高速将棋ライブラリ OpenShogiLib の作成

田 中 哲 朗[†]　副 田 俊 介[††]　金 子 知 適[††]

将棋に関する研究は近年，大変な進歩を遂げているが，一からプログラムを作成して研究を開始することが難しくなっている．

そこで我々はコンピュータ将棋研究の基盤となることを目指して将棋ライブラリ OpenShogiLib の作成を開始した．評価関数，手生成器，利き管理等の部品が何種類も用意されているため，条件を変えた様々な実験を行なえる．また，C++の template 機能を有効に使うことにより，速度を損なわずに柔軟に組み合わせて使うことが可能になっている．

OpenShogiLib– construction of an efficient shogi library

Tetsuro Tanaka,[†] Shunsuke Soeda[††] and Tomoyuki Kaneko[††]

Shogi is a very interesting target in game programming research. But it is not easy to make efficient shogi programs.

We started to make a shogi library named OpenShogiLib, which is aimed to be an infrastructure of computer shogi research. It involves many components such as evaluation functions, move generators, effect tables, which can be flexibly composed, and researchers can easily make experiments with this library.

1. はじめに

コンピュータ将棋に関する研究を新たに開始する場合には以下の点が問題になる．

- 千日手，二歩，打ち歩詰め等，例外的なルールが多いため，ルール通りにプレイする手生成プログラムを作るだけでも労力を要する．
- 高速なプログラムを作るのは困難である．また，ある環境で高速なプログラムを作ったとしても，別な環境 (64 ビット化等) でも高速に実行できるかどうか実際に走らせてみないとわからない．
- 標準的な実装が公開されていない．新しいアイデアがあっても，すでに十分強いプログラムを持っている人しか意味のある実験をおこなうことができない．

gnushogi のようにソースの公開されたプログラムはいくつかあるが，実験をおこなう際には，プログラム全体に手を入れなくてはいけないことが多い．

そこで，我々はコンピュータ将棋研究の基盤となることを目指して，ソースの公開を前提にした将棋ライブラリ OpenShogiLib の作成を開始した．評価関数，手生成器，利き管理等の部品を何種類も用意し，組み合わせて使うことにより，条件を変えた様々な実験を行なえるように設計している．

OpenShogiLib は以下の特徴を持つ．

- 駒の動きなどに関する知識は 1 箇所で記述して，バグの発生を最小限におさえつつ，C++の template の機能を使い，プレイヤや駒の種類に応じた特殊化した高速な関数を生成する．
- 将棋盤，手生成器などを，template 引数で扱うことにより，複数の実装を切り替えて使うことを可能にしながら，仮想関数呼び出しによる速度低下を起こさない．
- unit test 用のコードも提供されるので，自分で実装を追加する際のミスを押さえることができる．
- 研究用なので，主記憶やキャッシュメモリが少ない古い環境ではなく，最新の環境での実行を想定している．
- 現状の C++コンパイラの template への対応状況が遅れているため，通常のプログラムよりもコンパイル時間の面では不利である．実行速度の速い環境でコンパイル作業をおこなう必要がある．

以下では，OpenShogiLib の特徴的な実装に関して説明し，簡単な性能評価をおこなう．

2. 手番，駒の種類による特殊化

2.1 手番による特殊化

将棋のプログラムには手番によって非対称な処理をおこなう場合が多い．たとえば，あるマス (x,y) が成れる場所かど

† 東京大学情報基盤センター
Information Technology Center, The University of Tokyo
ktanaka@ecc.u-tokyo.ac.jp
†† 東京大学大学院総合文化研究科
Graduate School of Arts and Sciences, The University of Tokyo
{kaneko,shnsk}@graco.c.u-tokyo.ac.jp

うかは，先手では y<=3 という条件式で判定して，後手では 7<=y という条件式で判定する．変数 pl が先手で 1, 後手で-1 になるとすると，(5-y)*pl>=2 という一つの関数で判定できるが，乗算を含むので実行時間の上では不利となる．

YSS[1] では同じ関数に対して，先手用と後手用と特殊化した関数を手で書いて用意している．また我々の以前の研究では，C 言語風の言語で書いたプログラムを，部分計算器により特殊化して，特殊化した C 言語の関数を作成する方法で，性能の向上を確認したが[2]，特別な処理系を用いることなく，手作業を少なくするために，C++の template 機能を用いた特殊化をおこなうことにした．

今回作成するライブラリでは，手番は

```
enum Player{ BLACK=0, WHITE= -1 };
```

のように，enum 型で定義するので，template 引数に用いることができる．C++言語では enum -> int という型変換は cast 演算子なしに実行できるが，int -> enum という型変換は cast 演算子を使わないとコンパイルエラーが出るので，整数型ではなく列挙型にすると，プログラムのバグをコンパイル時に発見する可能性を高めることになる．

さきほどの，マスの y 座標が成ることが可能な場所かどうか判定する関数は，下のような template 関数の形で書ける．

```
template<Player P>  bool canPromoteY(int);
template<>
inline bool canPromoteY<BLACK>(int y){ return y<=3; }
template<>
inline bool canPromoteY<WHITE>(int y){ return y>=7; }
```

これで，template 引数 P により手番が指定されている環境では，canPromote<P>(y) のように呼び出すことで，適切な呼び出しが，速度の低下なしに実現できる．

手番がコンパイル時に決定していない場所での呼び出しのために，

```
bool canPromotePosition(Player player,Position pos){
  if(player==BLACK)
    return canPromotePosition<BLACK>(pos);
  else return canPromotePosition<WHITE>(pos);
}
```

のような template を使わない関数も用意する．こちらの関数は条件分岐を必要とするため遅いので，速度を重視する場合は可能な限り template 番の関数を用いる必要がある．

2.2 駒の種類による特殊化

駒の種類は 14 種類だが，EDGE(盤外) と EMPTY(空白) を含めて

```
enum Ptype { PTYPE_EMPTY=0, (中略)
  SILVER=13, BISHOP=14, ROOK=15,
};
```

のように列挙型で表現する．駒の種類に関する属性は，

```
template <>
struct PtypeTraits<GOLD>{
  static const bool isBasic=true;
  static const bool canPromote=false;
  ...(中略)
  static const int moveMask=
  DirectionTraits<UL>::mask|DirectionTraits<U>::mask
  |DirectionTraits<UR>::mask|DirectionTraits<L>::mask
  |DirectionTraits<R>::mask|DirectionTraits<D>::mask;
};
```

のように，policy class を用いて定義する．ここで定義された情報は，コンパイル時に参照することが可能になっている．

3. 盤面表現と効率的な実装

3.1 座標の表現

盤面上のマスは 2 次元上の座標 (x,y) で表されるが，相対位置を足したりする操作を考えると，整数値 1 つで表すと便利である．これに対応する型として，enum 型の Position を用いる

```
enum Position{};
```

Position から，x 座標,y 座標を求める際の操作を除算を行なわずに and 演算やシフト演算のみでおこなえるように☆，盤面の幅や高さを 2 の巾乗に切り上げる方法が多く用いられる．YSS[1] では 16*x+y としている．ただし，この場合は座標の差を取った時に，(-1,8) と (0,-8) とが同じ値になってしまう．この問題に対応するために，KFEnd[3] では，17*x+y のような座標系を用いている．この場合は，Position から x 座標,y 座標を求める際に，速度の低下は避けられない☆☆．この 2 つを満たすためには，32*x+y のようにすれば良いが，このようにすると Position の取りうる領域が 8 ビットでおさまらなくなってしまうため，8 ビット単位でのシフトが高速に実行できるプロセッサではメリットを生かすことができない．

そこで，OpenShogiLib では 図 1 のよう Position を16*x+y+1 と定義した上で，座標の差を取る時だけ，

```
int position16to32(Position pos) {
  return (pos&0xf0)+pos;
}
```

という演算 32*x+y の形式に変換する方法を採用した．

```
(A0)  ......................... (00)
(A1)  ......................... (01)
(A2) 92 82 72 62 52 42 32 22 12 (02)
(A3) 93 83 73 63 53 43 33 23 13 (03)
(A4) 94 84 74 64 54 44 34 24 14 (04)
(A5) 95 85 75 65 55 45 35 25 15 (05)
(A6) 96 86 76 66 56 46 36 26 16 (06)
(A7) 97 87 77 67 57 47 37 27 17 (07)
(A8) 98 88 78 68 58 48 38 28 18 (08)
(A9) 99 89 79 69 59 49 39 29 19 (09)
(AA) 9A 8A 7A 6A 5A 4A 3A 2A 1A (0A)
(AB) ...........................(0B)
(AC) ...........................(0C)
```

図 1 Position と盤面の対応

3.2 局面の表現

State に関しては，探索の途中では多くの場合，move を do(実行) して，何かを呼び出した後で undo するという操作

☆ Position を添字としたテーブル参照で求める方法もあるが，現在のプロセッサでは (たとえ 1 次キャッシュ上に載るデータであっても) メモリ参照が余計に入ることによる遅延が無視できない

☆☆ 17 で割ったり剰余を取るのは遅くて話にならない．Position の取りうる値が限られていることを利用すると，シフトと加減算だけで計算することは可能であり，テーブル参照よりは有利なことが多い．

が多い．do, undo 操作で同じチェックをおこなうのを避けるために，do 操作，呼び出し，undo 操作をおこなうための Helper クラスを使う設計とした．

Helper クラスには，以下の例のように，do と undo の間でやりたいことを記述する．

```
// Helper の定義
template<class State, Player P>
struct DoUndoHelper{
  State& state;
  DoUndoHelper(State& s) : state(s){}
  void operator()(Position p){
    nextMoves<P,State>(state);
  }
};
template<Player P,typename State>
void nextMoves(State& state){
// Helper のオブジェクトを作成
// PlayerTraits<P>::opponent で P の相手の手番が得られる
  DoUndoHelper<State,PlayerTraits<P>::opponent> helper(state);
// (略)  move を作成
//
  ApplyMove<P>::doUndoMove(state,move,helper);
// 略
}
```

呼び出す際は，ApplyMove に State, Move, Helper を渡す．ApplyMove は，指手の種類 (打つ手，取る手，動くだけの手) によって，別のメソッドを呼び出す．

```
template<Player P>
struct ApplyMove {
  template <class State, class Func>
  static void doUndoMove(State& state, Move move, Func &f){
    Position from=getFrom(move);
    Position to=getTo(move);
    if (isOffBoard(from)){
      ApplyDoUndoDropMove<P,State>::template
        doUndoDropMove<Func>(state,to,getPtype(move),f);
    }
    else{
      Piece captured=state.getPieceAt(to);
      if (captured != PIECE_EMPTY){
        ApplyDoUndoCaptureMove<P,State>::template
          doUndoCaptureMove<Func>
            (state,from,to,captured,getPromoteMask(move),f);
      }
      else{
        ApplyDoUndoSimpleMove<P,State>::template
         doUndoSimpleMove<Func>
          (state,from,to,getPromoteMask(move),f);
      }
    }
  }
};
```

このようにすることで，do と undo の際に 2 回 move の種類を判定する必要がなくなっている☆.

4. 手生成器

すべての手を生成する手生成器だけを用意すれば良いという考え方もあるが，

- 詰将棋中では，王手をかける手，王手を逃れる手だけを生成すれば良い
- 多くの将棋プログラムでは，手のカテゴリなどによって，優先順位をつけたり，実現確率を変えたりする．直前に動いた駒を取るといったカテゴリに関しては特殊化した手生成器を用いた方が効率が良い．

などの理由があるため，特殊化した手生成器も用意した方が良い．そこで，すべての手を生成する手生成器の他に，

- 指定したマスに移動する手
  そこに敵の駒がある場合は, 敵の駒を取る手
- 指定したマスに利きをつける手
  そこに敵の玉がある場合は，王手
- 指定した駒に敵の利きがある場合に，逃れる手
  王手がかかっている場合には逃げる手

などの特殊化した手生成器を用意している．これは自動的には作れずに，人間が考えてプログラムを作成している．

手生成器を呼び出す場合，典型的には以下のような template 引数を持つ

- Player P
  どちらの手番の手を生成するか
- class State
  どのクラスの盤面に対して手を生成するか.
- class MoveAction
  手生成の際の callback 関数のファンクタ

MoveAction の部分がわかりにくいかもしれないが,

```
struct MoveAction {
 /** コマをとらない Move */
 void simpleMove(Position from,Position to,
        Ptype ptype, bool isPromote,Player p);
 /** コマを取るかもしれない Move */
 void unknownMove(Position from,Position to,
                  Piece captured, Ptype ptype,
                  bool isPromote,Player p);
 /** コマを打つ Move */
 void dropMove(Position to,Ptype ptype,Player p);
};
```

のような callback を持つ．典型的なものは，move を作って配列に入れていくという action だが，ここでいきなり再帰呼び出しをする action を記述することも可能であるし，何らかのチェックをして，チェックに通った時のみ本物の Action を呼び出すというフィルタアクションもかける．

5. 探索・詰将棋

探索アルゴリズムは，MTDF[4] に基づく iterative deepning，alpha beta 等を試験的に実装している．どちらも，激指[5] で提案されたカテゴリに基づく実現確率探索に対応し

☆ 激指[5] のように undo の際には，アドレスと old value が詰まれた trail stack から書き戻すようにすると，判定の必要はないがメモリ参照は増える

ており，どのカテゴリをどの順番で使うかを探索クラスに与える template 引数により切り替えることができる．例えば，確率固定で全ての手を生成するカテゴリのみを与えれば全幅探索となる．また，指手生成と確率付与を同時に行なうカテゴリと，生成された指手に対して分類と確率付与のみを行なうカテゴリの両方が利用可能である．取り返しは専用の指手生成関数が利用可能なので前者のタイプにするなど，各カテゴリの性質に応じてどちらのかを決めることができる．

評価関数は，駒の損得を計算する単純なものと，駒の関係に基づくものが用意されている．玉の危険度などはまだ書かれていない．

詰将棋ルーチンとしては定評のある df-pn をベースにして，一部で，df-pn^{+}[6] を取り入れている．また，詰将棋ルーチンで読んだ結果の証明木を保存して，類似局面での詰みの有無を調べる際に利用する仕組も用意されている．しかし，優越関係を利用した効率化などはまだできていない．また，必至探索もまだ作っていない．

6. 評　　価

作成したライブラリの性能に関する評価をおこなう．組み合わせてプログラムを作る際の容易さを評価の尺度としたいが難しいので，現時点での性能に関する評価をおこなう．実験には，Opteron 240(1.4GHz), Turbolinux 8 for AMD64, gcc 3.3.1 を用いた．

まずは，手生成に要する時間を計測する．図 2 の左を初期局面として，深さ 3 まで全探索した際の深さ 3 のノード数は 3,090,402 で，実行サイクルは約 2700 万サイクルである．そのノードですべての手の生成 (のみ) をおこなった場合，作成される手は 492,832,101 個で，実行サイクルは約 53 億サイクルとなる．したがって，1 手の生成に平均 10.8 サイクルかかっていることになる．これは，実験環境においては，1 秒に約 1 億 3 千万手生成しているという計算になる．

次に，1 手を進めて戻すのに有する時間を計測する．さきほどの実験から進めて，深さ 4 まで計測した時の所要サイクルは約 370 億サイクルとなる．1 手進めて戻る操作が，492,832,101 個加わったという計算なので，1 回あたりの所要サイクルは 65.0 サイクルとなる．実験環境においては，手生成と合わせると，1 秒に約 1800 万局面を探索できるという計算になる．

利きをつける手，逃げる手を生成する手生成器も評価するために，図 2 の右を初期局面として，後手が詰みを逃れるかどうか，ハッシュを使わずに最良優先探索もしない単純な詰め将棋ルーチン (後手が逃げる手があるか) の計測をしてみた．この結果，チェックした局面 1 つにつき 350 サイクル程度かかり，1 秒に約 400 万局面を探索できることが分かった．

ただし，ハッシュ，証明数を組み合わせた詰将棋ルーチンの方が探索ノード数が少なくなり，結果的に探索時間が少なくなると考えられるので，実戦で使用する詰将棋ルーチンはこれよりも秒あたりの探索ノード数は数分の 1 のものになっている．

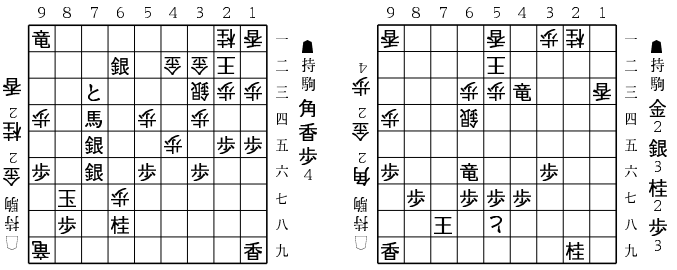

図 2　実験の初期局面

7. おわりに

OpenShogiLib は BSD 風ライセンスのもとで http://gps.tanaka.ecc.u-tokyo.ac.jp/osl/で公開している．このライブラリを使って商用ソフトも自由に作成できることを目指すからである．

OpenShogiLib を使った将棋プログラム「GPS 将棋」は http://gps.tanaka.ecc.u-tokyo.ac.jp/gpsshogi/ で GPL のもとに公開している．こちらは，研究の際のリファレンスとなることを目的としている．派生物をバイナリ配布する場合はそのソースの頒布とソースの自由な利用が義務づけるライセンスに基づいての公開になるため，これをベースに商用ソフトを作るのは向かない．

参 考 文 献

1) 山下宏: YSS – そのデータ構造，およびアルゴリズムについて, 松原仁編著「コンピュータ将棋の進歩 2」, pp. 112 – 142, 共立出版 (1998).
2) 田中哲朗: 部分計算を用いた手生成の高速化, 情報処理学会研究会資料 2000-GI-3, pp. 25 – 32 (2000).
3) 有岡雅章: 将棋プログラム KFEnd における探索, 松原仁編著「コンピュータ将棋の進歩 4」, pp. 18 – 40, 共立出版 (2003).
4) A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Best-first fixed depth minimax algorithms. *Artificial Intelligence*, 87:255–293, 1996.
5) 鶴岡慶雅: 将棋プログラム「激指」, 松原仁編著「コンピュータ将棋の進歩 4」, pp. 1 – 17, 共立出版 (2003).
6) Ayumi Nagai and Hiroshi Imai: Application of df-pn^{+} to Othello endgames, Proceedings of GPW'99, pp. 16 – 23(1999).