オセロの盤面の評価方法の改良

640672I 理科一類 1 年 24 組 服部桃子

[1]授業で作成した改良プログラムの成果と問題点

授業では，盤面上のマスの位置の重要さを表す重みベクトル w を導入し，重みベクトルと特徴ベクトルの内積を盤面のスコアとした。この方法では，評価方法を改良するための手段として，①重みベクトルの成分(NUM_FEATURES)を増やす，②各成分の値を適正に定める，の 2 つがあった。評価関数を改良するにあたり，最初にこれらの方法に従うことを考えた。

まず，重みベクトルの成分を増やした。授業で与えられたプログラムでは，四隅と中央のマスについて特別に評価をしていたが，さらにこれに付け足して，四隅の隣のマス 8 つについての評価を以下のように加えた。

```
int feature3() {
    return board[1][0]+board[6][0]+board[0][1]+board[7][1]+board[0][6]+board[7][6]+board[1][7]+board[6][7];
}
int evaluate() {
    return weight[0]*feature0()+weight[1]*feature1()+weight[2]*feature2()+weight[3]*feature3();
}
```

次に，n 回の selfplay によって求められた重みベクトルの値 w[i]を改良した。得られた重みベクトルを用いて行われた 1000 回の対戦成績は以下のようになった。

n	重みベクトル				対戦成績(/1000 回)		
	w[0]	w[1]	w[2]	w[3]	黒の勝利数	白の勝利数	引き分けなど
0	100	0	0	0	458	285	257
1000	100	97.6	-69.4	-35.1	464	264	272
10000	100	610.2	-112.3	-194.9	467	265	732

表 1. 重みベクトルの改良とそれに基づいた評価関数での対戦成績

表より，重みベクトルを改良しても，黒の勝率はそれほど上がらないことが分かる。これには，オセロというゲームの性質が関係していると考えられる。オセロでは，序盤から中盤にかけては目先の石の数を闇雲に増やそうとしてもあまり効果がない。いくら最初から石を沢山確保していても，打つ手が悪ければ相手に一度にひっくり返されてしまう。最終的な石の数が相手を上回れば良いので，最初の方では目先の石の数にとらわれるのではなく，自分の打てる場所を増やし，相手の打てる場所を減らすことを考えなければならない。よって，盤面上の石の位置を利用したこの評価方法はあまり適切でないと考える。

そこで今回は，改良策として，合法手の数(nmoves)による盤面の評価を組み合わせる方法を試すことにする。

[2]nmoves による評価方法を組み合わせる方法

まず，例えば 50 番目の石が置かれるまでは盤面の評価が nmoves に比例するとし，それ以降は[1]のように盤面の特徴によって評価することとする。

```
int evaluate(int turn, int last_stage) //last_stage=50 とする
{
    int nstones=64; //盤面の石の数
    int c,r;
    for(c=0;c<8;c++){
        for(r=0;r<8;r++){
            if(board[c][r]==0) nstones--;
        }
    }

    if(nstones>last_stage){
        return weight[0]*feature0()+weight[1]*feature1()+weight[2]*feature2()+weight[3]*feature3();
    } else {
        IntPair legal_moves[60];
        return generate_legal_moves(-turn, legal_moves);
    }
}
```

重みベクトルの成分を，先程 10000 回の selfplay で得られた値 w[1]=610.2, w[2]=-112.3, w[3]=-194.9 とすると，1000 回の対戦成績は，黒 600 勝，白 132 勝となり，先程に比べて大きく向上したといえる。

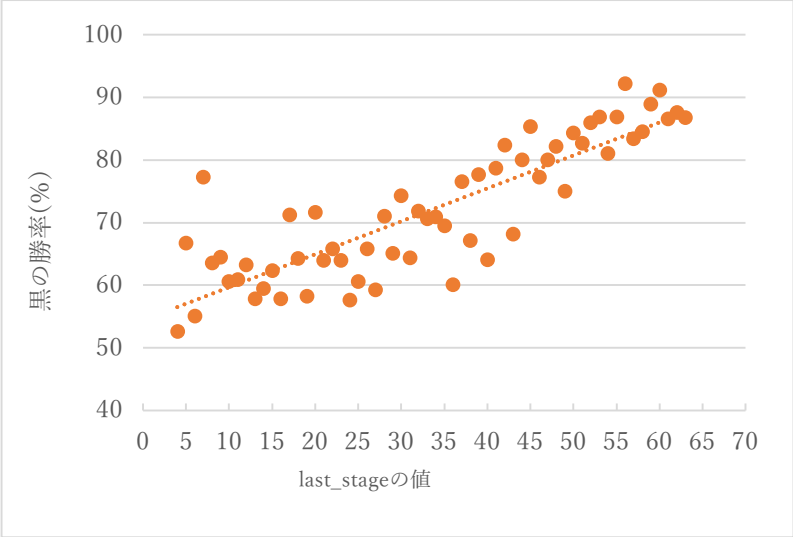
次に，何番目の石が置かれたら終盤と判断できるのかについて考える。ローカル int 型変数 last_stage について，盤上にある石の数が last_stage を超えたら終盤，つまり nmoves ではなく盤面の特徴による評価方法に切り替えると

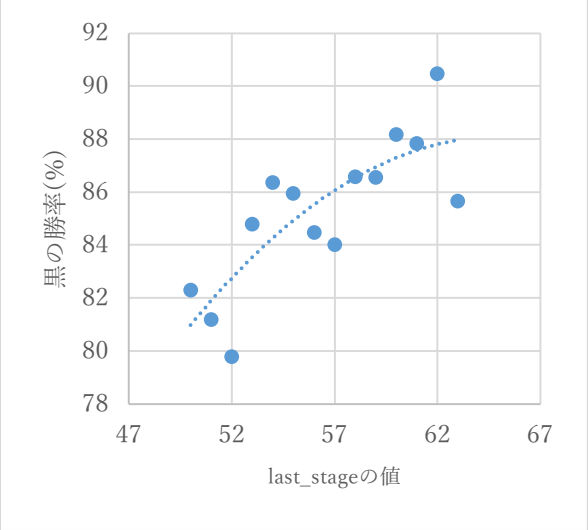
する。プログラムの main 関数の部分は次のようになる。

```
int main(int argc, char **argv)
{
    srand(12345);
    const int human_side = (argc >= 2) ? atoi(argv[1]) : 0;
    int nblackvictory[60], nwhitevictory[60]; //黒が勝った回数,白が勝った回数

    int i,j;
    for(j=0;j<60;j++){
        nblackvictory[j]=0;
        nwhitevictory[j]=0;
        for (i = 0; i < 100; i++) {
            if(selfplay(j+4)==1) nblackvictory[j]+=1;     //selfplay の引数は last_stage
            else if(selfplay(j+4)==-1) nwhitevictory[j]+=1;
        }
        printf("When last_stage=%d, ", j+4);
        printf("black won %d games out of %d games¥n",nblackvictory[j],
                                                        nblackvictory[j]+nwhitevictory[j]);
    }
    exit(0);
}
```

last_stage を 4 から 63 まで変化させたときの黒の勝率を 100 回の対戦で求めると，結果は以下の図 1 のようになった。





(左)図 1.　黒の勝率(100 回対戦させた時)　直線は 1 次近似
(右)図 2.　黒の勝率(1000 回の対戦させた時)　曲線は 2 次近似

いずれも勝率は(黒の勝利数)/((黒の勝利数)+(白の勝利数))*100 で計算。

図 1 より，試行回数が 100 回と少ないため多少の誤差はあるものの，黒の勝率は last_stage の値に概ね比例しているといえる。より正確に last_stage の最適値を求めるために，各 last_stage に対する試行を 1000 回に増やした結果が図 2 である(計算量が膨大になるため，調べる last_stage の値は 49 以上とした)。統計的誤差の影響があるため正確な断定はできないが，勝率を最大化する last_stage の値は 50 台後半から 60 台前半まであたりであると思われ，当初の予想より高い値であると分かった。

〈参考文献〉
「リバーシの基本戦術」　http://www.hasera.net/reversi/kotsu.html