Functional and logic programming lab 13th report

Yoshiki Fujiwara, 05-191023

1　実装について

1.1　全体の概要

今回の実装では、大きく分けて三つの操作を行った。(1,2) に準ずる場所という書き方をしている箇所は (1,2),(2,1),(1,7),(7,1),(8,7),(7,8),(2,8),(8,2) のことを表している。

1.1.1　定石の導入

https://skatgame.net/mburo/logistello.tar.gz　のリンクにある、logistello という定石を導入した。この作業は友人と行った。まず、logistello 同士の対戦の log の最初の 20 手を取得し、その盤面を出力する。それを int64 の pair にして、csv file にする。csv file は白の盤面 (64bit 整数), 黒の盤面 (64bit 整数), 次に打つべき座標、というデータとなっている。対局データは d3 から始まるものしか存在しなかったため、対称移動したものについてもデータに加えた。

int64 に整形して入れた理由は、hash 化した時の探索が行いやすい形にするためである。同時に board から int64 の pair を生成する generate_bit_from_board 関数も実装した。これで現在の board を int64 に変換し、hash table から検索する。

play 関数が初めて呼び出された時に、hash に値を追加する。Hashtbl という library 関数を用いた。

create_hash 関数は hash と text file を受け取って、text file の中に記述されているデータを hash table にいれる関数である。一回めの loop に入った時に、この関数を呼ぶ。対戦データ盤面を同一の hash table に入れても問題がなかったため、同一の hash table に入れた。

各場所での然るべき操作に入る前に、この hash table をみて、もしデータが入っていれば、探索などをおこなわず、これに決める。

1.1.2　探索

探索については min-max 法を base に用いた。min-max 以外にも、色を変えて max を撮り続ける実装などを行ったが、min-max が一番強かったため、min-max を採用した。

approximate 関数を用いて、n 手先の予測を行った。まず valid_moves 関数を用いて、合法的な手をリストにして出力し、その中の最適手を出力する。最適手のせんたくには、decide 関数を用いる。decide では、value_approximate 関数を用いて、その関数値がもっともよかった手を選ぶものとなっている。

value_approximate 関数では、n-1 手先を読む作業を行う。

1.1.3　枝刈り

この読む作業については、alpha-beta 法を用いた。alpha-beta 法を実現するために、value_approximate 関数の引数に acc を渡し、比べて、その木を切ることができれば、切れるように工夫した。こうすることで、alpha-beta 法が完成する。

また、alpha-beta 法で探索を行うと同時に、探索を行う board の順番も sort した。まず、一番初めの読み、すなわち、decide 関数内で、value_approximate の初めの case となる board については、2 手先までの評価で sort した。それ以外については、合法手の数が少ない順で sort した。

他にももっと深く探索を行う方法や、毎回探索を行う方法も実装してみたが、あまり早くならなかったため、この実装に至った。

1.1.4　評価関数

評価関数は序盤と中盤と終盤に分けて作成した。

序盤については、着手可能数と盤面評価で評価関数を実装した。盤面評価については、端から 1 つ内側の列について低い値を当てて、盤面の中心に高い評価値を当てた。端については取れればとるようにしたいため、端にも正の評価値を当てた。そして、コマについてのその評価値の平均をとった。

着手可能数については、自分の打てる数を調べた。valid_moves で合法手の list を作り、その長さを求めた。

係数については両方の効果が現れるように係数を調整した。

中盤については、盤面評価関数と確定石の個数で評価した。盤面評価については、(2,2) と (1,2)、そしてそれに準ずる場所についての評価値を低く設定した。また、端の評価値

を大きくして、端の隣の列を低くし、真ん中の値を正の値にとった。

確定石の個数については kakutei という関数を用いて実装した。kakutei 関数では 4 端から 2 方向に、合計 8 方向、順に辿っていく。途中で被ってしまうが、係数をつければ関係ないので、途中での被って数えてしまうことは気にせず実装した。端から続いているかどうかの判定は flag を用いて行った。

この 2 つの関数を係数を調整して評価関数を作成した。係数については序盤と同じく、両方の影響が現れる係数比にした。

終盤については、評価関数を自分の石の数にして先読みを行うことで読み切りを実装した。終盤は評価関数の計算が軽く、合法手の数が少ないため、10 手先を読むように工夫した。

評価関数は count 関数を用いた。これは自分の石の数を数える関数である。終盤は数を取りに行く。

2　Play 関数の中身について

上記工夫を行った play 関数について記述する。

9 手以下については定石を確認して、なければ、序盤の評価関数を用いて 6 手読みを行う。

10〜13 手については、定石を確認して、なければ、序盤の評価関数を用いて 4 手読みを行う。

14〜24 手については、中盤の評価関数を用いて 4 手先読みを行う。

25 手以降については終盤の評価関数を用いて 10 手先読みを行う。

3　このオセロのコードについて

3.1　強い点

このオセロは自分から危険な場所に置かないようになっている。というのも、delete_danger で端の隣に置かないようになっているからである。他の人と違ってここで明記しておき、候補から外しておくことで、自ら相手に端を渡すことをしなくなっている。負けに繋がるような手をあまり打たないため、random とは 100 戦 100 勝が可能である。

相手の盤面の評価値を見ると、序盤は良い戦いをしていると思われた。

3.2　弱い点

また、オセロ AI の特徴でもあるが、中盤の読みが強い。https://github.com/uhyo/is-othello を用いて、自分の AI オセロと戦ったが、(1,2) を置かないため、(1,2) - (1,7) を占領されてしまうという弱点があった。

また、お互いに、(1,2) に準ずる場所、(2,2) に準ずる場所を譲り合った場合、粘り負けしてしまうという弱点があった。相手が 6 手先を読んでいる場合に顕著であった。この場所で 6 手読むという戦略も考えたが、候補手が大きくなってしまうため、時間は最後の読み切りの方に使うことにした。

相手の評価値を見ると、終盤の読み切りになった段階で相手の評価値が上がった。やはり、読み手の少なさがあだになってしまっているようである。

定石データはハッシュの容量を気にして、20 手までしか取らなかったが、思っていたよりスムーズに find などが行えたため、もう少し多くの手数をとってもよかったと感じた。実際、同じ定石を入れていた友人はより効果的に定石を使えていた。