コンピュータ囲碁講習会

山下 宏

2015年

8月1日（当日配布資料＋α）

電気通信大学

西9号 3F AVホール

全体の流れ

- **1日目（8月1日）**
  - **碁盤の表示**
  - **ルールどおり打てるまで**
  - **モンテカルロとゲーム木探索**
- 2日目（8月8日）
  - UCT探索の実装
  - 改良しやすいサンプルの解説
  - GUIソフトで動かす
- 3日目（9月12日）
  - 参加者同士によるミニ大会

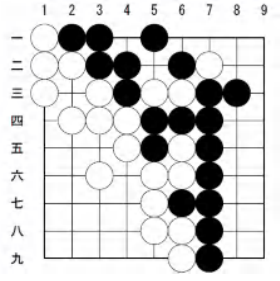
基本はサンプルを実行、解説します

- 1からコーディングすると
  - データ構造の設計とかで悩む
- バグ取りの時間が半端なくかかる
  - 4時間では足りない
- なので簡単なサンプルにそって話を進めます

とりあえずサンプルを実行してみる

- 碁盤を表示する /dentsu/01/go01.c をコンパイル、実行してみてください。
- 碁盤らしきものが表示されましたか？
  - ×が黒石
  - Oが白石
  - ＃は盤外を示しています。
    - ＃が表示されるのは正しくないのですが、その修正はもう少し後で・・・

表示される碁盤（実際はバグで少し違います）



黒：大橋5段
白：Zen

コミ 7目
白、Zenの2目勝ち

2012年3月17日対局

大橋5段が公式に打って
9路で唯一ソフトに負けた碁

データ構造の解説

- 講習では9路盤を使います
  - 実際の碁盤は19x19
- 9路盤は9x9
  - board[9][9]　　が直感的
  - board[11][11]　盤の外に枠があると便利
  - board[y*11+x]　一次元配列を使う
    - 実際は11x11の盤にして、盤外を示す値で囲む
      - 石が囲まれた判定で便利

空白は0、黒石は1、白石は2、盤外は3で表す

定数

- #define B_SIZE　9
  - Board Size の意味
  - 19路盤では19に
- #define WIDTH　(B_SIZE + 2)
  - 実際の横幅。両端の枠を含めて＋2
- #define BOARD_MAX (WIDTH * WIDTH)
  - 碁盤のデータサイズ 11x11 なので121

データ構造の解説

```
int board[BOARD_MAX] = {
  3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
  3, 2, 1, 1, 0, 1, 0, 0, 0, 0, 3,
  3, 2, 2, 1, 1, 0, 1, 2, 0, 0, 3,
  3, 2, 0, 2, 1, 2, 2, 1, 1, 0, 3,
  3, 0, 2, 2, 2, 1, 1, 1, 0, 0, 3,
  3, 0, 0, 0, 2, 1, 2, 1, 0, 0, 3,
  3, 0, 0, 2, 0, 2, 2, 1, 0, 0, 3,
  3, 0, 0, 0, 0, 2, 1, 1, 0, 0, 3,
  3, 0, 0, 0, 0, 2, 2, 1, 0, 0, 3,
  3, 0, 0, 0, 0, 0, 2, 1, 0, 0, 3,
  3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3
};
```

board[y*11+x]

1 <= x <= 9
1 <= y <= 9
で盤上の位置を

x=0,x=10, y=0,y=10 は盤外

例 x=5,y=3 で
board[3*11+5] = 2（白石）
を示す

print_board() 関数

```
int x,y;
const char *str[4] = { ".","X","O","#" };

printf("   ");
for (x=0; x<B_SIZE; x++) printf("%d",x+1);
printf("¥n");
for (y=0; y<B_SIZE; y++) {
  printf("%2d ",y+1);
  for (x=0; x<B_SIZE; x++) {
    int c = board[ (y+1)*WIDTH + (x+1) ];
    printf("%s",str[c]);
  }
  printf("¥n");
}
```

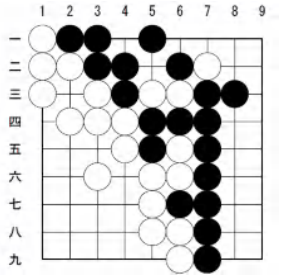
文字列の先頭ポインタ、の配列

for (x=0; x<9; x++)
なのでxは0から8まで

+1 して 1 から 9 まで

サンプルが下の結果になるように修正

```
   123456789
1 0XX.X....
2 00XX.XO..
3 0.0X00XX.
4 .000XXX..
5 ...0X0X..
6 ..0.00X..
7 ....0XX..
8 ....00X..
9 .....0X..
```



囲碁のルールを教える（go02．c）

- 難易度が飛躍的に上がります・・・
- 囲碁のルール
  - 囲めば取れる
    - 相手の石を自分の石と盤端で囲む
  - コウは禁止
  - 自殺手は禁止
    - 打った瞬間に何も取らずに取られる

追加のデータ構造（上下左右の移動量）

- int dir4[4] = { +1,+WIDTH, -1,-WIDTH };

```
int board[BOARD_MAX] = {
  3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
  3, 2, 1, 1, 0, 1, 0, 0, 0, 0, 3,
  3, 2, 2, 1, 1, ↑, 1, 2, 0, 0, 3,
  3, 2, 0, 2, ←, Z, →, 1, 1, 0, 3,
  3, 0, 2, 2, 2, ↓, 1, 1, 0, 0, 3,
  3, 0, 0, 0, 2, 1, 2, 1, 0, 0, 3,
  3, 0, 0, 2, 0, 2, 2, 1, 0, 0, 3,
  3, 0, 0, 0, 0, 2, 1, 1, 0, 0, 3,
  3, 0, 0, 0, 0, 2, 2, 1, 0, 0, 3,
  3, 0, 0, 0, 0, 0, 2, 1, 0, 0, 3,
  3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3
};
```

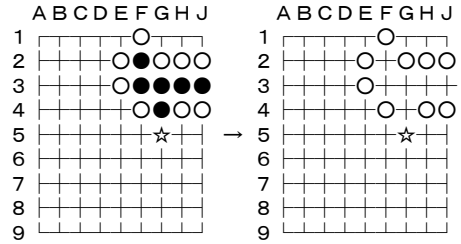
z + dir4[3]
↑
z + dir4[2] ← z → z + dir4[0]
↓
z + dir4[1]

for (i=0; i<4;i++) zz = z + dir4[i];
で4方向を検索できる

（WIDTH=11）

石を取る関数の作成

- 取る石は囲まれているとする
- 縦横に繋がった石を全部まとめて消す



縦横に繋がった石を消す

- 全検索で調べる
  - データ構造が単純なので
  - 欠点は遅いこと
- ある位置を基準にして上下左右の石を探す
  - すでに調べた石は消す
  - これを繰り返す
- 再帰関数、を使うときれいに書ける

再帰関数とは？

- 自分自身を呼ぶ関数のこと
- 下は石を取る関数
  - 短いが複雑な動作をする

```
void take_stone(int tz, int color)
{
  int z,i;

  board[tz] = 0;
  for (i=0; i<4; i++) {
    z = tz + dir4[i];
    if ( board[z] == color ) take_stone(z, color);
  }
}
```

ここで自分を呼んでいる

再帰関数はややこしいけど

- ゲーム木探索では必ず出てくるので避けては通れない

石を取る関数（再掲）

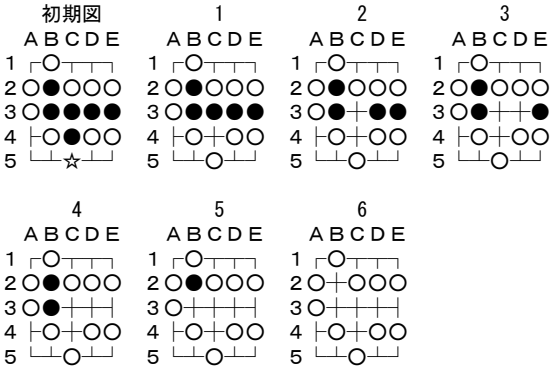
```
void take_stone(int tz, int color)
{
  int z,i;

  board[tz] = 0;
  for (i=0; i<4; i++) {
    z = tz + dir4[i];
    if ( board[z] == color ) take_stone(z, color);
  }
}
```

まず現在の位置の石を消す

現在の位置から4方向を調べる

自分の石が存在すれば、その位置を現在位置にして自分自身を呼ぶ

石が消えていくので必ず終わる



1. まずC4の位置から調べ始める。C4の石を消した後、4方向を D4, C5, B4,C3の順に調べる。(右、下、左、上の順で調べる) この場合、C3が●なのでC3を次の位置にして自分自身を呼ぶ。

2. C3ではC3の石を消した後、D3, C4, B3,C2の順に調べる。 最初のD3が●なのでD3を次の位置にして自分自身を呼ぶ。

3. D3ではD3を消した後、E3, D4,C3, D2を調べる。最初のE3が●なのでE3を呼ぶ。

4. E3では4方向に●がなくなったので、3. に戻る。3. でも4方向に●がないので 2. に戻りB3を見つける。

5. B3を消してB2を見つけて呼ぶ。

6. B2を消した後、どんどん関数を戻っていき、1. まで戻って終了する。

再帰関数の動作確認（fixed／go02．c）

- take_stone()関数の
  - int z,i;　の次の行に下の3行を追加

```
printf("tz=%d¥n", get81(tz));
void print_board();
print_board();
```

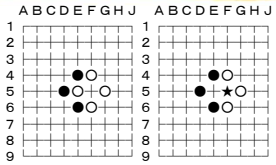
ダメの数と石数を数える関数

- 石を消す関数と同じ再帰関数
- count_liberty()関数を見てください
  - フラグを初期化
  - 石数とダメの数を同時に数える
  - 構造は石を取る関数と同じ

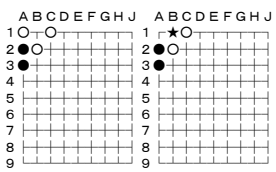
石を置く関数

- put_stone()関数を見てください
  - 石を置く場所の4 方向を調べて、そこにある石の色と石数，ダメ数を調べる
  - 相手の石が取れる場合はコウの可能性があるので位置を覚えておく。
  - 自殺手、コウはエラーを返す
  - 相手石を取り、石を置く
  - 置いた後の石がダメ1、石数1、さらに石を1 個取っていればコウ

コウになる例

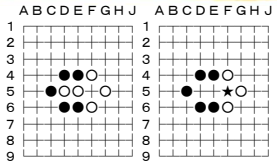


1子取って、その1子がアタリ

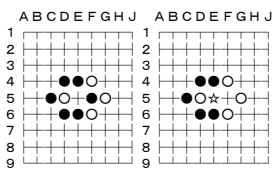


隅でも同じ。

コウにならない例



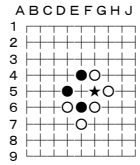
2子取っている



1子取ったが2子取られる

置いた後の石がダメ1、石数1、さらに石を1個取っていればコウ

- 実はこの条件でコウが判定できるか、というのは怪しい。



1子を取ったあとに、単独の石が2子以上取られる、左のようなケースも考えられる。

ただ、この形を中央の白1子を取って、作るのは無理。

長年この判定ルーチンで例外は起きてないので問題ない。経験から。

main() のコメントを2行づつ外す

- 1回目は右上の黒6子を取ります。
- 2回目は左下の白を1子取ってコウになるのを確認します。
- 3回目は白はすぐに左下を取り返しますがコウでエラーを確認します。
- 4回目は右下の黒を1子取りますがコウにならないのを確認します。

ランダムに石を打つ思考ルーチンを作成します。(**go03.c**)

- 追加するデータ構造
  - #define MAX_MOVES 1000
  - int record[MAX_MOVES];
  - int moves = 0;
  - 棋譜と、手数

追加する関数

- 空白の場所を1箇所乱数で返す
  - int get_empty_z()
- 実際に打ってみる。100回試してダメならパスする
  - int play_one_move(int color)

黒白交互に打って対戦

- main()関数
  - パスが2回続けば終了
- 無限に続くのを確認

- 終わらない原因は
  - 「眼」を自分で埋めて自爆してしまう！

眼には打たないようにする

- //if (wall + mycol_safe == 4) return 3; // eye
  - この行のコメントを外す
- 眼の定義
  - 4方向がダメ2以上の自分の石か壁
  - 唯一の囲碁の知識
- 正しく終了するのを確認

srand()関数

- //srand((unsigned)time(NULL));
  のコメントを外すと時刻によってseedが
  変わり毎回違った乱数を生成

  デバッグには毎回同じ乱数が便利

プレイアウトを行う関数（**go04.c**）

- 前回のランダム思考ルーチンそのもの
- put_stone()関数を拡張
  - 眼に打つ手をエラーとするかどうか
  - const int FILL_EYE_ERR = 1;
  - const int FILL_EYE_OK = 0;
  - プレイアウト中のみ、眼に打つ手をエラーにする

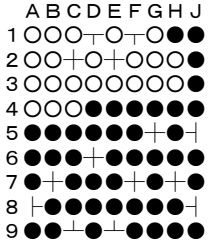
playout() 関数

- 空白を乱数で選ぶ方法が前回から少し変わっています。
- 最初に空白の位置を全部調べて一個ずつ打てるか試します。
- エラーの場合は削除してもう一度選ぶ

地を数える関数（**go05.c**）

- count_score()関数をみてください
  - double komi = 6.5; を追加

プレイアウトの最終局面を判定



例えばこの局面ならば

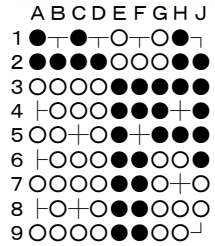
黒の陣地=55（石数=45, 地=10）
白の陣地=26（石数=22, 地= 4）

合計 黒の+29目
コミを含めても黒の勝ち win=1
（+29 − 6.5 = +22.5 > 0）
を返します。

地を数える関数

- playoutでPASS、PASSの局面を想定
- すべての石は活きている、とする
- 同じ色で囲まれている空白は地
  - 白、黒、どちらにも隣接する空白は地ではない
    - セキ

セキ



D1の地点はどちらの石にも接しているので地としない

※ 現在のplayoutではアタリに突っ込む手(D1)を黒も白も打つので、セキは出現しません。

地を数える関数

- 盤上の石の数＋自分の石で囲まれた地の数を数える
  - コミを含めて黒が勝ちなら1を、負けは0
- 1回playoutをして、正しく勝敗判定できているかを確認
- 白地の計算が間違ってるバグを修正

原始モンテカルロ囲碁の作成（**go06.c**）

- 着手可能な場所に30回playoutを行う
- 一番勝率が高い手を選ぶ
  - 黒番では最大の勝率
  - 白番では最小の勝率
- まずは動かしてみましょう

primitive_monte_calro()関数

- playoutを行うと盤面が壊れるので保存
  - 探索開始前の局面を保存(1)
    - 1手打った後の局面を保存(2)
    - playout()
    - (2)を戻す
  - (1)を戻す

原始モンテカルロ囲碁をNegaMaxで（**go07.c**）

- count_score() 関数に1行追加
  - if (turn_color == 2) win = -win;
- 黒番ではそのまま、白番では符号を反転させる
- win = -playout(flip_color(color));
  - 手番を反転させて、符号も反転させる

符号を反転させて何が便利？

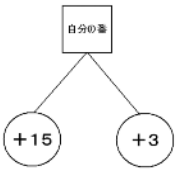
- primitive_monte_calro() 関数での黒番と白番の場合分けがなくなる
- 「常に大きい値の手を選ぶ」と統一できる
- 優勢な方が手番に関わらず、常に大きい値
  - 黒番　黒勝ち +1
  - 　　黒負け　0
  - 白番　白勝ち　0
  - 　　白負け -1

NegaMax形式

- このような書き方は「NegaMax形式」と呼ばれゲーム木探索ではよく使われる
- こういう形なんだと丸暗記
  - 慣れるしかないです
- 重要な点は
  - 再帰で自分を呼ぶときに符号を反転
  - 評価関数（プレイアウト結果）も手番で符号を反転

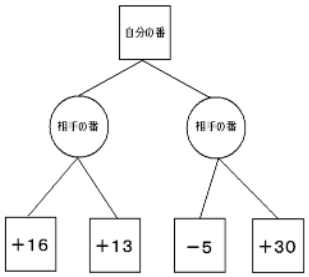
コンピュータはどうやって手を選ぶか

- どちらの手を選びますか？
- 点数が高いほど自分に有利



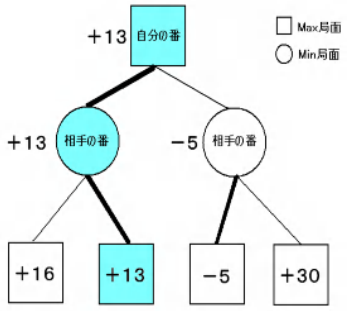
ではもう1手深く読むと？

- 右の手を選べば＋30が得られそうですが…
- 相手も自分に有利な手（点数の小さい手）を選ぼうとする

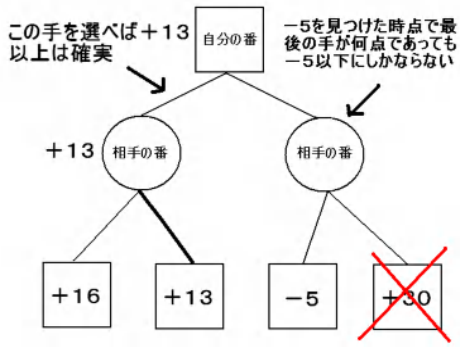


Min-Max法

1. Max局面では最大の手を、
2. Min局面では最小の手を選んで
3. 下から繰り上げていく



αβ法 （読む必要がない手を省略）



NegaMax法

1. 符号を反転させて繰り上げる
2. Max局面、Min局面、どちらでも最大の手を選ぶ
3. 評価関数も手番で反転させる