修　士　論　文

グラフィックエンジンを用いたゲーム探索の高速化

Speeding Up Game Search with Graphic Engines

指導教員　**近山 隆 教授**

東京大学大学院工学系研究科
電気系工学専攻融合情報学コース

氏　名　**37-086933 田野 文彦**

提　出　日　**平成22年2月9日**

概要

コンピュータゲームプレイヤでは，ゲームの進行をゲーム木と呼ばれる木で表現し，その中で有利な手を求めることを行う．ゲーム木の探索には膨大な計算が必要であり，効率よくゲーム木探索は強いゲームプレイヤをつくるための重要な要件の 1 つである．また，今日グラフィックエンジン（GPU）の性能向上が著しく，画像処理だけでなく CPU で行うような汎用計算も GPU で行うことが試みられている．本研究では囲碁のモンテカルロ探索に GPU を用いることにより，9 路での初期局面からのプレイアウト数毎秒 70,660 回で実現したことを報告する．これはよく知られたオープンソースの囲碁プログラムである GNU Go 3.8 を CPU 上で実行した場合と比べて 8.5 倍の速度であった．

目次

概要 i

第 1 章 序論 1
- 1.1 背景・目的 1
- 1.2 論文の構成 2

第 2 章 GPGPU 3
- 2.1 CUDA 3
- 2.2 CUDA のプログラミングモデル 3
- 2.3 GPU のアーキテクチャ 5
  - 2.3.1 演算器の構成 6
  - 2.3.2 シェアードメモリ 6
  - 2.3.3 グローバルメモリ 6
  - 2.3.4 SIMD 6
- 2.4 GPU の最適化の方法 6
  - 2.4.1 レイテンシの隠蔽 7
  - 2.4.2 ワープ 7
  - 2.4.3 コアレスアクセス 7

第 3 章 関連研究 8
- 3.1 ゲームの種類 8
- 3.2 ゲーム木探索 9
  - 3.2.1 探索手法 9
  - 3.2.2 評価関数 9

3.3 Minimax 型 10
3.3.1 Minimax 法 10
3.3.2 α-β 法 10
3.3.3 反復深化法 12
3.3.4 0.5 手延長アルゴリズム 12
3.3.5 実現確率探索 13
3.4 モンテカルロ型 15
3.4.1 モンテカルロ法 15
3.4.2 多腕バンディット問題 17
3.4.3 UCT 17
3.4.4 パターンを導入した UCT 18
3.5 それぞれの探索の特徴 19
3.6 一般的なコンピュータ囲碁プレイヤ 21
3.6.1 難しさ 21
3.6.2 コンピュータでの探索方法 22
3.6.3 CPU を用いた探索でのデータ構造 22
3.7 FPGA での囲碁の実装 23
3.7.1 TLPG 23
3.7.2 実験 24
3.8 GPU での囲碁の実装 25

第 4 章 提案手法 26
4.1 GPU によるプレイアウトの課題 26
4.2 GPU の ECC の検討 27
4.3 データ構造 27
4.4 局面更新の論理演算 28
4.5 プレイアウトの流れ 30
4.6 プレイアウトの例 30

第 5 章 実験と評価 35
5.1 環境 35
5.2 方法 35

5.3 結果 36

第 6 章 結論 45

6.1 まとめ 45

6.2 今後の課題 45

発表文献 48

謝辞 49

参考文献 50

表目次

3.1 0.5 手延長探索 13
3.2 実現確率探索の勝率（従来手法は常に 10[s/手]） 15
3.3 純粋な UCT とパターンを使った UCT の勝率 20
3.4 Minimax 型とモンテカルロ型 21
3.5 FPGA での囲碁のプレイアウト性能 25

5.1 GNU Go の毎秒のプレイアウト数 37
5.2 1block あたりの局面数と毎秒のプレイアウト数 41
5.3 GPU へ送る block 数と毎秒プレイアウト数 43

図目次

2.1 プログラミングモデル 4
2.2 GPU の処理の流れ 4
2.3 GPU（GT200）の構成 5
2.4 GPU（GT200）の Streaming Multiprocessor(SM) 5

3.1 Minimax 法の例 11
3.2 α-β 法の例 12
3.3 0.5 手延長アルゴリズムの例 14
3.4 実現確率探索 15
3.5 モンテカルロ法 16
3.6 UCT 探索 [6]. 18
3.7 時間が限られた α-β 探索. 19
3.8 UCT のパターンの例：ハネ・パターン 19
3.9 UCT のパターンの例：キリ 1 パターン 20
3.10 囲碁の局面の例 22
3.11 連の情報を付加した囲碁の局面の例 23
3.12 FPGA での連の呼吸点数のカウント 24

4.1 局面データの格納方法 27
4.2 GPU を用いた囲碁のプレイアウトの流れ 32
4.3 生存フラグの更新 1 33
4.4 生存フラグの更新 2 33
4.5 生存フラグの更新 3 33
4.6 生存フラグの更新 4 33

4.7 生存フラグの更新 5 33
4.8 生存フラグの更新 6 33
4.9 生存フラグの更新 7 34
4.10 生存フラグの更新 8 34
4.11 生存フラグの更新 9 34

5.1 合法になるまでランダムに手を打ったときの回数の n 手目の平均 38
5.2 合法になるまでランダムに手を打ったときの回数の n 手目の分散 39
5.3 合法手を打ったときの生存フラグの伝播回数の n 手目の平均 40
5.4 合法手を打ったときの生存フラグの伝播回数の n 手目の分散 40
5.5 1block あたりの局面数と毎秒のプレイアウト数の測定結果 42
5.6 GPU へ送る block 数と毎秒プレイアウト数の測定結果 44

6.1 局面データの格納方法の改善案 47

第 1 章

序論

1.1 背景・目的

人工知能研究の目的とする複雑な条件下での意思決定法のテストベッドとしての有力な手法の 1 つにコンピュータゲームプレイヤの研究がある．実生活における意思決定を支援するような人工知能システムをつくるためのモデルケースとして，ゲームは有用であり，いまだにコンピュータで扱うことが難しいゲームは数多く存在する．

コンピュータは現在，将棋や囲碁といったゲームは人間のプレイヤには勝つことができていない．確率の要素が入っていないゲームでは，すべての組み合わせを計算し尽くせば原理的には解くことができ，あらゆる状況での最善手が求められる．しかし，現在の計算機をもってしても将棋や囲碁などの必勝手順を導くことは不可能である．これは手の数が多く，考えられる組み合わせが爆発的に増えるのが原因である．したがって探索の無駄を省く枝刈り手法や，良さそうな手付近を集中的に探索する手法などが数多く提案されている．このようなアルゴリズムの優劣はゲームの勝敗によって決定することができるため，客観的な評価が可能である．

コンピュータゲームプレイヤでは，ゲームの進行をゲーム木と呼ばれる木で表現し，その中で有利な手を求めることを行う．ゲーム木の探索には膨大な計算が必要であり，効率のよいゲーム木探索は強いゲームプレイヤをつくるための重要な要件の 1 つである．

一方で，コンピュータゲームプレイヤにとって重要な要素に探索の速度がある．これが無限に速ければ全探索できるはずであるし，有限であっても速ければ速いほど同じアルゴリズムを適用したとしてもほとんどの場合有利な展開になることは明らかである．

従来，CPU（Central Processing Unit）の能力は単位時間当たりにコアが何回命令を実

行できるかという動作周波数で測られ，動作周波数の増大に伴って CPU の能力は高くなっていった．しかし，CPU の単一コアの動作周波数を大きくしていくというアプローチは近年では停滞している．それは発熱量などの物理的な制約により決定されるものであり，今後も大きく伸びることは期待できない．

そこで近年では 1 個の CPU の中に複数のコアを封入するマルチコア化がおこなわれている．CPU のマルチコア化が進む中で，CPU 以外の Cell/B.E.（Cell Broadband Engine）や GPU（Graphics Processing Unit）といった複数のコアを備えているプロセッサも，高速に演算できることからさまざまな用途に利用されている．中でも GPU は，パソコンなどでグラフィックを表示する用途にはよく使われ，プロセッサの進歩を表す指標の 1 つであるトランジスタ数の増大が CPU よりも速い．

近年の GPU の性能向上に伴い，GPU を画像処理だけでなく汎用的に使おうとする試みがなされている．このような GPU の利用を GPGPU（General Purpose GPU）と呼ぶ．GPGPU では多数のコアを動作させるため並列的な計算の場合 CPU より高速に計算をすることができる．GPU は並列処理に優れることから，多くの並列性を有するゲーム探索においても有用であると考えられる．そこで本研究では GPU を用い，CPU で計算したときよりも高速なゲームプレイヤを構築することを目的とする．

本研究ではゲームの種類は，ゲームの中でも探索空間が広く，探索が困難であるといわれる囲碁を対象とした．囲碁のモンテカルロ探索に GPU を用い，ランダムにプレイアウトを行った．

1.2　論文の構成

本論文の構成を示す．まず囲碁の一般的な実装，GPGPU について述べ，次にゲーム探索の種類や CPU 以外での囲碁の実装の関連研究を述べる．そして本研究の提案手法，実験と評価，まとめと今後の課題について述べる．

第 2 章

GPGPU

2.1 CUDA

近年，グラフィックスの演算用に開発された GPU のピーク性能は CPU を上回るといわれている．GPU をグラフィックス以外の計算に利用することができれば，計算資源を有効に活用することができる．実際に GPU に汎用的な計算をさせようという試みは存在し，GPGPU と呼ばれている．Cg（C for graphics）[11] などの言語を用いることで，GPU に汎用的な計算をさせることができる．しかし，もともと GPU はグラフィックス用に開発されたものであるため，汎用の計算をグラフィックス用の命令に置き換えるなど作業が煩雑となる．画像処理専用チップである GPU の命令体系は通常の CPU とは大きく異なり，その命令体系で直接プログラミングをするのは，プログラムを開発する側にとって負担の大きいものであり，複雑な汎用の処理を GPU で行うことは困難な作業である．

NVIDIA が発表した CUDA（Compute Unified Device Architecture）[12] は，GPU 上で行うプログラミングを，画像処理を意識せずに書くことを目的とした統合開発環境である．CUDA を用いることにより，行いたい計算を画像処理用の命令に人手で変換するという困難な作業を大幅に軽減することができ，行いたい計算部分の開発に集中することができる．

2.2 CUDA のプログラミングモデル

GPGPU の概要を述べる．GPGPU とは画像処理用のプロセッサである GPU に画像処理以外の汎用的な計算を行わせる技術であり，FFT，銀河の衝突や高分子の挙動の計算な

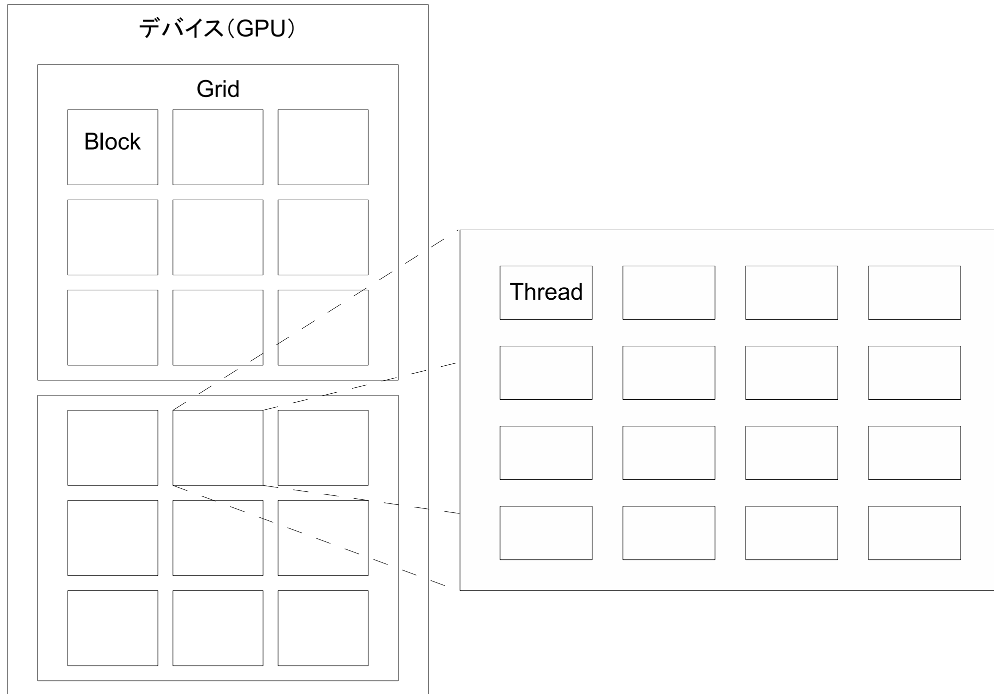

図 2.1　プログラミングモデル

どの科学技術の分野に応用されている [8][9][10]．GPU は多数の処理ユニットを並列に用いることで高速な処理を実現している．これは画像処理は比較的計算負荷が大きい処理であるが，画素単位など，並列的に計算しやすいという特徴があり，このような負荷に対して並列処理が有効なためである．

図 2.2 に示すような順序で GPU へデータを転送し，実行する．

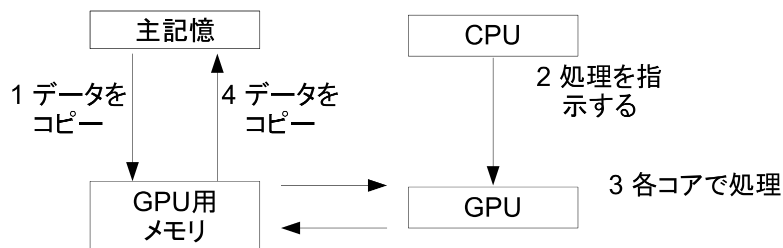

図 2.2　GPU の処理の流れ

2.3　GPU のアーキテクチャ

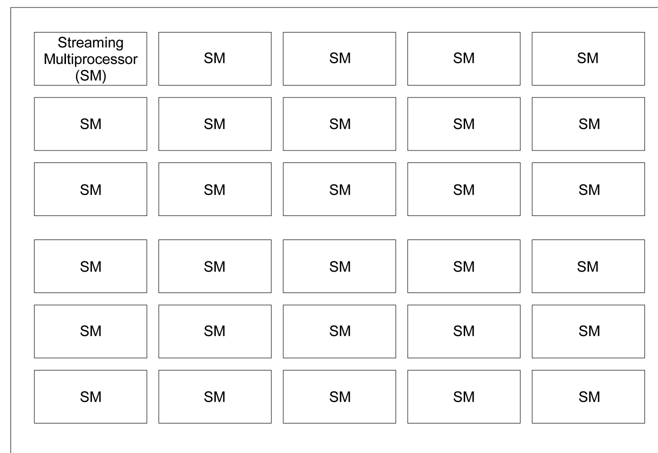

図 2.3　GPU（GT200）の構成

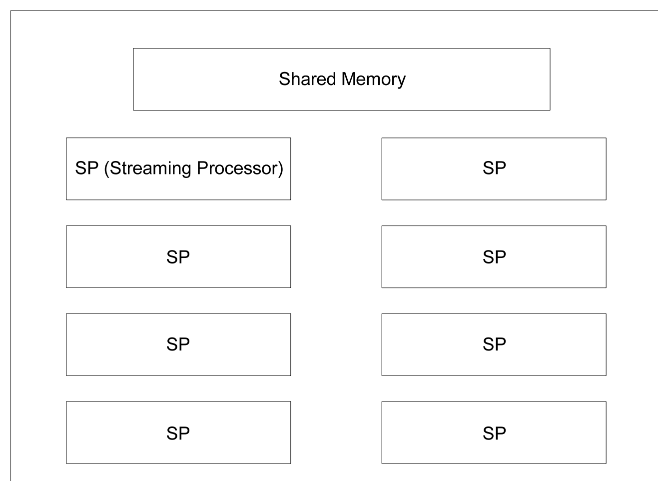

図 2.4　GPU（GT200）の Streaming Multiprocessor(SM)

図 2.3，図 2.4 に GPU とその中の Streaming Multiprocessor(SM) の構成を示す．CPU と比べてコアが多いこと，階層的なメモリアクセスを意識しなければならないことが特徴

としてあげられる．GPU のアーキテクチャを述べる．

2.3.1　演算器の構成

演算器の最小単位をストリーミングプロセッサ（SP）と呼ぶ．SP は一定個数（8 個）ごとにまとめられストリーミングマルチプロセッサ（SM）という集まりで扱われる．

2.3.2　シェアードメモリ

同一 SM 内の SP は同一のシェアードメモリと呼ばれる高速なメモリにアクセスできる．シェアードメモリは高速であるが容量が小さい（現在では 1 個あたり 16KB）．このため大量のデータを扱うプログラムの場合には後に述べるグローバルメモリへのアクセスの最適化が重要となる．

2.3.3　グローバルメモリ

グローバルメモリは低速であるが容量が大きく，すべての SM からアクセスできる．低速であるがコアレスアクセスと呼ばれる方法で最適化することでメモリアクセスの回数を減らすことができる．

2.3.4　SIMD

同一の MP 内の SP はすべて同じ命令を実行する．この点で GPU のアーキテクチャは SIMD（Single Instruction Multiple Data）であるといえる．同一 SM 内の SP の分岐が異なった場合，すべての分岐を通ることになり計算に時間がかかってしまう．したがって条件分岐を多用するようなプログラムの書き方は避けなければならない．

2.4　GPU の最適化の方法

GPU は CPU と異なりコアがそれぞれ完全に独立した命令を実行できるわけではない．各コアでどのように計算がなされるかに注意することで高速に実行することが可能となる．この最適化の方法について述べる．

2.4.1　レイテンシの隠蔽

CPU では通常スレッドは多く起動するとオーバーヘッドが大きくなり性能が低下する．しかし GPU の場合は性能を引き出すために，スレッド数はコア数よりはるかに多い数万，数十万単位で起動する．これは GPU は多くのスレッドを立ち上げることでパイプラインのようにレイテンシが隠せるからである．したがって GPU では数万単位以上の並列性がある問題が適しているといえる．

2.4.2　ワープ

先ほど述べたように SM 内の複数個の SP が SIMD のように同一の命令を実行する．しかし，さらに GPU の仕様として 4 サイクル同一の命令を実行する．つまり全部で 32 スレッドが同じ命令を実行すると考えてよい．したがって 1 個の SP 内の起動スレッド数は 32 の倍数であると性能を引き出しやすいといえる．

2.4.3　コアレスアクセス

大量のデータを扱う場合，シェアードメモリだけでは容量が足りない．そのため容量の大きいグローバルメモリを使う．しかし低速なため頻繁にアクセスすると実行時間が大幅に伸びてしまう．したがって大きいデータを扱うアプリケーションでは，アクセス回数をなるべく少なくし多くのデータにアクセスする必要がある．

GPU ではメモリアクセスの命令は 16 スレッドごとに行われるがこの各スレッドがどのアドレスにアクセスするかによって同じ大きさのデータであってもアクセスに必要な時間が異なる．グローバルメモリにアクセスする際，一定の大きさ（32 バイト，64 バイト，128 バイト）のデータごとに行われる．

コアレスアクセスと呼ばれる方法を使い効率的にアクセスする方法がある．以前の GPU ではコアレスアクセスになるための条件が多かったが，GPU の性能向上に伴って次第に緩和されつつある．

第 3 章

関連研究

3.1　ゲームの種類

ゲームの局面や考えられる手を木構造であらわしたものをゲーム木と呼ぶが，ゲーム木における探索手法を述べる．

最初にゲームの種類やゲーム木について述べる．ゲーム木の探索には大きく Minimax 型とモンテカルロ型の 2 つに分けることができるが，それぞれの具体的な探索手法である α-β 法，0.5 延長アルゴリズム，実現確率探索，UCT などについて述べる．

ゲームといっても，種類はさまざまであり，いくつかの分類方法がある．

プレイヤの人数　プレイヤの人数によって分けることができる．チェスなどは 2 人であるが，ダイヤモンドゲーム，麻雀など 3 人以上がプレーするゲームもある．

完全情報かそうでないか　各プレイヤが自分の番でこれまでの各プレイヤの行動について知ることができるゲームを完全情報ゲームという．

将棋やチェス等のボードゲームでは、各プレーヤーが他のプレーヤーの状況を把握でき，どのような手を差したのかもわかるため完全情報ゲームである．

しかし麻雀やポーカー等のカードゲームの多くでは，各プレーヤーの手牌や手札を他のプレーヤーが見ることができず，他のプレーヤーの状況を知ることができないため不完全情報ゲームという．

確定かそうでないか　偶然の要素が入り込まないゲームを確定ゲームといい，そうでないものを不確定ゲームという．

囲碁、将棋は確定ゲームに分類される．双六のようなサイコロでランダムにコマを

進めるゲームは，不確定ゲームである．

零和かそうでないか　将棋やチェスなどのゲームでは勝利を 1、引き分けを 0、敗北を −1 のように状態に数字をつけて考えることができる．この数字の総和が 0 になるゲームを零和ゲームという．
ルーレットのようなカジノで行うゲームでは、各プレーヤーの利得の合計は必ずしも 0 あるいは一定の数値になるとは言えないため、零和のゲームとはいえない．
将棋やチェスのような二人ゲームで，あるプレーヤーからみた場合の終了時の状態が，勝利，引き分け，敗北となるゲームについてはゲームの利得表の合計は常に 0 となる．

ここでは以後，チェス，将棋，囲碁などの二人零和確定完全情報ゲームを扱うものとする．

3.2　ゲーム木探索

ゲームの局面をノードとし，可能な手を枝とした木構造をゲーム木という．これからの議論ではこのゲーム木を中心に述べることとする．

3.2.1　探索手法

ゲーム木の探索手法にはさまざまなものがある．探索にはコストがかかるから悪そうな手をあまり探索せず，良さそうな手を重点的に探索し，順位をつけ，一番良い手を打つ．したがって，勝敗は効率よく探索をするかどうかに大きく関わってくる．

3.2.2　評価関数

プレイヤはすべての手の組み合わせが勝敗を決定するまで計算することはできない．したがってプレイヤはよい手なのか悪い手なのかを限られた探索で判断し，できるだけ自分が勝ちそうな，良い局面に状況を持っていこうとする．このためある局面がどの程度良いのか，悪いのかを判断する方法が必要である．この判定を評価関数によって求める．

例えばオセロでは，局面の角をとると有利になるため角の評価値を高くしたり，そのすぐ隣は不利であるためマイナスの評価値としたりする．このような関数を用いて駒の点数の合計を局面の点数とし，この点数が高くなるようにプレイヤは競っていく．

このように駒の位置のみの情報を使っても評価関数は作ることができるが，現在どのような駒の配置になっているかなどを取り入れることも重要である．

3.3 Minimax 型

3.3.1 Minimax 法

たとえばプレイヤ 1 とプレイヤ 2 がゲームをしていて，プレイヤ 1 が 3 手先まですべての手が読めたとする．プレイヤ 1 としては 3 手先での自分にとっての評価値が一番高い局面にゲームをもっていきたいと考える．しかしプレイヤ 2 も自分にとって最もよい局面になるような手を指してくる．

したがってプレイヤ 1 が先読みする場合，自分の番では自分にとって最も良い局面を指すとし，相手の番では相手にとって最も良い手，つまり自分にとって最も悪い手を指すと仮定しなければならない．この仮定において先読みする方法を Minimax 法といい，ゲーム探索において基本となる考え方である．図 3.1 に 3 手先までの例を示す．

3.3.2 α-β 法

ここで，$\min(x_1, x_2)$ は x_1 と x_2 のうち最小のものを返し，$\max(x_1, x_2)$ は x_1 と x_2 のうち最大のものを返す関数とする．

たとえば

$$\min(3, \max(5, x)) \tag{3.1}$$

という式を考える．上式の中で，x は未知数とする．未知数があるにもかかわらず，上式の値は 3 と求めることができる．

まず $\max(5, x)$ に注目する．すると，未知数 x を含んでいるが，5 と x のうち大きいほうを返すのであるから，少なくとも 5 以上であることがわかる．なぜなら，x が 5 未満の場合には $\max(5, x) = 5$ となり，x が 5 以上の場合には $\max(5, x) \geq 5$ となるからである．つぎに min 関数の部分では 3 と「少なくとも 5 以上の値」との最小値であるから 3 と求められる．

このように x が求められなかったり，計算困難な複雑な関数であったりしてもその値を計算することなく全体の関数の値を求めることができる．これをゲーム木に使うことで，評価しなくてもよいノードを計算しないことによって計算量を削減することができる．

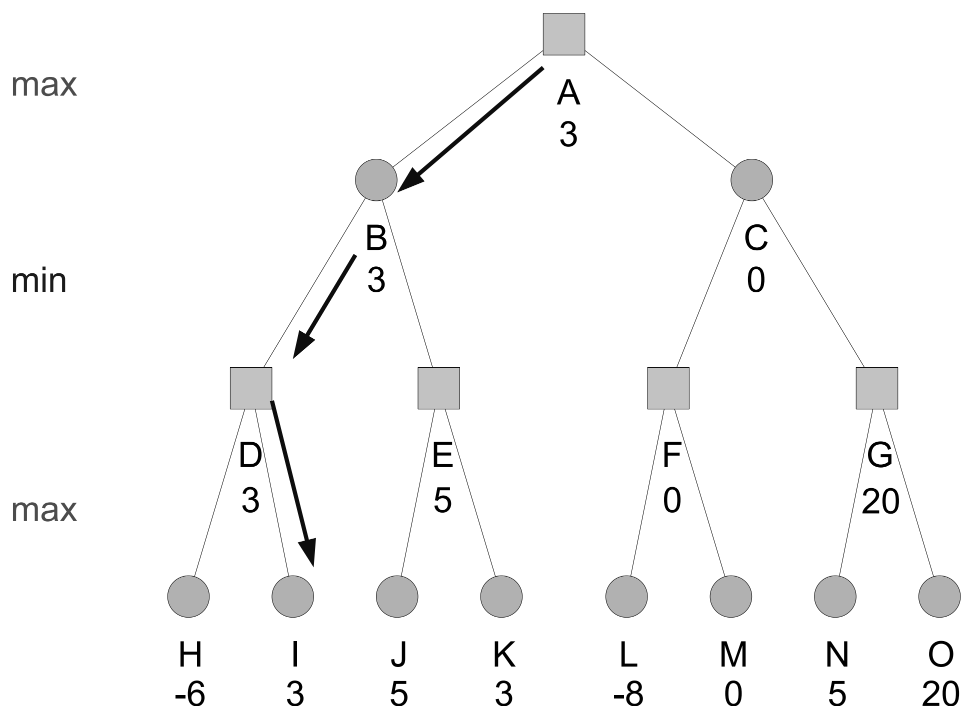

図 3.1　Minimax 法の例

以上の考え方を図 3.1 のゲーム木に適用する．この木に Minimax 法を使うことは

$$\max(\min(\max(H,I),\max(J,K)),\min(\max(L,M),\max(N,O))) \tag{3.2}$$

を求めることと等価である．これを先ほど述べた考え方を使うことで計算量を削減する．一番左のリーフである H からトーナメントのように評価関数の最大値（最小値）を選んでゆく．すると図 3.2 のように枝刈りを行うことができる．

J を評価した時点で E は少なくとも 5 以上であるので，D と E の最小値を選ぶ B は B=D となる．ここで，K を評価せずに B を決定している．F を評価した時点で C は少なくとも 0 以下であるので B と C の最大値を選ぶ A は A=B となる．ここで，N，O，G を評価せずに A を決定している．

つまり，K，N，O，G を評価せずに，Minimax 法とまったく同様の結論を得ることができた．この方法を α-β 法 [1] という．

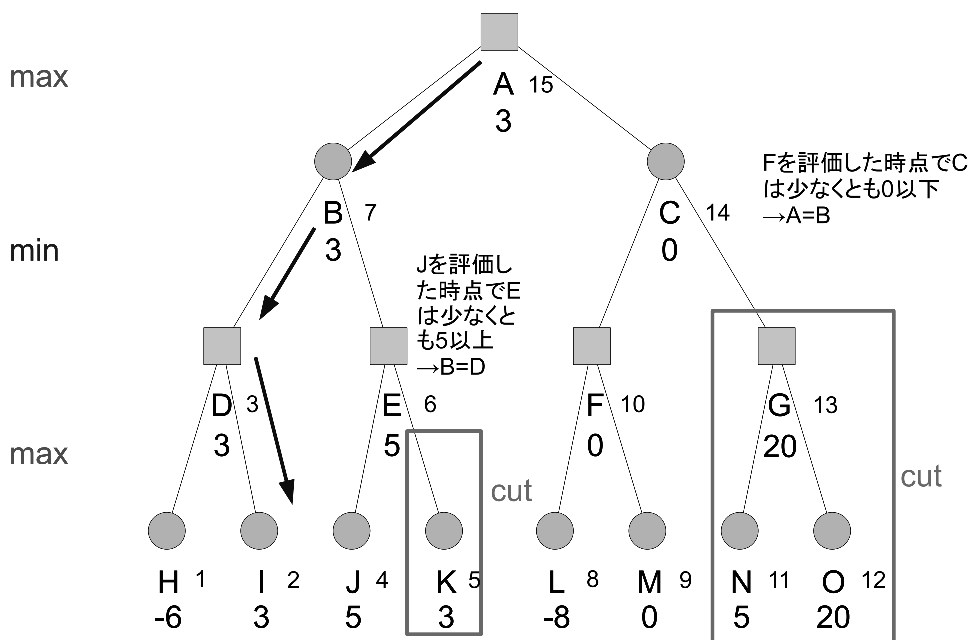

図 3.2　α-β 法の例

3.3.3　反復深化法

通常の Minimax 法などでは，たとえば 3 手先のリーフの評価値から調べていくが，反復深化法 [2] では，最初に 1 手先を調べ，そこで最大の評価だったものから順に 2 手目を探索し，といったようにする．このようにすることによって，時間制限がある場合に，局所的な探索で時間切れになってしまうことを防ぐことができる．

3.3.4　0.5 手延長アルゴリズム

図 3.3 のように，はじめにたとえば 3 手先まで探索すると設定する．これを基本深さという．次に 1 手先を探索し，残りの探索はあと 2 手である．しかし，1 番評価が高かった局面のときだけ，0.5 手延長することができるようにし，あと 2.5 手探索することができる設定とする．最後の 0.5 手は切り捨てとする．

このアルゴリズムの特徴は基本深さ n で探索したとき最大 $2n-1$ の深さまで探索する

可能性を持つが，末端局面の数は通常の 2〜3 倍程度で済むということである．

分岐数 20 で，1 手のみ 0.5 手延長した場合のかかる時間と最大探索深さを表 3.1 に示す [2]．

例えば，各局面の分岐数を 20 とし基本深さ 8 の探索をした場合，最大 15 手先まで読むにもかかわらず探索時間は 2.35 倍にしかならない．

表 3.1　0.5 手延長探索

深さ	時間	最大深さ
2	1.04	3
3	1.14	5
4	1.28	7
5	1.46	9
6	1.70	11
7	2.00	13
8	2.35	15
9	2.79	17
10	3.32	19

このような手法を 0.5 手延長アルゴリズム [2] という．将棋プログラムである YSS が用いている手法である．

このアルゴリズムを使うことによって，評価の低い手はあまり読まずに，評価の高い手を重点的に読むことができる．評価の高い手付近は実際に指す手とする可能性が高く，重点的に探索し手に順位付けをし，よりよい手を探すことが重要である．しかし，評価が低い手付近の手の順位付けするために時間をかけて探索することは特に重要ではない．したがってよりよい手を探すために，良い評価が出ている局面付近を探索するアルゴリズムが重要となってくる．

3.3.5　実現確率探索

将棋プログラムである激指が使用している手法である．人間であればありそうもない手というのはそれ以上読まない．しかしコンピュータではどんなに人間が指しそうにないて

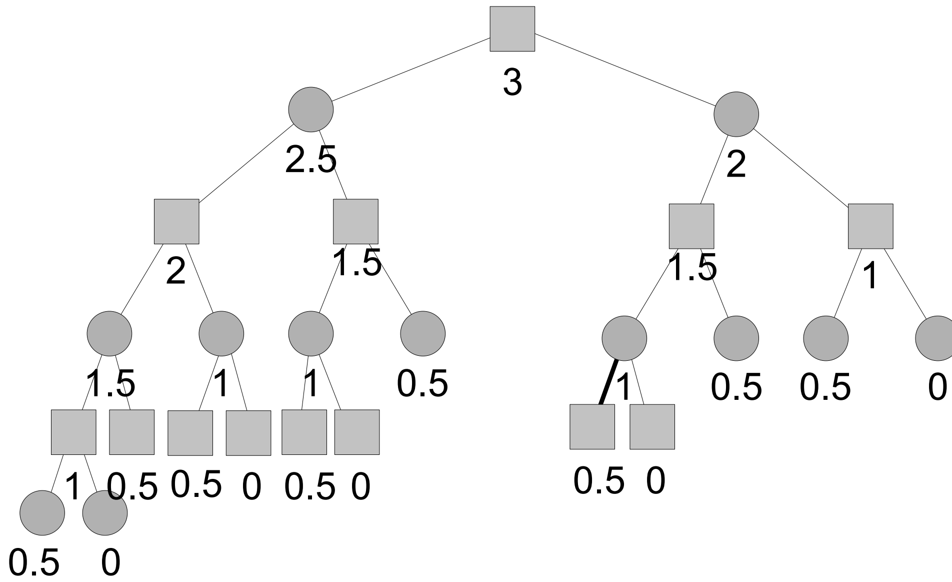

図 3.3　0.5 手延長アルゴリズムの例

であったとしても評価が高ければ指してしまう．これは人間とはかけ離れた動作である．この点を改善し，人間らしい手をさせるようにした手法が実現確率探索である [3]．

この手法では棋譜を学習し，それぞれの手の実現する確率を求める．閾値を設定し，確率の低い手を読まないことにより探索を効率化する．

ノード x の実現確率 P_x はノード x' の実現確率を $P_{x'}$，x' から x への遷移確率を P_m とすると

$$P_x = P_m P_{x'} \tag{3.3}$$

となる．ここで求めた確率 P_x が閾値を下回った場合に枝刈りを行う．図 3.4 に実現確率と遷移確率を示す [3]．

実現確率探索を実装したプログラム（提案手法）とそうでないプログラム（従来手法）を対戦させた結果を表 3.2 に示す [3]．従来手法は常に 10[s/手] と設定してある．提案手法は従来手法の半分の思考時間で勝率が 55%，同じ 10[s/手] では 71% の勝率であり，効果がある手法であるといえる．

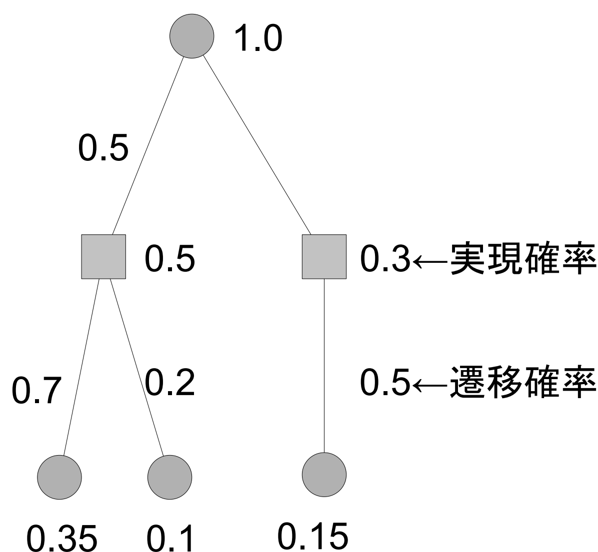

図 3.4　実現確率探索

表 3.2　実現確率探索の勝率（従来手法は常に 10[s/手]）

	勝利数	引き分け	敗北数	勝率 [%]
5[s/手]	109	0	91	55
10[s/手]	137	8	55	71

3.4　モンテカルロ型

3.4.1　モンテカルロ法

モンテカルロ法 [4] はシミュレーションや数値計算を乱数を用いて行なう手法の総称であり，元々は，中性子が物質中を動き回る様子を探るためにジョン・フォン・ノイマンにより考案された手法である．このゲームへの応用は，乱数を用いて決める着手に従ってゲームを進めることで勝敗を決定し，勝負を十分な回数行うことで，どの手が一番勝ちやすいのかを判断する方法である．

この手法の特徴としては，ランダムに手を生成しゲームの終了時まで打っていくプレイ

アウトと呼ばれる試行を行うため，評価関数が必要ないということがあげられる．したがって評価関数がつくりにくいとされる囲碁などで効果的な手法である．

図 3.5 にモンテカルロ法の例を示す．このゲーム木のリーフは勝敗が決定しているものとする．勝ちを 1，負けを 0 とした．ランダムにゲーム木をルートから下っていくが，たとえば最初に A-C-G-N というノードをたどったとする．するとリーフ N は 0，つまり負けのノードであった．すると最初に A から C を選んだという行動に点数がつかず，0 点とする．次にランダムにノードをたどると A-B-E-J だったとする．すると今度は J は勝ちのノードである．したがって A から B を選んだという行動に 1 点加算される．このように十分な回数ランダムに勝敗を決定していくと，A という局面からは B を選んだ行動が点数が高いのか，C を選んだ行動のほうが点数が高いのかがわかる．この点数が高いほうをよい局面とする．このような乱数を使った手法をモンテカルロ法という．

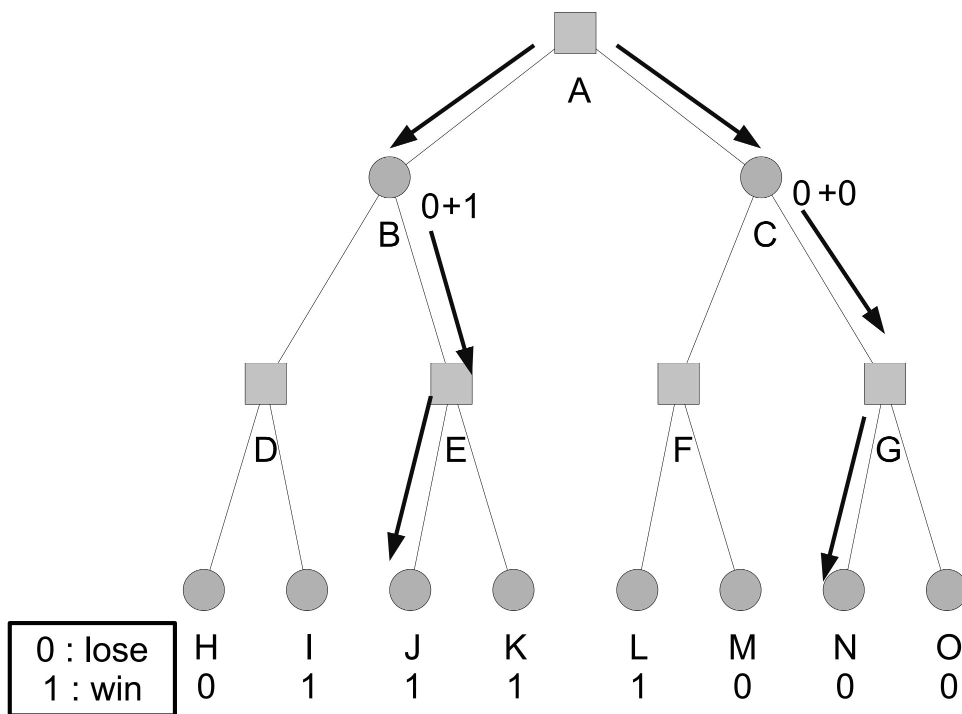

図 3.5　モンテカルロ法

3.4.2　多腕バンディット問題

多腕バンディット問題は，多数の腕を持つスロットマシンで最大の報酬を得ようとしたときに考えられる機械学習の問題である．n 本の腕のあるスロットマシンを考える．スロットマシンのそれぞれの腕は報酬に関する固有の確率分布を持つ．

このスロットマシンで得られる報酬を最大にしたい場合，もっとも高い報酬を得られる腕でプレイしたい．ある程度プレイしたとき，現在分かっている高い報酬を得られる腕だけでプレイすると，一定の報酬は得られるが，まだほかにも高い報酬を得られる腕があるかもしれないため，他の腕も試したい．これは収穫と探検のジレンマと呼ばれている．

ゲームにおける合法手を腕，報酬を勝敗と考えると，モンテカルロ法と多腕バンディット問題は類似した問題として扱うことができる．したがってモンテカルロ法に多腕バンディット問題のアルゴリズムを適用する場合が多くある．

3.4.3　UCT

モンテカルロ法のひとつに UCT[5] と呼ばれるものがある．評価の高い手付近を重点的に探索する手法である．これは MoGo という囲碁のプログラムに実装されている．

先に述べたモンテカルロ法で，図 3.5 において A から B，または C を選ぶ際の点数のみを計算したが，同様にほかのノードへ移る際の点数も計算できる．ランダムにノードを選んでいるため，選ばれた回数で点数が高くなったり低くなったりしないように，あるノード i が選ばれて勝った回数をノードが選ばれた数で割った数 X_i を勝率と定義する．また，任意のノードを選んだ回数を n，ノード i が選ばれた数を $T_i(n)$ とする．そして

$$X_i + \sqrt{\frac{2\log n}{T_i(n)}} \tag{3.4}$$

を各ノードの子ノードについて計算し，この値が 1 番高いノードを選び探索する．この式の第 1 項は勝率を表し，勝率が高いノードを重点的に探索するという目的にあっている．第 2 項ではいままであまり選ばれていない子ノードが大きくなりやすい．したがって勝率の高いノード付近を探索しながら，あまり選ばれていないノードも探索し，ほかの可能性も探ることができる．

図 3.6 に UCT の探索例を示す．図に示すように探索は非対称に進んでいく．UCT は良い手をより深く探索していく．また，時間制限にも強い．図 3.7 に α-β 法で探索時間が短

くてあまり探索できなかったゲーム木の例を示す [6]．‘?’ はまだ探索されていないノードでありこれは完全な探索が不可能な巨大な木の探索ではよく起きる．反復深化はこの問題を部分的に解決する α-β 法に比べて UCT はいつ止めてもそれなりに良い結果がでるアルゴリズムである．

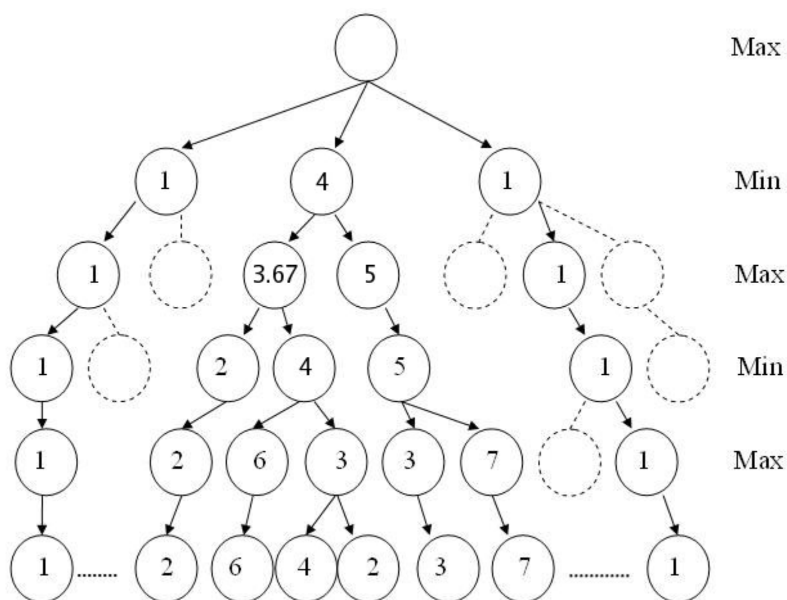

図 3.6　UCT 探索 [6].

3.4.4　パターンを導入した UCT

UCT にヒューリスティックな調整としてパターンを導入する [6]．パターンにマッチする局面の場合にそのノードを選びやすくするものである．図 3.8，図 3.9 に示すような囲碁における有利になると思われる 3×3 のパターンを使う．図 3.8 ではどれかにマッチすれば真を返す [6]．右端のパターンで □ が乗った黒石は，周囲 8 点がマッチしかつ黒番の時に限り真を返すことを意味する．図 3.9 のキリ 1 のパターンは 3 パターンから成る [6]．最初のパターンがマッチし残りの 2 つがマッチしない時真を返す．

表 3.3 に通常の UCT とパターンを導入した UCT についての勝率を示す [6]．パターンを導入したほうがしないより 2 倍ほど勝率が上がっていることがわかる．

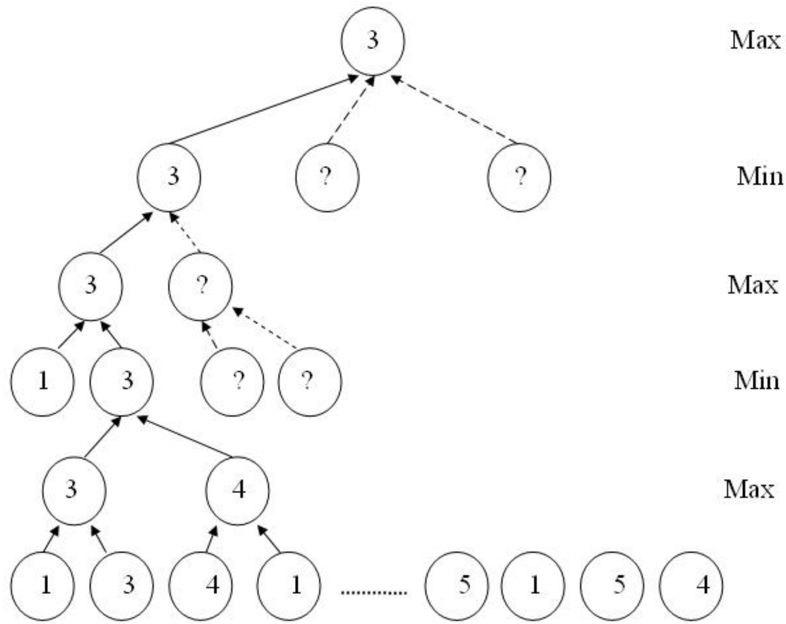

図 3.7　時間が限られた α-β 探索.

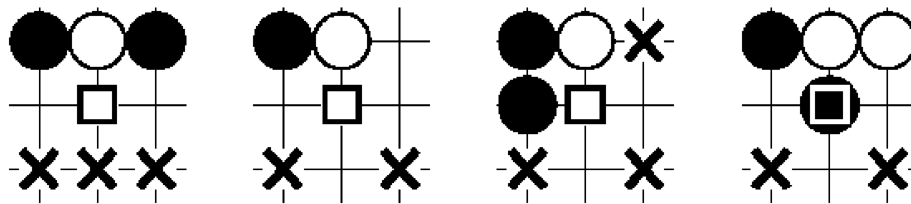

図 3.8　UCT のパターンの例：ハネ・パターン

3.5　それぞれの探索の特徴

ゲームの種類やゲームの展開を木構造として表現するゲーム木を述べた．また，具体的なゲーム探索手法である α-β 法，0.5 手延長アルゴリズム，実現確率探索，UCT などについて述べた.

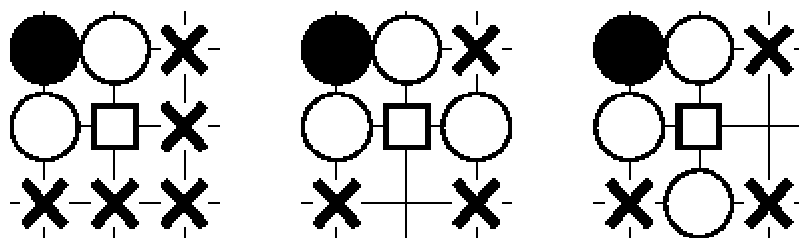

図 3.9　UCT のパターンの例：キリ 1 パターン

表 3.3　純粋な UCT とパターンを使った UCT の勝率

ランダムモード	勝率 [%]
純粋 UCT	41.2
パターンを使った UCT	80.0

これまで述べたように，ゲーム木は巨大な木となってしまうために，どのように良い手だけを探索するかが重要である．その点から α-β 法は Minimax 法と全く同様な結果を得ることができ，最も良い手を選ぶ探索の 1 つであると思われる．

しかし，現実のゲームでは待ち時間に制限があるため，なるべくよさそうな手のみを探索し，計算時間を減らしたいという要求がある．したがって 0.5 手延長アルゴリズムなどのように短時間で，良い手の近くを探索できたほうが効率がよく，同じ時間が与えられた場合，Minimax 法より優れていることが多い．

実現確率探索は棋譜を用いて確率的に棋士が打ちそうな手かどうかを参考にしている．したがってプロに近い手を打つことが可能となっている．

またモンテカルロ法は評価関数が必要ないアルゴリズムであり，評価関数の調整により優劣が決まることがない．囲碁などのように評価関数の作成が難しいゲームでは威力を発揮すると考えられる．

ゲーム木の探索を大きく Minimax 型とモンテカルロ型の 2 つに分け，それぞれの特徴を表 3.4 にまとめた．評価関数の有無などはアルゴリズムによって異なるが，対象とするゲームについての知識がどれだけ蓄積されているかなどによって，適切なアルゴリズムを選択することが重要であると考えられる．

3.6　一般的なコンピュータ囲碁プレイヤ

3.6.1　難しさ

本研究ではゲームとして囲碁を扱う．囲碁がチェスや将棋などのゲームの中で特に難しいといわれるが，これはほかのゲームに比べて

- 合法手がたくさんあること
- 形勢の判断が難しいこと

があげられる．

合法手とはルール上許される手のことである．たとえばオセロの場合を述べる．駒は「白」,「黒」という順で左から並んでいたとする．その右側に何も置いていない場合，そこに「白」を置くことができるが,「黒」を置くことは許されない．このようにオセロやチェス，将棋では合法手がある程度絞られるが，囲碁の場合，あまり石を打つ場所が制限されないため，ゲーム木の枝分かれが膨大な数になってしまい，探索の計算量が大きくなってしまう．

さらに，囲碁はプレイアウトの途中で自分が勝っているのか，負けているのかを判断するのが容易ではない．オセロでは局面の駒の位置やパターンである程度の判断ができ，将棋の場合は駒の種類でも判断ができる．しかし囲碁の場合，地の判断が必要であり，これは優劣がすぐに入れ替わってしまうために判断が難しい．このため形勢判断をする評価関数をつくるのが困難である．

これらの理由により，現在囲碁ではコンピュータはプロの棋士に及ばない．

表 3.4　Minimax 型とモンテカルロ型

	評価関数	ゲームごとの事前の知識
Minimax 型	必要である	評価関数の作成や，実現確率探索などで枝刈りに必要
モンテカルロ型	必要無い	UCT でのパターン作成に必要

3.6.2　コンピュータでの探索方法

囲碁は許される手の数が多く，評価関数の作成が難しいためどの手が良い手なのかの判断が困難である．そこでモンテカルロ法により確率的にどの手がよいのかを判定することが多い．確率的に良い手を求めるため，石を打つ試行回数は重要な要素である．

3.6.3　CPU を用いた探索でのデータ構造

CPU での実装では囲碁は連と呼ばれるデータ構造を用いることがよく行われる．例として図 3.10 のような局面を考える．連の情報を付加すると図 3.11 のようになる．ここで□は連番号 4 の連の呼吸点を表している．

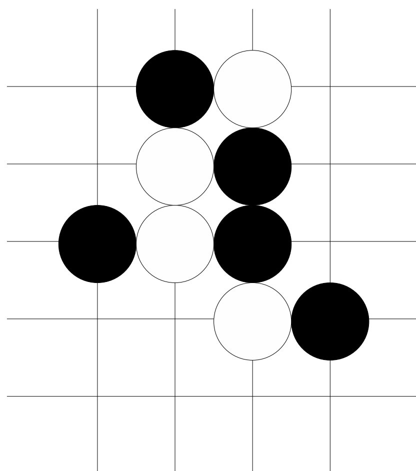

図 3.10　囲碁の局面の例

連は呼吸点を共有するため，相手の石に囲まれて自分の石がとられるかどうかという判断は同一の連に属していれば同一の結果となる．したがって連という単位は局面を CPU で計算するときには重要な概念となる．データ構造を連ごとで考え，連に属する石の座標，呼吸点の数，隣合う連などを保持している．この連などのデータ構造を用いゲームを進めていく．

囲碁のルール上，相手の石で上下左右を囲まれた場合，石がとられる．これをコンピュータで扱いやすいようにする．連の上下左右を相手の石で囲まれることは，連の呼吸点の数

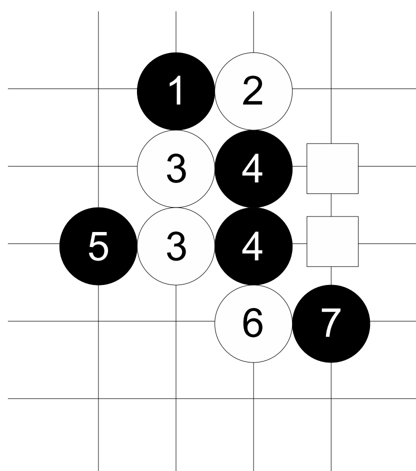

図 3.11　連の情報を付加した囲碁の局面の例

が 0 になることと同じことである．したがって連ごとに隣り合う空点である呼吸点の数を管理する．すると囲まれたかどうかという何ステップもかかる判断が，呼吸点の数が 0 かどうかで判断することができる．例として図 3.11 では連番号 4 の連の呼吸点が 2 個である．この 2 個の呼吸点の位置に相手の石が打たれた場合，連番号 4 の連の呼吸点が 0 個となり，とられる．

局面の状態を石の配置だけで保持しておくのではなく，連の情報をキャッシュしておくことで少ないステップ数で局面の処理をすることが可能となり，一定時間内でのプレイアウト数の増加につながる．

3.7　FPGA での囲碁の実装

3.7.1　TLPG

CPU 以外での囲碁の実装は FPGA（Field Programmable Gate Array）上において TLPG（Triple Line-based Playout for Go）と呼ばれるハードウェア実装を用いた研究 [7] がある．

TLPG で局面の情報を 3 行分読みこみ，パイプライン処理によって 9 路盤で毎秒 13,104 プレイアウト，19 路盤で毎秒 2,055 プレイアウトを実現している．

TLPG の詳細を述べる．連が取れるかどうかの判定に必要な連の呼吸点のカウントを

する．

局面を横方向に並列，縦方向にパイプラインで処理する．

1 行ずつ局面を取り出して 3 段のシフトレジスタに入力する．図 3.12 のように連の呼吸点の計算を行う [7]．

1. 局面の上端から下端までを入力していき，連の呼吸点を発見したらその連の呼吸点の数を加算していき保存する
2. 下端まで調べたら，次は前回と逆向きに下端から上端へ局面を入力していき，保存していた呼吸点の情報を連全体に反映させる

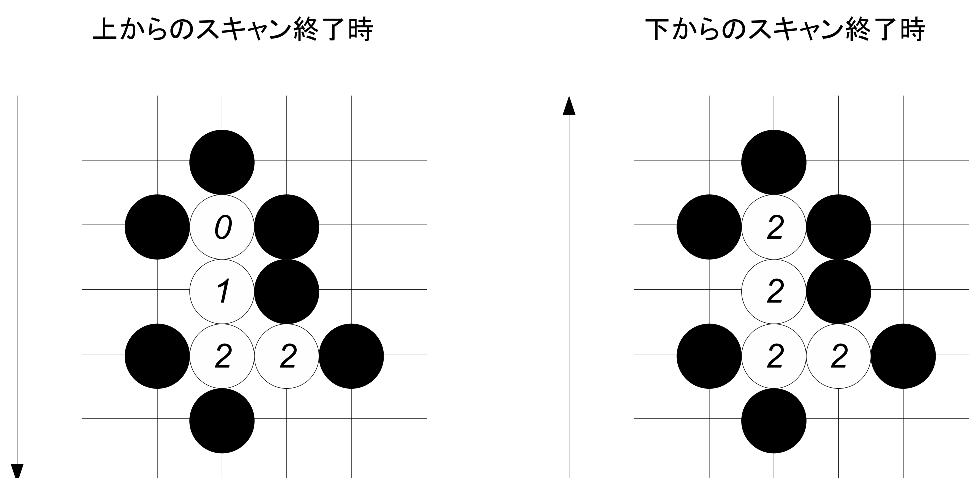

図 3.12　FPGA での連の呼吸点数のカウント

3.7.2　実験

実験環境

実験に用いた環境は以下のものである．

TLPG　Xilinx 社の FPGA（Virtex-5 XC5VFX70T-1FF1136）上に実装
論理合成　Xilinx ISE 11.2

実験結果

実験結果を表 3.7.2 に示す [7].

表 3.5　FPGA での囲碁のプレイアウト性能

	9 路盤	19 路盤
回路遅延	9.255ns	15.393ns
動作周波数	104MHz	62.5MHz
プレイアウト速度	13104po/s	2055po/s

3.8　GPU での囲碁の実装

GPU で囲碁を実装したコードがメーリングリスト[*1]で公開されている．これによると 9 路盤が GPU で毎秒 17,300 プレイアウトで実行されるという報告がある．この実装では int 型 1 個を 1 個の交点とし，1thread を 1 交点，1block を 1 局面としている．

メーリングリストでの別の実装[*2]では 9 路盤が GPU で毎秒 170,000 プレイアウトで実行されるという報告がある．

[*1] Petr Baudis, “[computer-go] CUDA implementation of the per-intersection GPGPU approach”, http://computer-go.org/pipermail/computer-go/2009-September/019433.html

[*2] Christian Nentwich, “[computer-go] CUDA and GPU Performance”, http://computer-go.org/pipermail/computer-go/2009-September/019422.html

第 4 章

提案手法

4.1 GPU によるプレイアウトの課題

GPU を用いた囲碁のプレイアウトについて述べる．

本研究ではモンテカルロ法を行う際の，次々に石を打っていくプレイアウトと呼ばれる処理を GPU で実行し高速に処理する．囲碁のプレイアウトをする場合，合法手であるか，相手の石は取れるかといったことや，連を定義した場合どの連をとることができるか，など判断すべき事柄は多い．通常こうした判断をコンピュータでは条件分岐の組合せにより表現する．しかしながら GPU の場合は GPU 内のコアが異なる方向に分岐した場合，同一ストリーミングマルチプロセッサ内のコアは両方の分岐を通っただけの時間を消費する．この制約によって GPU では条件分岐をできるだけ使わないアルゴリズムのほうが性能を生かすことができる．

通常 CPU であれば連と呼ばれる同じ色の石が縦横に連なったものを定義する．しかし GPU の性質から連のような属する石の数や連の数が状況によって変化するデータ構造を扱いにくい．また一度に処理するデータが多いほうが性能を発揮する．

そこで本研究ではこのような事態に対処するため多数の局面の進行を同時に管理し，実行時に大きさが固定のデータを扱うために連での管理をせず局面の石の配置を基礎にプレイアウトを行う．局面の更新は石を置くたびに計算し，取れる石を判定する．

4.2　GPU の ECC の検討

GPU には ECC（Error Correcting Code，誤り訂正符号）が実装されていないものが多くある．最新の Fermi という GPU では搭載されているが，そうでない GPU がまだ数多く存在する．本研究に用いた GPU も ECC は備えていない．GPU はグラフィックス用途に用いられる場合，1 ビット誤っていてもあまり影響のない場合が多い．しかし科学技術計算などに GPU を用いる場合は 1 ビットの誤りが許されないことが多い．そこで GPU にソフトウェア処理で ECC を実装する研究もある [14]．

本研究ではプレイアウトに GPU を用いている．プレイアウトはモンテカルロ法に応用され，実際のゲームに用いられることを想定している．したがって確率的にどの手が良いかを無数のプレイアウトの結果から求めるため，GPU で ECC がなかったとしても 1 ビットの誤りは深刻な結果の誤りを引き起こすとは考えにくい．ゆえに GPU に ECC がないという点は本研究のプレイアウトにとってはほとんど影響はないと言っていい．

4.3　データ構造

図 4.1 のように 32 ビットの int 型整数値を 11 個を 1 組としてこれを 3 組使い，9 路盤 1 局面を構成している．3 組の整数のうち，2 組で各交点の状態を管理する．各交点は空点，黒石，白石，番兵の 4 状態をとる．残りの 1 組で後に述べる生存フラグを各交点に定義する．

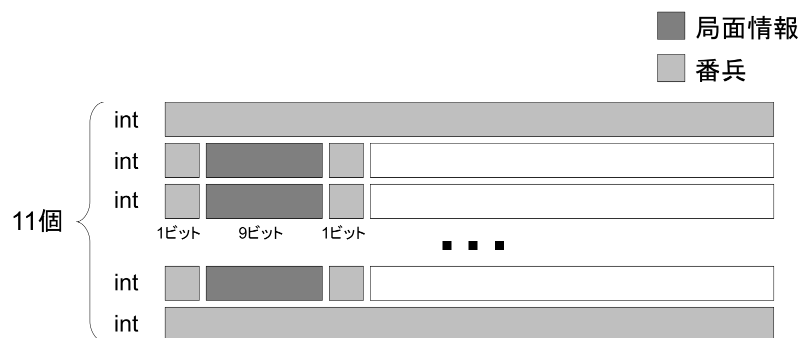

図 4.1　局面データの格納方法

番兵とは実際の囲碁には存在しない領域であるが，コンピュータで局面を作る際にはよく用いられるものである．各交点の上下左右を調べるときに，すべての交点が同様の動きをしたほうがコードが簡単になり GPU のアーキテクチャの面からも都合がよい．もし局面の角の交点の外側に変数が定義されていない場合を考える．この場合，角の交点の外側にアクセスすることは許されない．したがって局面の内側と角や辺とではプログラムを変えなくてはならない．これを防ぐために番兵と呼ばれる仮の領域を用意する．番兵があるため，局面の角などであっても内側の交点と同様に上下左右すべてにアクセスするコードが使える．

1 局面は 9 路盤であれば交点は 9^2 で 81 個である．したがって 32 ビットの int 型 3 個が 3 組あれば局面の状態と生存フラグはデータ容量としては十分に定義できる．しかし，その場合 int 型 3 個の中での交点の並べ方が問題となる．局面上で交点の上下左右にアクセスしたときにつながっているようにする処理が必要となる．各交点へのアクセスは頻繁におこるので，単純なほうがよい．したがってある交点から上下左右が自然な形でアクセスできる 11 個の整数を使う．

スレッドを各整数に割り当て，局面の更新など行う．

4.4　局面更新の論理演算

局面の更新を行う方法を述べる．各交点をビットで表しているため，状態更新をビットごとの論理演算で行う．各交点をそれぞれ比較していくよりもビットごとの論理演算で 1 行ごと比較していくほうが速度の面から有利である．

ここで局面の交点 (x, y) での $e_{x,y}$ を空点かどうかを表す 2 値変数，$b_{x,y}$ を黒石かどうかを表す 2 値変数とおく．真の値を T，偽の値を F とあらわすことにすると

空点　$e_{x,y} = T$，$b_{x,y} = F$
黒石　$e_{x,y} = F$，$b_{x,y} = T$
白石　$e_{x,y} = F$，$b_{x,y} = F$
番兵　$e_{x,y} = T$，$b_{x,y} = T$

となる．ただし，空点と番兵の $b_{x,y}$ には任意性があるが，上のように定めた．また，交点 (x, y) での生存フラグを $l_{x,y}$ とする．

まず，自分が色 c_1 の石（黒石ならば $c_1 = T$，白石ならば $c_1 = F$）を打つ．次に，相手の色 $c_2 (= \overline{c_1})$ の石がとれるかどうかの判定処理に移る．

各交点 (x,y) が相手の石で，かつ各交点の上下左右に空点があるかを判定し，結果を $l_{x,y}$ の値とする．

各交点 (x,y) について相手の色と同じかどうか

$$\overline{c_2 \otimes b_{x,y}} \cap \overline{e_{x,y}} \tag{4.1}$$

局面の各交点の上下左右が空点かどうか

$$e_{x+1,y} \cap \overline{b_{x+1,y}} \tag{4.2}$$
$$e_{x,y+1} \cap \overline{b_{x,y+1}} \tag{4.3}$$
$$e_{x-1,y} \cap \overline{b_{x-1,y}} \tag{4.4}$$
$$e_{x,y-1} \cap \overline{b_{x,y-1}} \tag{4.5}$$

を判定し，まとめて $l_{x,y}$ の値とする．つまり $l_{x,y}$ は

$$\begin{aligned} l_{x,y} =& (\overline{c_2 \otimes b_{x,y}} \cap \overline{e_{x,y}}) \\ & \cap \{(e_{x+1,y} \cap \overline{b_{x+1,y}}) \cup (e_{x,y+1} \cap \overline{b_{x,y+1}}) \cup (e_{x-1,y} \cap \overline{b_{x-1,y}}) \cup (e_{x,y-1} \cap \overline{b_{x,y-1}})\} \end{aligned} \tag{4.6}$$

となる．

つぎに $l_{x,y}$ を上下左右に伝搬していく処理に移る．

1 回の伝搬処理を実行するとき，伝搬前の生存フラグを $l_{x,y}$，伝搬後の生存フラグを $l'_{x,y}$ とする．$l'_{x,y}$ が T となるための条件は (x,y) の石が生存しているか，または隣接する上下左右の石が生存していることである．(x,y) に相手の石が存在しているかどうかは $\overline{c_2 \otimes b_{x,y}} \cap \overline{e_{x,y}}$，生存しているかどうかは $l_{x,y}$ の値で判定でき，隣接する石も同様に $l_{x+1,y}$，$l_{x,y+1}$，$l_{x-1,y}$，$l_{x,y-1}$ の値で判定できる．したがって $l'_{x,y}$ は

$$l'_{x,y} = (\overline{c_2 \otimes b_{x,y}} \cap \overline{e_{x,y}}) \cap (l_{x,y} \cup l_{x+1,y} \cup l_{x,y+1} \cup l_{x-1,y} \cup l_{x,y-1}) \tag{4.7}$$

である．式 (4.7) をすべての交点で繰り返し実行し，$l_{x,y} = l'_{x,y}$ となったとき，式 (4.7) の適用を終了する．すると，生きている石はすべて $l_{x,y} = T$ となる．

ここから石をとる処理に移る．

ここで，1 回の石をとる処理で空点にする交点 (x,y) を真偽値 $e_{1x,y}$ とする．石がとれる条件は相手の石が存在し，生存していないことであるから $\overline{l_{x,y}} \cap c_2(\overline{c_2 \otimes b_{x,y}} \cap \overline{e_{x,y}})$，またもともと空点である場合 $e_{x,y} \cap \overline{b_{x,y}}$ である．これらをまとめて

$$e_{1x,y} = \{\overline{l_{x,y}} \cap c_2(\overline{c_2 \otimes b_{x,y}} \cap \overline{e_{x,y}})\} \cup (e_{x,y} \cap \overline{b_{x,y}}) \tag{4.8}$$

とる処理をした後の空点かどうかを $e'_{x,y}$，黒石かどうかを $b'_{x,y}$ であらわすと

$$e'_{x,y} = e_{1x,y} \cup e_{x,y} \tag{4.9}$$
$$b'_{x,y} = \overline{e_{1x,y}} \cap b_{x,y} \tag{4.10}$$

となる．

4.5 プレイアウトの流れ

実際の流れを述べる．全体は図 4.2 のようになる．

局面は GPU 内のシェアードメモリに管理する．

ランダムに手を選択し，合法であるかどうかを判断する．合法なら打ち，合法でなかったらもう一度ランダムに手を選択する．これを今回は 10 回繰り返し，すべて合法でなかった場合にパスをすることにした．

そしてとれる石の判定と石をとる処理に移る．とれる石の判定は生存フラグを定義して行う．生存フラグは，各石に定義され真と偽の 2 値をとる．すべての生存フラグの初期値を偽とする．

自分が石を打ったとき，相手の色の石すべてに対して初期値が偽の生存フラグを定義し，とれる石の判定を行う．とられることがないことが確定した場合生存フラグを真とする．まず相手の石それぞれの上下左右に空点があるかどうかを調べる．上下左右に空点があった場合，必ずその石は生きている．とられることはないから，生存フラグを真とする．

生きている石はほかにもある可能性がある．それは上下左右に空点があった相手の石と接している相手の石である．さらにその石と接している相手の石も同様に生きている．このように生きているという判定は伝搬していくことが分かる．そこで生存フラグを隣の石へ伝搬していく処理を行う．これにより生きているすべての石の生存フラグが真となる．したがって伝搬が収束したとき生存フラグが偽の石は死んでいると判定され，とることができる．このように処理をすることで連を使わずに局面の更新ができる．

今回は簡単のため自殺手は考慮していない．

4.6 プレイアウトの例

例として図 4.3 から図 4.11 のような，最後に黒石を打った局面を考える．すると

1. 黒の手番とする

2. 黒石を打つ
3. 白石の生存フラグを定義．初期値は偽（F）
4. 白石はそれぞれ四方の空点または生存フラグが真の同じ色の石を探す．更新 1 回目で左側の白石は下に呼吸点があった．
5. 右の白石には四方に空点または生存フラグが真の同じ色の石がなかった．
6. 左側の白石は呼吸点があったため生存フラグが真（T）となる．右側の白石は四方に空点または生存フラグが真の同じ色の石がなかったため F のまま．生存フラグ更新の 1 回目が終了．
7. 2 回目の生存フラグの更新を開始．左側の白石はすでに生存フラグは T であるから，四方によらず T である．
8. 右側の白石は四方を探す．左側の白石が生存フラグが T である．
9. 左側の白石は生存フラグが T のまま．左側の白石が T だったため右側の白石は F から T となる．更新の 2 回目が終了．同様に 3 回目の更新も行う．しかし 3 回目では生存フラグに変化がない．したがって生存フラグは収束したとみなし，生存フラグの更新を終了する．

のようになる．

注目している白石の四方に空点があるかどうかによって呼吸点の有無を判定している．もし一か所でも呼吸点が存在すれば，その石は取られることはない．呼吸点があったという情報を隣の白石に伝え，右の白石も四方を石に囲まれているが，取られることはないという判定ができる．この呼吸点の情報をすべての行，列に行き渡らせるためにすべての生存フラグが変更されなくなるまで更新処理を行う．更新を行い，前回と生存フラグが変化していなければそれは更新が収束したと考えられる．

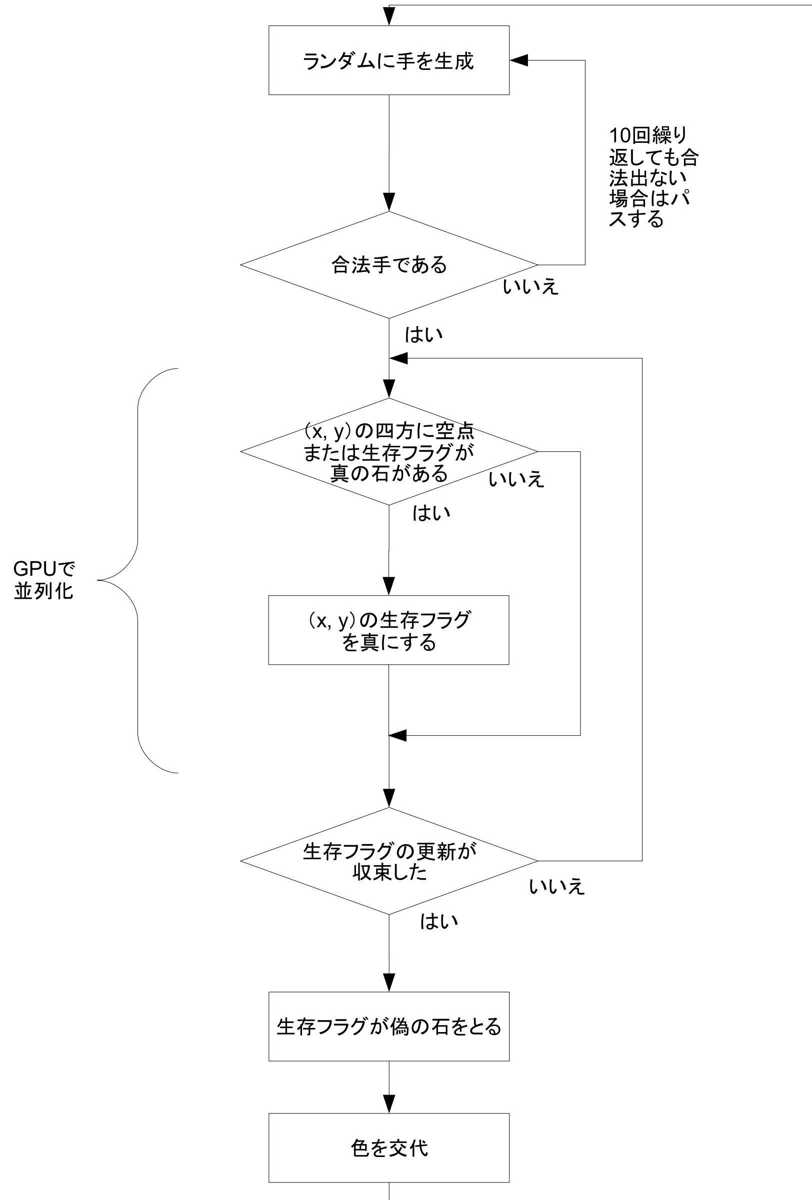

図 4.2　GPU を用いた囲碁のプレイアウトの流れ

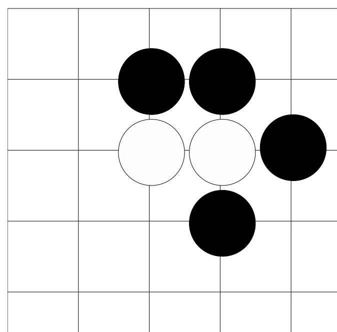

図 4.3　生存フラグの更新 1

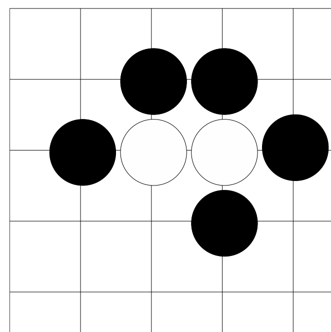

図 4.4　生存フラグの更新 2

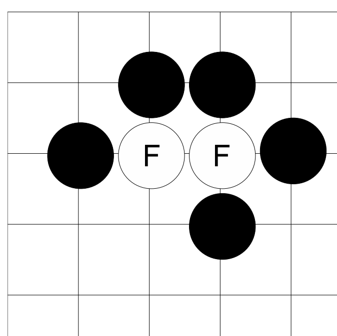

図 4.5　生存フラグの更新 3

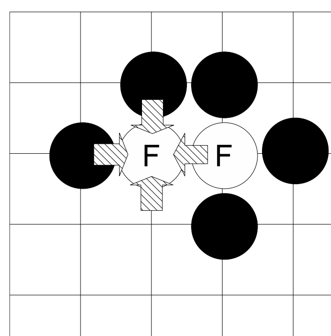

図 4.6　生存フラグの更新 4

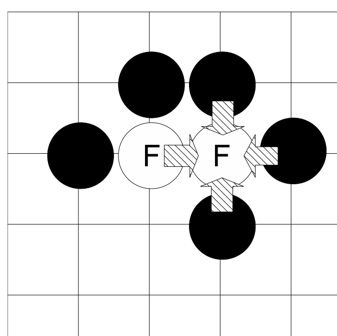

図 4.7　生存フラグの更新 5

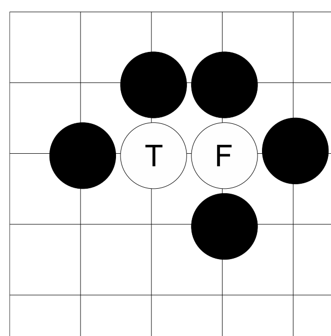

図 4.8　生存フラグの更新 6

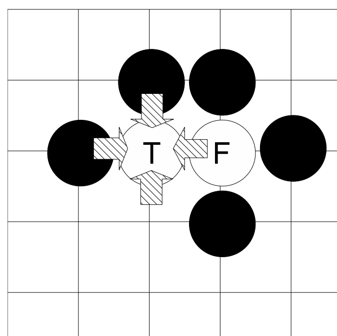

図 4.9　生存フラグの更新 7

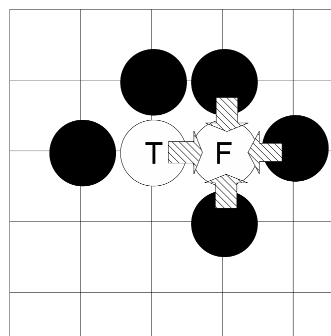

図 4.10　生存フラグの更新 8

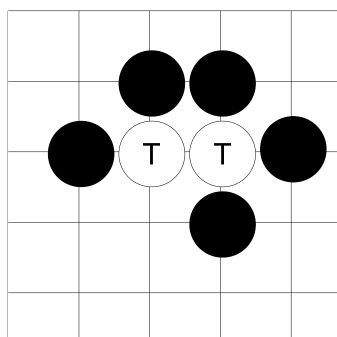

図 4.11　生存フラグの更新 9

第 5 章

実験と評価

5.1 環境

プレイアウトを実施した環境は次のとおりである.

CPU Core i7 920 2.66GHz
メモリ 6GB
GPU Tesla C1060
- **ストリーミングプロセッサコアの数** 240
- **プロセッサコアの周波数** 1.296GHz
- **デバイスメモリ** 4GB
- SP **数** 240
- SM **数** 30

GPU 用の開発環境は NVIDIA 社の CUDA2.1[12] を用いた.

5.2 方法

予備実験として，まず CPU でのプレイアウト数を調べ，次に GPU でランダムに手を打ったとき合法な手を選ぶまでの回数と生存フラグの更新の回数がどのような分布になるのかを調べた.

ランダムな手の選択は GPU 内の律速により，1 番多い回数で 1 ワープ内の局面の処理がかかってしまう．この手の選択回数が少ないほど動作は速いが，回数が多かったとして

も 1 ワープ内の更新回数にばらつきがなければ無駄なく次の手へ移ることができる．ばらつきが大きかった場合，たとえば 2 回の手の選択回数の局面と 10 回の選択の局面が同じワープ内にあったとする．すると 2 回の手選択の局面も 10 回の手選択の局面が更新が終わるまで待たされてしまう．したがってばらつきは少ないほうがよい．そこで n 手目でのランダムに選択する回数を調べた．1,000 回プレイアウトを行い，これを標本として平均，分散を n 手目ごとに調べた．

生存フラグの伝搬回数のばらつきも手の選択と同様に GPU 内での無駄が生じる原因となる．そこで n 手目での伝搬が行われる回数を調べた．1,000 回プレイアウトを行い，これを標本として平均，分散を n 手目ごとに調べた．

次に 9 路盤で初期局面からのプレイアウトにかかる時間を計測し，100 回の平均を算出した．GPU で一度に扱う局面数を増やしていき，単位時間当たりのプレイアウト数の変化を測った．1 個の block に格納する局面の最適値を求め，この最適値を 1block に指定して block 数を変化させプレイアウトの速さを測定した．

5.3　結果

予備実験の結果を示す．CPU でのプレイアウト数は GNU Go 3.8[15] を用い，これを改変してプレイアウト数と消費時間を調べた．10 回をとった結果を表 5.1 に示す．平均は 8,294po/s であった．1 コアで実験しているため，本研究で用いている CPU を 4 コアすべて使えばさらに性能が伸びると考えられる．

表 5.1　GNU Go の毎秒のプレイアウト数

プレイアウト回数	時間 [s]	毎秒のプレイアウト [po/s]
78921	9.45	8351
78868	9.35	8435
78817	10.35	7615
78765	9.16	8599
78730	9.04	8709
78756	9.40	8378
78964	9.56	8260
78848	9.42	8370
79008	9.84	8029
78989	9.64	8194

合法になるまでランダムに手を打ったときの回数の n 手目の平均を図 5.1 に示し，その分散を図 5.2 に示す．合法になるまでの回数はランダムに番兵の部分も選択するため 1 手目であっても失敗することがある．80 手目付近で最も合法でない手を選択する確率が高くなっている．これは終盤であるから石が混んでいるためである．90 から 100 手目で分散は最大となっており，平均のピークより少し後に分散のピークがある．これは終局した局面が全く手を生成しないのに対し，終局に近い局面は石が込み合っており，多くの手生成が起こるためであると考えられる．

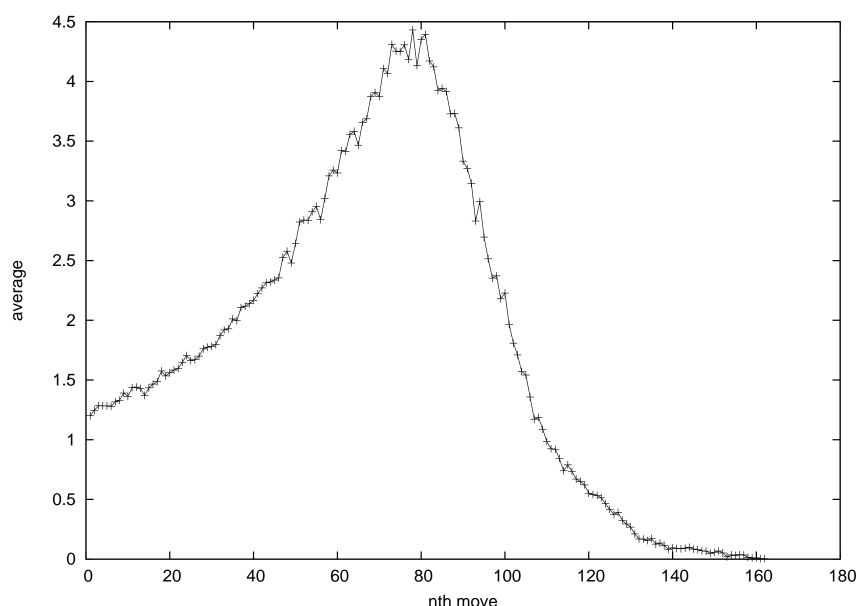

図 5.1　合法になるまでランダムに手を打ったときの回数の n 手目の平均

合法手を打ったときの生存フラグの伝播回数の n 手目の平均を図 5.3 に示し，その分散を図 5.4 に示す．20 手目を越えたあたりで徐々に伝搬回数が上昇し始め，80 手目付近で最大になっており終盤になると伝播回数が多くなっていることがわかる．そして 80 手目付近から徐々に下降し始める．これは終局した局面が多くなったことを示すと考えられる．徐々に石が混んでいると伝播回数の分散が大きくなっている．90 から 100 手目で最大となっている．これも手生成の回数と同様に終局した局面が全く伝搬させないのに対し，終局に近い局面は石が込み合っており，多くの伝搬が起こるためと考えられる．したがって平均のピークより少し後に分散のピークがある．

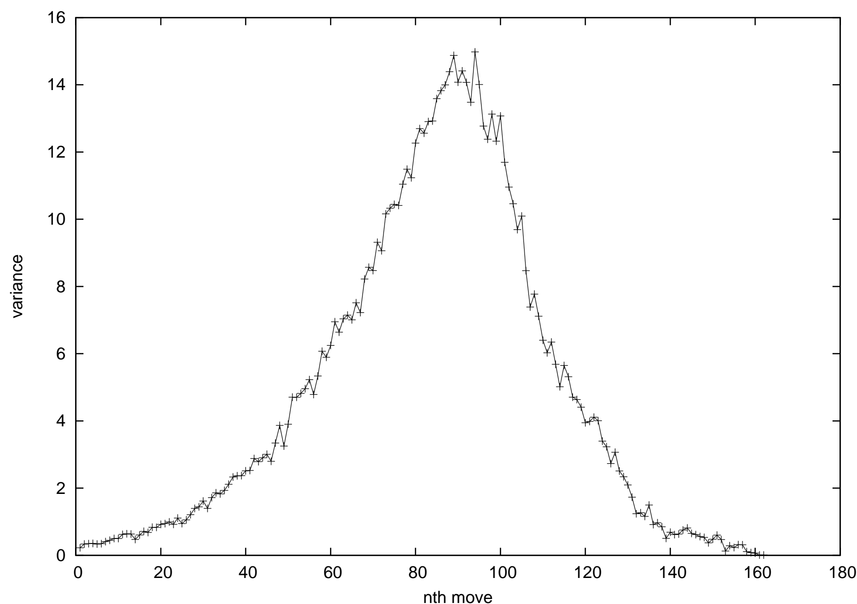

図 5.2　合法になるまでランダムに手を打ったときの回数の n 手目の分散

1block あたりの局面数を変化させたときの結果を表 5.2，図 5.5 に示す．最もプレイアウト数が大きくなるのは 1block あたりの 9 局面のときで毎秒 70,233 プレイアウトであった．次に 1block に 9 局面を入れ，これを GPU へ送る block 数を変化させプレイアウト数の変化を測定した．結果を表 5.3，図 5.6 に示す．処理する局面数を増やすことで単位時間当たりのプレイアウト数が増加し，block 数が 7,000 くらいから飽和することが分かる．block 数を 41000 と指定したときに毎秒 70,660 プレイアウトであった．

GNUGo が 1 コアで 8,294po/s であったが，もし 4 コア用いて約 4 倍の性能向上を成し得たとしても GPU の優位性を示すことができる．

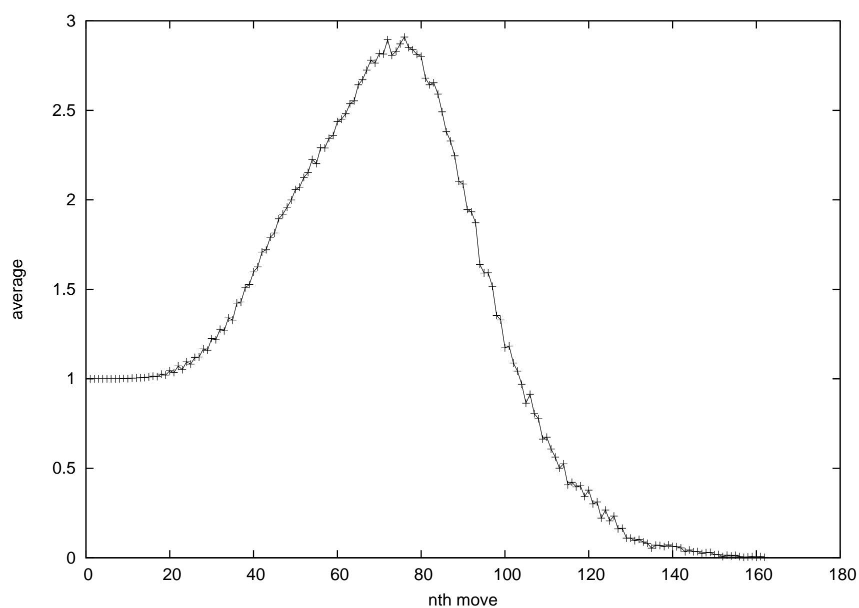

図 5.3　合法手を打ったときの生存フラグの伝播回数の n 手目の平均

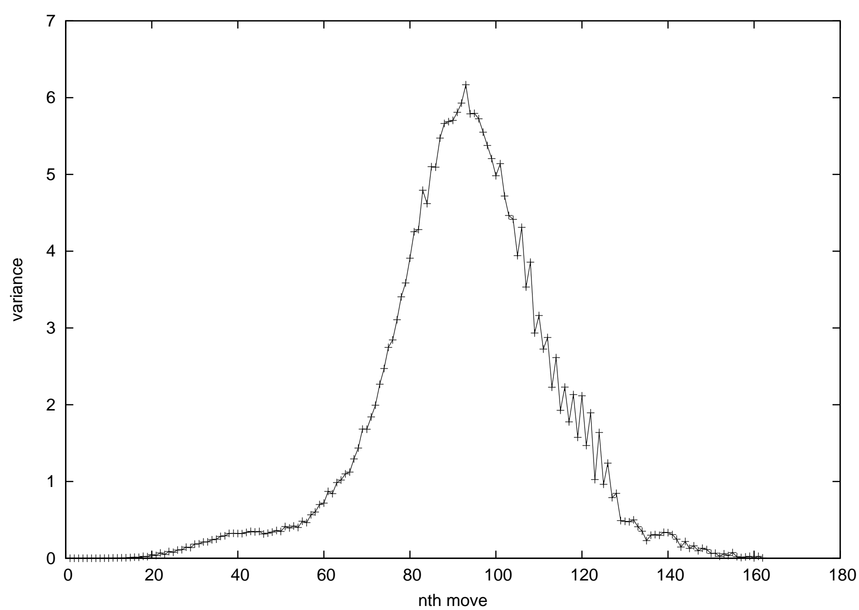

図 5.4　合法手を打ったときの生存フラグの伝播回数の n 手目の分散

表 5.2　1block あたりの局面数と毎秒のプレイアウト数

1block あたりの局面数	毎秒プレイアウト数
3	66253
6	68825
9	70233
12	69597
15	63157
18	67363
21	67546
24	51360

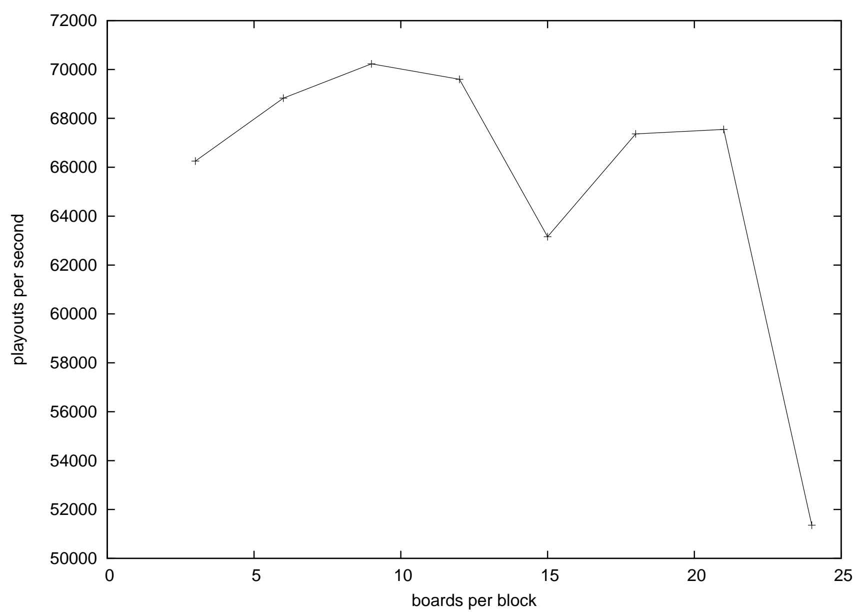

図 5.5　1block あたりの局面数と毎秒のプレイアウト数の測定結果

表 5.3　GPU へ送る block 数と毎秒プレイアウト数

GPU へ送る block 数	毎秒プレイアウト数	GPU へ送る block 数	毎秒プレイアウト数
1000	63496	26000	70609
2000	66524	27000	70585
3000	67899	28000	70584
4000	68764	29000	70592
5000	69139	30000	70577
6000	69219	31000	70633
7000	69594	32000	70640
8000	69852	33000	70588
9000	69988	34000	70597
10000	70232	35000	70540
11000	70372	36000	70520
12000	70318	37000	70586
13000	70250	38000	70611
14000	70320	39000	70583
15000	70330	40000	70660
16000	70458	41000	70660
17000	70473	42000	70528
18000	70434	43000	70520
19000	70477	44000	70563
20000	70385	45000	70478
21000	70377	46000	70556
22000	70454	47000	70577
23000	70497	48000	70528
24000	70527	49000	70565
25000	70577	50000	70569

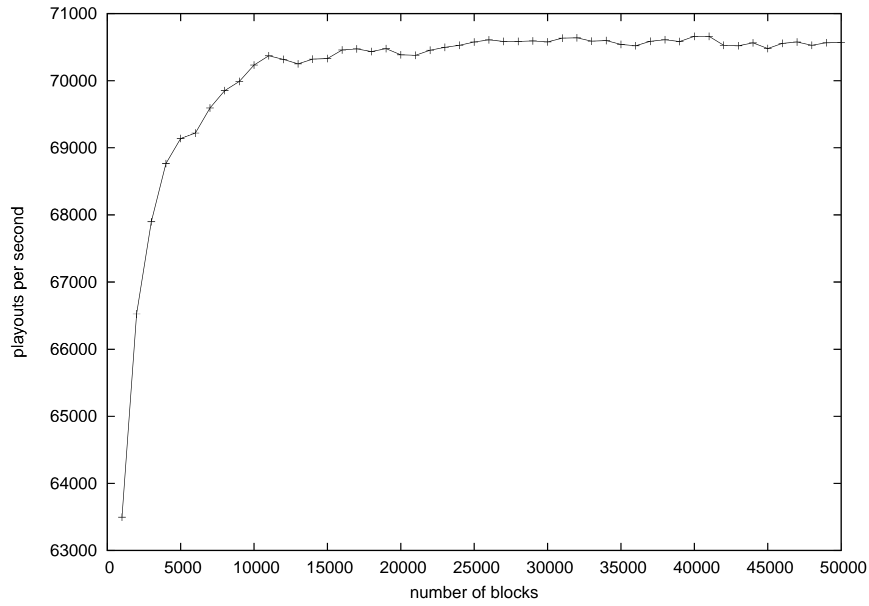

図 5.6　GPU へ送る block 数と毎秒プレイアウト数の測定結果

第 6 章

結論

6.1 まとめ

囲碁は可能な手の数が多いことや形勢が読みにくいことからゲームの中でも特に難しいとされている．強いゲームプレイヤをつくるためには膨大な計算を短時間で行う必要がある．そこで本研究では高性能化の著しい GPU を用い，囲碁のプレイアウトを実装した．アルゴリズムを適したものにすることにより GPU のアーキテクチャを生かし性能を引き出すことができた．CPU を用いたものや GPU の先行例で用いている連という石の連なりを，本研究では用いることなしに GPU で高速なプレイアウトを実現した．

今回の実装では自殺手判定をしていないものの，CPU を用いた GNU Go 3.8 に比べ 8.5 倍，関連研究で示した FPGA を用いたものに比べ 5.4 倍，GPU を用いた Petr Baudis の実装に比べ 4.1 倍の速度向上を示すことができた．

一方で GPU を用いた Christian Nentwich の実装よりも 2.4 倍実行時間がかかってしまっている．しかし我々の実装は，現在は整数 11 個に 1 局面を格納しているが，図 6.1 のように整数 11 個で 3 局面を同時に格納できる．これを行った場合，現在の 1 局面に対して行っている処理と同じ処理で同時に 3 局面を更新することが可能であり，十分に先行例の実装と同等の性能を示すことができると考えられる．

6.2 今後の課題

今後は更なる高速化や，モンテカルロ法による対戦の評価，UCT への組み込みが考えられる．

UCT へ組み込むことを考えた場合に，CPU で UCT の木構造を作り，GPU ではプレイアウトを行うとすると，GPU は並列的にプレイアウトを行う．したがって逐次的な UCT とは異なる計算結果になってしまう．これをどの程度の並列度まで許容できるかという課題がある．

図 6.1　局面データの格納方法の改善案

発表文献

1. 田野文彦，三輪 誠，横山大作，近山 隆,『GPU 開発環境 CUDA を用いたゲーム探索の高速化』，第 13 回ゲームプログラミングワークショップ 2008，pp.104-107

謝辞

本研究を進めるにあたって，適切なご指導をいただきました近山隆教授，田浦健次朗准教授に心から感謝を申し上げます．

また近山・田浦研究室の皆様にたくさんのご助言をいただきました．ここに，心より感謝の意を表します．

参考文献

[1] Donald E. Knuth, Ronald W. Moore, “An Analysis of Alpha-Beta Priming”, Artificial Intelligence, 1975

[2] 山下 宏,『コンピュータ将棋の進歩 2』, 共立出版, 1998

[3] Yoshimasa Tsuruoka, Daisaku Yokoyama, Takashi Chikayama, “Game-Tree Search Algorithm Based on realization Probability”, ICGA Journal, 2002

[4] Bernd Brügmann, “Monte Carlo Go”, Max Planck Institute of Physics, 1993

[5] Levente Kocsis and Csaba Szepesvari, “Bandit based Monte-Carlo Planning”, European Conference on Machine Learning pp. 282-293, 2006

[6] Sylvain Gelly, Yizao Wang, Rémi Munos, Olivier Teytaud, “Modification of UCT with Patterns in Monte-Carlo Go”, Institut National de Recherche en Informatique et en Automatique, 2006

[7] 小泉賢一, 石井康雄, 美添一樹, 三好健文, 菅原豊, 稲葉真理, 平木敬,『FPGA 基板を用いたモンテカルロ碁の高速化』, SWoPP2009, 2009

[8] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, John Manferdelli, “High performance discrete Fourier transforms on graphics processors”, Proceedings of the 2008 ACM/IEEE conference on Supercomputing, 2008

[9] 大石篤哉, 長尾雅也, 小野崎洋, 吉村忍,『GPU による弾性波伝播シミュレーション』, 2007

[10] Daniel Cederman, Philippas Tsigas, “A Practical Quicksort Algorithm for Graphics Processors”, 2008

[11] Cg Toolkit - GPU Shader Authoring Language, `http://developer.nvidia.com/object/cg_toolkit.html`

[12] Cuda Zone, `http://www.nvidia.com/object/cuda_home.html`

[13] OpenCL, http://www.khronos.org/opencl/

[14] 丸山直也, 松岡聡, 尾形泰彦, 額田彰, 遠藤敏夫『ソフトウェア ECC による GPU メモリの耐故障性の実現と評価』, 情報処理学会研究報告 2008-HPC-111, SWoPP2008, 2008

[15] GNU Go, http://www.gnu.org/software/gnugo/