは各 pass でタスクを送信する。この送信は前回の pass で同じタスクを送信していたとしても行われる。このようにすることで、ワーカはマスタが現在行っている pass に含まれているタスクを速やかに知ることが可能となり、それらのタスクを他のタスクより優先的に実行することができる。さらに、マスタは、一回の pass が終わったときに、その終了をワーカに伝える

[1]decided values は APHID の元論文では出てこないが、便宜上名前を付けた。

ためのメッセージ（pass message と呼ぶ）を送信する。この pass message を受信することにより、ワーカはこの pass 中に受信しなかったタスクを無視することができる。例えば、図 5.17 の右図においては、ワーカは pass message の受信によって、タスク A が不要であることが分かる。このとき、マスタはワーカに A を中止するための通知を明示的に送る必要はない。

ワーカはマスタ上の最小探索木の変更を即座に検知する必要がある。このために、ワーカはたとえタスクである部分木を探索途中であっても、マスタからのメッセージを頻繁に調べるようにした。もし、受け取ったメッセージによって、その時点でワーカが実行中のタスクより他のタスクの方が重要であることが分かったならば、ワーカは現在のタスクを中断して、最も重要な他のタスクの実行を開始する。この仕組みを実現するために、[105] で用いた方法を用いた。ワーカがマスタからタスクを受け取ったとき、ワーカは探索をいったん中断し、受け取ったタスクを優先度付きキューに挿入したのち、中断した探索を再開する。もし、受け取ったタスクの優先度が中断されたタスクの優先度より高かった場合には、ワーカは中断したタスクを再開する代わりに、受け取ったタスクの実行を開始する。中断されたタスクはタスクキューに残ったままであり、優先度が最大になったならば再び最初から探索される。このとき、いったん中断したタスクを最初から開始しても、トランスポジションテーブルがあるために、再探索のコストは最小限に抑えることができる。

7.2.1　マスタの実装の詳細

図 7.1 はマスタプログラムの擬似コードである。INF は十分大きい整数とする。本節ではまず、7.2.1.1 節で、関数 MasterSearch() による、マスタの探索の全体の流れを説明する。次に、関数 MasterAlphaBeta() の説明を行う。MasterAlphaBeta() は図 2.8 中の AlphaBeta() を修正したものであり、3 つの点が大きく異なるため、それぞれについて節を分けて述べる。7.2.1.2 節では、マスタの探索における探索窓の使い方について述べる。7.2.1.3 節はマスタがその探索木のリーフに到達したときの操作と、ワーカに送られるタスクについて述べる。7.2.1.4 節では、マスタの探索途中における子ノードの並び替えの方法について述べる。最後に、本研究の実装では、マスタが負荷分散を行うため、7.2.1.5 節において、その負荷分散の方法について述べる。

7.2.1.1　マスタの探索の流れ

マスタは関数 MasterSearch() の中で pass を実行し、自身の探索木を探索する。このとき、もし uncertain な値が見つからなければ、マスタは探索を終了し、評価値を返す（5 行目から 6 行目）。uncertain な値が一つでもあったら、マスタは、一回の pass を終了したことを知らせるために、pass message をすべてのワーカに送る（7 行目）。マスタは次の pass を開始する前に、ワーカからの結果を受信し、マスタの探索木のリーフの情報を更新する（8 行目）。今回の pass 中でマスタの探索木に変更がないことが明らかな場合、マスタは次に新しい情報をワーカから受信するまでは次の pass を開始する必要はない。実際の実装では、最後にワーカからの結果を受信した後、2 回 pass を実行してもワーカから新しい結果が来なかった場合は、次の pass を開始する必要はないものとした。2 回

```
1  int MasterSearch(position, depth){
2    d = depth;
3    while(true){
4      score = MasterAlphaBeta(position, depth, −INF, INF, −INF, INF);
5      if(no uncertain values have been found)
6        return score;
7      SendPassMessages();
8      RecvMessagesFromWorkers();
9    }
10 }
11 int MasterAlphaBeta(position, depth, a1, b1, a2, b2){
12   if(position is a terminal position)
13     return Evaluate(position);
14   if(depth == d−d') return EvalAndSendTask(position, depth, a1, b1, a2, b2);
15   if(position is visited first time) MasterAlphaBeta(position depth−2, a1, b1, a2, b2);
16   val, clist = SortChildren(position, a1, b1);
17   a1 = max(a1, val);
18   a2 = max(a2, val);
19   if(b1 <= a1) return b1;
20   foreach(child of clist){
21     score = max(a1, −MasterAlphaBeta(child, depth−1, −b1, −a1, −b2, −a2));
22     a1 = max(a1, score);
23     if(no uncertain values were found in this child)
24       a2 = max(a2, score);
25     if(b1 <= a1) return b1;
26   }
27   return a1;
28 }
```

図 7.1: マスタプログラムの擬似コード

としたのは、1 回目の pass でマスタの探索木の内部ノードの情報を更新し、2 回目の pass で、最小探索木だけを探索するようになるからである。

7.2.1.2　マスタで用いる探索窓

図 2.8 の AlphaBeta() と図 7.1 の MasterAlphaBeta() の違いの一つ目は、MasterAlphaBeta() では 2 種類の探索窓を用いることである。一つ (a1 と b1) は pass を実行するために用いられる pass window であり、もう一つ (a2 と b2) はワーカに送るタスクの探索窓として用いられる task window である。pass window は uncertain な値と decided な値の両方で更新される（22 行目）のに対し、task window は decided な値でのみ更新される（24 行目）。これらの更新のために、リーフノードだけでなく内部ノードの探索結果に対しても、そのノードの探索が指定された深さで終了したかどうかによって、uncertain か decided かというラベルをマスタは付ける。

図 7.2 に pass window と task window の例を示す。まず、ノード N を訪問したときの pass window が (20,80)、task window が (-10,80) だとする。定義より、pass window は task window と同じかそ

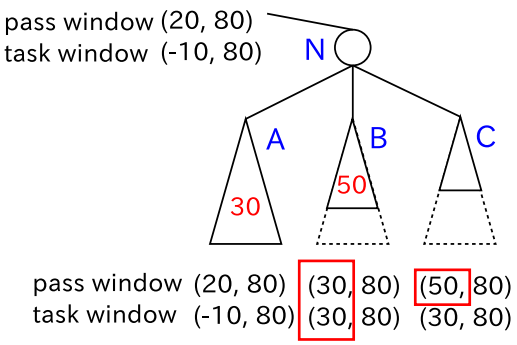

図 7.2: Pass window と task window の例

れより狭くなることに注意する。ここで、部分木 A を訪問するときの窓は N を訪問したときの窓をそのまま使う。探索終了後、部分木 A は決められた深さの探索が終了したことが判明し、その評価値は 30 だとする。この場合、pass window も task window も狭められ、両方とも (30,80) になる。この窓を用いて部分木 B を探索する。部分木 B はまだ決められた深さの探索は終了していないが、この pass では 50 の評価値が返ってきたとする。このときは、pass window だけ狭められ、(50,80) となり、task window は (30,80) のままである。この窓を用いて部分木 C を探索することになる。

ここで、task window を用いた理由について述べる。ワーカに送る探索窓としては、pass window を用いることも考えられる。ただし、pass window は予測された探索窓であるため、マスタのリーフノードの評価値が変わると探索窓も変わってしまう。したがって、pass window を用いてタスクの実行が終了しても、探索窓が変わることにより、再びタスクの実行を行わなければならないことがある。APHID では探索窓として、現在のルートノードの評価値から予測した探索窓を用いているが、予測したものである以上、タスクの再実行が起こりうる点では同じである。本研究では、性能の解析をより行いやすくすることを考慮し、decieded な結果だけを用いた探索窓を task window として用いた。これは、探索が進むにつれて狭くなることはあっても、広がることはない[2]。広い探索窓で探索した結果は狭い窓でも用いることができるので、再探索が起こることはなくなる。

7.2.1.3　マスタが保持する探索木のリーフノードの処理とタスクの送信

図 2.8 の AlphaBeta() と図 7.1 の MasterAlphaBeta() の違いの二つ目はマスタは残り探索深さが $d-d'$ になると、タスクをワーカに送信する点である（14 行目）。関数 EvalAndSendTask() は APHID と同様にはじめに今までにその局面について得られている最も深い結果を調べる。もし、その評価値が pass window の範囲で利用することができるならば、この関数はその評価値を返す。このとき、利用できるかどうかの判断は、2.2 節で述べたトランスポジションテーブルの結果を使うときの判断と同様である。もし結果を利用できないならば、この関数は次に深い探索を順番に調べていく。も

[2] 7.5 節では、深い結果を用いることにより、探索結果が変わることがあり、その場合は task window も広がる可能性がある。

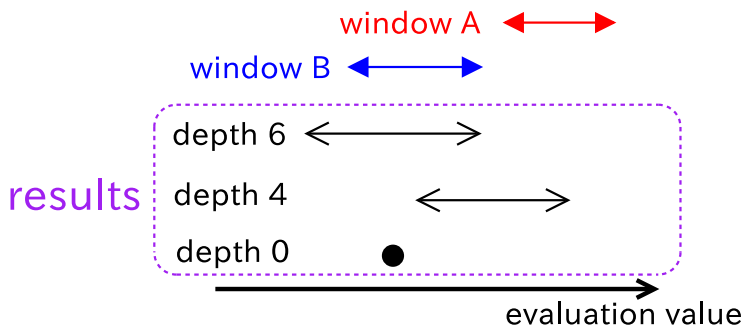

図 7.3: マスタでのタスクの結果の利用

し全ての結果が pass window では用いることができないのであれば、この関数は静的評価値を返す。もし、EvalAndSendTask() が uncertain な値、つまり探索が終了しておらず途中結果を返すときは、マスタは同時にワーカにタスクを送信する。そのタスクの探索窓は task window(a2 と b2) にセットされる。以前の pass において、そのタスクがすでに送信されていたとしても、マスタはそのタスクを再び送ることに注意する。

タスクは、局面、指定された探索深さ $(d - d')$、探索窓、signature の 4 つの情報からなる。同じ局面でも探索窓が異なる複数のタスクが存在しうることに注意する。signature はルートからそのノードまでのエッジにつけられた番号からなるリストである [55]。エッジに付けられた番号は、その子ノードが兄弟ノードの中で何番目に有望かを表す。この signature は [96] において優先度として用いられている。この優先度はタスクを深さ優先探索順に実行するように設計されている。本章では signature をタスクの優先度の一部として用いる。

ワーカからマスタに送られる結果は、局面、実行した探索深さ、その結果得られた評価値の上界値と下界値からなる。マスタがそのリーフノードにおいて、利用できる結果の調べるときの例を挙げる。図 7.3 では、すでにこのリーフノード以下の部分木の探索結果が深さ 6 と深さ 4 について分かっている場合を示している。まず、現在の pass window が図中の window A の場合を考える。深さ 6 の結果により、評価値が探索窓の範囲外にあることが分かるので、この結果は利用可能である。したがって、この結果を用いる。次に現在の pass window が図中の window B の場合を考える。深さ 6 の結果では探索窓の範囲の値について情報を与えないので用いることができない。深さ 4 の結果も同様に用いることができない。したがって、この場合、マスタは深さ 0、すなわち静的評価値を用いることになる。静的評価値は上界値と下界値が一致した値であり、範囲ではないため、静的評価値は必ず用いることができることに注意する。

7.2.1.4　マスタでの子ノードの並び替え

図 2.8 の AlphaBeta() と図 7.1 の MasterAlphaBeta() の違いの三つ目は、子ノードの並び替えについてである。反復深化はその局面が過去の pass も含めて、はじめて訪問されたときにだけ行

われる（15 行目）。この反復深化はマスタの中だけで完結するものであり、浅い探索の途中では、EvalAndSendTask() でワーカにタスクを送信せず、静的評価値を用いるだけとした。以降の pass ではマスタは以前の pass で得られた結果を用いる。ある子ノードについて pass window の範囲で decided な値がすでに分かっていれば、マスタはその値を用いて pass window と task window の両方を狭めてから、残りの子ノードの探索を行う（17 行目と 18 行目）。

子ノードの並び替えについて、さらに詳しく述べる。マスタのゲーム木中のノードは、必要な深さの探索が終了したノード、つまり、decided な結果が得られているノードでは、その評価値の下界値と上界値を保持している。これらはトランスポジションテーブルの結果と同様に更新される。また、decided な結果とは別に、そのノードの予測評価値と予測上界値も保持している。この予測評価値と予測上界値も decided な結果と同様な方法で更新されるが、その方法は少しだけ異なる。ノードがはじめて訪問されたときは予測評価値はそのノードの静的評価値[3]、予測上界値は INF に設定する。ノードの探索が終了したときは、既に保持されている予測評価値と予測上界値をそれぞれ s_e と u_e とすると、これらは次のように更新される。

- $v_p \leq \alpha_1$ のとき、s_e はそのまま、$u_e \leftarrow \min\{u_e, \alpha_1\}$
- $\alpha_1 < v_p < \beta_1$ のとき、$s_e \leftarrow v_p$、$u_e \leftarrow v_p$
- $\beta_1 \leq v_p$ のとき、s_e はそのまま、$u_e = \mathrm{INF}$

ただし、α_1 と β_1 はそのノードの探索開始時の pass window、v_p はそのノードの探索が終了したときの評価値である。v_p は uncertain であることもある。この方法で更新を進めていくと、予測上界値だけが小さくなることがあるため、予測評価値が予測上界値より大きくなり矛盾することがある。しかし、これらの予測値は並び替えに用いるだけであり、そのためには、現在得られている最新の結果を用いることが重要であるため、この矛盾は問題ではない。

実際に子ノードを並び替えるときの手順について述べる。子ノードの上界値は、負号をとることにより、親から見れば下界値となることに注意し、親から見たときの上界値と下界値を用いて以下の説明を行う。まず、必要な深さの探索が終了している decided な子ノードについて、得られている下界値の最大値 ($l_{\max}$) を求める。この最大値が pass window の b_1 を超えていれば、子ノードを探索する必要はないので、並び替えを終了する。次に、decided な子ノードのうち、その上界値が $l_{\max}$ を超えていないものについては、探索する必要がないため、並び替えの対象から外す[4]。最後に、残った decided な子ノードと全ての uncertain な子ノードを並び替える。基本的には予測下界値が大きいものほど先に探索されるように並び替える。予測下界値が同じものが複数存在する場合（ほとんどの場合、その値は –INF である）の中では、予測評価値が大きいものが先に探索されるようにする。予測評価値と予測下界値の二種類の予測値を並び替えに用いるのは、次の二つの理由からである。一つ目は、探索窓に含まれなかったときの評価値は正確には分からず、上界値や下界値を用いるだけでは、複数の子ノードで同じ値になってしまうため、並び替えに用いることができず、予測評価値が必要となる。二つ目は、枝刈りが前回の pass で発生したときは、今回の pass では枝刈りを引き起

[3]静的評価値に初期化されるのは評価実験の項で述べる人工木の場合。将棋のゲーム木の場合は、静的評価値を求めず遷移確率を代わりに入れておく。ただし、遷移確率が入ったままの子ノードより、評価値が入っている子ノードの方が先に探索されるようにする。

[4]実際には、次の段落で述べる signature を必要以上に変えないようにするために、並び替えは他の子ノードと同様に行うが、探索はしないようにしている。

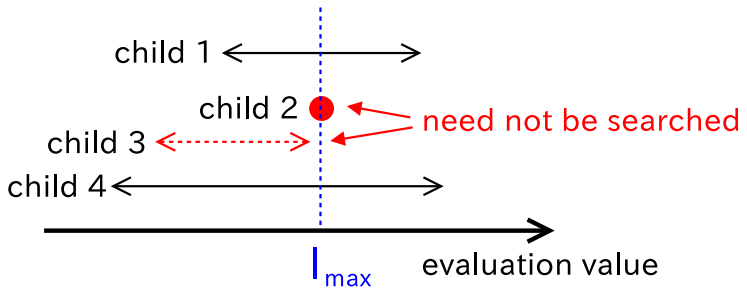

図 7.4: 子ノードの並び替えの際の decided な子ノードの結果の利用

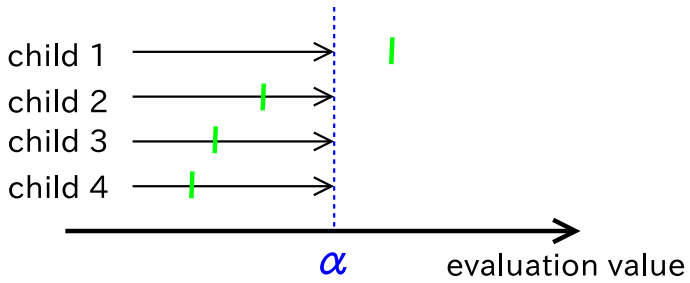

図 7.5: 親ノードが ALL ノードの場合の子ノードの並び替え

こした子ノードを優先的に探索したいが、そのためには予測下界値が必要となるからである。また、並び替えには decided な値は用いない。探索が終了したときしか更新されない decided な値よりも、毎回の pass で更新される予測値の方が新しい情報であるからである。

また、図 7.1 の SortChidlren() は並び替えられた子ノードのリストだけではなく、評価値も返す。評価値を返すのは、すでに decided な子ノードがいた場合、探索しない代わりにその評価値を探索窓に反映するためである。評価値としては、上述した $l_{\max}$ が返される。decided な子ノードがいなかった場合は $l_{\max} = -\mathrm{INF}$ とする。

子ノードの並び替えについて図を用いて説明する。まず、decided な結果を調べる操作の例を図 7.4 に示す。横軸は評価値であり、decided 子ノードのそれぞれについて上界値と下界値を示している。このうち、子ノード 2 は上界値と下界値が等しく、評価値が範囲ではなく値であることを表す。このとき、$l_{\max}$ は子ノード 2 の評価値である。また、子ノード 2 と子ノード 3 はその上界値が $l_{\max}$ を超えていないので、この pass では探索する必要がない。代わりに現在の探索窓の α が $l_{\max}$ より小さければ、子ノード 2 の結果を用いて、α を $l_{\max}$ まで大きくする。ここで用いるのは decided な値なので、pass window と task window、両方の α の値を更新する（図 7.1 の 17 行目と 18 行目）。子ノード 1 と子ノード 4 は decided ではあるが、この pass でも探索される。

次に、decided な結果を使った上で、残った子ノードの並び替えについて図を用いて説明する。まず、親ノードが ALL ノードだったときを考える。図 7.5 に親から見た子ノードの予測値を示す。ALL

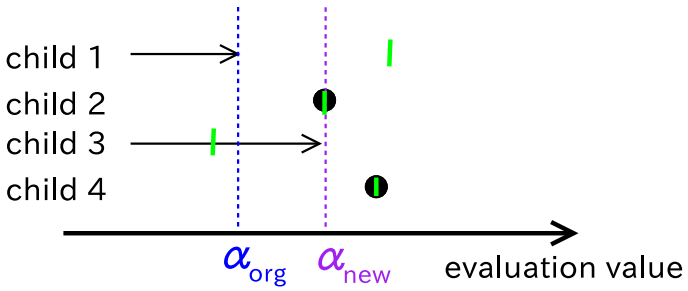

図 7.6: 親ノードが PV ノードの場合の子ノードの並び替え

ノードなので、前回の探索窓の α を、どの子ノードの評価値も上回らなかったことになり、予測上界値はすべての子ノードについて α となっている。しかし、現在の実装では、この予測上界値、子ノードから見れば予測下界値は保持しない。たとえ保持していても、子ノードの並び替えに用いることができないことはは図から分かる。このような場合に、予測評価値を保持していれば並び替えを行うことができる。図では緑の線が予測評価値を示している。並び替えのときは、まず予測下界値で並び替えるが、ここではすべて –INF だとすると並び替えができない。そこで予測評価値を用いて並び替えを行う。もし、予測下界値が –INF ではないものがあった場合、それは予測評価値に関わらず、先に探索されることになってしまう。しかし、予測下界値の更新のときに –INF を入れるようにしているため、予測下界値が –INF でないのは、違う経路で同じ局面に至った子ノードが存在する場合に限られるため、悪影響は少ないと考えた。以降で述べる、親ノードが PV ノードや CUT ノードの場合も、違う経路で同じ局面に至った子ノードが存在した場合に予測下界値がセットされている可能性があるが、その影響は少ないと考えた。

親ノードが PV ノードだったときの例を考える。図 7.6 に親から見た子ノードの予測値を示す。前回の pass では子ノード 1 から 4 の順に探索されたとする。このとき、子ノード 1 と子ノード 3 については、探索窓を超えなかったため、予測上界値しか得られなかった（そしてそれは保持されていない）。子ノード 2 と子ノード 4 は前回の pass で範囲ではなく値が得られた場合を示している。この場合、緑で示した予測評価値も更新されたものを示している。このとき、まずは予測下界値を用いて並び替えを行うので、子ノード 4 が最初に探索され、次に子ノード 2、残った二つは予測下界値が –INF だとすると、予測評価値を用いて子ノード 1、子ノード 3 の順に探索される。

最後に、親ノードが CUT ノードだったときを考える。図 7.7 に親から見た子ノードの予測値を示す。図 7.6 とは子ノード 4 の結果が違うだけの場合である。この場合、子ノード 4 は、親ノードから見たときの予測下界値を保持している。そしてこれは他の子ノードの予測下界値より大きい。したがって最初に子ノード 4 が探索されることになる。前回の pass で枝刈りを起こした子ノードから探索することができる。

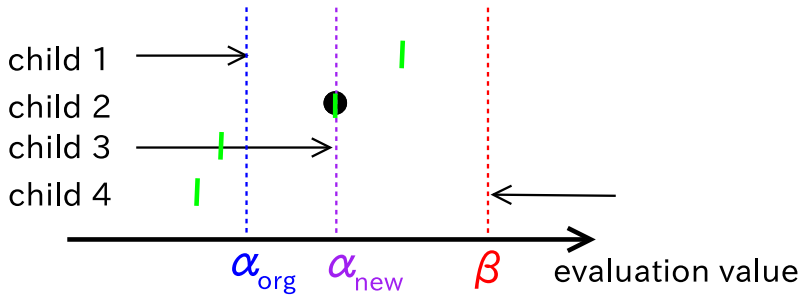

図 7.7: 親ノードが CUT ノードの場合の子ノードの並び替え

7.2.1.5　タスクの負荷分散

マスタは負荷分散の観点からタスクを送信するワーカを決定する。つまり、マスタはタスクを持っていないアイドルなワーカの数を減らそうとする。このために、マスタは各 pass において、各ワーカに局面を何個ずつ送ったかを数えておく。局面について数えるため、同じ局面に対する 2 つ以上のタスクがあった場合は、一回しかカウントされない。これは同じ局面に対するタスクは結果が使いまわせることも多く、あまり負荷にならないと考えられるためである。以前の pass でまだ送信されたことがないタスクの局面については、マスタは直前の pass で送信された局面の数が最も少なかったワーカを選び、そのワーカに対するカウントを 1 増やす。もし候補となるワーカが複数いた場合は、ラウンドロビン方式で選ぶ。局面が以前の pass ですでに送信されたことがあった場合は、マスタは、基本的には、そのタスクが前に送信されたワーカ（オーナー）にそのタスクを再び送信する。これは、トランスポジションテーブルを有効活用するためには、同じ局面は同じワーカに送信するべきであるからである。しかし、負荷分散のために、以下の 3 つの条件が全て満たされる場合は、オーナーが変更され、違うワーカが選ばれる。

- その局面は今回の pass でまだどのワーカにも送信されていない
- 前回の pass で局面（つまりタスク）を受け取っていない、アイドルと思われるワーカが少なくとも一つは存在する
- その局面のオーナーであるワーカは、今回の pass で少なくとも 1 つの局面をすでに受信している

オーナーが変更された場合、アイドルだったワーカがその局面に対する新しいオーナーとなり、その局面については以降その新しいオーナーとなったワーカに送信される。

7.2.2　ワーカの実装の詳細

図 7.8 にワーカプログラムの擬似コードを示す。ワーカはタスクを入れる優先度付きキューを管理する（図 7.8 における taskQueue）。関数 RecvMessages() は他のプロセスからのメッセージを調べ、受信する。この受信操作は関数 WorkerAlphaBeta() の実行中にも、各ノードを訪問するたびに行わ

```
1  void WorkerSearchLoop(){
2    while(the master continues the search){
3      do{
4        RecvMessages();
5      }while(taskQueue.empty());
6      task = taskQueue.top();
7      d = GetSearchDepth(task);
8      if(task need not be executed){
9        tasksQueue.dequeue();
10       continue;
11     }
12     //message checking is also performed in this funtion
13     score = WorkerAlphaBeta(task.position, d, task.alpha, task.beta)
14     if(this search was not aborted){
15       SendReslt(task.position, d, score);
16       SetTaskResult(task.position, d, score);
17       if(d == task.depth) TaskQueue.dequeue();
18     }
19   }
20 }
```

図 7.8: ワーカプログラムの擬似コード

れる。ワーカはタスクを受信すると、そのタスクを自分のタスクキューに追加する。Pass message を受信したときは、前回の pass message を受信してからこれまでに受信しなかったタスクをタスクキューから削除する。これらのタスクキューの変更の結果、WorkerAlphaBeta() が実行していたタスクが、最も優先度が高いタスクでなくなることがある。この場合、実行していた WorkerAlphaBeta() は中止される。ワーカは最も優先度の高いタスクを得る（6 行目）。関数 GetSearchDepth() は、ある局面がそのワーカによってどの深さまで探索されているかを調べる。タスクによって指定された深さと探索窓の範囲で、利用可能な結果が得られていた場合はそのタスクは無視される（8 行目から 11 行目）。必要な深さの結果が得られていない場合は、利用可能な結果がどの深さで実行されたかを調べ、その深さに 2 を足して次の深さの探索を実行する。このために、ワーカは局面の探索結果を保存しておく（16 行目）。関数 WorkerAllphaBeta() は図 2.8 の $\alpha\beta$ 探索の擬似コードと基本的に同じであるが、4 行目の直前に RecvMessages() が挿入され、この関数により関数が中止される処理を行っている。もし、探索が RecvMessages() によって中止されることなく終了した場合は、ワーカはマスタに結果を返す（15 行目）。このとき、実行した探索の深さが $d - d'$ ならば、タスクをタスクキューから削除する。探索の深さが $d - d'$ に足りない場合は、そのタスクはタスクキューに残され、後で少し深い探索が実行されることになる。

ワーカ間でのトランスポジションテーブルの共有は並列ゲーム木探索では重要な問題である。本章の実装では深い探索結果、つまりタスクとなる部分木のルートに近い部分だけの結果を共有するという APHID の方法を元にした実装を行った。各ワーカは結果をトランスポジションテーブルに格納すると同時に、トランスポジションテーブルの該当エントリへのポインタを他のテーブル（ポインタテーブルと呼ぶ）に格納する。ワーカは仮想的にリング状の通信経路を構築する。各ワーカ

は自分の直前のワーカから受信し、自分の直後のワーカに送信することによって、結果を共有する。最初のワーカが結果を送信するときは、ポインタテーブル中のポインタに指されている全てのエントリを次のワーカに送信する。送信される各エントリにはそのエントリを追加したワーカの ID も保存される。ワーカは直前のワーカからエントリを受信すると、全てのエントリの内容を自分のトランスポジションテーブルに反映させる。その後、受信したエントリと自分のポインタテーブルに指されているエントリの両方を次のワーカに送信する。ただし、受信したエントリのうち、保存されている ID が次のワーカの ID であるものについては、送信されずに捨てられる。すでに全てのワーカで共有されたエントリであることが分かるからである。

タスクの優先度は以下のように決めた。まず、最後の pass message を受信した後に受信したタスクは他のタスクより優先度が高い。次に、各タスクについて探索が終了した深さが浅いほど優先度を高くする。次にタスクの signature をチェックし、深さ優先探索で先に探索されるものが優先される。最後に探索窓を調べ、より狭い探索窓のタスクが優先される。先に β の値を調べ、それが小さい方が優先、次に α の値を調べ、それが大きい方が優先されるようにした。

7.3 ゲーム木の探索時間による評価

評価実験では二種類のゲーム木を用いた。一つは人工木であり、もう一つは激指の指し手の生成、並び替えを行う関数と評価関数（以下静止探索を含む）によって生成された将棋のゲーム木である。前者は探索アルゴリズムを単純な場合について分析するのに用い、後者は前者と比較することによって、より現実的なゲーム木の場合については何が異なるのかを調べるために用いた。

7.3.1 人工ゲーム木

人工木としては [93] で用いられている incremental random tree を用いた。これは木のエッジに乱数が割り当てられたものである。リーフノードの評価値はルートノードからそのノードまでの経路上にあるエッジの値の和とする。[93] ではエッジの乱数として一様乱数が用いられているが、本章ではより現実的な分布を用いた。用いた分布は、親ノードと子ノードで評価関数の値の差の分布であり、次節で述べる将棋のゲーム木を探索させることによって統計をとった。

ゲーム木中のノードは何度も探索されることになる（多重反復深化やマスタの探索など）。よって同じノードは必ず同じ評価値を返さなければならない。これを実現するために、[70] と同じような、ルートからそのノードまでの経路に依存する乱数の種を用いる手法を用いた。結果として生成された人工木は合流のある DAG(Directed Acyclic Graph) ではなく、合流のない木構造となる。つまり、異なる二つの経路を経て到達した二つのノードは必ず異なる状態を指す。この場合でも、同じ局面が何度も訪問されることから、トランスポジションテーブルは依然として必要であることに注意する。

ゲーム木の分枝数は並列プログラムの性能に大きな影響を与えることがある。YBWC の性能は分枝数に大きな影響を与える（分枝数が 2 のときは逐次探索と同じであることからも分かる）が、APHID の性能は分枝数に影響されないことが報告されている [24]。本章では分枝数が 5 と 20 の人

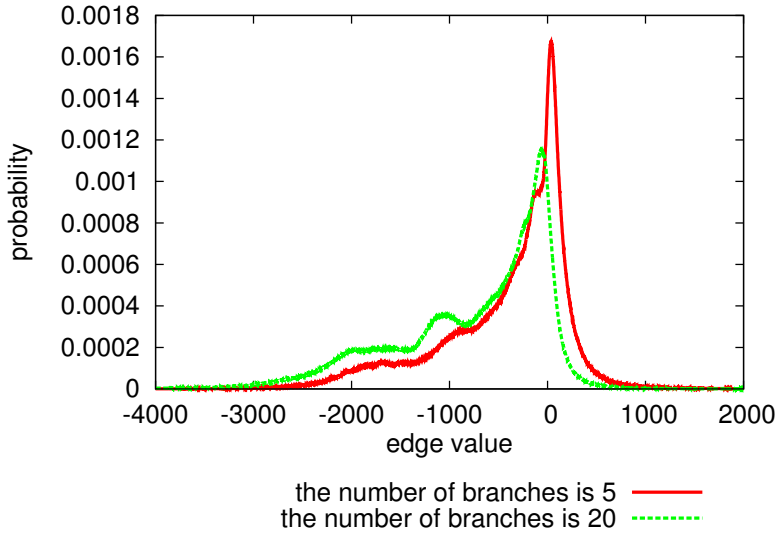

図 7.9: 人工木のエッジに与えられる乱数の分布

工木を生成した。これらは全てのノードが同じ数の子ノードを持つ均一な木である。将棋のゲーム木の平均分枝数は約 80 であるが、実際の将棋プログラムで考慮される有効分枝数はずっと少ない [48]。

分枝数が 5 と 20 のときのエッジに与える乱数の分布を 7.9 に示す。これらのデータは、分枝数を制限せずに探索し、各ノードの子ノードの評価値を全て保存しておき、浅い探索で最善だった子ノードと、それ以外で評価値の良い子ノードから順に選んで、全部で 5 個や 20 個の子ノードを選び、その評価値を調べたものである。横軸は親ノードと子ノードの評価値の差、縦軸は出現確率である。実際の将棋では合法手の多くは考える必要のない手だと言われているため、分枝数が多いほど、分布全体が評価値が小さいほうにシフトしていることが分かる。

生成された人工木の性質を調べたところ、並び替えの精度が高いことが分かった。つまり、浅い探索で最善だと判断された子ノードは次の少し深い探索でも最善である可能性が高かった。並列探索の性能は並び替えの精度が高い方が性能が出やすい。これは最小探索木をより簡単に予測し、訪問ノード数を減らすことができるからである。

7.3.2　将棋のゲーム木

より現実的なゲーム木を対象として、激指の評価関数、指し手生成、指し手の並び替えの 3 つを用いて、$\alpha\beta$ 探索プログラムを実装した。激指は他にさらなる性能向上のための工夫を多く用いているが、今回の実験ではこれらを用いることはしなかった。並列探索の振る舞いの解析をより簡単に行うことが目的であるためである。以降激指の関数によって得られたこのゲーム木を「将棋のゲーム木」と呼ぶことにする。

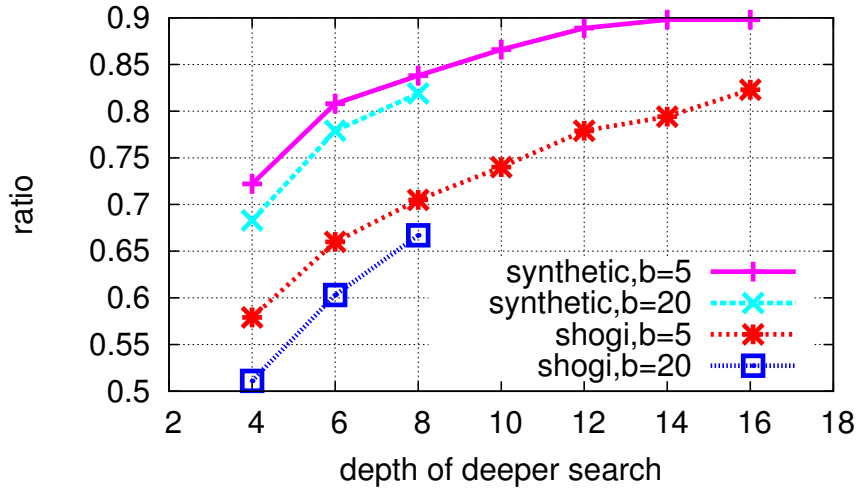

図 7.10: 浅い探索で最善と判断された子ノードが深い探索でも最善と判断される割合

人工木との比較を目的として、将棋のゲーム木でも分枝数を同様に 5 と 20 に制限した。各ノードでは、激指の関数を用いて子ノードを生成、並び替えを行った後に、最も有望な 5 個（20 個）の子ノードを選んだ。将棋では、王手がかかっている局面など、子ノードの数がこの指定した数より少なくなることがあることに注意する。

人工木と将棋のゲーム木の違いについて説明する。まず、将棋のゲーム木では異なる経路を経て到達した二つのノードが同じ局面を指すことがある。このため、ワーカ間でのトランスポジションテーブルの共有が重複探索を避けるために重要となる。次に将棋のゲーム木は人工木ほど指し手の並び替えの精度が良くない。図 7.10 に人工木と将棋のゲーム木の指し手の並び替えの精度の違いを示す。それぞれ 3200 個のゲーム木を探索したときのルートノードに着目して調べたものである。縦軸は深さが 2 だけ浅い探索の最善の子ノードが、深い探索でも最善だった割合を示しており、横軸はこのときの深い方の探索深さである。例えば、分枝数が 5 の場合、将棋のゲーム木では、深さ 14 の探索で最善だと判断された子ノードの内の 82.3%が深さ 16 でも最善であった。これが人工木では 89.8%になっている。分枝数が 20 の場合でも、深さが 6 と 8 の探索の間では、将棋のゲーム木では 66.7%と低いが、人工木では 81.9%と高い。最後に、探索時間が部分木間で大きく異なることがある点が将棋のゲーム木の特徴である。この主な理由としては王手の局面など、ノードによって子ノードの数が変わることが挙げられる。

7.3.3　深い探索結果の利用

トランスポジションテーブルではより深い探索結果が利用可能な場合には、それを用いることを 2.1.2 節で述べた。本実験では、人工木を用いた実験では深い探索結果は用いないことにした。将棋のゲーム木を用いた場合では、ワーカのトランスポジションテーブルに限り、深い結果を用いること

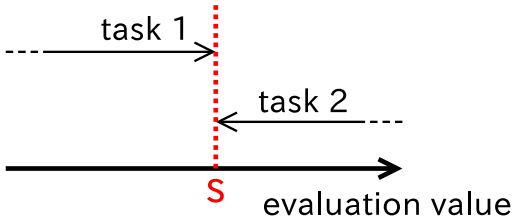

図 7.11: タスクの評価値が変わっているかもしれない例

にした。

トランスポジションテーブルを用いることで、探索結果が変わる可能性があることはすでに述べた。これにより、いったん終了したタスクの結果が変わることがありえる。これによって起こりうる例を図 7.11 に示す。このワーカはまずタスク 1 を実行し、図に示すように評価値が s 以下の範囲にあることが分かったとする。次にタスク 2 を実行した場合、例えばタスクの探索窓が $(-\mathrm{INF}, \mathrm{INF})$ であったとしても、評価値の範囲は、s 以下であることが分かっているから、窓を $(-\mathrm{INF}, s)$ に狭めてから探索したところ、s が返ってくることがある。この場合、実際に評価値が s なのか、それとも評価値が変わって s 以上になったのかはこれだけでは分からない。ここでの実験では、簡単のため、評価値は変わらないとみなし、この場合には s が評価値であることにした。なお、評価値が変わっているかどうかを調べるためには、$(-\mathrm{INF}, s+1)$ という探索窓で探索する方法があり、後述する 7.5.1.1 節ではそのようにしている。

7.3.4 実験設定

本章の実験では CPU の速度を揃えるため、表 4.1 のうち、E5530、E5620、E5-2665 を用いて実験を行った。ハイパースレッディングが有効になっているノードもあるが、実験では 1 ノードあたりのプロセス数は最大でノードのコア数とした。実装には MPI を用い、処理系としては MPICH2 を用いた。

比較対象となる逐次プログラムのトランスポジションテーブルのエントリ数は人工木を用いた実験では、1,000,000,000 とした。将棋のゲーム木を用いた実験では、7.4.2 節で述べる方法により、マスタに相当する部分と、ワーカに相当する部分で、1,000,000,000 エントリのテーブルを二つ用意した[5]。人工木、将棋のゲーム木ともに、並列プログラムでは、ワーカ一つあたりのエントリ数は 31,250,000 とした。マスタは自身が管理する探索木のノードについて全ての情報を保存するテーブルを保持する。これらの設定は人工木、将棋のゲーム木で共通とした。

実験で探索する局面は人工木、将棋のゲーム木ともに 32 局面とした。人工木については乱数の種を変えることにより、32 個のゲーム木を探索させた。将棋のゲーム木に関してはプロの棋譜から序盤と終盤を除いて 1 棋譜から 1 局面ずつランダムにとってきた 32 局面を用いた。それぞれのゲーム木は一回ずつ探索させた。

[5]分枝数が 5 のときは $d' = 12$、分枝数が 20 のときは $d' = 6$ としている。

表 7.1: 人工木を用いた結果 ($b = 5$, $d = 24$, $d' = 12$)

# of processes	method	search time [seconds]	speedup	# of leaves [$\times 10^6$]	start-up idle time [seconds]	other idle time [seconds]
1	sequential	19438.92	1	790		
512	w/ updates	57.13	340	1171	0.63	0.42
	w/o updates	50.32	386	1010	0.62	0.47
1536	w/ updates	26.04	746	1394	0.85	0.69
	w/o updates	28.81	675	1409	0.85	2.33

評価では以下の 3 つの方法を比較した。一つ目は最小探索木の予測の動的な修正を行うのに対し、二つ目は行わない。動的修正を行わない場合は、図 7.1 の関数 GetSearchDepth() は常にタスクが要求する深さを返す。この場合でもワーカは反復深化を行うが、浅い探索の結果をマスタには報告しない。これら二つの方法ではワーカ間でトランスポジションテーブルの共有は行わないことにした。三つ目の方法は、最小探索木の動的修正とトランスポジションテーブルの共有を両方行うものである。ワーカはタスクの部分木のルートからある深さの結果までを共有することにした。この深さは分枝数が 5 の場合は 4、深さが 20 の場合は 2 とした。また、データサイズの大きすぎるメッセージを通信しないようにするために、一回のメッセージで送るエントリ数を 10,000 に制限した。これ以上のエントリを追加しそうになった場合は、それらを単に捨てるものとした。

性能測定結果を表 7.1 から表 7.5 に示す。探索時間は 32 個のゲーム木の探索時間の相乗平均である。速度向上比は逐次探索の実行時間を並列探索の実行時間で割ったものである。表には訪問ノード数の平均とワーカあたりの平均アイドル時間も示している。ワーカのアイドル時間は開始アイドル時間とその他のアイドル時間で分けて調べた。start-up idle time はワーカが最初のタスクを受信するまでの時間、other idle time はワーカがタスクを終了してから次のタスクを受信するまで、あるいは探索全体が終了するまでの時間である。

次節から、表 7.1 から表 7.5 の結果を、着目する点を変えながら少しずつ見ていくことにする。

7.3.5 人工木に対する結果

表 7.1 に分枝数 b が 5 の場合の、表 7.2 に分枝数が 20 の場合の人工木の結果を示す。マスタの探索深さ d' は予備実験によって決めた。

最小探索木の予測の動的な修正が性能を改善している傾向にあるが、その改善の度合いは 1 割程度と大きくはない。これは人工木では指し手の並び替えの精度が高いからだと説明できる。つまり最小探索木は深く探索しなくとも、精度よく予測できるため、ワーカの浅い探索の結果を用いても、大きな予測の修正は起こらない。結果として、予測の修正の影響が少なくなっていると考えられる。

表 7.2: 人工木を用いた結果 ($b = 20$, $d = 12$, $d' = 6$)

# of processes	method	search time [seconds]	speedup	# of leaves [$\times 10^6$]	start-up idle time [seconds]	other idle time [seconds]
1	sequential	7321.58	1	149		
512	w/ updates	20.83	351	208	0.17	0.55
	w/o updates	22.85	320	228	0.17	0.56
1536	w/ updates	11.26	650	291	0.44	0.58
	w/o updates	13.07	560	322	0.43	1.02

予測の動的な修正の改善の度合いとは逆に、速度向上比は 1536 プロセスで 650 倍以上と、先行研究（チェスにおいて YBWC の速度向上比は 1024 プロセスで 344 倍 [37]）と比較して大きい。これは予測の精度が正しいため、最小探索木に含まれるノードを中心に探索することが可能であるためである。

7.3.6 将棋のゲーム木に対する結果

表 7.3 に $b = 5$、$d = 24$ のときの将棋のゲーム木に対する結果を示す。さらに、$b = 20$ のときの結果については、表 7.4 に $d = 12$ のときの結果を、表 7.5 に $d = 14$ のときの結果を示す。この実験では、プロセス数とタスクの粒度との関係性も調べるために d' を変更している。

まず、最小探索木の予測の動的な修正の影響に焦点を当てる。図 7.12 に予測の修正による性能向上を示す。横軸は予測の修正を行う場合の実行時間を行わない場合の実行時間で割った比、縦軸はリーフノード数に対して同様に求めた比である。括弧の中の数字は (b、d、プロセス数) になっている。マスタの探索深さ d' については、$b = 5$ のときは 12、$b = 20$ のときは 6 のものを示している。予測の修正によって探索時間も訪問リーフノード数も減少していることが分かる。問題サイズ d が大きいほどこの性能改善の程度は大きい。このとき、実行時間は予測の修正によって半分程度になっていることが分かる。これは人工木のときよりも影響が大きい。この理由としては将棋のゲーム木においては、指し手の並び替えの精度は高くなく、最初の予測だけでは不十分なことが多いからだと説明できる。

次にトランスポジションテーブルの共有が並列探索の性能に与える影響について見る。表 7.6 に共有しない場合の実行時間に対する共有をした場合の実行時間の比を示す。64 プロセスの場合は、d' が小さければ、4%から 6%程度の性能改善になっている。d' が大きいと性能の改善は見られず、逆にわずかだが遅くなる結果となっている。1536 プロセスの場合には、$b = 5, d = 24$ のときと、$b = 20, d = 12$ のときは 2%程度の性能向上になっているが、$b = 20, d = 14$ のときは、10%程度の改善になっている。

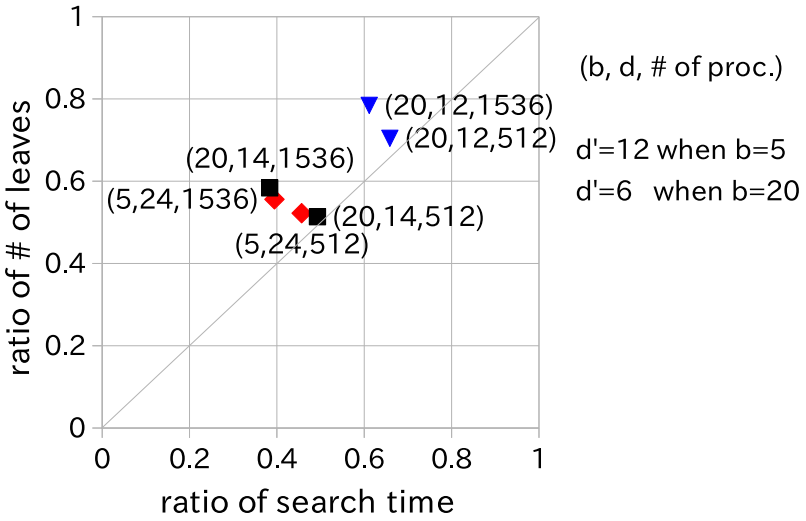

図 7.12: 最小探索木の予測の動的な修正を行った場合の、行わない場合に対する探索時間と訪問リーフノード数の比

最後に図 7.13 に予測の修正を行い、トランスポジションテーブルの共有は行わないプログラムの速度向上比を示す。d が十分大きく、d' が適切に決められていれば、プロセス数を増やすことで大きな速度向上が得られていることが分かる。しかし、約 250 倍という数字は人工木の場合よりずっと小さい。表 7.3、表 7.4、表 7.5 を見ると、速度向上が得られていないのは訪問リーフノード数が著しく増えているのが原因だと分かる。ワーカのアイドル時間も原因の一つではあるが、リーフノード数の増加に比べれば影響が小さいことが分かる。

7.4 速度向上を抑制する要因の分析

7.4.1 速度向上を抑制する要因

訪問リーフノード数の増加が速度向上の主な原因だと分かったが、この増加の要因は以下の 3 つに分けられる。

1. ワーカ間でのトランスポジションテーブルの共有がうまくいかなかった
2. 他のタスクの結果を得る前にタスクの実行を開始するので、ワーカが逐次探索より広い探索窓でタスクを実行している
3. 最小探索木の予測が不正確であるためにタスクの数が増えている

これらの要因は明確に区別できるものではない。たとえば、トランスポジションテーブルの共有がうまくいかず、最小探索木の予測の修正が遅れ、その結果タスクの数が増える、などといったことは

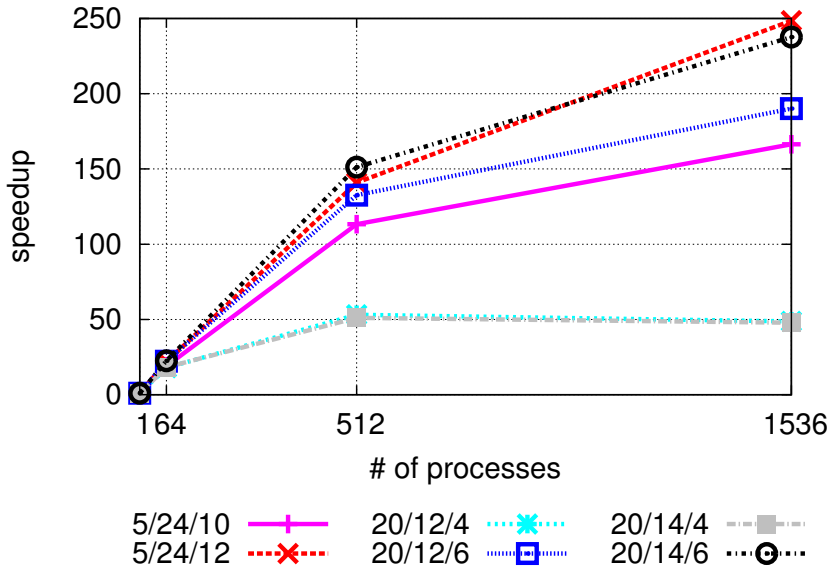

図 7.13: 並列プログラムの速度向上比

考えられる。しかし、本章ではこの 3 つの要因それぞれのリーフノード数の増加に与える影響を調べることを試みた。そこで、逐次探索プログラムを用いて追加実験を行った。

7.4.2　分析のための補足実験

まず、トランスポジションテーブルの共有がうまくいっていないことで、どの程度リーフノード数が増えるのかを調べるために、逐次探索プログラムに小さい変更を加えた。このプログラムは二つのトランスポジションテーブルを用いる。一つはマスタがワーカとは独立に探索木の情報を保持することを考慮したもので、ルートノードからの深さが d' 以下のノードの結果が格納される。もう一つはワーカのトランスポジションテーブルを真似たもので、ルートノードからの深さが d' 以上のリーフノードに近い部分の結果が格納される。ルートに近いノードを探索するときには一つ目のテーブルを、リーフに近いノードを探索するときには二つ目のテーブルを使えば、ワーカが完全にトランスポジションテーブルを共有したときの訪問ノード数を見積もることができる。逆に、ワーカ間でトランスポジションテーブルを全く共有しない場合の訪問ノード数を見積もるためには、二つ目のテーブルの利用を制限する。つまり、ルートノードからの深さが d' のノードの局面以下の部分木をタスクと見なし、違う局面のタスクの探索結果を用いないようにする。これはワーカ全ての異なるタスクがそれぞれ違いワーカで探索されるときをシミュレートしており、訪問リーフノード数が最大でどれだけ増えるかを見積もることができる。最後に 7.2 節で述べた共有手法の効果を見積もるため、逐次プログラムに異なる局面のタスクの結果も、部分木のルートに近いものに関しては利用可

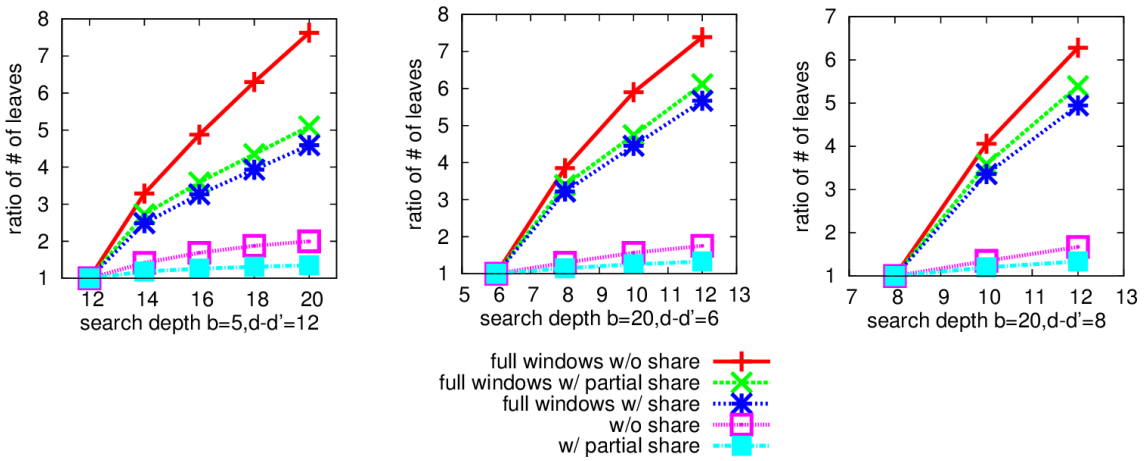

図 7.14: トランスポジションテーブルを共有しないことと、探索窓にフルウィンドウを用いることによる訪問リーフノード数の増加率

能にすることを行った。これを以下「部分共有 (partial share)」と呼ぶことにする。

次に、広い探索窓が使われたことによる訪問リーフノード数の増加の程度を調べるため、さらに小さい変更を逐次探索プログラムに行った。逐次探索で深さ $d-d'$ の部分木を探索を開始する直前に、探索窓を強制的にフルウィンドウに広げる。つまり $\alpha = -\mathrm{INF}$、$\beta = \mathrm{INF}$ とする。この部分木の探索が終了すると、探索窓は元に戻す。これにより、ワーカが全てのタスクをフルウィンドウで探索したときの訪問リーフノード数の増加を見積もることができる。もちろんこれは最悪のケースに対する見積りであり、実際の並列探索では、他のタスクの結果が使えるならば、タスクの探索窓は狭められる。

逐次探索プログラムを用いたそれぞれのシミュレーション方法に対して、訪問リーフノード数を数えた。ベースラインとしては二つのトランスポジションテーブルを用い、ワーカ間では完全に共有がされている状態をシミュレートしたものとした。ベースラインでは通常の探索窓で探索を行いフルウィンドウにすることはしない。図 7.14 に将棋のゲーム木を用いた場合の結果を示す。この図では $d-d'$ を固定し（$b=5$ のときは $d-d'=12$ であり、$b=20$ のときは $d-d'=6$ と $d-d'=8$ の 2 通りを試した)、d を少しずつ変えた。横軸は d であり、縦軸はベースラインに対して、各シミュレーションの訪問リーフノード数が何倍に増えたかを示している。

また、図 7.14 の詳細な数字と、問題サイズが大きくなったときの予測値を表 7.7 と表 7.8 に示す。問題サイズが大きい場合は、逐次では時間がかかりすぎるため、予測値とした。予測値には?をつけてある。この予測値は、深さを x として、関数 $a-e^{-bx+c}$ のパラメータ a, b, c を、データ点にフィッティングさせることにより求めた。表では、table(normal) は通常の探索窓を使ったときに、トランスポジションテーブルを共有しないことによるリーフノードの増加率を示す。これは図 7.14 における「w/o share」の値である。同様に table(full) は探索窓にフルウィンドウを使ったときの、トランスポジションテーブルを共有しないことによるリーフノードの増加率であり、図 7.14 において、「full

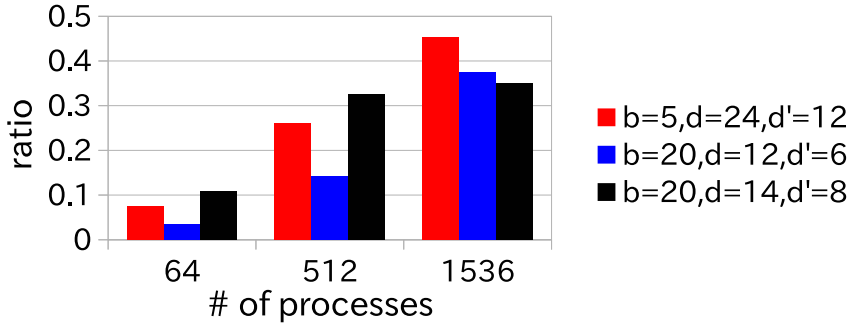

図 7.15: フルウィンドウを用いて実行が終了したタスクの数の実行が終了したタスクの数全体に対する割合

windows w/o share」を「full windows w/ share」で割った値である。full は探索窓にフルウィンドウを用いることによるリーフノードの増加率で、図 7.14 において、「full windows w/ share」の値である。

7.4.3　知見

図 7.14 を見ると、トランスポジションテーブルを全く共有しないことによって訪問リーフノード数が 2 倍程度に増えることが分かる。この増加率は分枝数が 20 の場合では少し小さくなる傾向がある。また、フルウィンドウを使ったときも増加率が小さいことが分かる。次に、トランスポジションテーブルを部分共有することで、共有しない場合と比べて、訪問リーフノード数が減っていることが分かる。しかし、実際の並列探索では共有の効果は最大でも 10%であった。また、先行研究の APHID でも、共有メモリでテーブルを共有すると 16 プロセスを用いたときで性能が 2 倍程度になるのに対し、分散環境で本章と同様の手法を用いた場合は 18%程度の性能向上であった [24]。これらのことを考慮すると、大規模な分散環境でトランスポジションテーブルを共有するのは、難しいと結論せざるを得ない。

図 7.14 はフルウィンドウによるリーフノード数の増加が大きいことも示している。これは最悪のケースなので、実際の増加の程度を知る必要がある。そこで、並列探索においてフルウィンドウを使って探索が終了したタスクの数を数えた。探索が終了したタスクの数全体に対するフルウィンドウを用いて探索が終了したタスクの数の比を、将棋のゲーム木の場合について、図 7.15 に示す。プロセス数が増加すると、フルウィンドウを用いて探索が終了するタスクの割合が増加していることが分かる。

最後に将棋のゲーム木の場合に 1536 プロセスを用いて探索した場合に、各要因によって訪問リーフノード数がどの程度増加しているかを見積もった。7.4.1 節で述べた要因は、(1) トランスポジションテーブルを共有しないこと (table)、(2) 広い探索窓を使うこと (windows)、(3) タスクの数が増え

ること (tasks)、の 3 つである。表 7.9 に各要因によってリーフノード数が何倍になっているかの見積りを示す。この表には全体で何倍に増えるか（各要因の増加率の積）の見積りと、実際の増加率も示している。トランスポジションテーブルを共有しないことで受ける増加率は、

$$(1-r)t_n + rt_f \tag{7.1}$$

で表される。ただし、r は図 7.15 から得られる割合、t_n と t_f は表 7.7 と表 7.8 で示した「table(normal)」と「table(full)」の値である。同様に、フルウィンドウを用いて探索を行うことによる増加率は、

$$(1-r) + rw \tag{7.2}$$

と表せる。ただし、w は表 7.7 と表 7.8 で示した「full」の値である。最後にタスクの増加率は逐次探索で実行されたタスク数と比較することで決定した。タスク数は逐次探索より少なくなっていることが多い。これは最終的に探索するタスクは最小探索木に含まれるものになることを示していると考えられる。

表 7.9 は全体の増加率が実際の増加率を大雑把に予測できていることが分かる。予測の増加率と実際の増加率の差の原因の一つとしては、今回の分析では、浅い探索は実行したものの、決められた深さの探索を完了する前に中止されて、最後まで実行されなかったタスクを無視していることが挙げられる。また表 7.9 は、広い探索窓を使うことによる訪問リーフノード数の増加率が他の要因よりも大きいことを示している。

64 コアを用いたときと、512 コアを用いたときに、表 7.9 と同様に、各要因による訪問リーフノード数の増加率を計算したものを表 7.10 と表 7.11 に示す。表 7.9 と合わせて比較すると、プロセス数が増えるほど、探索窓がフルウィンドウであることによる増加が大きくなっていることが分かる。

7.5 将棋プログラムとしての勝率の評価

本節では将棋プログラムとしての強さを評価することを目的として、プログラムにさらに変更を加え、激指と実際に対局させることにより勝率を評価した。前節まででは、並列探索プログラムの性質の分析を目的としていたため、分枝数を 5 や 20 に制限し、かつ実現確率探索を用いていないプログラムを用いて、ゲーム木を探索する時間を評価していた。しかし、このプログラムは簡単なプログラムであり、強いプログラムを作るためのものではなかった。

まず、7.5.1 節で、プログラムを強くし、対局できるようにするために追加した実装について述べる。主な実装の変更点は、実現確率の導入と時間制限に対応するための反復深化である。次に 7.5.2 節によって、実際に激指と対局させた結果を示し、考察を行う。

7.5.1 追加した実装

7.5.1.1 実現確率の導入

激指で用いられている実現確率探索は将棋プログラムの強さに大きく関わる手法である。そこで、並列プログラムに実現確率を導入した。まず、ワーカのプログラムは激指をそのまま用いることに

した。次に、マスタのプログラム内では、分枝数を制限することはせず、かつ実現確率を用いて探索深さを決めるように変更した。激指で用いられている他の探索手法の導入はしていない。実現確率を用いるときに問題となったのは、以下の 2 点である。

- より深い探索結果を使わなければならない点
- 実現確率探索における、有望な手の深い探索の実装

実現確率によって決められる深さを確率深さと呼ぶことにしているが、この確率深さはルートノードからの経路が異なると基本的に異なる値になる（図 2.4 を参照）。したがって、トランスポジションテーブルでは、同じ確率深さの結果はほとんど存在せず、本質的により深い結果を使わざるを得ない。深い結果を用いるということは、いったん実行したタスクでも、もう一度探索すると結果が変わることがあることを意味する。前節の実験では、探索性能の分析が目的だったため、結果が変わって以前の結果と食い違う可能性がある場合でも、結果は変わっていないとみなしていた。しかし、プログラムを強くすることを考えると深い結果を用いるべきであるし、ワーカとして激指を用いている以上、ワーカの探索の中ではトランスポジションテーブルによって深い結果を用いてしまうので、より深い結果を用いることは避けることができない。

そこで、実現確率を用いる場合については、いったん得られた評価値の範囲と矛盾した結果が得られたら、後から得られた結果の方を採用するようにした。7.3.3 節で述べた図 7.11 の例で説明する。この場合、評価値が $[-\mathrm{INF}, s]$ にあることが分かっている前提で、(s, INF) の探索窓で探索すると、s が返ってくることが問題であった。深い結果を用いる場合は、評価値が変わっていることに気づくために、タスクの探索窓を両側に 1 ずつ広げて探索することにした。この例だと、窓の下端は $-\mathrm{INF}$ なので、上端だけずらして $(-\mathrm{INF}, s+1)$ の探索窓で探索することにした。これによって評価値が s の場合は s が、評価値が変わっている場合は $s+1$ が返ってくるので判別することができる。

ワーカでより深い結果を用いることに合わせて、マスタでもより深い結果を用いるようにした。すでに得られていた評価値の範囲と矛盾する結果が得られた場合は、後から得られた結果の方を使うようにした。ただし、より深い探索結果を用いることで、結果が変わると、今まで終了していたと思っていた探索を再び実行しなければならなくなることがあるため、探索の終了が遅れる可能性がある。

実現確率探索においては、はじめは浅くしか探索していなかった部分木でも、評価値が良さそうなら深く再探索する。しかし、本研究では、簡単のため、すべての部分木の探索深さは実現確率だけで決定され、有望な手も深く探索しないものとした。[105] では、反復深化における浅い探索で最善だった子ノードだけは、実現確率を用いず深く探索するようにしているが、本研究はそれも行っていない。当然プログラムは弱くなるが、逐次探索を用いた予備実験により、マスタ部分で実現確率をまったく使わないものよりは強いことが判明したため、実現確率だけは用いることにした。

7.5.1.2　時間制限に対応するための反復深化

将棋プログラムの強さを評価する場合、1 手にかかる時間を固定して対局させることが多い。しかし、前節の実験では一つのゲーム木をある深さまで探索し終える時間で評価を行っており、任意のタイミングで探索を打ちきれるような実装にはなっていなかった。任意のタイミングで探索を打ちき

```
1  int IterativeMasterSearch(position, d'){
2    MasterSearch(position, d');
3    depth = d'+MIN_WORKER_DEPTH;
4    current_score = −INF;
5    while(true){
6      score = MasterSearch(position, depth);
7      if(time is up){ return current_score; }
8      current_score = score;
9      depth += 2;
10   }
11 }
```

図 7.16: 全体として反復深化を行うマスタの擬似コード

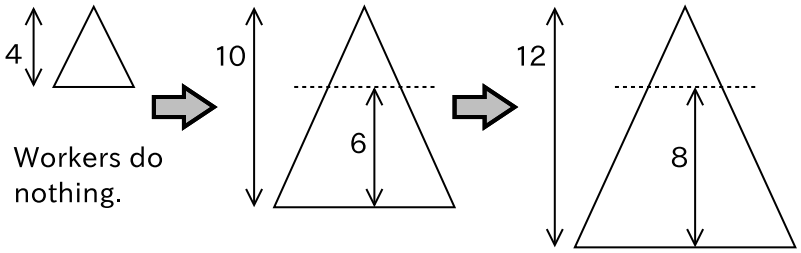

図 7.17: 全体として反復深化を行う場合の例

るためには、反復深化が用いられる。前節の実装では、ワーカでは反復深化を用いて探索を行うが、ワーカ全体が足並みを揃えて深さを少しずつ深くしていくわけではなかったため、探索全体としては反復深化になっていなかった。

任意のタイミングで探索を打ちきる簡単な方法として、各 pass においてルートでの最善手を保存しておき、制限時間になったら、最後に保存した最善手を返すという方法が考えられる。しかし、この方法では、たまたま浅くしか探索していない部分木が、その時間打ち切りの瞬間だけ最善に見えてしまっていたという可能性があるため、悪い手を選択してしまう可能性がある。高い勝率のためには、ひどく悪い手を指さないことが求められるため、この方法は時間打ち切りの方法として不適切である。

時間制限に対応するため、探索全体も反復深化させるような実装を追加した。その擬似コードを図 7.16 に示す。非常に浅い結果でも、ワーカがマスタに伝えてしまうと、通信が非常に多く発生してしまうため、マスタに結果を伝える最小の探索深さを決めているが、その深さを MIN_WORKER_DEPTH としている。探索の進行の例を図 7.17 に示す。この図では $d' = 4$、MIN_WORKER_DEPTH=6 の場合を示している。まずはじめに d' の深さをマスタだけで探索する。このときワーカにはタスクを送らない。この探索の目的は、マスタが保有する木を一通り構築することである。その後、探索深さを d'+MIN_WORKER_DEPTH に設定して探索し、以降ワーカが探索する深さだけを 2 ずつ増やす。マスタが探索する深さは全体の探索深さが大きくなっても変更しないことにした。

ワーカにおけるタスクの優先度にも変更を加えた。7.2.2 節では、すでに実行した深さが浅いタスクを優先していたが、このチェックを行わないようにした。これにより、深さ優先探索順と同じ順序でタスクが実行されていく。全体として反復深化をする場合、少しずつ探索の深さが大きくなっていくため、深さ優先探索と同じ順序で実行しても浅い探索の結果を用いるという意味では問題は少ない。それどころか、探索窓が早く狭まることになるため性能が良くなることも考えられる。実際、予備実験では探索時間が短くなったため、優先度を深さ優先探索順にした。

なお、APHID では、ワーカは自身が持っているタスクをすべて必要な深さまで探索した後は、投機的にタスクをより深く探索することを行っている。この投機的実行の結果は、現在の反復深化の深さが終了するまでは、マスタでは用いないとしていた。本研究の実装では、これらの投機的実行は行っていない。前述の通り、ワーカやマスタではより深い結果を用いることを避けることができないため、結果が不安定になりやすい可能性があったためである。

7.5.2 激指に対する勝率の評価

実際に激指と対局させてその勝率を調べた。実験設定は以下の通りである。

- 対局相手は 8 コアスレッド並列の激指
- 1 手 10 秒
- 相手手番時の予測読みは行わない
- プロの棋譜の 36 手後の局面から開始
- 各棋譜について先手後手入れかえて 2 局ずつ対局
- 評価値の絶対値が 5000 より大きくなって不利なら投了
- 200 手でも決着が付かない場合は引き分け
- 同じ局面[6]が 4 回登場したら引き分け

激指は表 4.1 の E5530 の計算ノード上で動かした。本研究の分散並列プログラムでは E5-2665 の計算ノード上で動かした。トランスポジションテーブルのサイズは激指で 250,000,000 エントリ、分散並列プログラムは 1 プロセスあたり、これの 1/8 である、31,250,000 エントリとした。また、マスタが探索する深さ d' は確率深さを用いた。ルートノードから確率深さが 6 以上になったノードがワーカに送信される。ただし、残り確率深さがなくなってしまった場合はワーカには送信せず、マスタで静的評価値を用いるだけにした。

対局結果を表 7.12 に示す。分散並列プログラムのプロセス数は 128、512、1536 の 3 通りを試した。対局数はプロセス数が 128 と 1536 のとき 100 局、512 のときに 140 局である。いずれの勝率も有意水準 5%の二項検定で有意ではない。ただし、128 プロセスでは 8 スレッド並列の激指に対して弱い傾向がある。512 プロセスと 1536 プロセスでは差は見られなかった。

[6]正確には局面のハッシュキー

7.5.3 勝率が低い理由の考察

512 プロセスを用いても、8 コア並列と勝率があまり変わらなかった理由について考察する。分散並列プログラムがあまり強くなかった理由として以下のようなものが挙げられる。

- 実現確率探索において有望な手を深く探索していない。
- その他、激指で用いられているような手法を用いていない。
- トランスポジションテーブルの共有をしていないため、深い探索結果が使えない。
- タスクの部分木の探索時間がばらつくため負荷分散が難しい。
- 激指では探索窓をパラメータとして、前向き枝刈りを行うが、探索窓が広いことが多い本研究の実装では、前向き枝刈りがされないため、訪問ノード数の増加が著しい。

はじめの二つは、そもそも並列化する前の逐次のプログラムが弱くなっているということである。本実装では、簡単のため、マスタのプログラムは単純なものとしていた。その中でも特に一つ目の実現確率探索において、有望な手を深く探索していないのは、プログラムの強さを下げている大きな理由として考えられる。他にも、二つ目にあるように、激指で用いられているような手法で導入していない。例えば表 5.2.2.6 で示した指し手の生成のうち、浅い読みの最善手は先に探索されるが、残りの指し手に関してはその他の指し手と同様にしか扱われないため、子ノードの並び替えの精度が悪い。

三つ目のトランスポジションテーブルの共有については、これまでも議論してきた。しかし、実現確率を用いる場合は、より深い結果を使うという目的もあるため、トランスポジションテーブルが共有されていないことで、さらに大きく不利になってしまっている。

最後の二つは、7.3 節で示した結果よりも並列化による速度向上が小さくなっている理由として考えられるものである。まず、7.3 節では分枝数を 5 や 20 に制限していたため、人工木よりは部分木の大きさがばらつくと言っても、そのばらつきはまだ大きくなかった。しかし、分枝数を制限しない場合、部分木によって大きく探索時間がばらつくようになる。さらに、実現確率を用いた場合、確率深さは急に深くなることもあり、タスクで探索することになる確率深さそのものもばらつくことになる。これらの原因により負荷分散が難しくなっており、7.3 節で示した結果より、アイドル時間が大きくなっていることが、並列性能を劣化させている原因となっている。

7.4 節では、探索窓が広いことで訪問ノード数がどの程度増えているかを示した。ところが、激指では探索窓が広いと、探索窓が適切に狭められている場合に比較して、7.4 節で示した以上に訪問ノード数が多くなることが分かった。これは、探索窓の範囲を明らかに超えそうなときは前向き枝刈りを行っているからだと考えられる。ワーカに激指を用いた場合では、訪問ノード数が著しく増えている可能性がある。

分散並列プログラムの元となった逐次プログラムが激指に比べてどの程度弱くなっているかを調べるために追加で対戦実験を行った。分散並列プログラムの元となった逐次プログラムを簡易逐次プログラムと呼ぶことにする。表 7.13 では、実現確率探索において有望な手の深い探索を行わない簡易逐次プログラムと、これを行う簡易逐次プログラムの対戦結果を示す[7]。後者のプログラムの持

[7]有望な手の深い探索を行う場合は、浅い探索で最善だった子ノードについては確率深さを 1 だけ減らす。これを行わないようにするためには、減らす確率深さを求めなければならないが、このためには激指の指し手生成の関数を呼ぶ必要があ

ち時間を 1 手 10 秒とし、前者のプログラムの持ち時間を変えて実験した。有望な手の深い探索を行わない場合、これを行うプログラムと同じ強さになるには、3 倍の時間をかける必要があることが分かる。次に、表 7.14 に、有望な手の深い探索を行う簡易逐次プログラムと、1 コアしか用いない逐次の激指を戦わせた結果を示す。激指は 1 手 10 秒で、簡易逐次プログラムの持ち時間を変えて実験した。2 倍程度時間をかけると激指と同程度の強さになることが分かる。したがって、並列化する前の逐次プログラムが、激指と同じ強さになるためには、少なくとも 6 倍程度の時間をかけないといけないことが分かった。

7.6 結論

並列 $\alpha\beta$ 探索では、探索する必要のないノードの探索を避けるために、浅い探索の結果を用いて最小探索木の予測を動的に修正するべきだと考えられる。しかし、この予測の修正の効果は、100 コアを越える大規模環境では評価されていなかった。本章では、この予測の修正を行う並列探索プログラムを実装し、これが性能にどの程度影響を与えているのかについて評価を行った。予測の修正を行っても速度向上は 1536 コアで約 250 倍であり、この主な原因である訪問ノード数の増加が発生する要因について詳しく調査を行った。本章で実装したワーカ間でのトランスポジションテーブルの共有手法が完全にうまくいけば、訪問ノード数を 60%程度に削減できる可能性があるが、大規模計算環境ではこの共有は難しいことが分かった。別の大きな要因としては、他のタスクの結果が使えないことで、逐次より広い探索窓を用いてタスクを実行してしまっていることがあることが分かった。この要因は、コア数が多いほど、影響が大きくなることを確かめた。

また、激指と対局して勝率を調べたところ、512 プロセスの本研究の並列プログラムが同程度の強さは、8 スレッド並列の激指と同程度であるにとどまった。主な原因としては、実現確率探索を用いたときの再探索を行っていないなど、並列化する前のプログラムがそもそも弱いという要因がまず挙げられる。さらに分枝数が局面ごとにばらつくため、タスクの大きさも大きくばらつくことによる負荷分散の難しいことや、ワーカに激指を用いることで、フルウィンドウで探索したときの訪問ノード数の増加が著しくなるといったことも問題であることが分かった。

今後の課題としては、広い探索窓を用いてタスクを実行しているのが問題だと分かった。特に、ワーカに激指を用いると、広い探索窓では著しく訪問ノード数が増えることが分かった。より有望な部分木の優先度を高くすることで、探索窓を早く狭めるための計算に多く資源を投入するといった方向性が考えられる。しかし、こうすることで、他の部分木の探索を後回しするため、最小探索木の予測が遅れる可能性がある。

また、強いプログラムを作るという観点からは、並列プログラムのマスタに、実現確率探索における有望な手の深い探索や、前向き枝刈りなどの手法を導入することも今後の課題である。また、実際の将棋プログラムでは、負荷分散が難しくなっていると考えられるため、タスクを細かく動的に分割するような負荷分散手法も必要である。

る。この関数を呼ぶ時間による勝率への影響をなくすため、表 7.13 では、浅い探索で最善だった子ノードの探索の前に、指し手生成の関数を呼ぶことで条件を揃えている。

表 7.3: 将棋のゲーム木を用いた結果 ($b = 5$, $d = 24$)

# of processes	d'	method	search time [seconds]	speedup	# of leaves [$\times 10^6$]	start-up idle time [seconds]	other idle time [seconds]
1		sequential	30767.09	1	165		
64	10	w/ updates	1626.35	18.92	434	0.48	29.85
		w/ share	1531.33	20.09	406	0.46	24.06
	12	w/ updates	1413.62	21.76	389	2.15	5.95
		w/ share	1454.47	21.15	386	2.19	6.30
512	10	w/ updates	271.54	113.31	670	0.56	35.96
	12	w/ updates	218.27	140.96	610	2.16	12.67
		w/o updates	477.92	64.38	1168	2.08	69.40
1536	10	w/ updates	184.88	166.42	842	0.97	77.37
	12	w/ updates	123.79	248.54	860	2.46	19.01
		w/o updates	313.85	98.03	1547	2.35	119.07
		w/ share	121.19	253.88	846	2.34	18.20

表 7.4: 将棋のゲーム木を用いた結果 ($b = 20$, $d = 12$)

# of processes	d'	method	search time [seconds]	speedup	# of leaves [$\times 10^6$]	start-up idle time [seconds]	other idle time [seconds]
1		sequential	4020.24	1	41		
64	4	w/ updates	223.54	17.98	116	0.02	20.30
		w/ share	214.96	18.70	111	0.02	19.00
	6	w/ updates	180.63	22.26	88	0.21	1.42
		w/ share	183.81	21.87	88	0.23	1.53
512	4	w/ updates	75.70	53.10	187	0.12	44.51
	6	w/ updates	30.32	132.58	162	0.28	2.89
		w/o updates	46.03	87.34	230	0.29	7.54
1536	4	w/ updates	82.53	48.71	185	2.02	69.06
	6	w/ updates	21.15	190.09	262	0.63	5.30
		w/o updates	34.59	116.21	334	0.61	14.12
		w/ share	20.64	194.75	256	0.64	4.95

表 7.5: 将棋のゲーム木を用いた結果 ($b = 20$, $d = 14$)

# of processes	d'	method	search time [seconds]	speedup	# of leaves [$\times 10^6$]	start-up idle time [seconds]	other idle time [seconds]
1		sequential	48773.45	1	457		
64	4	w/ updates	2683.06	18.18	1248	0.02	313.07
		w/ share	2518.67	19.36	1181	0.02	247.80
	6	w/ updates	2162.77	22.55	986	0.21	19.43
		w/ share	2169.77	22.48	970	0.22	20.04
512	4	w/ updates	954.84	51.08	2075	0.12	585.43
	6	w/ updates	322.53	151.22	1511	0.31	36.30
		w/o updates	654.27	74.55	2942	0.29	125.49
1536	4	w/ updates	1020.74	47.78	1963	5.55	881.65
	6	w/ updates	205.26	237.62	2324	0.62	58.87
		w/o updates	535.17	91.14	3980	0.63	278.62
		w/ share	184.49	264.36	2133	0.66	49.92

表 7.6: トランスポジションテーブル共有による探索時間の短縮

# of proc.	b	d	d'	ratio of search time
64	5	24	10	0.942
			12	1.029
	20	12	4	0.961
			6	1.018
		14	4	0.939
			6	1.003
1536	5	24	12	0.979
	20	12	6	0.976
	20	14	6	0.899

表 7.7: トランスポジションテーブルを共有しないことと、探索窓にフルウィンドウを用いることによる訪問リーフノード数の増加率の値と、問題サイズが大きいときの予測（分枝数 5）

ワーカの探索深さ 12				
b	d	table (normal)	table (full)	full
5	12	1.000	1.000	1.000
5	14	1.418	1.321	2.490
5	16	1.689	1.493	3.266
5	18	1.875	1.602	3.932
5	20	1.997	1.661	4.592
5	24	2.131?	1.713?	5.149?

表 7.8: トランスポジションテーブルを共有しないことと、探索窓にフルウィンドウを用いることによる訪問リーフノード数の増加率の値と、問題サイズが大きいときの予測（分枝数 20）

ワーカの探索深さ 6				
b	d	table (normal)	table (full)	full
20	6	1.000	1.000	1.000
20	8	1.297	1.196	3.220
20	10	1.566	1.325	4.452
20	12	1.753	1.303	5.670

ワーカの探索深さ 8				
b	d	table (normal)	table (full)	full
20	8	1.000	1.000	1.000
20	10	1.351	1.209	3.358
20	12	1.673	1.270	4.946
20	14	1.968?	1.287?	6.013?

表 7.9: 1,536 コアを用いたときの、各要因による訪問リーフノード数の増加率

b	d	d'	table	windows	tasks	estimated	actual
5	24	12	1.941	2.881	0.967	5.407	5.103
20	12	6	1.584	2.756	1.092	4.767	6.390
20	14	6	1.729	2.758	0.976	4.654	5.085

表 7.10: 64 コアを用いたときの、各要因による訪問リーフノード数の増加率

b	d	table	windows	tasks	est.	actual
5	24	2.099	1.314	0.893	2.463	2.358
20	12	1.738	1.156	0.878	1.764	2.146
20	14	1.894	1.548	0.829	2.431	2.158

表 7.11: 512 コアを用いたときの、各要因による訪問リーフノード数の増加率

b	d	table	windows	tasks	est.	actual
5	24	2.022	2.077	0.939	3.944	3.697
20	12	1.689	1.661	0.944	2.648	3.951
20	14	1.746	2.635	0.976	4.490	3.306

表 7.12: 分散並列プログラムの、8 スレッド並列の激指に対する勝率

プロセス数	勝ち	負け	引き分け	勝率
128	39	52	9	42.9%
512	65	60	15	52.0%
1536	50	43	7	53.8%

表 7.13: 実現確率探索で有望な手の深い探索を行わない簡易逐次プログラム（1 手 x 秒）の、深い探索を行う簡易逐次プログラム（1 手 10 秒）に対する勝率

x	勝ち	負け	引き分け	勝率
10	24	102	2	19.0%
20	47	74	7	38.8%
30	64	61	3	51.2%

表 7.14: 実現確率探索で有望な手の深い探索を行う簡易逐次プログラム（1 手 x 秒）の、逐次の激指（1 手 10 秒）に対する勝率

x	勝ち	負け	引き分け	勝率
10	35	84	9	29.4%
20	62	53	13	53.9%

第8章　おわりに

8.1　本研究のまとめ

本研究では、強いコンピュータ将棋プレイヤを作成するために重要な、指し手を選択するための探索と評価関数のパラメータ調整という異なる二つの処理の性能の改善を目的として、大規模計算環境をこれらに活用する手法をそれぞれ提案した。

コンピュータゲームプレイヤの評価関数のパラメータ調整では、訓練データの量を増やすために、コンピュータプレイヤ自身が時間をかけて探索して選んだ指し手を訓練データに追加することを提案した。探索木のリーフノードの局面を訓練局面として追加することで、コンピュータプレイヤの強さを向上させることができることを示した。また、オンライン学習の並列化手法を、評価関数のパラメータ調整に適用することを提案した。64 ノードを用いて並列化したところ、学習時間を 1 ノードを用いたときの 1/11.6 に短縮できることを示した。

コンピュータゲームプレイヤの探索に関しては、$\alpha\beta$ アルゴリズムを大規模計算環境上で効率的に並列化することを目的とした。実行が必要だとは予測されないタスクの投機的実行を効果的に制御するために、実行が必要となる可能性に着目したタスクの優先度付けを提案した。将棋の実探索では提案手法の有効性を示すことができなかったが、全てのタスクにかかる時間が同じだとみなした並列探索のシミュレーションによる評価を通して、提案手法が有効である可能性を示すことができた。

不要な部分木を探索してしまうことをできるだけ避けるために、探索が必要だと予測される部分木の予測を、探索途中に得られる浅い探索の結果も用いて動的に修正する手法を実装した。1,536 コアを用いた評価により、予測の修正により、問題サイズが大きい場合には、探索時間と訪問ノード数をおよそ半分にできることを示し、性能の改善に重要であることが分かった。また、このときの並列探索による性能向上は、予測精度が高い人工木を用いた場合で 650 倍以上、予測精度がより低い将棋の探索の場合で約 250 倍であった。さらに、並列探索の速度向上の妨げとなっている原因を、いくつかの要因に分けることにより、得られた実験結果を説明するためのモデルを構築し、そこから、大規模計算環境では、枝刈りを行うために必要な情報を得る前にタスクの実行を終了してしまっていることが多いことが大きな問題となっていることが分かった。

8.2　今後の展望

本研究では、将棋の評価関数のパラメータ調整の精度の向上のために、訓練データを増やすという点と、学習時間を短縮するという点の、二つの観点からの提案を行った。これらの提案手法は、精度のよい評価関数を実現するために、大規模計算環境を活用するための手法の一例ではあるが、そ

の有効性を示したことにより、今後の評価関数の自動調整の展望を広げた。例えば、さらに質の高い訓練データを生成するためには、局面に対して、訓練対象としての適切性についてなんらかの指標を設け、指し手を生成して訓練対象とする局面を絞り込むといった方法が考えられる。また、パラメータ調整にかかる時間を短縮できたことにより、訓練データを増やす方向以外に、時間はかかるが高い精度が得られる可能性のある学習アルゴリズムを試すことができるようになるため、より精度の良い評価関数が実現可能になると考えられる。これは、コンピュータゲームプレイヤに限らず、他のアプリケーションについても当てはまる。つまり、学習時にコストの高い計算を行う学習手法が選択できるようになる可能性を示したことになり、そのようなアプリケーションの性能の向上にもつながると考えられる。

並列探索では、部分木の探索が必要となる可能性に着目した並列探索手法を提案した。部分木の探索の必要性が分からないのは $\alpha\beta$ 探索に限らず、探索問題において共通の性質である。したがって、他の探索アルゴリズムでも、本研究での提案手法を適用できる可能性がある。本研究で提案した手法は、探索問題に対する知識を用いて探索が必要となる可能性を見積もっているため、同様に探索問題に対する知識を用いている探索アルゴリズムには、特に適用できる可能性が高い。例えば、A* アルゴリズムは問題に対する知識を用いたヒューリスティック関数を用いて探索を進めるなど、$\alpha\beta$ 探索に近い性質を持っており、両者の探索効率の改善の手法にも、共通部分が多い [92] ことを考慮すると、本研究の提案が有効である可能性が特に高い。

また、探索問題では情報の更新を即座に行うことが、訪問ノード数と探索時間を削減するために重要であることを示した。また、実験結果を分析することで、大規模計算環境では、タスクの実行中に枝刈りを行うための情報が得られにくくなることが大きな問題になることが分かった。このような知見は、大規模計算環境で探索のような問題を、簡単に記述でき高性能を実現することを目的とした並列処理系の研究にも活用できると考えられる。現状、並列処理系の研究は、タスクの生成にかかる時間など、並列化に伴うオーバーヘッドを削減することが目的である研究が多く、探索問題の性質に着目して訪問ノード数を減らすような探索制御を行うためのものは少ない。大規模計算環境を用いた様々な並列探索アルゴリズムの研究が進み、枝刈りや探索順序の制御において、探索アルゴリズムに依存しない共通の性質が存在することが分かれば、そのような性質に着目した並列処理系の研究も進むと考えられる。

参考文献

[1] TOP500, http://www.top500.org/.

[2] AmazonEC2, http://aws.amazon.com/ec2/.

[3] Windows Azure, http://www.microsoft.com/japan/windowsazure/.

[4] Graph500, http://www.graph500.org/.

[5] Apache Hadoop, http://hadoop.apache.org/.

[6] Jubatus, http://jubat.us/en/.

[7] 将棋プログラム「激指」,http://www.logos.t.u-tokyo.ac.jp/~gekisashi/.

[8] 将棋プログラム「GPS 将棋」,http://gps.tanaka.ecc.u-tokyo.ac.jp/~gpsshogi/.

[9] A. Agarwal and J. Duchi. Distributed delayed stochastic optimization. In *Advances in Neural Information Processing Systems 24 (NIPS '11)*, pp. 873–881, 2011.

[10] K. Aida, W. Natsume, and Y. Futakata. Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. In *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '03)*, pp. 156–163, 2003.

[11] L. V. Allis, M. van der Meulen, and H. J. van den Herik. Proof-number search. *Artificial Intelligence*, 66(1):91–124, 1994.

[12] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[13] E. B. Baum and W. D. Smith. A bayesian approach to relevance in game playing. *Artificial Intelligence*, 97(1-2):195–242, 1997.

[14] J. Baxter, A. Tridgell, and L. Weaver. Reinforcement learning and chess. In *Machines that learn to play games*, pp. 91–116. Nova Science Publishers, Inc., 2001.

[15] D. F. Beal and M. C. Smith. Temporal difference learning applied to game playing and the results of application to shogi. *Theoretical Computer Science*, 252(1-2):105–119, 2001.

[16] R. Bekkerman, M. Bilenko, and J. Langford. *Scaling up machine learning: Parallel and Distributed Approaches*. Cambridge University Press, 2011.

[17] H. Berliner. The B* tree search algorithm: A best-first proof procedure. *Artificial Intelligence*, 12(1):23–40, 1979.

[18] H. J. Berliner and C. McConnell. B* probability based search. *Artificial Intelligence*, 86(1):97–156, 1996.

[19] Y. Björnsson and T. A. Marsland. Multi-cut $\alpha\beta$-pruning in game-tree search. *Theoretical Computer Science*, 252(1-2):177–196, 2001.

[20] Y. Björnsson and T. A. Marsland. Learning extension parameters in game-tree search. *Information Sciences*, 154(3-4):95–118, 2003.

[21] L. Bordeaux, Y. Hamadi, and H. Samulowitz. Experiments with massively parallel constraint solving. In *21st International Joint Conference on Artificial Intelligence (IJCAI '09)*, pp. 443–448, 2009.

[22] B. Bošković, J. Brest, A. Zamuda, S. Greiner, and V. Žumer. History mechanism supported differential evolution for chess evaluation function tuning. *Soft Computing*, 15(4):667–683, 2010.

[23] M. Bowling, N. A. Risk, N. Bard, D. Billings, N. Burch, J. Davidson, J. Hawkin, R. Holte, M. Johanson, M. Kan, B. Paradis, J. Schaeffer, D. Schnizlein, D. Szafron, K. Waugh, and M. Zinkevich. A demonstration of the polaris poker system. In *8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS '09)*, pp. 1391–1392. International Foundation for Autonomous Agents and Multiagent Systems, 2009.

[24] M. G. Brockington. *Asynchronous Parallel Game-Tree Search*. PhD thesis, University of Alberta, 1998.

[25] M. G. Brockington and J. Schaeffer. APHID: Asynchronous parallel game-tree search. *Journal of Parallel and Distributed Computing*, 60(2):247–273, 2000.

[26] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[27] M. Buro. From simple features to sophisticated evaluation functions. In *1st International Conference on Computers and Games (CG '98)*, Vol. 1558 of *Lecture Notes in Computer Science*, pp. 126–145, 1999.

[28] M. Buro. Improving heuristic mini-max search by supervised learning. *Artificial Intelligence*, 134(1-2):85–99, 2002.

[29] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu. Deep blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

[30] W. Chrabakh and R. Wolski. GridSAT: A chaff-based distributed SAT solver for the grid. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '03)*, p. 37, 2003.

[31] C. E. Clark. The greatest of a finite set of random variables. *Operations Research*, 9(2):145–162, 1961.

[32] M. Collins. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *2002 Conference on Empirical Methods in Natural Language Processing (EMNLP '02)*, pp. 1–8. Association for Computational Linguistics, 2002.

[33] O. David-Tabibi, M. Koppel, and N. S. Netanyahu. Expert-driven genetic algorithms for simulating evaluation functions. *Genetic Programming and Evolvable Machines*, 12(1):5–22, 2011.

[34] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Ng. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25 (NIPS '12)*, pp. 1232–1240, 2012.

[35] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '09)*, p. 53, 2009.

[36] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *29th International Conference on Software Engineering (ICSE '07)*, pp. 3–12, 2007.

[37] R. Feldmann. *Game Tree Search on Massively Parallel Systems*. PhD thesis, University of Paderborn, 1993.

[38] D. B. Fogel, T. J. Hays, S. L. Hahn, and J. Quon. A self-learning evolutionary chess program. *Proceedings of the IEEE*, 92(12):1947–1954, 2004.

[39] H. Fujitani, K. Shinoda, T. Yamashita, and T. Kodama. High performance computing for drug development on K computer. In *24th IUPAP Conference on Computational Physics (IUPAP-CCP '12)*, Vol. 454 of *Journal of Physics: Conference Series*. IOP Publishing, 2013.

[40] J. Fürnkranz. Machine learning in games: A survey. In *Machines that learn to play games*, pp. 11–59. Nova Science Publishers, Inc., 2001.

[41] K. Gimpel, D. Das, and N. A. Smith. Distributed asynchronous online learning for natural language processing. In *Fourteenth Conference on Computational Natural Language Learning (CoNLL '10)*, pp. 213–222, 2010.

[42] T. Graf, U. Lorenz, M. Platzner, and L. Schaefers. Parallel monte-carlo tree search for HPC systems. In *Euro-Par 2011 Parallel Processing*, Vol. 6853 of *Lecture Notes in Computer Science*, pp. 365–376. Springer, 2011.

[43] K. B. Hall, S. Gilpin, and G. Mann. MapReduce/Bigtable for distributed optimization. In *Neural Information Processing Systems: Workshop on Learning on Cores, Clusters, and Clouds*, 2010.

[44] R. Halstead. Implementation of multilisp: Lisp on a multiprocessor. In *1984 ACM Symposium on LISP and Functional Programming*, pp. 9–17, 1984.

[45] Y. Hamadi, S. Jabbour, and L. Sais. Control-based clause sharing in parallel SAT solving. In *21st International Joint Conference on Artifical Intelligence (IJCAI '09)*, pp. 499–504, 2009.

[46] K. Himstedt, U. Lorenz, and D. P. F. Moller. A twofold distributed game-tree search approach using interconnected clusters. In *Euro-Par 2008 Parallel Processing*, Vol. 5168 of *Lecture Notes in Computer Science*, pp. 587–598. Springer, 2008.

[47] K. Hoki and T. Kaneko. The global landscape of objective functions for the optimization of shogi piece values with a game-tree search. In *13th International Conference on Advances in Computer Games (ACG '11)*, Vol. 7168 of *Lecture Notes in Computer Science*, pp. 184–195, 2012.

[48] K. Hoki and M. Muramatsu. Efficiency of three forward-pruning techniques in shogi: Futility pruning, null-move pruning, and late move reduction (LMR). *Entertainment Computing*, 3(3):51–57, 2012.

[49] R. M. Hyatt. The dynamic tree-splitting parallel search algorithm. *ICGA Journal*, 20(1):3–19, 1997.

[50] T. Ishiyama, K. Nitadori, and J. Makino. 4.45 Pflops astrophysical N-body simulation on K computer–the gravitational trillion-body problem. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12)*, pp. 1–0, 2012.

[51] V. Janjic and K. Hammond. Granularity-aware work-stealing for computationally-uniform grids. In *10th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '10)*, pp. 123–134, 2010.

[52] P. Jyothi, L. Johnson, C. Chelba, and B. Strope. Large-scale discriminative language model reranking for voice-search. In *NAAC LHLT 2012 Workshop: Will We Ever Really Replace the N-gram Model? On the Future of Language Modeling for HLT*, pp. 41–49, 2012.

[53] T. Kaneko. Parallel depth first proof number search. In *24th AAAI Conference on Artificial Intelligence (AAAI '10)*, pp. 95–100, 2010.

[54] T. Kaneko and K. Hoki. Analysis of evaluation-function learning by comparison of sibling nodes. In *13th International Conference on Advances in Computer Games (ACG '11)*, Vol. 7168 of *Lecture Notes in Computer Science*, pp. 158–169, 2012.

[55] A. Kishimoto. Transposition table driven scheduling for two-player games. Master's thesis, University of Alberta, 2002.

[56] A. Kishimoto, A. Fukunaga, and A. Botea. Scalable, parallel best-first search for optimal sequential planning. In *19th International Conference on Automated Planning and Scheduling (ICAPS '09)*, pp. 201–208, 2009.

[57] A. Kishimoto, A. Fukunaga, and A. Botea. Evaluation of a simple, scalable, parallel best-first search strategy. *Artificial Intelligence*, 195:222–248, 2013.

[58] A. Kishimoto and J. Schaeffer. Distributed game-tree search using transposition table driven work scheduling. In *International Conference on Parallel Processing (ICPP '02)*, pp. 323–330, 2002.

[59] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[60] Y. Kobayashi, A. Kishimoto, and O. Watanabe. Evaluations of hash distributed A* in optimal sequence alignment. In *22nd International Joint Conference on Artificial Intelligence (IJCAI '11)*, pp. 584–590, 2011.

[61] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *17th European Conference on Machine Learning (ECML '06)*, Vol. 4212 of *Lecture Notes in Computer Science*, pp. 282–293, 2006.

[62] R. E. Korf and D. M. Chickering. Best-first minimax search. *Artificial Intelligence*, 84(1–2):299–337, 1996.

[63] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '09)*, pp. 65–76. IEEE, 2009.

[64] R. Kumar and E. G. Mercer. Load balancing parallel explicit state model checking. *Electronic Notes in Theoretical Computer Science*, 128(3):19–34, 2005.

[65] B. C. Kuszmaul. The StarTech massively parallel chess program. *ICGA Journal*, 18(1):3–19, 1995.

[66] K.-F. Lee and S. Mahajan. A pattern classification approach to evaluation function learning. *Artificial Intelligence*, 36(1):1–25, 1988.

[67] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of VLDB Endowment*, 5(8):716–727, 2012.

[68] J. Mandziuk. *Knowledge-Free and Learning-Based Methods in Intelligent Game Playing*. Springer, 2010.

[69] G. Mann, R. McDonald, M. Mohri, N. Silberman, and D. Walker. Efficient large-scale distributed training of conditional maximum entropy models. In *Advances in Neural Information Processing Systems 22 (NIPS '09)*, pp. 1231–1239, 2009.

[70] V. Manohararajah. Parallel alpha-beta search on shared memory multiprocessors. Master's thesis, University of Toronto, 2001.

[71] T. A. Marsland. Relative efficiency of alpha-beta implementations. In *8th International Joint Conference on Artificial Intelligence (IJCAI '83)*, Vol. 2, pp. 763–766, 1983.

[72] T. A. Marsland and M. Campbell. Parallel search of strongly ordered game trees. *ACM Computing Surveys*, 14(4):533–551, 1982.

[73] T. A. Marsland and F. Popowich. Parallel game-tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-7(4):442–452, 1985.

[74] R. Martins, V. Manquinho, and I. Lynce. An overview of parallel SAT solving. *Constraints*, 17(3):304–347, 2012.

[75] D. A. McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence*, 35(3):287–310, 1988.

[76] D. A. McAllester and D. Yuret. Alpha-beta-conspiracy search. *ICGA Journal*, 25(1):16–35, 2002.

[77] R. McDonald, K. Hall, and G. Mann. Distributed training strategies for the structured perceptron. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL HLT '10)*, pp. 456–464, 2010.

[78] I. Melatti, R. Palmer, G. Sawaya, Y. Yang, R. M. Kirby, and G. Gopalakrishnan. Parallel and distributed model checking in Eddy. *International Journal on Software Tools for Technology Transfer*, 11(1):13–25, 2009.

[79] H. Miyazaki, Y. Kusano, H. Okano, T. Nakada, K. Seki, T. Shimizu, N. Shinjo, F. Shoji, A. Uno, and M. Kurokawa. K computer: 8.162 PetaFLOPS massively parallel scalar supercomputer built with over 548k cores. In *2012 IEEE International Solid-State Circuits Conference (ISSCC '12)*, pp. 192–194. IEEE, 2012.

[80] A. Nagai and H. Imai. Proof for the equivalence between some best-first algorithms and depth-first algorithms for AND/OR trees. *IEICE Transactions on Information and Systems*, E85-D(10):1645–1653, 2002.

[81] A. Narang, A. Srivastava, R. Jain, and R. K. Shyamasundar. Dynamic distributed scheduling algorithm for state space search. In *Euro-Par 2012 Parallel Processing*, Vol. 7484 of *Lecture Notes in Computer Science*, pp. 141–154. Springer, 2012.

[82] M. Newborn. Unsynchronized iteratively deepening parallel alpha-beta search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(5):687–694, 1988.

[83] T. Obata, T. Sugiyama, K. Hoki, and T. Ito. Consultation algorithm for computer shogi: move decisions by majority. In *7th International Conference on Computers and Games (CG '10)*, Vol. 6515 of *Lecture Notes in Computer Science*, pp. 156–165. Springer, 2010.

[84] K. Ohmura and K. Ueda. c-sat: A parallel SAT solver for clusters. In *12th International Conference on Theory and Applications of Satisfiability Testing (SAT '09)*, Vol. 5584 of *Lecture Notes in Computer Science*, pp. 524–537. Springer, 2009.

[85] P. S. Pacheco. *Parallel programming with MPI.* Morgan Kaufmann, 1996.

[86] A. Plaat. *Research Re:Search & Re-search.* PhD thesis, Erasmus University Rotterdam, 1996.

[87] A. Reinefeld. An improvement to the scout tree-search algorithm. *ICCA Journal*, 6(4):4–14, 1983.

[88] M. O. Riedl and A. Zook. AI for game production. In *IEEE 2013 Conference on Computational Intelligence in Games (CIG '13)*, pp. 1–8. IEEE, 2013.

[89] J. Romein, A. Plaat, H. Bal, and J. Schaeffer. Transposition table driven work scheduling in distributed search. In *16th National Conference on Artificial Intelligence (AAAI '99)*, pp. 725–731, 1999.

[90] Y. Sato, M. Miwa, S. Takeuchi, and D. Takahashi. Optimizing objective function parameters for strength in computer game-playing. In *27th AAAI Conference on Artificial Intelligence (AAAI '13)*, 2013.

[91] J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212, 1989.

[92] J. Schaeffer, A. Plaat, and A. Junghanns. Unifying single-agent and two-player search. *Information Sciences*, 135(3–4):151–175, 2001.

[93] Y. Shoham and S. Toledo. Parallel randomized best-first minimax search. *Artificial Intelligence*, 137(1–2):165–196, 2002.

[94] A. B. Sinha and L. V. Kalé. A load balancing strategy for prioritized execution of tasks. In *7th International Parallel Processing Symposium (IPPS '93)*, pp. 230–237. IEEE, 1993.

[95] Y. Soejima, A. Kishimoto, and O. Watanabe. Evaluating root parallelization in go. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):278–287, 2010.

[96] J. R. Steenhuisen. Transposition-driven scheduling in parallel two-player state-space search. Master's thesis, Delft University of Technology, 2005.

[97] T. Sugiyama, T. Obata, K. Hoki, and T. Ito. Optimistic selection rule better than majority voting system. In *7th International Conference on Computers and Games (CG '10)*, Vol. 6515 of *Lecture Notes in Computer Science*, pp. 166–175. Springer, 2011.

[98] Y. Sun, G. Zheng, P. Jetley, and L. Kalé. ParSSSE: An adaptive parallel state space search engine. *Parallel Processing Letters*, 21(3):319–338, 2011.

[99] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction.* MIT Press, 1998.

[100] G. Tesauro. Comparison training of chess evaluation functions. In *Machines that learn to play games*, pp. 117–130. 2001.

[101] G. Tesauro. Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134(1-2):181–199, 2002.

[102] G. Tesauro, V. Rajan, and R. Segal. Bayesian inference in monte-carlo tree search. In *26th Conference on Uncertainty in Artificial Intelligence (UAI '10)*, 2010.

[103] M. Thielscher. General game playing in AI research and education. In *34th Annual German Conference on Advances in Artificial Intelligence (KI '11)*, Vol. 7006 of *Lecture Notes in Computer Science*, pp. 26–37. Springer, 2011.

[104] Y. Tsuruoka, D. Yokoyama, and T. Chikayama. Game-tree search algorithm based on realization probability. *ICGA Journal*, 25(3):146–153, 2002.

[105] A. Ura, D. Yokoyama, and T. Chikayama. Two-level task scheduling for parallel game tree search based on necessity. *Journal of Information Processing*, 21(1):17–25, 2013.

[106] R. Valenzano, N. Sturtevant, J. Schaeffer, K. Buro, and A. Kishimoto. Simultaneously searching with multiple settings: An alternative to parameter tuning for suboptimal single-agent search algorithms. In *3rd Annual Symposium on Combinatorial Search (SoCS '10)*, 2010.

[107] B. van der Spoel. Algorithms for selective search. 2007.

[108] E. Vázquez-Fernández, C. A. Coello Coello, and F. D. Sagols Troncoso. An evolutionary algorithm coupled with the Hooke-Jeeves algorithm for tuning a chess evaluation function. In *2012 IEEE Congress on Evolutionary Computation (IEEE CEC '12)*, pp. 1–8, 2012.

[109] J. Veness, D. Silver, W. Uther, and A. Blair. Bootstrapping from game tree search. *Advances in Neural Information Processing Systems 22 (NIPS '09)*, pp. 1937–1945, 2009.

[110] J.-C. Weill. The ABDADA distributed minimax search algorithm. In *1996 ACM 24th Annual Conference on Computer Science (CSC '96)*, pp. 131–138, 1996.

[111] J.-C. Weill. The ABDADA distributed minimax-search algorithm. *ICCA Journal*, 19(1):3–16, 1996.

[112] B. Wilson, A. Parker, and D. Nau. Error minimizing minimax: Avoiding search pathology in game trees. In *2nd International Symposium on Combinatorial Search (SoCS '09)*, 2009.

[113] Y. Xu. *Scalable Algorithms for Parallel Tree Search.* PhD thesis, Lehigh University, 2007.

[114] M. Yokokawa, F. Shoji, A. Uno, M. Kurokawa, and T. Watanabe. The K computer: Japanese next-generation supercomputer development project. In *International Symposium on Low Power Electronics and Design (ISLPED '11)*, pp. 371–372, 2011.

[115] K. Yoshizoe, A. Kishimoto, T. Kaneko, H. Yoshimoto, and Y. Ishikawa. Scalable distributed monte-carlo tree search. In *4th Annual Symposium on Combinatorial Search (SoCS '11)*, 2011.

[116] G.-X. Yuan, C.-H. Ho, and C.-J. Lin. Recent advances of large-scale linear classification. *Proceedings of the IEEE*, 100(9):2584–2603, 2012.

[117] K. Zhao and L. Huang. Minibatch and parallelization for online large margin structured learning. In *The 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL HLT '13)*, 2013.

[118] 浦晃, 横山大作, 近山隆. 投機を用いた並列ゲーム木探索の効率化. 第 15 回ゲームプログラミングワークショップ, pp. 134–141, 2010.

[119] 横山大作. 「激指」におけるゲーム木探索並列化手法. 人工知能学会誌, 26(6):648–654, 2011.

[120] 金子知適. コンピュータ将棋の評価関数と棋譜を教師とした機械学習. 人工知能学会誌, 27(1):75–82, jan 2012.

[121] 金子知適, 田中哲朗. 最善手の予測に基づくゲーム木探索の分散並列実行. 情報処理学会論文誌, 53(11):2517–2524, 2012.

[122] 高見明秀, 鍋島英知, 岩沼宏治. 分散並列型 SAT ソルバにおける探索空間の分割手法の提案. 電子情報通信学会技術研究報告. ソフトウェアサイエンス, SS2008(52):23–28, 2009.

[123] 小谷善行, 岸本章宏, 柴原一友, 鈴木豪. ゲーム計算メカニズム. コロナ社, 2010.

[124] 小幡拓弥, 杉山卓弥, 保木邦仁, 伊藤毅志. 将棋における合議アルゴリズム：既存プログラムを組み合わせて強いプレイヤを作れるか？ 第 14 回ゲームプログラミングワークショップ, pp. 51–58, 2009.

[125] 田中哲朗, 金子知適. 将棋プログラムの大規模並列実行. 情報処理学会研究報告, 2010-GI-24(2):1–8, 2010.

[126] 平石拓, 八杉昌宏, 馬谷誠二. 動的負荷分散フレームワーク Tascell の広域分散およびメニーコア環境における評価. 先進的計算基盤システムシンポジウム (SACSIS 2012), pp. 55–63, 2011.

[127] 保木邦仁. 局面評価の学習を目指した探索結果の最適制御. ゲームプログラミングワークショップ, pp. 78–83, 2006.

[128] 保木邦仁, 伊藤毅志. 疑似乱数を利用した df-pn 探索の簡易分散並列計算. 第 16 回ゲームプログラミングワークショップ, pp. 116–119, 2011.

本研究に関連する発表文献

査読付き論文誌

1. Akria Ura, Daisaku Yokoyama, Takashi Chikayama. Two-level Task Scheduling for Parallel Game Tree Search Based on Necessity. *Journal of Information Processing*, Vol. 21, No. 1, pp. 17–25, 2013

査読付き会議論文

1. 浦晃, 横山大作, 近山隆, 投機を用いた並列ゲーム木探索の効率化. 第 15 回ゲームプログラミングワークショップ, pp. 134–141, 2010.
2. 浦晃, 三輪誠, 横山大作, 田浦健次朗, 近山隆. 探索が必要となる確率を用いた並列 $\alpha\beta$ 探索のスケジューリング. 第 16 回ゲームプログラミングワークショップ, pp. 68–75, 2011.
3. 浦晃, 鶴岡慶雅, 近山隆. 探索窓の予測分布を用いた並列 $\alpha\beta$ 探索. 第 17 回ゲームプログラミングワークショップ, pp. 68–75, 2012.
4. Akira Ura, Makoto Miwa, Yoshimasa Tsuruoka, Takashi Chikayama. Comaprison Training of Shogi Evaluation Functions with Self-Generated Training Positions and Moves. *The 8th International Conference on Computers and Games (CG 2013)*, 2013.

その他の発表文献

査読付き会議論文

1. 林伸也, 浦晃, 三輪誠, 田浦健次朗, 近山隆. 自己対戦棋譜を利用した半教師あり学習による将棋の評価関数の学習. 第 16 回ゲームプログラミングワークショップ, pp. 143–149, 2011
2. 藤田康博, 浦晃, 三輪誠, 鶴岡慶雅, 近山隆. General Game Playing におけるモンテカルロ木探索のシミュレーション方策の学習. 第 17 回ゲームプログラミングワークショップ, pp. 38–45, 2012.
3. 亀甲博貴, 浦晃, 三輪誠, 鶴岡慶雅, 近山隆. 局面情報からの探索信頼性の推定による将棋の ProbCut の性能向上. 第 17 回ゲームプログラミングワークショップ, pp. 92–99, 2012.
4. 古居敬大, 浦晃, 三輪誠, 鶴岡慶雅, 近山隆. 相手の抽象化による多人数ポーカーでの戦略の決定. 第 17 回ゲームプログラミングワークショップ, pp. 211–218, 2012.
5. 水上直紀, 中張遼太郎, 浦晃, 三輪誠, 鶴岡慶雅, 近山隆. 降りるべき局面の認識による 1 人麻雀プレイヤの 4 人麻雀への適用. 第 18 回ゲームプログラミングワークショップ, 2013.
6. 亀甲博貴, 浦晃, 三輪誠, 鶴岡慶雅, 森信介, 近山隆. 将棋解説の自動生成のための局面からの特徴語生成. 第 18 回ゲームプログラミングワークショップ, 2013.
7. 川上裕生, 浦晃, 三輪誠, 鶴岡慶雅, 近山隆. 将棋の評価関数の学習に有用な局面の自動選択. 第 18 回ゲームプログラミングワークショップ, 2013.
8. 中張遼太郎, 水上直紀, 浦晃, 三輪誠, 鶴岡慶雅, 近山隆. LinUCB の 1 人麻雀への適用. 第 18 回ゲームプログラミングワークショップ, 2013.

査読なし論文

1. 藤田康博, 浦晃, 三輪誠, 鶴岡慶雅, 近山隆. GGP における強さとバランスを両立したモンテカルロ木探索の方策の学習”. 第 29 回ゲーム情報学研究会, 2013.

謝辞

本研究を進めるにあたり、多くの方にお世話になりました。

近山隆教授には、指導教員として大変お世話になりました。特に、博士課程 3 年の夏以降は、一緒に議論をさせていただく時間を毎週設けていただき、多くのご助言、ご指導をいただきました。田浦健次朗准教授には、本研究のメインである並列処理に関して、数多くのご助言をいただきました。鶴岡慶雅准教授には、論文の執筆に対するアドバイスを多くいただきました。研究を進めるにあたって困ったときの相談や、激指における細かい質問などに気軽に応じていただきました。横山大作助教には、計算環境を使用するにあたって、いろいろとお世話になりました。また、研究に関する議論にも応じていただきました。三輪誠さんには、研究、特に機械学習に関わる質問や議論に気軽に応じていただきました。また、論文の書き方や発表資料の作り方について、多くのご助言をいただきました。ここに、心より感謝を申し上げます。

近山隆教授、相田仁教授、中山雅哉准教授、田浦健次朗准教授、鶴岡慶雅准教授には、本論文の審査をしていただきました。大変ありがとうございました。

近山・鶴岡研究室、田浦研究室の皆様にもお世話になりました。特に田浦研究室の中島潤君には、修士時代から同じ研究室の同期の学生ということもあって、ちょっとした質問や雑談にのってもらいました。ありがとうございました。

最後に、博士課程に進学することを許可してくれ、研究でつらいときには励ましてくれた家族には大変感謝しています。