連載講座: 高生産並列言語を使いこなす(3)
ゲーム木探索問題

田浦健次朗
東京大学大学院情報理工学系研究科, 情報基盤センター

目 次

1　概要

今号では, ゲーム木探索問題の一種であるオセロゲーム [6] の詰み探索 (End Game Solver) を取り上げる. オセロの局面が与えられ, それを最終局面 (ゲーム終了局面) まで探索し, 手番側の必勝か, 必敗か, 引き分けかを判定する. より正確には, 手番側が最大何石差で勝てるか (または最小何石差で敗けるか) を判定する.

オセロのプログラムに限らずゲーム木探索は, 枝刈りによって探索ノード数を少なくすること, そのための指し手の順序付け (move ordering) の経験則が重要になる. また, 探索ノードあたりの速度を高速化するための手法も数々存在するため, 並列化に当たってそれらを犠牲にしないことも重要である. 本記事では, オセロプログラムとして定評のある Zebra[2] の作者 Gunnar Andersson が公開しているオセロの End Game Solver [1] を元にして, 逐次ではそれと全く同じ動作をするプログラムを作り, それを並列化した.

2　ゲーム木探索

最初にゲーム木探索の基本について説明する. 以下の議論は, 二人のプレイヤーが交互にプレイをする, 完全情報ゲームであればほぼ同様に当てはまる. そのようなゲームの例としてはチェス, 将棋, 囲碁, チェッカーなどがある.

2.1　必勝・必敗・引き分け

ある局面が, **「手番側の必勝」**であるというのは,

- 手番プレイヤーがうまい手を選べば,
- 相手がどんな手を選んでも,

手番プレイヤーが勝てる, ということである.

逆に**「手番側の必敗」**であるというのは,

- 手番側がどんな手を選んでも,
- 相手がうまい手を選べば,

手番側が敗けるということである. 短く言えば「両者が最善の手を選んだときの結果」ということである.

ある局面 g が手番側の必勝もしくは必敗であることをそれぞれ $\mathrm{win}(g)$, $\mathrm{lose}(g)$ と書くと,

$$
\begin{aligned}
\mathrm{win}(g) \quad &\Longleftrightarrow \quad g \text{ が最終局面のとき : } g \text{ が手番の勝ち} && (1)\\
&\qquad\quad\ \ g \text{ が最終局面でないとき : } g \text{ の「次の局面」} h \text{ のどれかが } \mathrm{lose}(h) \text{ を満たす} && (2)\\
\mathrm{lose}(g) \quad &\Longleftrightarrow \quad g \text{ が最終局面のとき : } g \text{ が手番の負け} && (3)\\
&\qquad\quad\ \ g \text{ が最終局面でないとき : } g \text{ の「次の局面」} h \text{ のすべてが } \mathrm{win}(h) \text{ を満たす} && (4)
\end{aligned}
$$

である. ここで h が g の「次の局面」であるとは, g から合法な手 (パスも含める) を一つ打って h に到達できるということであり, 以降この関係を $g \to h$ と書く.

2.2 盤面の評価値

上記の考察を一般化したものが, 盤面の評価値という考え方である. まず g が最終局面のとき, その評価値 $E(g)$ は,

$$E(g) \quad = \quad E_{\mathrm{leaf}}(g) \quad = \quad \begin{cases} 1 & : \text{手番の勝ち} \\ 0 & : \text{引き分け} \\ -1 & : \text{手番の負け} \end{cases} \tag{5}$$

と定める. g が最終局面ではないときは, 評価値 $E(g)$ を

$$E(g) = \max_{h:g\to h}(-E(h)) \tag{6}$$

で定める. この定義式より例えば,

$$\begin{aligned} E(g) \geq 1 \quad &\Longleftrightarrow \quad \max_{h:g\to h}(-E(h)) \geq 1 \\ &\Longleftrightarrow \quad g \to h \text{ なるどれかの } h \text{ に対し}, -E(h) \geq 1 \\ &\Longleftrightarrow \quad g \to h \text{ なるどれかの } h \text{ に対し}, E(h) \leq -1, \end{aligned}$$

同様に,

$$\begin{aligned} E(g) \leq -1 \quad &\Longleftrightarrow \quad \max_{h:g\to h}(-E(h)) \leq -1 \\ &\Longleftrightarrow \quad g \to h \text{ なるすべての } h \text{ に対し}, -E(h) \leq -1 \\ &\Longleftrightarrow \quad g \to h \text{ なるすべての } h \text{ に対し}, E(h) \geq 1 \end{aligned}$$

であり, これら 2 式は $E(g) \geq 1$ を $\mathrm{win}(g)$, $E(g) \leq -1$ を $\mathrm{lose}(g)$ と読み替えれば式 (2.1) そのものである. つまり $E(g)$ を評価することで, g の必勝, 必敗, 引き分け, が判定できる.

今は必勝, 必敗, 引き分けだけに興味があるため, 最終局面の E_{leaf} を式 (5) と定義したが, ここを適切に変えるだけで, $E(g)$ に別の有用な意味を持たせることができる. 例えば

$$E_{\mathrm{leaf}}(g) = \text{手番の石数} - \text{相手の石数}$$

と定義し, これをそのゲームの「得点」であると定義すれば, 一般に $E(g)$ は,

- 手番プレイヤーがうまい手を選べば,
- 相手がどんな手を選んでも,

手番プレイヤーが得ることができる点の最大値, となる. 本稿で行う実験はこの得点を計算している.

また, 詰み探索以外の一般のゲーム木探索では, 最終局面まで探索できることは少ない. その場合, g から適当な手数 (深さ) で到達できる局面に対して, E_{leaf} をその局面の静的な評価値 (その局面のいくつかの特徴量からすぐに計算できる値で, その局面の手番にとっての「望ましさ」を近似するように定めた値) として, あとは式 (6) に従って $E(g)$ を評価すれば, g 自身の望ましさが評価できる.

本記事では詰み探索を対象としているが, 上のように問題の本質はすべて式 (6) に表されており, 評価手法も共通部分が多い.

2.3 $\alpha\beta$ 法

$E(g)$ を評価する最も素直なプログラムは, 再帰呼び出しを用いた以下のようなものである.

```
eval(g) {
  if (g が最終局面) {
    return eval_leaf(g); /* 石数の差 */
  } else {
    E = {}; /* 空集合 */
    for (g → h なるすべての h) {
      E = E ∪ { −eval(h) };
    }
    return max(E);
  }
}
```

eval(g) を計算するのに, $g \to h$ となる h に対して再帰的に eval を呼び出すため, 評価のための再帰呼び出しは, 局面をノード, 局面間の推移関係 $g \to h$ を親子関係とした木構造 (ゲーム探索木) をなす. そこで以下では局面のことをノード, ある局面 (ノード) から一手動かして到達する局面 (ノード) のことを, そのノードの子供と呼ぶことにする.

上で述べた方法は, $E(g)$ を計算するのに, g の子孫すべての評価値を求めていることになる. しかし常にそれが必要なわけではない. 以下が基本的な観測である.

1. 定義式
$$E(g) = \max_{h:g\to h}(-E(h))$$
を見ればわかる通り, $E(g)$ の値に影響を及ぼすのは, g の子供の中で評価値が「最小」のものだけである ($*$).

2. $E(g)$ を計算する過程で, g のどれかの子供 h の評価値が, $E(h) = e$ とわかったとする. このとき,
   - $E(g) \geq -e$ が確定している.
   - g の h 以外の子 h' の評価値を計算する際は, $E(h') < e$ でない限り, $E(g)$ に影響を与えない ($*$ に注意). 従って, $E(h') \geq e$ が確定した時点で $E(h')$ の計算を終了してよい.

つまり,

- ある局面の評価値を計算する際, その子供をいくつか評価するだけで, 自分の評価値がある値「以上」であることは確定する
- ある局面 g の評価値 $\geq a$ が確定したら, g の子は, その評価値 $\geq -a$ 以上であることが確定した時点で, g の結果に影響を与えない. つまりその時点で計算を打ち切って良い.

以上の観測を元にした探索手法が $\alpha\beta$ 法である. 以下で定義された eval(g, α, β) は, 以下の値を返す.

1. $E(g) \geq \beta$ のとき: $\geq \beta$ を満たすある値
2. $E(g) \leq \alpha$ のとき: $\leq \alpha$ を満たすある値 (実際には α)

```
/* E(g) を評価するが,
   E(g) ≤ α ⇒ ある ≤ α を返す
   E(g) ≥ β ⇒ ある ≥ β を返す */
eval(g, α, β) {
  if (g が最終局面) {
    return eval_leaf(g); /* 石数の差 */
  } else {
    for (g → h なるすべての h) {
      α = max(α, eval(h, −β, −α));
      if (α ≥ β) return α; /* E(g) ≥ β が確定 */
    }
    return α;
  }
}
```

図 1: $\alpha\beta$ 法

3. $\alpha < E(g) < \beta$ のとき: $E(g)$

つまりは eval(g, α, β) は, $\alpha < E(g) < \beta$ の場合にだけその正確な値に興味がある, という評価であり, この $[\alpha, \beta]$ のことを「探索窓」と呼ぶ.

2. を言い換えれば, 最初から $E(g) \geq \alpha$ が「確定」した状態で計算が始まる, ということである. 以上に注意してコードにしたものが図 1 である. α は, これまでに「確定」している返り値である.

2.4 指し手の順序付け (move ordering)

$\alpha\beta$ 法においては, あるノードの子供をどの順序で探索するかが, 計算量に大きな影響を与える. 明らかに, g の最初の子供の評価値が確定した時点で, $E(g) \geq \beta$ が確定し, そこで計算を終了出来れば最も効率が良い. 直感的に言えば「手番にとって最も良い手 (= 相手にとって最も悪い手 = 評価値の低い子ノード)」の評価値から先に求めるのがよい. もちろんそのような手が実際に子ノードを評価せずにわかるわけではない. そこで, 子ノードの良し悪しを判断する経験則が必要になる.

説明は省略するがその経験則が一番うまく働いたときの探索ノード数について, 以下が分かっている [5, 4].

- 探索木が深さ l (偶数: 根ノードとその子供のみからなる木を深さ 2 と数える) の完全 d 分木であるとする. すなわち探索木の全ノード数は $1 + d^1 + \cdots + d^{l-1} = (d^l - 1)/(d - 1)$.
- ∞ (および $-\infty$) を, $E(g)$ がとりうるどの値よりも大きい (小さい) 値とする.
- このとき, 評価値の最低な子ノードを, 常に最初に選択できたとすると, eval(g, $-\infty$, ∞) を計算するのに必要なノード数は, $2d^{l/2} - 1$.

つまり大雑把には, $\alpha\beta$ 法は最もうまく行くと, 木の深さを半分にする効果がある. したがって, $\alpha\beta$ と, よく当たる順序付けの効果は非常に大きく, これを無視して並列化を行っても台数効果は探索ノード数の増加により, 容易に打ち消されてしまう.

Andersson の詰み探索で用いる順序付け方法は 3 節で述べる.

3	53	9	21	19	7	51	1
55	59	45	37	35	43	57	49
11	47	15	29	27	13	41	5
23	39	31			25	33	17
22	38	30			24	32	16
10	46	14	28	26	12	40	4
54	58	44	36	34	42	56	48
2	52	8	20	18	6	50	0

図 2: 静的な手の順序付け

3 Andersson の詰み探索およびその並列化

3.1 Andersson の詰み探索コード

本記事で並列化を施した元となった詰み探索コードは, Zebra のホームページ中の Basic End Solver として記述されているもの [1] で,

> Basic endgame solver: When I started working on computer Othello I downloaded an endgame solver created by Warren Smith and later improved by Jean-Christophe Weill. I then improved it myself, the resulting solver is found here.

とある. 後に述べる実験結果と同ホームページ中に報告されている, Zebra の詰み探索ルーチンの速度や探索ノード数 http://radagast.se/othello/ffotest.html によると, この Basic endgame solver と, Zebra の詰み探索ルーチンの間にはまだ開きがあるようである. しかしソースが入手可能であったことからこの Basic endgame solver を用いている.

3.2 指し手の順序付け

指し手を並べ替える経験則としては, 残りの空マス数に応じて 3 種類を用いている. 基本は, 残りの空マス数が少ないときは「下手な考え休むに似たり」で, とにかく生成が高速な方式を用い, 多いときは多少時間をかけても, 当たりやすい経験則を用いるというものである.

ケース (1) 残り空マス数 ≤4 のとき: 図 2 に示された数字が少ないマスへの着手を, よりよい手とみなす (関数名: NoParEndSolve). 例えば角が良い手, その隣は悪い, などの知られた経験則である.

ケース (2) 4< 残り空マス数 ≤7 のとき: 連結な空マスが作る領域を「穴」と呼ぶ. 例えば図 3 には 5 つの穴がある (右上の, 斜めに隣り合っているマス同士も, 同じ穴に属すとみなす). この時, 奇数個の空マスを持つ穴中のマスへの着手を, 偶数個の空マスを持つ穴中のマスへの着手よりも, 良い手とみなす. この基準で同点のマスは, ケース (1) の場合と同様に優劣をつける (関数名: ParEndSolve).

ケース (3) 残り空マス数 >7 のとき: 相手の着手可能なマスの数が少なくなる手を良い手とみなす. この基準で同点のマスは, ケース (1) の場合と同様に優劣をつける (関数名: FastestFirstEndSolve).

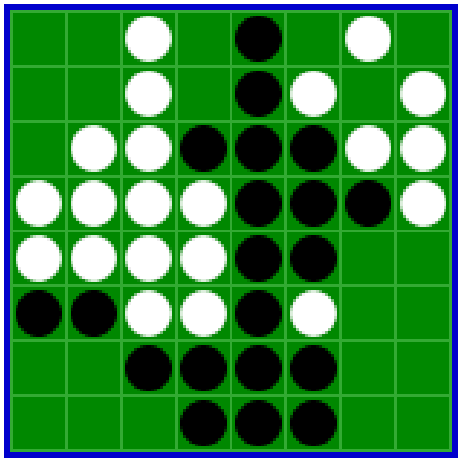

図 3: 5 つの穴を持つ盤面. 左上 (5 マス), 左下 (5 マス), 真ん中上 (2 マス), 右上 (3 マス), 右下 (8 マス). 左上, 左下, 右上への着手は,

盤面に存在する穴の数はゲーム進行に応じて変わりうる. 各空マスが所属する穴, およびそこに残された空マスの数 (の偶奇) を局面ごとに毎回計算するのは大変である. そこで初期局面において一度だけその計算を行い, それ以降, 穴の数は変化しないものとする. つまり, 実際にはゲームが進むについれて分断される穴があっても, それは引き続き一つの穴とみなす. そうであれば, 空マス → 穴の対応関係は変化せず, かつ各穴に残された空マス数の偶奇も簡単に更新できる (ある穴のマスに石がおかれたら, その穴の偶奇を変更すれば良い).

3.3 局面の表現

オセロにおける局面は, 論理的には以下の 3 つだけで記述できるものである. 括弧内は Andersson によるプログラムにおける C 変数の型と名前である.

- 盤面におかれた石の状態 (大域変数 unsigned char board[91])
- 手番 (白か黒; int color)
- 最後の手がパス以外であったか否か (int prevmove)

与えられた局面において石を打てる場所が存在しなかった場合, 最後の手がパスであれば, その時点でゲーム終了, そうでなければパスが唯一の合法手になる. したがって, 最後の手がパス以外であったか否かは局面の一部と考えるべきである.

図 4 に盤面の 91 要素の配列の使われ方を示す. 本来オセロの盤面を表すには, 各マスを 1 バイトで表すとしても, 64 バイトあれば十分であるが, 各マスからどの方向へ行っても, 盤面をはみ出す前にダミー要素に遭遇するように, ダミー要素が付け加わっている.

Andersson のプログラムでは, これ以外に以下を局面の状態として保持する. それらは上記 3 つの情報から生成しうるものであるが, 局面が変化するたびに再計算するのを避けるために, 明示的に保持されている.

- 石の数の差 = 手番の石数 − 相手の石数 (int discdiff)
- 空マスの数 (int empties)

d	d	d	d	d	d	d	d	d	
d	0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	
d	1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	
d	2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7	
d	3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	
d	4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7	
d	5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7	
d	6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7	
d	7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7	
d	d	d	d	d	d	d	d	d	d

図 4: 盤面の表現. 数字が実際に石がおかれるマス, d は番兵の役割を果たすマスで, 白, 黒, 空きのどれでもない値で埋められる. この両者が 91 要素の一次元配列として表現される. 空欄は配列中に保持されていないマス.

- 空マスのリスト. 以下の大域変数 EmHead および Ems.

```
struct EmList
{
    int square;           /* マスの番号. board への index */
    int hole_id;          /* そのマスが属する穴 */
    struct EmList *pred; /* リスト中の前要素へのポインタ  */
    struct EmList *succ; /* リスト中の後続要素へのポインタ  */
} EmHead, Ems[64];
```

- 各穴に残された空マス数の偶奇 (大域変数 int RegionParity) [1]

3.4 末端付近の最適化

木構造探索においては, 多くのノードが末端 (leaf) である. 例えば完全 d 分木では全体のほぼ $(d-1)/d$ のノードが末端である. 従って末端近くのオーバーヘッドを減らすことは重要である. オセロの詰め探索では, 空きマス数が 1 となった時点で, 可能な手は 1 つ以下 (その空きマスもしくはパス) なので, 図 1 にある for 文は省略でき, 他にも多くの部分が簡略化できる. そして関数呼び出しも消去してインライン展開してしまえば, さらにオーバーヘッドが削減できる. これらも Andersson のプログラムに取り入れられている.

4 並列化のための準備: 逐次の基本コード作成

4.1 次の局面の生成法

ある局面 g から一手動かした後の局面 h をどこに生成するか—メモリをどう確保するか—には大きく二つの方法が考えられる.

コピー方式: 一手ごとに g をコピーし, そのコピーを更新して h_i を生成する方法

[1] オセロの盤は 64 のマスを持つため, 穴の数は最大で 32 である. 最大 32 の偶奇をそれぞれ 1 bit で表現している

Undo 方式: 一手ごとに g を直接更新して h_i を作り (この時点で g は一旦失われる), h_i の評価がすんだら h_i から g を復元 (undo) する方法

の二つが考えられる. 一般には状態をコピーするよりも undo の計算の方が速いため, undo 方式の方が速い. Andersson のプログラムも undo 方式を用いている. 一方この方法はすべての評価を局面一つ分のメモリで実行するため, 並列実行は不可能である. 最低限, 時間的にオーバーラップして評価される局面は, 異なるメモリに割り当てなくてはならない. そこで適切なタイミングで局面データをコピーすることになる.

具体的にいつどのようにコピーするのが自然であるかは, 並列化に用いる構文や言語の機能にも依存する. OpenMP の tasks 構文, Cilk の spawn, TBB のタスク並列などの, タスク並列をサポートするプログラミング言語では, eval 関数中の再帰呼び出し毎にタスクを生成する記述を効率よくサポートする. それらの再帰呼び出しは実際に並列に (時間的にオーバーラップして) 実行されるか否かはわからないが, 少なくともそうなる可能性がある以上, 局面データはタスク生成時にコピーするのが自然である. これは再帰呼び出し時に局面全体を, 構造体の値渡しを使ってコピーできればそれが手っ取り早い. 過去に Cilk によるチェスプログラムの並列化 [3] の際にもこの方式が用いられた.

局面データのコピーオーバーヘッドが問題になる時は, ある程度末端に近いところ—オセロの詰み探索では, 空きマス数により, 比較的容易に末端への距離が予想できる—では undo 方式に切り替え, 同時に再帰呼び出しの並列実行をやめてしまうのが自然である. 局面コピーのオーバーヘッドの大小にかかわらず, 最終局面近くで逐次実行に切り替えることは, オーバーヘッド削減に貢献することが多いので, そのついでに逐次探索 + undo 方式に切り替える.

MPI や UPC のような SPMD モデルや, Pthreads でコア数分のスレッドを生成する場合など, プログラマがタスクとスレッドのマッピングを明示的に管理するモデルでは, それらのスレッドごとに一局面分のメモリを用い, スレッド内の逐次実行は undo 方式を用いるのが自然である. 実際に記述したものの比較は次号で行う.

4.2 コピー方式に伴う問題

Andersson によるプログラムを, 局面のコピー方式で並列実行する際, 実際のプログラム書き換えに当たって問題となるのは, 以下の点であった.

- 上で述べた変数のうち, 配列および構造体—盤面の配列 (board), 空マスのリスト (EmHead および Ems), 各穴に残された空マス数の偶奇 (RegionParity)—は, **大域変数**として, プログラム中に一つだけ確保されており, 関数への引数としては渡されていない. 従ってタスク生成時にコピーするとはいっても, 探索関数自身の変数参照部分の変更も必要である.

- 空マスのリストは, リンクリストとして表現されている. リンクリストは, 前後のセルへのポインタを用いてセルがリンクされており, 従って, 局面を全体をコピーしても空きマスのセルは依然として共有されたままである.

- 局面の評価が, 深さに応じて呼び出される 3 つの関数 (NoParEndSolve, ParEndSolve, FastestFastEndSolve) で実現されており, 並列化に際してのコードの書き換えが多い

結局これらの問題を解決するには, 指し手の順序付けを忠実に保存しながらも, 局面のデータ構造や局面の評価などは全面的な書き換えをするのが, むしろ近道であった.

4.3 並列化のもととなるコード

上記の方針に基づいて書かれた逐次プログラムの概形を示す. 局面全体を一つの構造体で表現したものが以下である.

```
typedef struct game_state {
  char color_to_move;           /* 手番 (白または黒)*/
  char last_move_was_pass;      /* 最後の手がパス? */
  char n_empties;               /* 空マス数 */
  char disc_diff;               /* 手番マス数 - 相手マス数 */
  unsigned int region_parity;   /* 各穴の残り空マス数の偶奇 */
  square_list_node empty_squares[MAXEMPTIES]; /* 空マスリスト */
  unsigned char board[91];      /* 盤面 */
} game_state, * game_state_t;
```

空マスのリストは, 上記の構造体を簡単にコピーできるよう, ポインタを直接用いずに, 上記 empty_square 中の添字で互いをリンクする.

```
typedef struct square_list_node {
  char square;                  /* マス番号 */
  char hole_id;                 /* 穴番号 */
  char pred_idx;                /* リスト中の前の要素の添字 */
  char succ_idx;                /* リスト中の次の要素の添字 */
} square_list_node;
```

これにより, g, g' を game_state へのポインタとしたとき, 代入文*g' = *g で, 局面のコピーができる.

評価は, 与えられた盤面そのものを評価する関数 eval (図 4.3) と, そこから呼び出される, 「与えられた盤面から一手動かしてその盤面を評価する」関数 move_and_eval から構成される.

図 4.3 において,

```
int n_moves = gen_moves(g, moves);
```

は, 配列 moves に, 合法手 (マスの番号もしくはパスに対しては −1) を, 手番にとって有望な手が先に来るように整列した状態で格納する. eval のポイントは, 再帰呼び出し時に

```
int e = -move_and_eval(*g, i, moves[i], -beta, -alpha);
```

として, 局面の構造体渡しにより局面のコピーを生成していることである. この再帰呼び出しは, g に i 番目の手 moves[i] を適用した盤面を評価する. 実質的には以下だけの関数である.

```
/* es_idx は石を置くマスの位置もしくは-1 (パスの場合).
   指定された手を打ち, 打った後の局面の評価値を返す. */
int move_and_eval(game_state gg, int move_idx, int es_idx,
                  int alpha, int beta) {
  game_state_t g = &gg;
  apply_move(g, es_idx);
  return eval(g, alpha, beta);
}
```

```
/* 局面 g の評価値を返す */
int eval(game_state_t g, int alpha, int beta) {
  int moves[MAXEMPTIES];
  int n_moves = gen_moves(g, moves);
  if (n_moves == 0) {
    return g->disc_diff;
  } else {
    int i;
    for (i = 0; i < n_moves; i++) {
      int e = -move_and_eval(*g, i, moves[i], -beta, -alpha);
      if (e > alpha) {
        alpha = e;
        if (e >= beta) break;
      }
    }
    return alpha;
  }
}
```

図 5: eval 関数

apply_move は盤面 g をその場で書き換える．ただし 3.4 節で述べたとおり, 再帰呼び出しの末端 (leaf) 付近のオーバーヘッドを減らすことは重要である．move_and_eval において一手動かした後, 残りの空きマスがひとつしかなければ eval を呼び出さずにそのまま評価を行う．ただし, Andersson のコードをよく見ると, それを行っているのは, その手がパスでなかった場合だけである．特に必然性はないが, ここでは両者が完全に同一の探索木を探索したことを確かめるため, そこを含めて忠実に再現している．それを施したコードは図 6 のとおりである．

以上の方針で書かれたプログラムを, 基本コード (base code) と呼ぶ．次号ではこの基本コードを元に並列化を施す．

5 逐次性能の測定

今回は Andersson のコードと基本コードの逐次性能測定を行う．評価環境は以下である．

- CPU: Intel Nehalem-EX (E7540) 2.0GHz (6 core/12 スレッド × 4 ソケット)
- L3 cache: 18MB
- memory: 24GB DDR3

盤面には, Andersson の提供する, The FFO endgame test suite (http://radagast.se/othello/ffotest.html) を用いた．図 7 がそれらの局面である．

表 1 は, Andersson のコードおよび基本コードが探索したノード数を 10^6 単位で示したもので, 両者が 1 の位まで一致することは確認している．ここでの探索ノード数は, つまりは基本コードにおけ

```
/* 局面 gg から，手 (es_idx) を打ちその局面の評価値を返す */
int move_and_eval(game_state gg, int move_idx, int es_idx,
                  int alpha, int beta) {
  game_state_t g = &gg;
  apply_move(g, es_idx);
  if (g->n_empties != 1 || g->last_move_was_pass) {
    return eval(g, alpha, beta);
  }
  /* 残りは一手のみ */
  /* 相手がそこに置けた場合 */
  if (apply_move(g, first_move(g))) return -g->disc_diff;
  /* 相手がパス */
  apply_move(g, -1);
  /* 相手のパス後，自分がおけた場合 */
  if (apply_move(g, first_move(g))) return g->disc_diff;
  /* 相手のパス後，自分もパスで終了 */
  return -g->disc_diff;
}
```

図 6: move_and_eval 関数

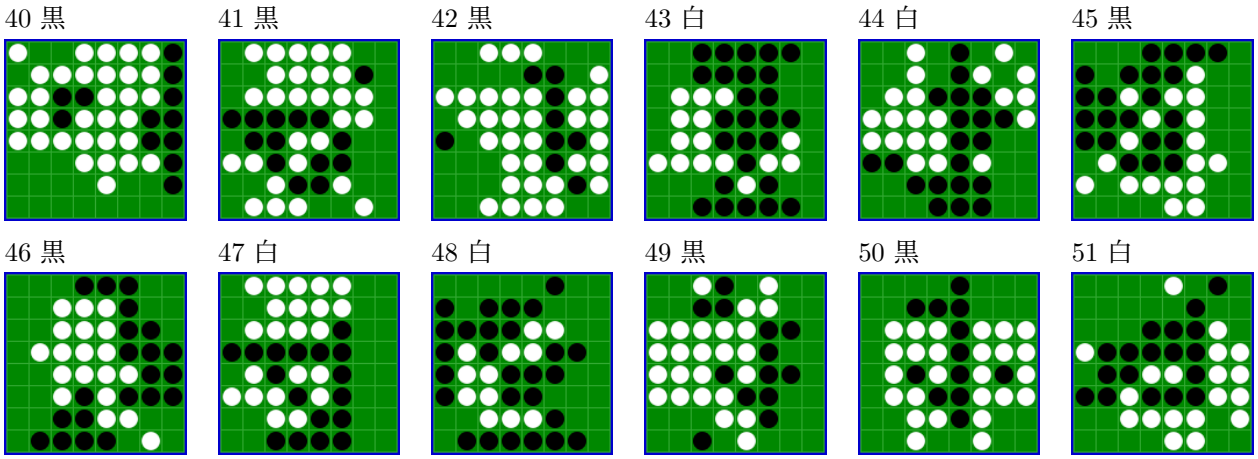

図 7: 評価に用いた盤面と手番 (http://radagast.se/othello/ffotest.html より). いずれも直前の手はパスではない.

表 1: 問題番号と探索ノード数

問題番号	探索ノード数 ($\times 10^6$)
40	53
41	379
42	328
43	341
44	347
45	4552
46	3112
47	1266
48	11046
49	22586
50	15637
51	14084

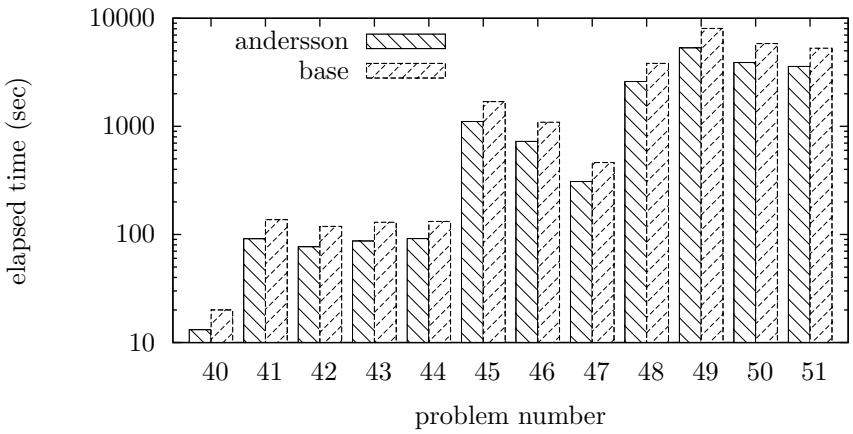

図 8: Andersson のオリジナルコード (andersson) および, 今回作成したコード (base) が各問題を解くのに費やした時間

る eval が呼び出された回数ということで, 3.4 節で述べた末端付近の最適化が施された結果, eval を呼び出さずに評価されたノードは数えられていない.

図 8 はそれぞれの問題に対して両者が費やした時間 (秒単位. 対数軸), 図 9 は両者の比を示したものである. 総じて, Andersson のコードは 600 万ノード/秒, 基本コードは 400 万ノード/秒の局面探索速度を持っており, 50%のオーバーヘッドがある. 探索するノードはその数, 順番とも同一なので, オーバーヘッドの原因は, 局面の生成や参照の基本速度の差によるものである. 一つの要因は 4.1 節で述べたとおり, 局面をコピーしていることに起因していると考えられるが, 一方, 基本コードとほぼ同様のデータ構造で, undo 方式をナイーブに実装したコード自身はむしろ, 基本コードよりも遅くなっており, より詳細な調査及び基本コードの最適化は次号で述べる予定である.

6 まとめ

タスク並列の題材として, オセロの詰み探索の並列化を試みる準備として,

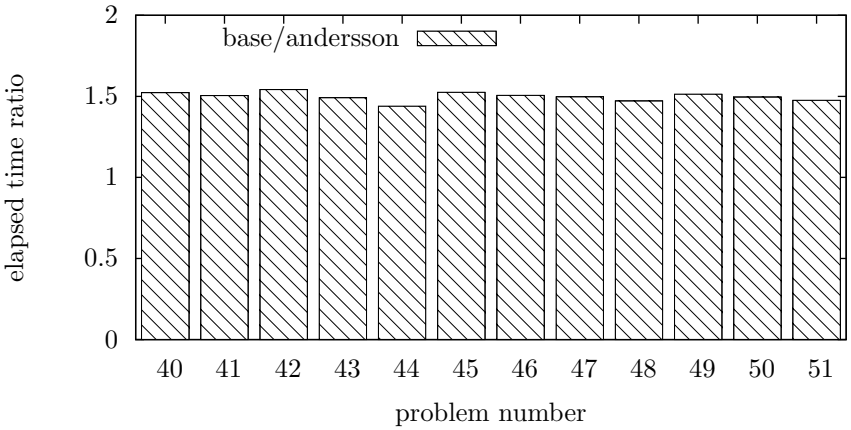

図 9: Andersson のオリジナルコード (andersson) と今回作成したコード (base) の経過時間の比

- Andersson による move ordering と $\alpha\beta$ 法を忠実に再現し,
- 次局面を生成するのに局面コピーを用いた

逐次コード (基本コード) を作成し, その逐次性能を評価した. 結果, Andersson の 50%程度のオーバーヘッドの基本コードが得られた. 次号以降で, 各種の並列プログラミングの構文を用いて, 基本コードを並列化し, その速度を評価していく.

参考文献

[1] Gunnar Andersson. Andersson's endgame.c. http://radagast.se/othello/endgame.c.

[2] Gunnar Andersson. Zebra. http://radagast.se/othello/zebra.html.

[3] Don Dailey and Charles E. Leiserson. Using cilk to write multiprocessor chess programs. In *Advances in Computer Games 9*, 2001.

[4] Donald E. Knuth. *An Analysis of Alpha-Beta Pruning*, chapter 9. Cambridge University Press, 2000.

[5] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975. reprinted as [4].

[6] Wikipedia. オセロ (遊戯). http://ja.wikipedia.org/wiki/AF