2016

たこっと

第 4 回将棋電王トーナメント PR 文書

内宮大志　大場寿仁　瀧川正史

概要

「たこっと」は，電王戦を見てコンピュータ将棋に興味を持った筆者らが，フルスクラッチで実装した（している）将棋プログラムです。

Web 上の解説記事や論文，Stockfish, Apery, やねうら王, Bonanza 6.0 のソースコードを 参考にしています。

第 26 回世界コンピュータ将棋選手権（WCSC26）から進化した部分を中心に記述します。

WCSC26 時点のたこっとは実装が間に合わず，SEE や 1 手詰みルーチンなど現代の将棋プログラムに必要な機能が不足していました。また，バグもたくさんありました。例えるなら，F1 のエンジンを搭載した軽自動車みたいな状態だったといえるでしょうか。

この文書の作成時点で 1 手詰みルーチン以外は必要だと思われる機能の実装が完了しています。把握している不具合も全て修正しました。ようやくまともなシャシーを手に入れ，本来のポテンシャルが発揮できる状態に近づいたといえます。

第 4 回将棋電王トーナメントでは持てる力を全て解放したたこっとの戦いぶりをご覧ください。

※ WCSC26 のたこっとについては第 26 回世界コンピュータ将棋選手権アピール文書をご参照ください

特徴

- 各種処理が AVX2 命令で実装されていて高速
- AVX2 命令を適用しやすいデータ構造（非ビットボード）
- Stockfish 風の探索アルゴリズム

ライブラリの使用について

- この文書の作成時点で使用しているライブラリは Apery の評価関数バイナリのみです
- 今後，やねうら王の学習データと学習ルーチンを使用し，評価関数バイナリの学習に挑戦する予定
- 定跡バイナリの使用に関しては未定（念のためにライブラリ使用申請しています）

※ ライブラリを提供していただき，Apery の平岡様，やねうら王の磯崎様には感謝いたします
※ 対局に使用する将棋プログラム自体にライブラリを使用する予定はありません

第一章「こんにちはバイトボード」

※　序章は WCSC26 アピール文書をご参照ください

来る日も来る日も指し手生成ルーチンのベンチマークを取っていました。どんどん速くなるので，あまりの楽しさに将棋のことはどうでもよくなっていました。

ある日，歩を打つ指し手の生成ルーチンの実装に取り掛かりました。打ち歩詰めを回避するために駒の利きを調べる必要があることに気が付きました。

「駒の利きを調べるのって面倒～」

面倒なだけでなく，計算負荷もそれなりにあります。しかし，指し手生成のパフォーマンスは犠牲にしたくありません。少ない計算量で簡単に利きを求める方法がないかと考え続けました。

それはもうずいぶん考えました（実験もしました）。

「なんか出た～」

そうです。思いついてしまったのです。

バイトボードの誕生です。

ついでに将棋のことも思い出し，止まっていた開発も進み始めました。

続く

並列化

CPU の動作クロックが頭打ちになりつつある現在，処理速度を向上させるには並列化するしかありません。既存の将棋プログラムでも

- マルチスレッド化
- PC のクラスタ化

は取り組まれています。

しかし，命令レベルの並列化は遅れているように感じます。たこっとは SIMD 命令を活用し，命令レベルの並列化に取り組んでいます。

現在の Intel CPU には AVX2 命令が実装されています。一度に 256bit のデータを処理できる SIMD 命令です。また，今後は 512bit 幅の AVX-512 命令も実装される予定です。

※　AVX2 命令や AVX-512 命令に関しては下記 URL を参照してください
https://software.intel.com/sites/landingpage/IntrinsicsGuide/

ここでは AVX-512 命令まで視野に入れ，たこっとで実装した「バイトボード」について簡単に解説します。

バイトボード

※　バイトボードの基本については WCSC26 アピール文書を参照してください

今回はバイトボードの応用について取り上げます。

WCSC26 のときに下図のような飛び駒（龍，馬，飛，角，香）の利きを調べる方法を掲載しました。

```
   9   8   7   6   5   4   3   2   1
+-------------------------------------+
| 20   ・   ・   ・  04   ・   ・   ・  28| 一
|  ・  19   ・   ・  03   ・   ・  27   ・| 二
|  ・   ・  18   ・  02   ・  26   ・   ・| 三
|  ・   ・   ・  17  01  25   ・   ・   ・| 四
| 12  11  10  09  Tg  13  14  15  16| 五
|  ・   ・   ・  29  05  21   ・   ・   ・| 六
|  ・   ・  30   ・  06   ・  22   ・   ・| 七
|  ・  31   ・   ・  07   ・   ・  23   ・| 八
| 32   ・   ・   ・  08   ・   ・   ・  24| 九
+-------------------------------------+
```

```
tmp1    = proponentByteBoard.Pack(TABLE::shuffleCrosses[squareTarget]);
tmp2    = proponentByteBoard.Pack(TABLE::shuffleDiags[squareTarget]);
line1   = _mm256_permute2x128_si256(tmp1, tmp2, 0x20);
```

同様に相手側の opponentByteBoard も Pack() し，これを line2 とします。この line1（自駒）と line2（相手駒）には Tg へ利きがある可能性の PieceBit が桂馬以外すべて含まれています。別途 Tg に利きのある桂馬の個数も調べておくこととします。何かに気が付かないでしょうか？

そうです。Tg のマスにおける駒の取り合いを調べることができるのです。つまり Static Exchange Evaluation (SEE) を実現できるのです。

一例として取り合いに参加する自駒の銀を処理する疑似コードを掲載します。

```
/* eight には中心に利きのある 8 近傍の PieceBit だけが格納される */
eight       = _mm256_and_si256(line1, TABLE::maskPackedPieceBitEights[squareTarget]);
tmp         = _mm256_set1_epi8(PIECE_BIT_SILVER);
pieceMask   = _mm256_cmpeq_epi8(eight, tmp);
moveMask    = _mm256_movemask_epi8(pieceMask);

if (moveMask) {
    _BitScanForward(&index, moveMask);
    line1   = _mm256_andnot_si256(TABLE::maskPackedIndices[index], line1);

    /*
        ここを通過する場合，squareTarget に利きがある銀が存在する
        また，取り合いに参加した銀が line1 から除去される
    */
}
```

実際にはもう少し工夫が必要ですが，電王トーナメントに向けてこのアルゴリズムをベースに SEE を実装しました。

もうビットボードに負い目はありません。