5 五将棋をプレイするゲーム AI の強化

MI/CS 実験第二 3a

第 5 回

December 11, 2023

types.hの修正

types.h(841行目あたり)に次のようなコードを加えてみる
これにより**特定の升目に移動する**指し手が比較的簡単に得られる

```
  explicit MoveList(const Position& pos)
   : last(generateMoves<GenType>(pos, mlist)) {}

  explicit MoveList(const Position& pos, Square sq)
   : last(generateMoves<GenType>(pos, mlist, sq)) {
    static_assert(GenType == RECAPTURES || GenType == RECAPTURES_ALL);
  }

  // 内部的に持っている指し手生成バッファの先頭
  const ExtMove* begin() const { return mlist; }

  // 生成された指し手の末尾のひとつ先
  const ExtMove* end() const { return last; }
```

```
void user_test(Position& pos, std::istringstream& is)
{
 for (ExtMove m : MoveList<RECAPTURES_ALL>(pos, SQ_34))
   std::cout << m.move << " " << pos.legal(m.move) << std::endl;
}
```

types.h の修正を反映させないとコンパイルが通らない

- Position のメンバ関数 legal() は指し手が合法かを判定する
  引数は Move

isready コマンド後に user コマンドを入力すると，初期局面から 3 四の升目に移動する指し手を出力する

指し手生成器

```
void user_test(Position& pos, std::istringstream& is)
{
 // 合法手をすべて出力する
 for (ExtMove m : MoveList<LEGAL_ALL>(pos)) // 範囲 for 文
   std::cout << m.move << " " << pos.legal(m.move) << std::endl;
}
```

isready コマンド後に user コマンドを入力すると，初期局面の合法手すべて (14 個) を出力する

LEGAL_ALL の部分を変更すると，生成する指し手を**限定**できる

MOVE_GEN_TYPE は生成する指し手の種類を表す列挙型
(types.h で定義)

指し手の種類の例

CAPTURES_PRO_PLUS_ALL	駒を取る指し手 $\vee$ 歩を成る指し手
NON_CAPTURES_PRO_MINUS_ALL	駒を取らない指し手 $\wedge$ 歩を成る指し手でない
EVASIONS_ALL	王手を回避する手
NON_EVASIONS_ALL	王手の回避でない手
RECAPTURES_ALL	特定の升目への移動への指し手
LEGAL_ALL	合法手すべて

(MOVE_GEN_TYPE の説明続き)

各末尾の_ALL を除去すると，以下の指し手が**除外**される

- 角の不成 (成らず)
- 飛の不成

※ 5 五将棋では歩の不成はない

基本的にメリットがない [1] 不成を除外する

$\longrightarrow$ **探索の効率化**

実装上の注意

- `LEGAL` と `LEGAL_ALL`
  内部的に関数 legal() を呼び出しているため，合法手であることが保証されている
- それ以外
  生成された指し手がすべて合法であるとは限らない
  $\longrightarrow$ **必ず探索部で legal() を呼び出す**

静止探索の実装

指し手を限定したミニマックスを行う

例

- 王手がかかっていないとき：**駒の取り合い**のみ調べる
- 王手がかかっているとき：王手を回避する手をすべて調べる

- 通常探索の末端 (深さ 0) で静止探索を呼び出す
- 評価関数は静止探索で呼び出す
- 王手がかかっているかは position のメンバ関数 **checkers**() で判定できる

search.cpp に関数 qsearch() を定義する

```
// プロトタイプ宣言
Value qsearch(Position& pos, Value alpha, Value beta,
  int depth, int ply_from_root);
```

関数 search() の冒頭を変更する

```
// 通常探索
Value search(Position& pos, Value alpha, Value beta,
  int depth, int ply_from_root)
{
  // 末端では静止探索を呼び出す
  if (depth <= 0)
    return qsearch(pos, alpha, beta, depth, ply_from_root);

  // 以降はほぼ前回と同じ
}
```

関数 qsearch() の実装の一部を紹介 (やや複雑)

```
Value qsearch(Position& pos, Value alpha, Value beta, int depth, int ply_from_root)
{
  /* 中略 */
  MovePicker mp(pos, move_to(pos.state()->lastMove));
  Move move;
  while ((move = mp.nextMove()) != MOVE_NONE)
  {
    // 合法かを確認する
    if (!pos.legal(move))
      continue;
  /* 中略 */
  }
  // 詰みのスコアを返す
  if (InCheck && move_count == 0)
    return mated_in(ply_from_root);

  return alpha;
}
```

時間制御

探索時間の計測結果 (初期局面)

- アルファベータ探索
- 静止探索は未使用

探索深さ	経過時間 (ms)
5	1
6	19
7	37
8	563
9	1509
10	**18890**

- 探索深さが大きくなると，探索局面数が指数関数的に増加する
- ゲーム木の分岐の数は局面の状況によって大きく変わるため，探索時間の事前予測が困難

探索中別のスレッドで時間を管理させる
C++では並列処理の標準ライブラリ **std::thread** が用意されている

`Search::Stop` は探索を中止するフラグ

- `Search::Stop == true` のとき，即座に探索をやめる

関数 search(), qsearch() にこの処理を追加実装する

ムーブオーダリング

候補手の中で**評価値の高そうな**指し手から順に探索を行っていく

これにより探索範囲の下限 (α 値) が大きくなるため，より早い段階で**枝刈り**が発生する (探索効率が上がる)

反復深化

探索深さを徐々に深くしていく手法

探索深さ 1 で思考 $\rightarrow$ 探索深さ 2 で思考 $\rightarrow$ 探索深さ 3 $\rightarrow \cdots$

ムーブオーダリング

探索深さ d で探索終了すると，各合法手に対する評価値が得られる

$\longrightarrow$ 評価値が高そうな指し手がわかる

$\longrightarrow$ 探索深さ $d+1$ で思考する際にはこの順に探索を行っていく

プレイヤの作成

以下の改良が考えられる

- 評価値の変更 (易)
- 評価関数の変更 (易〜標準)
- アルファ・ベータ法 (標準)
- 反復深化 (標準)
- `rootMoves` のオーダリング (標準)
- 静止探索 (難)

これらの改良がうまくいけば `AI` の強化が見込めるが，何か独自の手法を組み込んでみても面白いと思います

第7回の大会について

- 事前にソースコードを提出してもらう (当日は観戦のみ)
- TA がランダムプレイヤと対戦させて動作確認する
- 持ち時間は 1 手 10 秒とする