

# コンピュータ囲碁講習会



山下 宏

2015年

8月1日、8月8日、9月12日

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
- 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

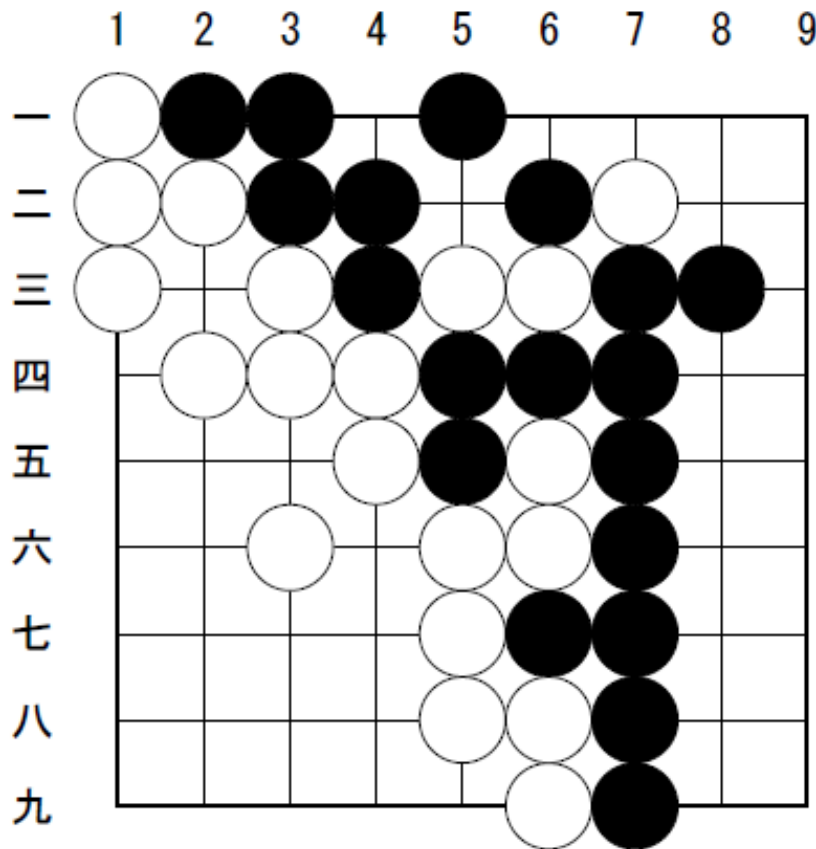
# 基本はサンプルを実行、解説します

- 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
  - ○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  
};
```

$\text{board}[y*11+x]$

$1 \leq x \leq 9$

$1 \leq y \leq 9$

で盤上の位置を

$x=0, x=10, y=0, y=10$  は盤外

例  $x=5, y=3$  で

$\text{board}[3*11+5] = \textbf{2}$  (白石)  
を示す



# print\_board() 関数

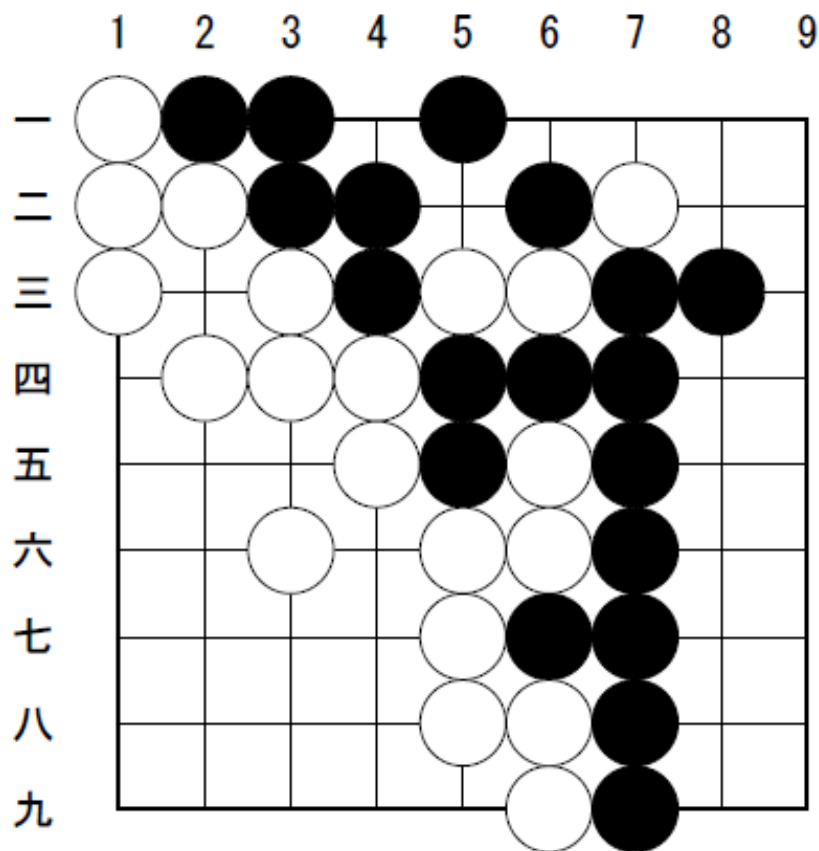
```
int x, y;
const char *str[4] = { ".", "X", "O", "#" };
printf(" ");
for (x=0; x<B_SIZE; x++) printf("%d", x+1);
printf("¥n");
for (y=0; y<B_SIZE; y++) {
    printf("%2d ", y+1);
    for (x=0; x<B_SIZE; x++) {
        int c = board[ (y+1)*WIDTH + (x+1) ];
        printf("%s", str[c]);
    }
    printf("¥n");
}
```

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

	1	2	3	4	5	6	7	8	9
1	0	X	X	.	X	.	.	.	.
2	0	0	X	X	.	X	0	.	.
3	0	.	0	X	0	0	X	X	.
4	.	0	0	0	X	X	X	.	.
5	.	.	.	.	0	X	0	X	.
6	.	.	0	.	0	0	X	.	.
7	.	.	.	.	.	0	X	X	.
8	.	.	.	.	.	0	0	X	.
9	.	.	.	.	.	.	0	X	.



# 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  
};
```

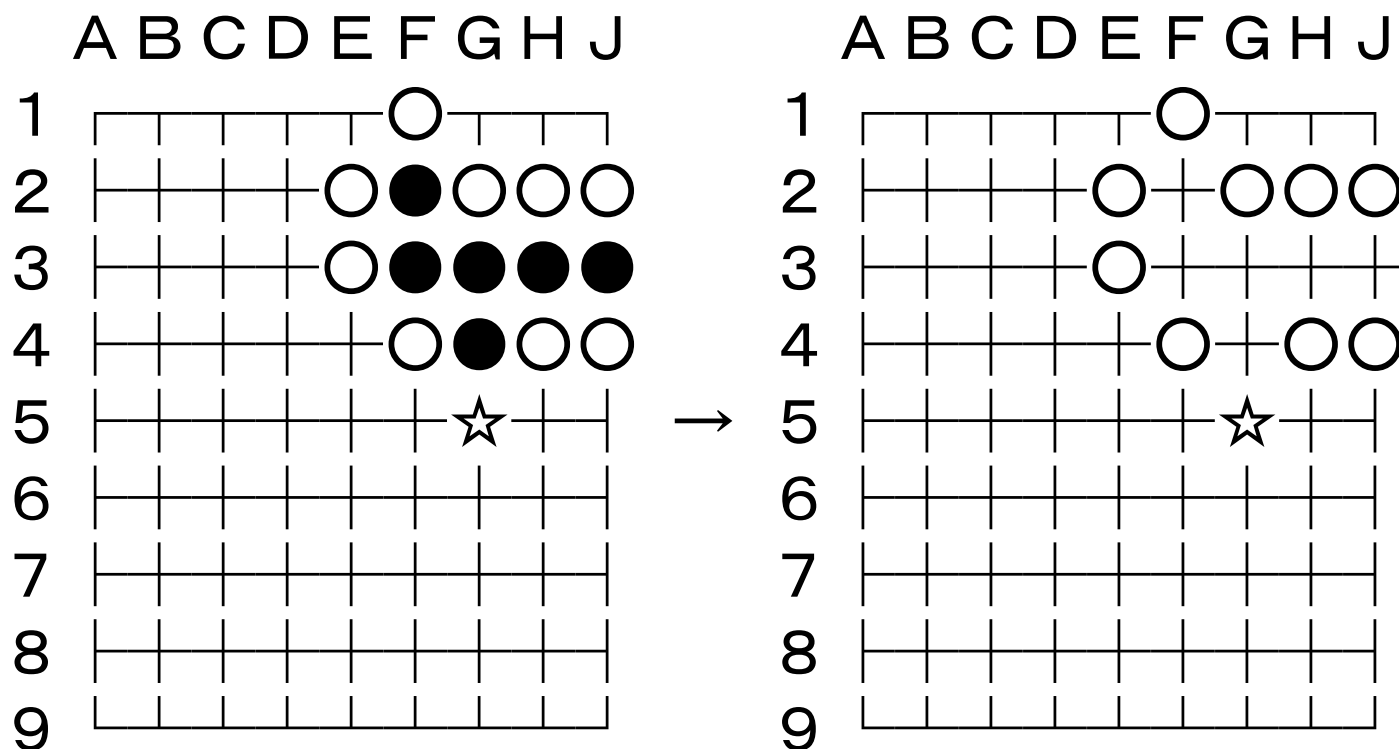
$z + \text{dir4}[3]$   
↑  
 $z + \text{dir4}[2] \leftarrow z \rightarrow z + \text{dir4}[0]$   
↓  
 $z + \text{dir4}[1]$

`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

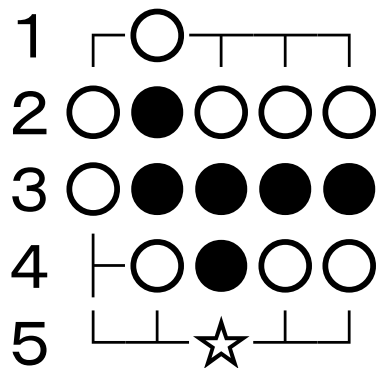
現在の位置から4  
方向を調べる

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

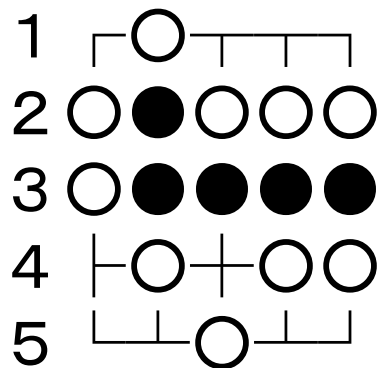
# 初期図

A B C D E



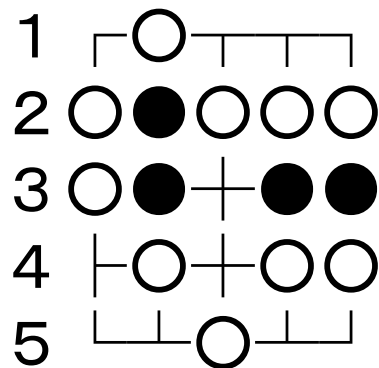
1

A B C D E



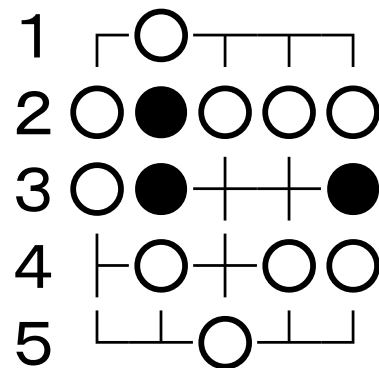
2

A B C D E



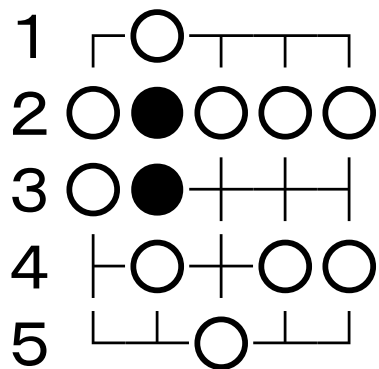
3

A B C D E



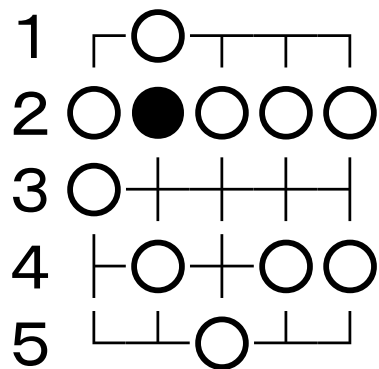
4

A B C D E



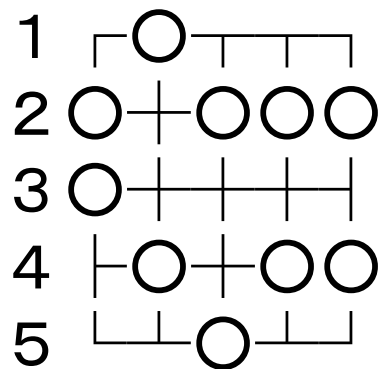
5

A B C D E



6

A B C D E



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();
```

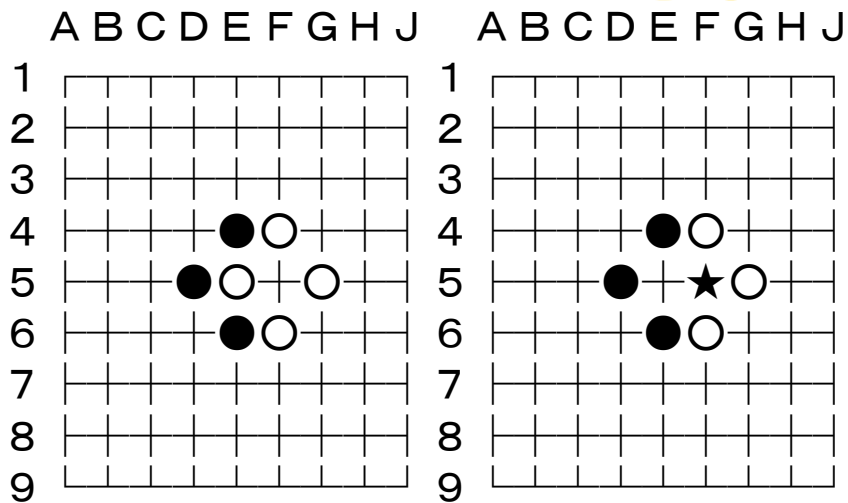
# ダメの数と石数を数える関数

- 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

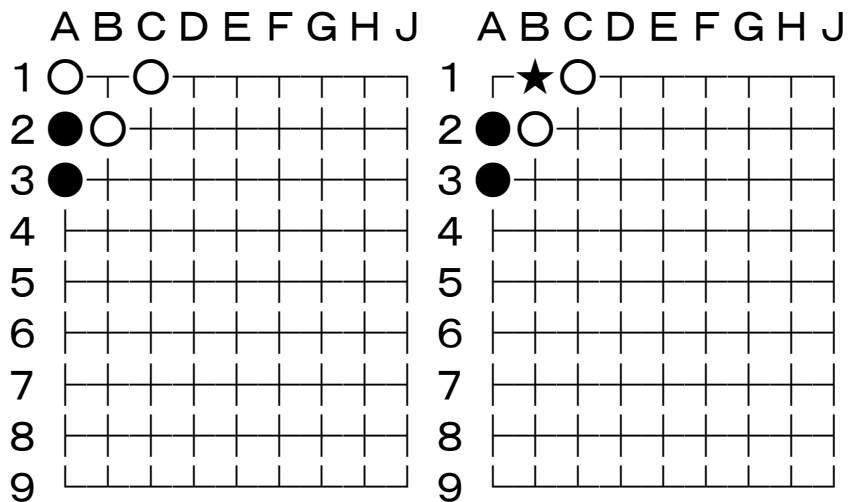
# 石を置く関数

- 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

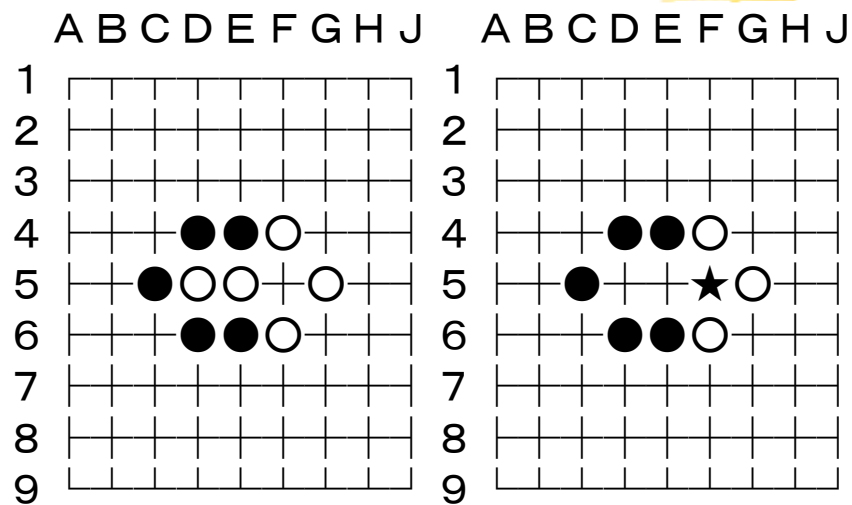


1子取って、その1子がアタリ

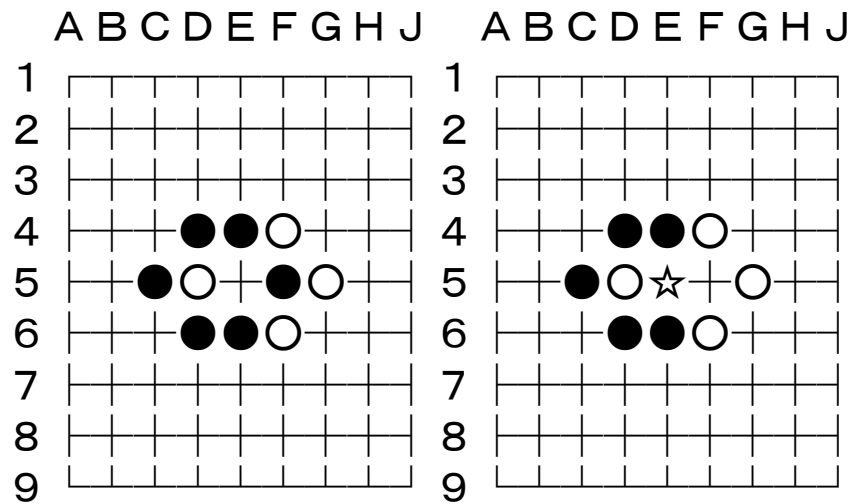


隅でも同じ。

# コウにならない例



2子取っている

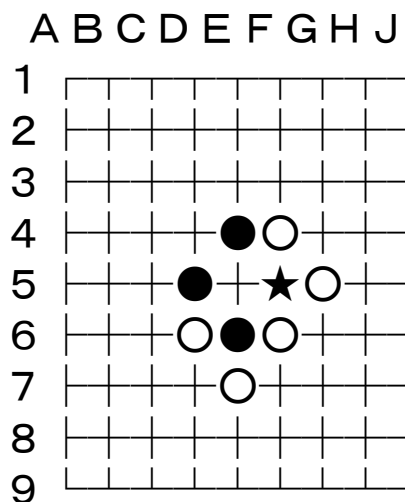


1子取ったが2子取られる



# 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行ずつ外す

- 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()` 関数をご覧ください
  - `double komi = 6.5;` を追加

# プレイアウトの最終局面を判定

	A	B	C	D	E	F	G	H	J
1	○	○	○	┐	○	┐	○	●	●
2	○	○	+	○	+	○	○	○	●
3	○	○	○	○	○	○	○	○	●
4	○	○	○	●	●	●	●	●	●
5	●	●	●	●	●	●	+	●	┐
6	●	●	●	+	●	●	●	●	●
7	●	+	●	●	●	+	●	+	●
8	┐	●	●	●	●	●	●	●	┐
9	●	●	┐	●	┐	●	●	●	●

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

	A	B	C	D	E	F	G	H	J
1	●	┐	●	┐	○	┐	○	●	┐
2	●	●	●	●	○	○	○	●	●
3	○	○	○	○	●	●	●	●	●
4	┐	○	○	○	●	●	●	+	●
5	○	○	+	○	●	+	●	●	●
6	┐	○	○	○	●	●	○	○	●
7	○	○	○	○	●	●	○	+	○
8	┐	○	+	○	●	●	○	○	○
9	○	○	○	○	●	●	○	○	┐

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

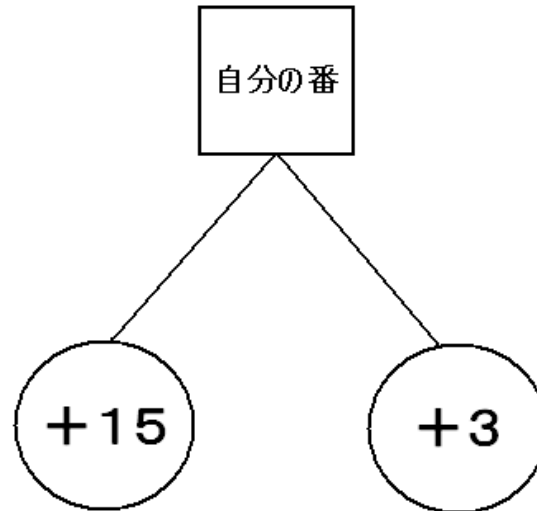
# NegaMax形式



- 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

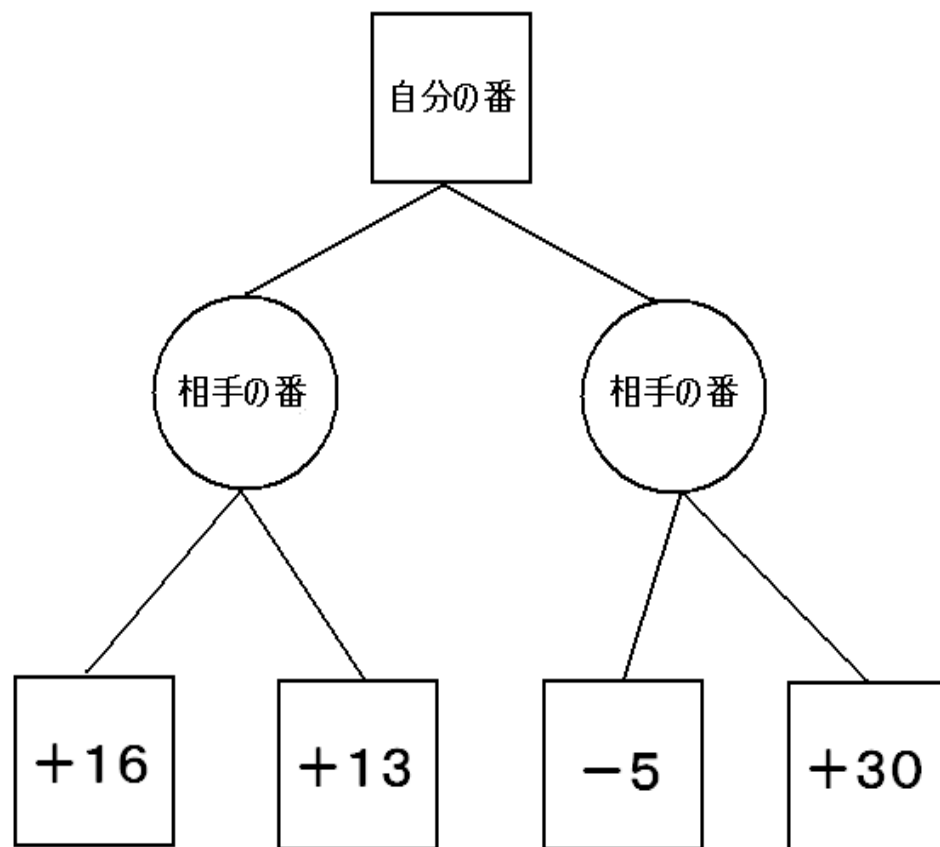
# コンピュータはどうやって手を選ぶか

- どちらの手を選びますか？
- 点数が高いほど自分に有利



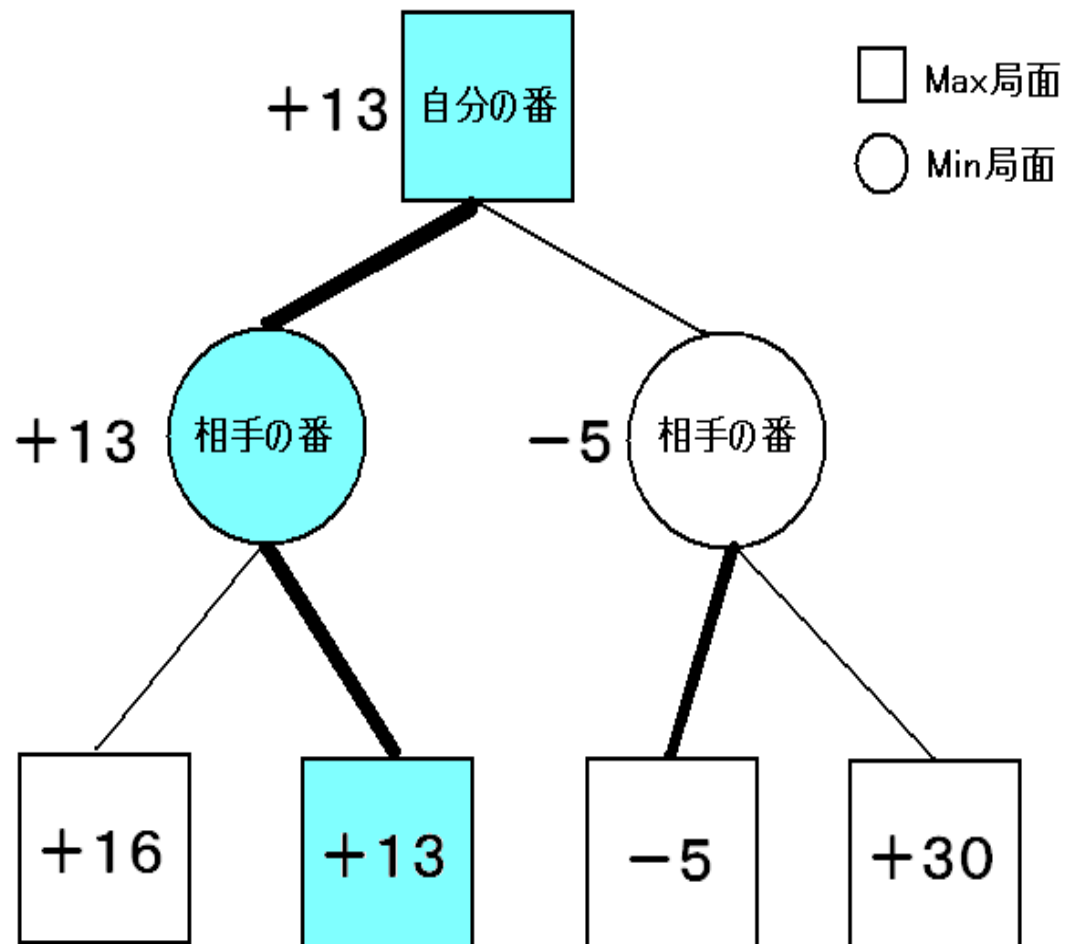
# ではもう1手深く読むと？

- 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

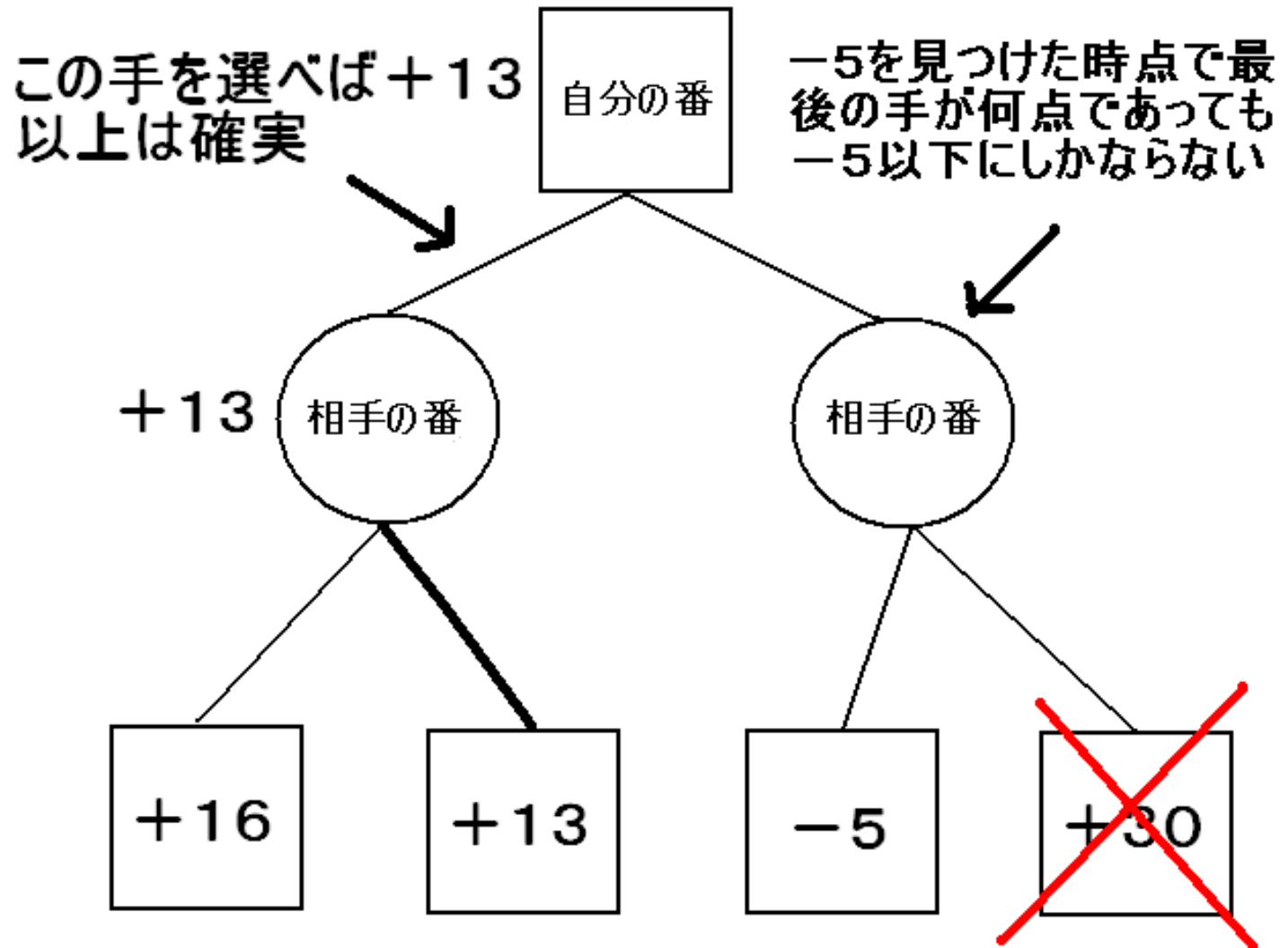


# Min-Max法

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



# $\alpha\beta$ 法 (読む必要がない手を省略)

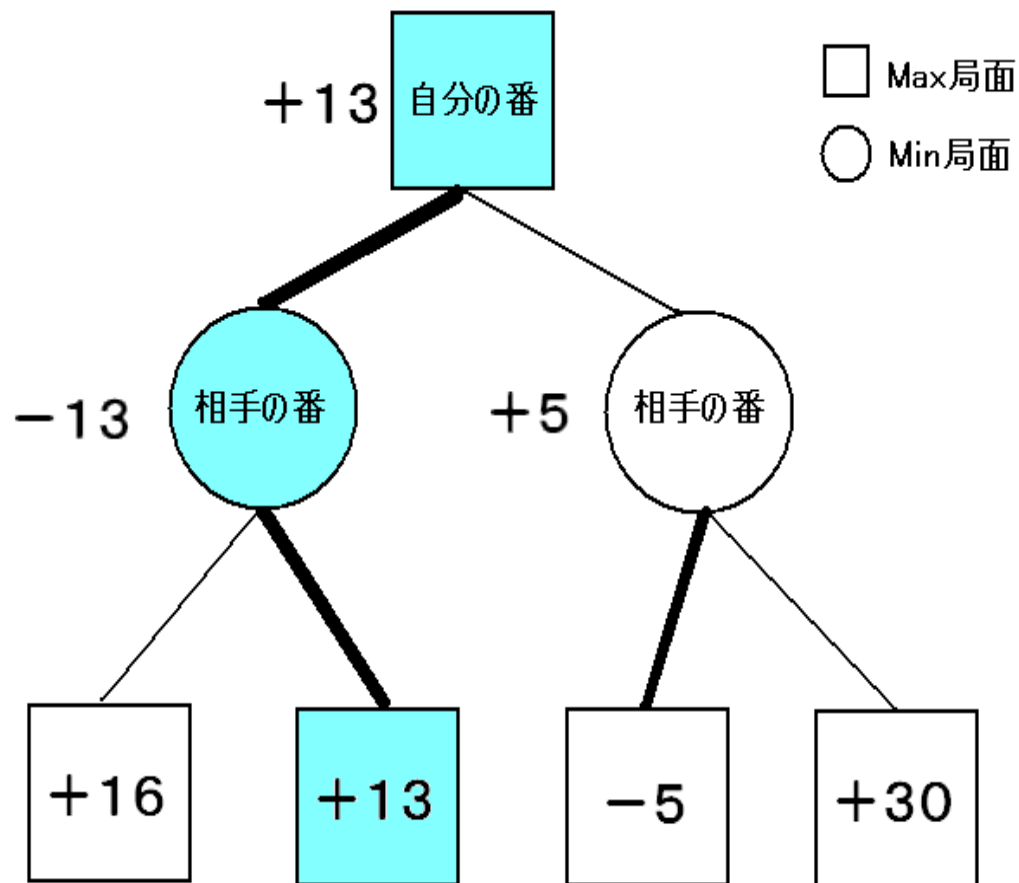


# NegaMax法

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



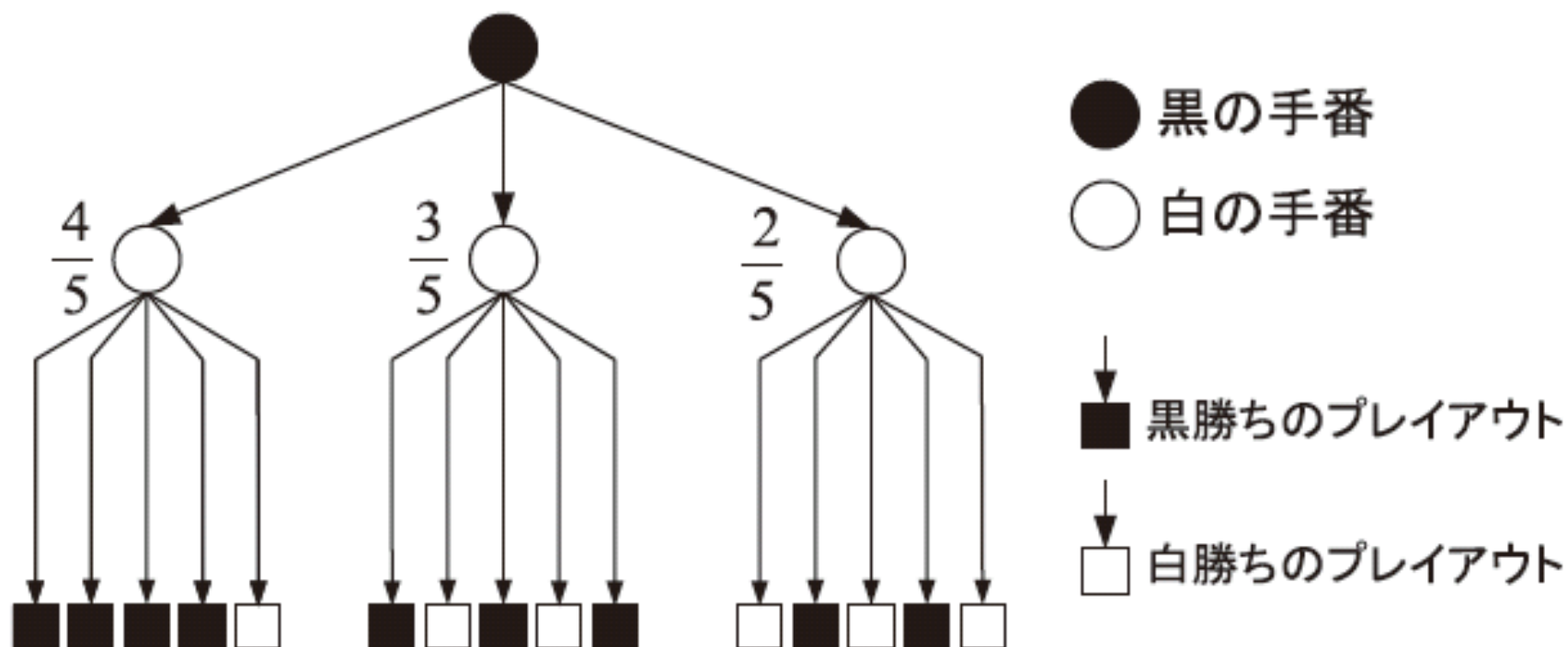


# UCTで探索するプログラム (**go08.c**)



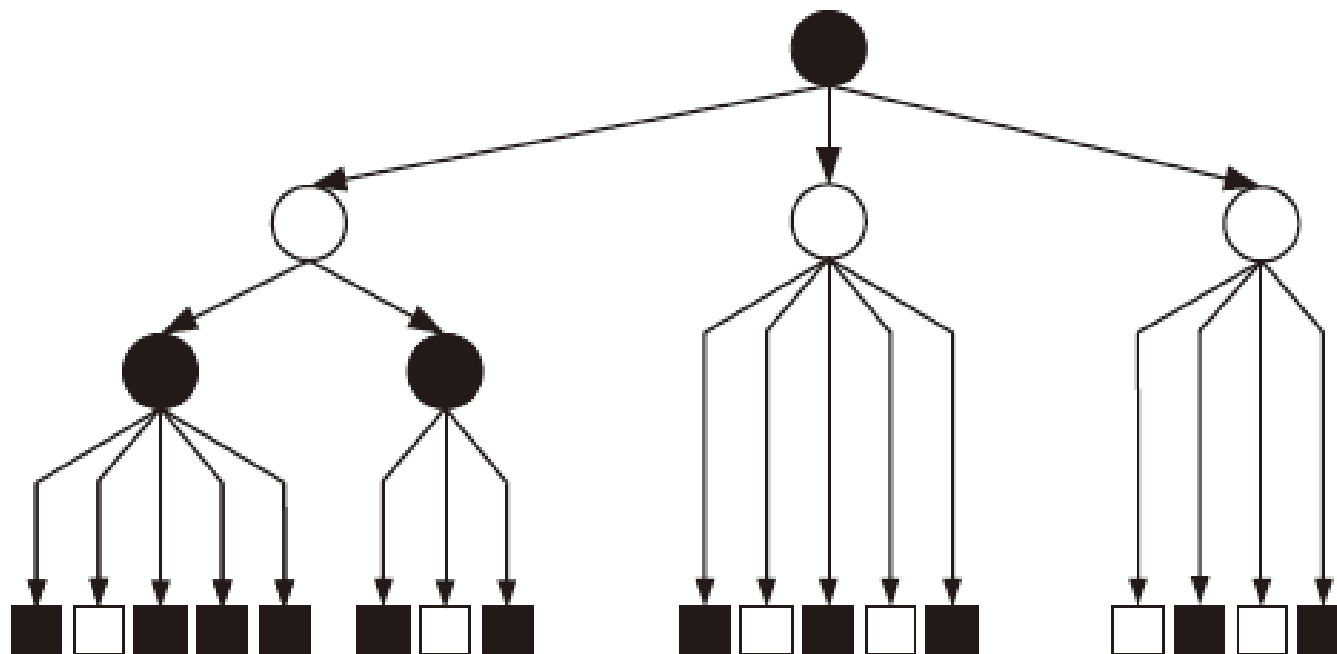
- モンテカルロ法を使って木探索をするプログラムの解説をします。
- MCTS(Monte Carlo Tree Search)と呼ばれています。

# 原始モンテカルロ囲碁のイメージ



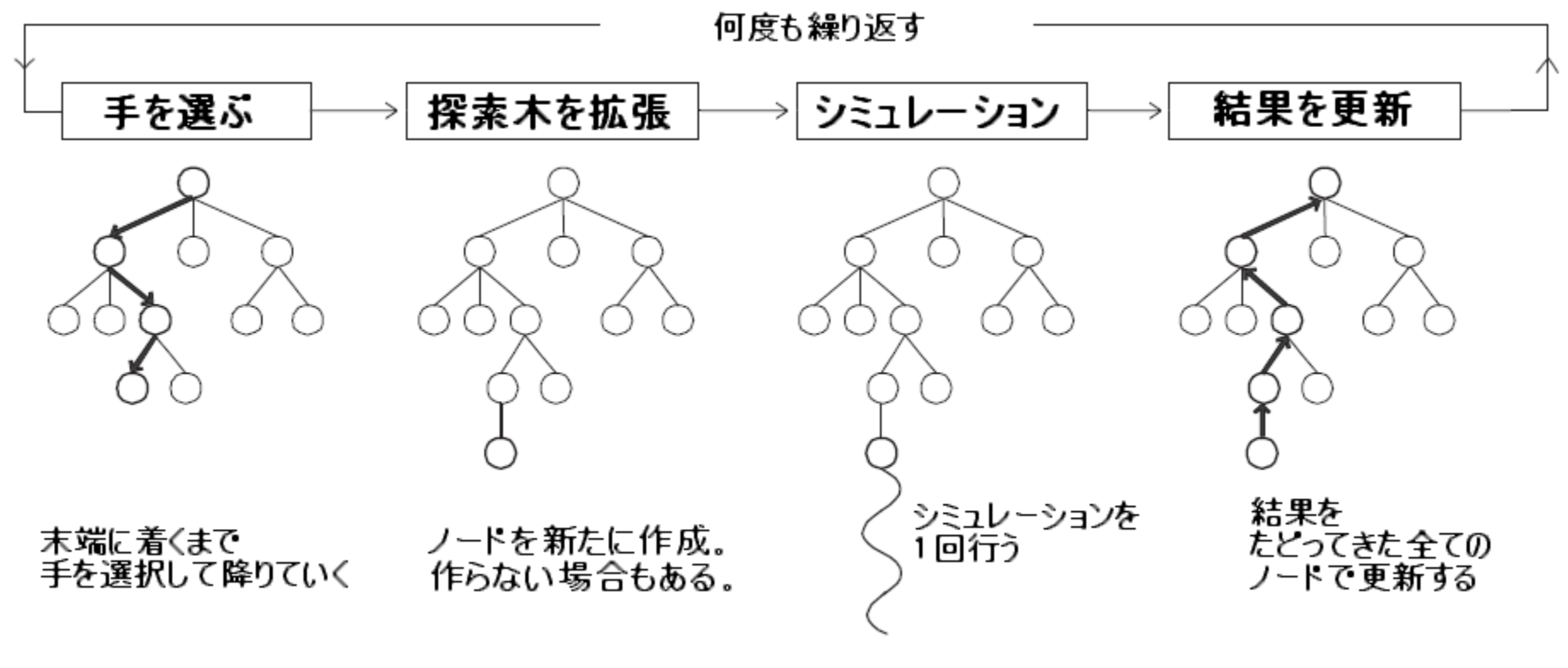
1. 勝率の低い手も同じ回数試していて効率が悪い
2. 相手の好手を深く調べない。事実上1手読み

# UCTでの木探索のイメージ

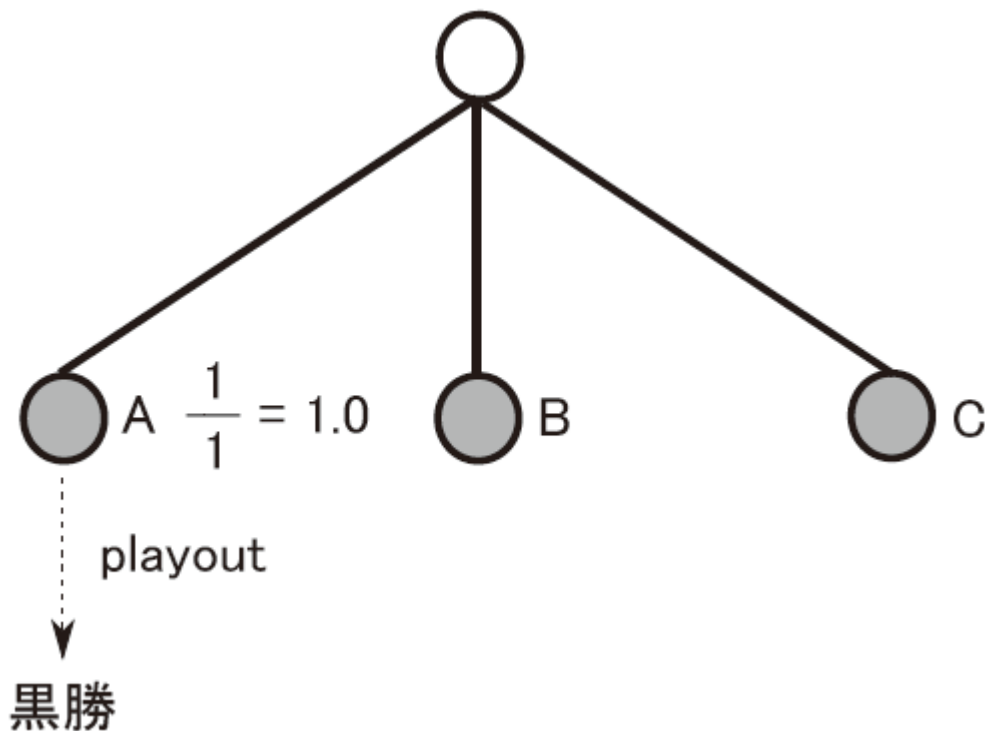


1. 勝率の高い手をたくさん調べる
2. 良さそうな変化を展開して深く調べる。

# 探索木を展開する様子

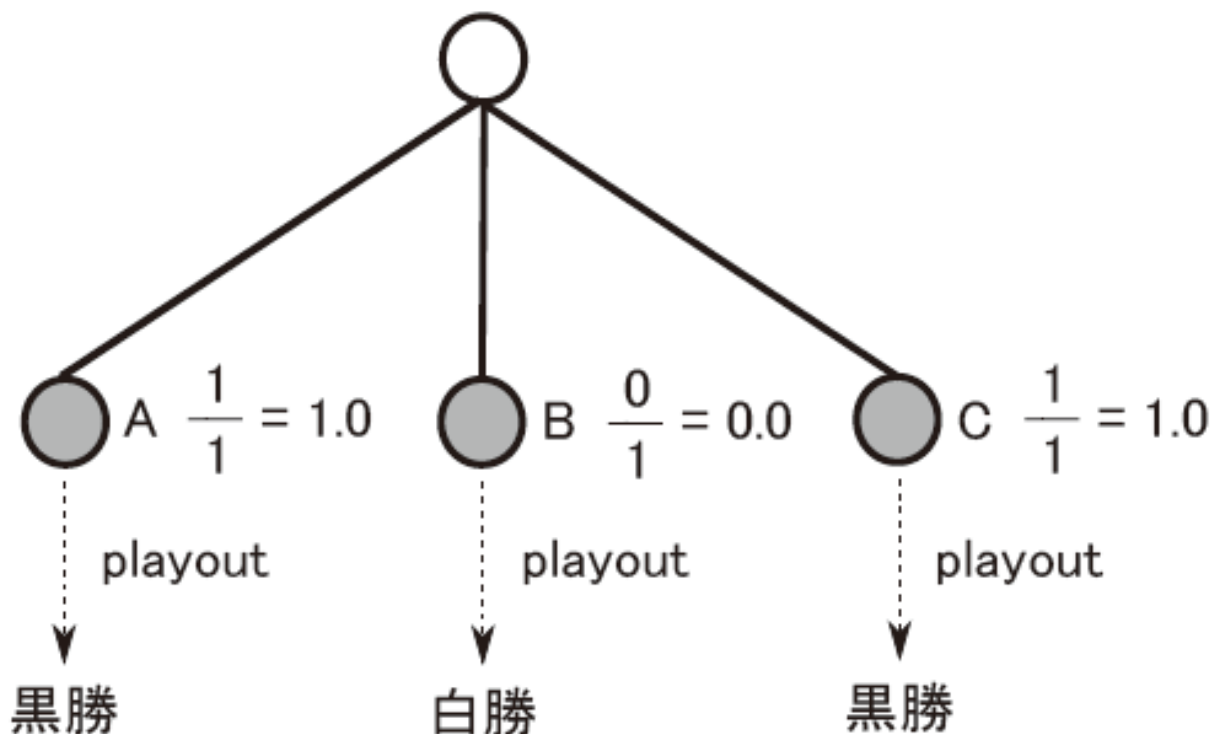


# UCBの式の復習



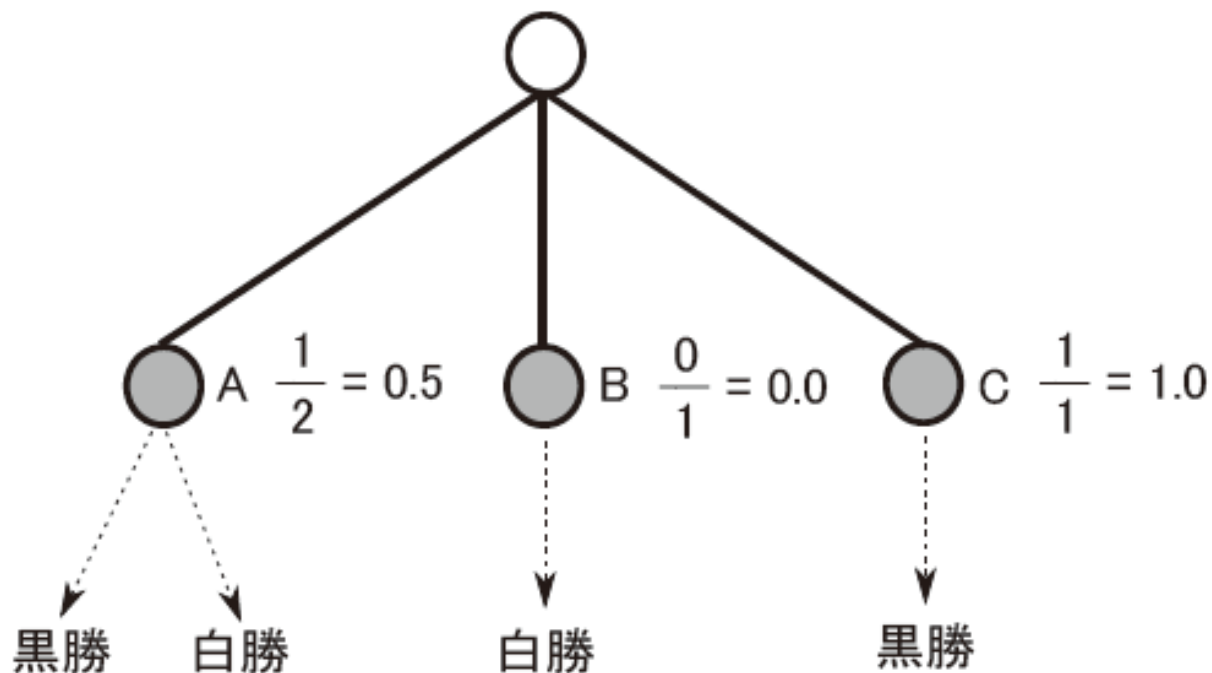
候補手が3手ある。黒番で一番いい手を選びたい。  
まず、左の手から選んでプレイアウト。黒勝ち。

# 全部の手を一度は試す



1. 全部の手を一度はプレイアウト
2. A と C の手が勝ったので次は A を選ぶとする

# 次に選ぶべきは、どの手？



1. Aの手は2回目は負けた
2. 次はどれを選ぶ？？？

# UCBの式がさっそうと登場

$$\text{UCB} = \text{その手の勝率} + K \sqrt{\frac{\log(\text{全部の手の合計回数})}{\text{その手を調べた回数}}}$$

それぞれの手のUCB値を計算して一番高い手を選ぶ！

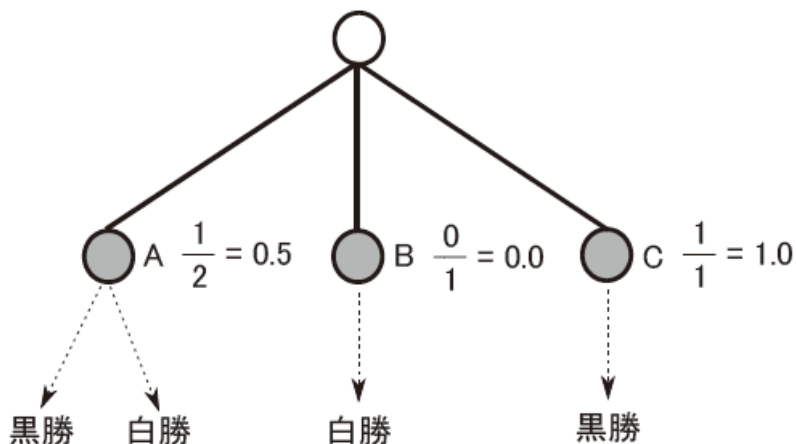
式の深い意味はあまり気にしないでも大丈夫です  
私も分かっていません・・・

Upper Confidence Bound (信頼上限)の略



# UCB値が最大のCを選ぶ

(Aの勝率は0.5、全体で4回試した)



$$A_{\text{UCB}} = 0.5 + K \sqrt{\frac{\log 4}{2}} = 0.5 + K \times 0.83$$

$$B_{\text{UCB}} = 0.0 + K \sqrt{\frac{\log 4}{1}} = 0.0 + K \times 1.18$$

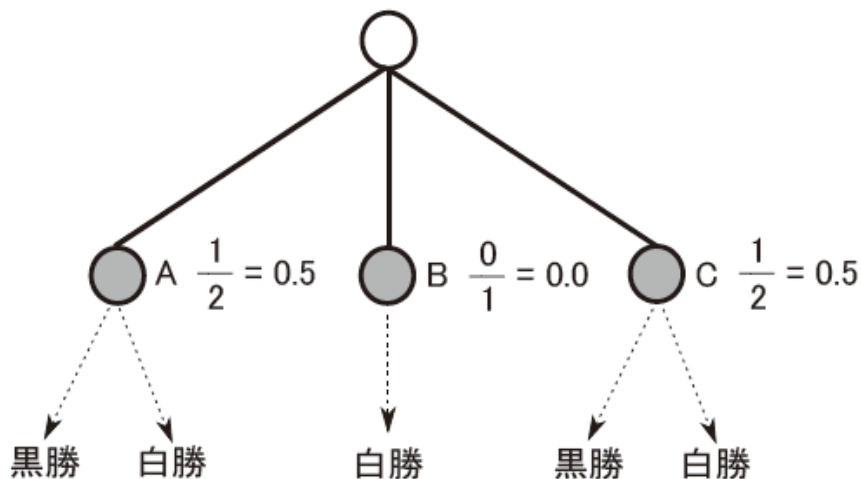
$$C_{\text{UCB}} = 1.0 + K \sqrt{\frac{\log 4}{1}} = 1.0 + K \times 1.18$$

K は正の定数なので、ここでは C のUCB値が最大

Cの手を選ぶ

# Kの値で探索の性格が決まる

(Aの勝率は0.5、全体で5回試した)



$$A_{UCB} = 0.5 + K \sqrt{\frac{\log 5}{2}} = 0.5 + K \times 0.90$$

$$B_{UCB} = 0.0 + K \sqrt{\frac{\log 5}{1}} = 0.0 + K \times 1.27$$

$$C_{UCB} = 0.5 + K \sqrt{\frac{\log 5}{2}} = 0.5 + K \times 0.90$$

K が1.35以上ならBを選び、それ以下ならAかCを

Kの値が大きいと冒険的に調べ、小さいと現在の勝率が高い手を保守的に調べる

# 実際の実装

- `select_best_ucb()` 関数を見てください

```
ucb = c->rate + C * sqrt(  
    log( pN->child_games_sum )  
    / c->games );
```

まったく同じ形で計算しています。

# 追加するデータ構造



- 木探索局面のすべての手の勝率を覚えておく必要がある

# 手を保存するための構造体

```
typedef struct {  
    int z;           // 手の場所  
    int games;       // 試した回数  
    double rate;     // 勝率  
    int next;        // 次のノード番号  
} CHILD;
```

# 局面を保存する構造体

```
// 最大の子数。9路なら82個。+1 はPASS用  
#define CHILD_SIZE (B_SIZE*B_SIZE+1)
```

```
typedef struct {  
    int child_num;           // 実際の子供の数  
    CHILD child[CHILD_SIZE];  
    // 何回このノードに来たか（子の合計）  
    int child_games_sum;  
} NODE;
```

# 探索木全体を保存



```
// 最大10000局面まで  
#define NODE_MAX 10000
```

```
NODE node[NODE_MAX];
```

```
int node_num = 0;    // 登録局面数
```

# 関数の説明



## ■ add\_child() 関数

- 手を追加
- この手を打った後のノードは、なし

## ■ create\_node() 関数

- 空白を全部追加
- PASSも追加
- 作られたノード番号を返す。最初は0から。



# select\_best\_ucb()関数



- UCB値が最大の手を返す
- 一度も試していない手は優先的に選ぶ
- 定数  $C$  は実験で決める
- PASS、があるのですべての手がエラー、はありえない

# search\_uct() 関数

- ucb値が最大の手を調べる
- 打ってみる
  - エラーなら ILLEGAL\_Z をセットしてもう一度
- 1回目は必ずplayout
  - `c->games <= 10` とかすればメモリを節約できる
  - `c->games <= 0` より強くなる場合もあり
- 次のノードがなければ作成
- `win = -search_uct();`
  - NegaMax形式で自分自身を呼ぶ。再帰関数。

# 勝率の更新

```
c->rate = (c->rate * c->games + win)
          / (c->games + 1);
```

- $c \rightarrow \text{rate} * c \rightarrow \text{games}$

- 勝率が 0.60 で 100回試したなら 60。勝った回数

- もし  $\text{win} = 1$  (勝った)なら 61回勝って101回試したので

- ( 60+1 ) / (101) = 0.604 になる

# get\_best\_uct() 関数



- まず現在局面のノードを作成
- search\_uct()を1000回呼ぶ
- 最大回数の手を選ぶ
  - 最大勝率ではない？
  - 3回試して3連勝、よりは100回試して0.60の方が安定した結果になる
  - MCTSでは一番選ばれた回数が多い手を打つ

# サンプルを実行してみる(**go08.c**)

- 黒で1000回UCTを繰り返す
- 白でも1000回。
- 定数  $C$  を 0、0.3、2、10 に変えて games の分布がどう変わるか、を確認
  - 値が大きいほど均等に選ぶ

# UCTと原始モンテカルロを自己対戦させてみる(**go09.c**)

- selfplay()関数を見てください
  - 黒番は  $fUCT = 0$ , 原始モンテカルロ
  - 白番は  $fUCT = 1$ , UCT探索
- 前回の講習で追加した selfplay()関数の
  - $fUCT = 0$ ; の行をコメントアウト
- get\_computer\_move()
  - $fUCT$ の値でどちらかを選ぶ

# UCT(白)が(多分)勝つのを確認

- `add_moves()`
  - 実際に着手して棋譜に1手を追加
- UCTは1000 playout
- 原始モンテカルロは30playout
  - 初期局面ではUCT 1000回、原始  $30 \times 81 = 2430$  回
  - 相合計だと同じくらいの回数
- この条件だと76%ほどUCT(白)が勝つ

# プレイアウト数を増やした時の伸び は同じくらい



原始モンテカルロ vs UCT (150局)

一箇所に	合計で	UCTの勝率
30 playout vs	1000 playout	76%
300 playout vs	10000 playout	67%
3000 playout vs	100000 playout	76%


プレイアウトが単純乱数だとUCTはあまり伸びない？



# SGFの棋譜をgoguiで再生

- 対局終了時の  
(;GM[1]SZ[9]KM[6.5]PB[]PW[];B[ha];W[h  
e];B[bb]; ... B[di];W[];B[])
- の部分をコピー
- SGFは囲碁の標準棋譜形式
  - 拡張子は \*.sgf

# SGFコピーの仕方



- Visual C++のDOSプロンプトの画面からコピーするには
  - 画面左上のアイコンを右クリック
  - 「編集」「範囲指定」で指定してEnter、でコピー
- goguiに貼り付けるには
  - 「ファイル」「インポート」「クリップボードからSGF形式読み込み」
- cgfgobanには画面上で「Ctrl+V」

# プレイアウトの中身を確認



- `main()`の`test_playout()`のコメントを外す
- 初期局面から`playout`の結果をSGFで出力

# 改良用のサンプル(**go10.c**)

- 9月12日はミニ大会を開いて皆さんのプログラム同士で対戦してもらいます。
- このソースを基本に改良してもいいですし、1から作り直してももちろんOKです。別の言語でもかまいません。
- 高速化のみでも歓迎です。
- 3x3のパターンを1個追加する、など何か少しでも工夫をお願いします。

# このソースで追加されていること

- GTP (Go Text Protocol) でgoguiから動かす
- 3x3のパターン認識
- プレイアウトで着手を確率分布から選ぶ
- 少しずつ探索する手を増やしていく(Progressive Widening)
- UCBの式に手のボーナスを追加
  - ノード作成時に個々の手のボーナスを計算

# GTPでプログラムを操作



- 実行して、"genmove b" と入力してEnterを押してください。genmove と b の間にはスペースが入ります。
- 次に "play w d3" (Enter) と入力。
- この繰り返しで手入力でも対局できます

# GTPの座標

- "D6","g3" など
  - 大文字、小文字、どちらでもOK
  - 左下が"A1"、右上が"J9"。("I" は抜く)

	A	B	C	D	E	F	G	H	J
9									
8									
7									
6									
5									
4									
3									
2									
1									

# GTPの主なコマンド

- `genmove b` ... 黒番で手を打て。白なら "w"
- `play w d3` ... 白をD3に置く
- `name` ... プログラム名を聞く
  - "your\_program\_name" をあなたの名前に変えてコンパイルしなおして下さい。
- `version` ... バージョンを聞く
- `clear_board` ... 盤面を初期化する
- `quit` ... 終了



# GTPでgoguiから動かす

## ■ goguiでは

- 「プログラム」「新規プログラム」「コマンド」に
  - 「c:¥dentsu¥10¥Release¥go10.exe」
- 「ワーキングディレクトリに」に
  - 「c:¥dentsu¥10」

## ■ cgfgobanでは

- 「Setting」「GTP Setting」に下記を入力
  - 「c:¥dentsu¥10¥Release¥go10.exe」

# プログラムを起動



## ■ goguiでは

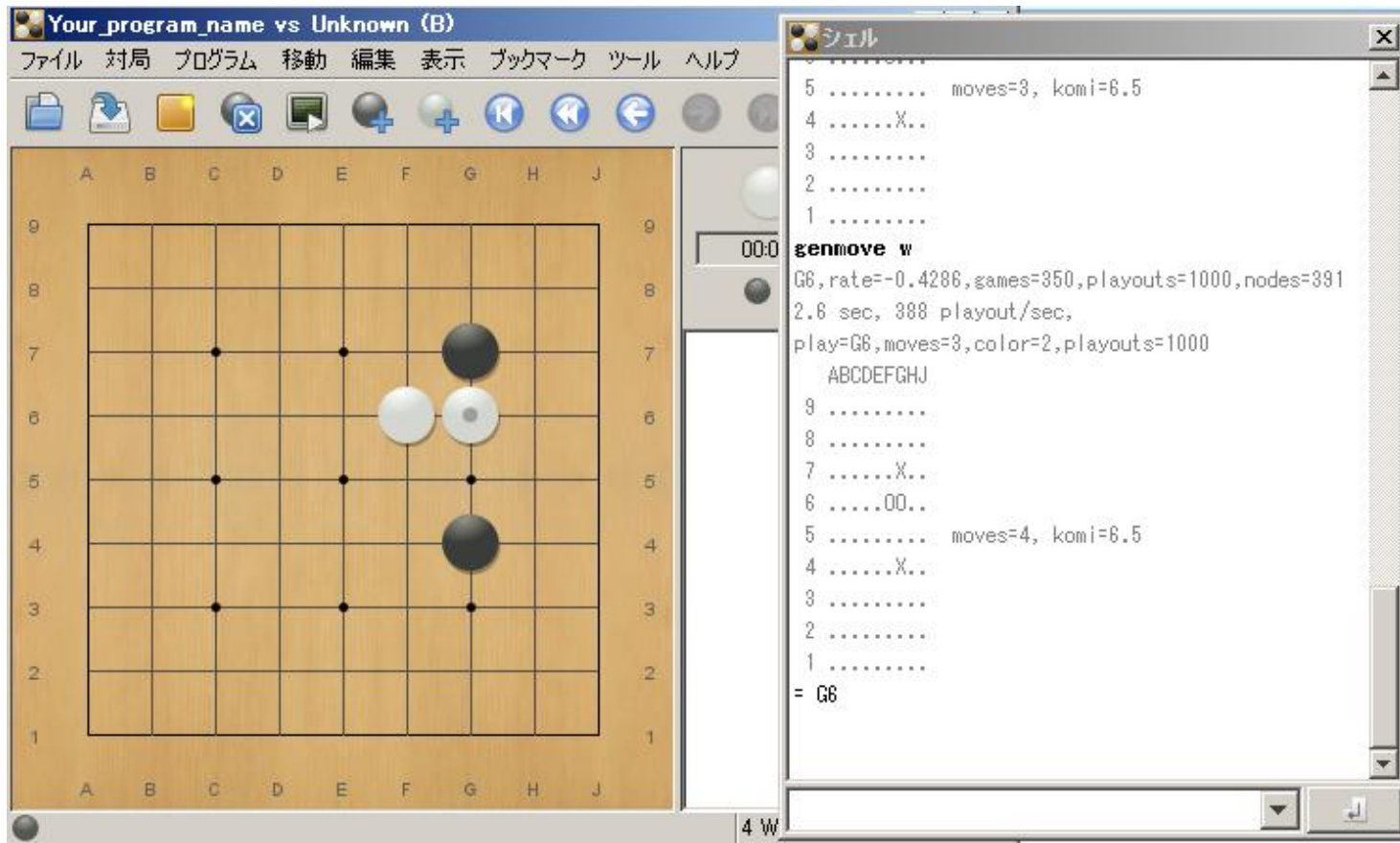
- 「プログラム」「プログラムの起動」「`your_program_name`」を選択
- どこか盤面をクリック
- 「ツール」「GTPシェル」で通信内容を表示

## ■ cgfgobanでは

- 「Play」「Start Game」で「white」を「Computer (GTP)」

# デバッグ情報も表示できるgoguiがお勧めです

- stderr に書けば表示してくれる



# 別のプログラムと通信



## ■ stdin

- 標準入力 ... 通常はキーボードからの入力

## ■ stdout

- 標準出力 ... 通常は画面に表示

## ■ stderr

- 標準エラー出力 ... 通常は画面に表示

- 他のプログラムから起動することでstdin、stdoutで通信できる

# GTPの実装

- main()関数に書いています
- stdin, stdout 経由で通信
  - send\_gtp() で stdout に出力
  - fgets() で gogui からの命令を stdin で受け取る
- stderr に書くと gogui では表示される
  - 今までのprintf()での情報表示はprt()関数で
    - void prt(const char \*fmt, ...) {}
      - printf()のような可変個数の関数定義の書き方
- 改行(¥n)が2回連続で応答完了

# printf()関数を置き換える

```
void prt(const char *fmt, ...)  
{  
    va_list ap;  
  
    va_start( ap, fmt );  
    vfprintf( stderr, fmt, ap );  
    va_end( ap );  
}
```

printf() は stdout に出力するのを stderr に変更する

# GTPでの文字列操作など

- `strtok()` スペースを区切りに文字列から文字列を取り出す
  - "genmove b" なら
    - | `sa[0] = "genmove"`
    - | `sa[1] = "b"`
- `tolower()` 文字列を小文字に変換
- `get_char_z()` 座標を"G6" などに変換
  - 1行で2回使うとバグるので注意
  - ✗ `prt("%s,%s¥n",get_char_z(z0),get_char_z(z1));`

# 3x3のパターン認識

```
char *pattern3x3[] = {  
    "X0?"  
    "   "  
    "..."  
    "???"  
};
```

●○?  
| | |  
? ? ?

真ん中に●を置く  
これは“ハネ”のパターン

"X"	...	黒石
"0"	...	白石
"?"	...	何でもOK (don' t care)
"#"	...	盤外



# 回転、対称で8パターンに展開

- `expand_pattern3x3()` 関数
- `#define EXPAND_PATTERN_MAX (8*100)`
  - パターンの最大数は100(8倍になるので)
- 90度回転を3回。線対称。また90度3回。

● ○ ?	124
++	000
? ? ?	444

90度回転



? + ●	401
? + ○	402
? + ?	404

“?” は 4

# パターンマッチングを高速化

9個の数値を1個ずつ比較すると時間がかかる

124

000

444



1, 2, 4, 0, 0, 0, 4, 4, 4

1個を2bit、計18bitで表し、dont care(4)を消すmaskを作る

1, 2, 4, 0, 0, 0, 4, 4, 4

bit 01 10 00 00 00 00 00 00 00

0x18000

mask 11 10 00 11 11 11 00 00 00

0x3cfc0

if ( (pattern\_bit & mask) == bit ) で判定できる。

しかしパターンの個数だけループするのでやっぱり遅い

## 3x3パターンを全部持つ(発展)

- 周囲8箇所が空白、黒、白、壁、で4通りなので  $4^8 = 65536$  通り。対称形を考慮するともっと少ない。
- 高段者やプロの棋譜から出現確率を調べる
- playoutで1手打つたびに8箇所を差分更新
- ハッシュ法で高速に検索
  - 時間があれば後述

# 棋譜の入手



- KGSの高段者の棋譜(毎月1000局ぐらい)
  - <http://www.u-go.net/gamerecords/>
  - 無料
- プロの棋譜 (GoGoD)
  - <http://gogodonline.co.uk/>
  - 15\$(約1,800円)
  - 84,000局ぐらい
  - 著作権的に微妙。私は買ったことはありません。

# プレイアウトで着手を確率分布から選ぶ

- 5手あって、その着手確率が
- 30%、10%、40%、15%、5%
- 1から100まで目ができるサイコロを振って、先頭から足して比較
  - 31なら10%の手
  - 98なら5%の手
- `playout()`関数の `empty[][]` あたりで実装

# 個々の手の着手確率を計算

- `get_prob()` 関数
- 基本は100。大きいほど打たれやすい。
- 下の2行は役に立っていないのでコメントアウト
















```
if ( sc[un_col] >= 3 && sc[   col] == 0 ) pr /= 2;  
if ( sc[   col] >= 3 && sc[un_col] == 0 ) pr *= 2;
```
- 3x3のパターンは直前手の8近傍だけチェック
- `MAX_PROB` より大きな値にしないように
  - `int`の範囲を超えてバグります

# 改良点



- 直前手の周囲8近傍に打つ手の確率を上げる。相手の石にくっつけて打つように。
- 盤の1線、2線に打つ確率を下げる。
- 3x3のパターンを増やして、この形は絶対に打たせたい、という手は10000倍とか大きく。
- アタリにされた石の周囲にあるダメ1の石を取る手の確率を上げる。
  - これはデータ構造が単純なので実装は困難

# 3x3全パターンを使った黒の着手確率 の例

1	1	1	1	1	9	47	165	3
9	3	6	25	10	3			3
1			391	48	1			2
9	130			1	1			3
1	1			1	48	475	440	3
1	5			27	57	57	10	1
1	31	602		857	57	57	10	1
1	10	33	38	31	10	10	10	1
1	1	1	1	1	1	1	1	1

数値が大きいほど着手確率が高い



# playoutの中身を確認



- `get_prob()` 関数を変更したらplayoutで実際にどういう手が打たれているか、を確認。
  - `main()` で `test_playout()`関数を

# 少しずつ探索する手を増やしていく

- playoutが30回なら上位4手ぐらい。1000回で上位12手ぐらいしか調べない
- 手の仮評価が正しいほど有効に働く
  - 相手の直前手の近くは高く、1線は低く、など
- Progressive Widening（徐々に増やす）と呼ばれてる。

# 手の仮評価の計算



- `get_bonus()` 関数
  - ノード作成時に手の仮評価(bonus)を計算
- 仮評価の大きさを手でソート
- `select_best_ucb()`関数
  - `pw_num` で上位何手かを設定

# get\_bonus() 関数



- 1線、2線は小さく
- 直前手にくっつけて打つ手は大きく
- 値は0～10くらいまで
  
- 改良点
  - 3x3パターンの評価を追加
  - 直前手からの距離で

# UCBの式に手のボーナスを追加

- 手のボーナスを直接UCB式に組み込む
  - ボーナスが大きい手は勝率が悪くても何度も試される

```
C = 1;  
B0 = 0.1;  
B1 = 100;  
plus = B0 * log(1 + c->bonus)  
        * sqrt( B1 / (B1 + c->games) );  
ucb = c->rate + C * sqrt( log(pN->child_games_sum)  
                          / c->games ) + plus;
```

# UCBの式の改良点

- 定数の  $C$  や  $B_0$ ,  $B_1$  を変えてみる
- 式の形を変えてみる

$$\text{plus} = B_0 * c \rightarrow \text{bonus} * ( B_1 / c \rightarrow \text{games} );$$

- $\text{playout}(c \rightarrow \text{games})$  が増えるほどボーナスの効果を減らすところがミソ

# 自己対戦で勝率を調べる

## ■ go10同士を100局対戦させる場合

■ dentsu.txt からコピーして下さい。

```
java -jar gogui-twogtp.jar -black  
"c:¥dentsu¥10¥release¥go10.exe" -white  
"c:¥dentsu¥10¥release¥go10.exe" -games 100 -size 9 -  
alternate -sgffile go10test -auto
```

## ■ 結果をhtmlで表示

```
java -jar gogui-twogtp.jar -analyze go10test.dat
```

# 終局判定にGNU Goを使う

- go10同士を100局対戦させる。

```
java -jar gogui-twogtp.jar -black  
"c:¥dentsu¥10¥release¥go10.exe" -white  
"c:¥dentsu¥10¥release¥go10.exe" -games 100 -size 9 -  
alternate -sgffile go10test -auto -referee  
"c:¥go¥gnugo¥gnugo.exe --mode gtp --chinese-rules"
```

- GNU Go のWindowsバイナリ

<http://gnugo.baduk.org/>



# 以下は発展です



- 時間があれば解説します

# RAVE (**go11.c**)



- 1手を選び、playoutを1回行う
  - 更新される値はこの1手だけ
  - もったいない！
- ●E3、と打つ。playout が
  - ○E2●D3 ○C3●B5・・・ の場合
  - ● E3の1手だけを更新するのではなく、●D3、●B5の手も更新する

# 打たれた手の順番を無視する

- 「手順中のすべての黒の手が『最初に打たれた』と仮定する」
  - All Moves As First (AMAF) と呼ばれる。
  - RAVEは(Rapid Action Value Estimate)の略
- 通常の勝率とは別に保存
  - 手順前後無視で信頼性はない
  - 評価速度は9路だと50倍ほど速い
- 好手はどのタイミングで打ってもだいたい好手、という囲碁特有の知識を使っている？

# UCBとRAVEを組み合わせる

- ucb\_rave が最大の手を選ぶ
- 試した回数が少ないときはraveの値を優先し、増えればucbを優先する
  - beta という変数でコントロール

```
ucb_rave = beta * rave + (1 - beta) * ucb;
```

```
rave = raveでの勝率
```

```
beta = rave_games / (rave_games + games +  
rave_games*games / 3000);
```

# RAVEの実装

## ■ 手の構造体にraveの勝率、回数を追加

```
typedef struct {  
    int    z;           // move position  
    int    games;       // number of games  
    double rate;        // winrate  
    int    rave_games;  // (RAVE) number of games  
    double rave_rate;   // (RAVE) winrate  
    int    next;        // next node  
} CHILD;
```

# UCT、playoutの手順を記憶

- `int path[D_MAX]` に 手順を
- `int depth` に開始局面からの手数を
- `update_rave()` 関数で更新
  - 同じ場所に2回打たれたら最初の石の色のみを
- `select_best_ucb()` で`ucb_rave`を計算
  - PASSを選ばないように調整

# RAVEが圧勝する



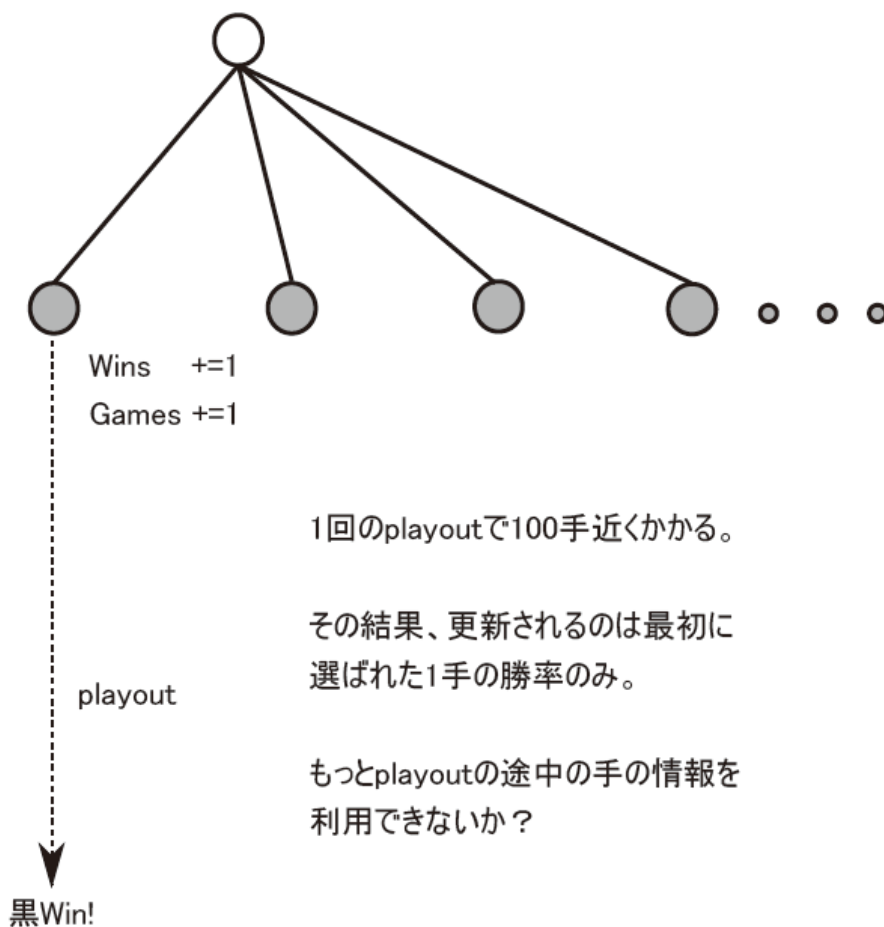
- RAVEありとなしでは1000playoutでは「あり」が勝率91%で圧勝する
- 手の並び替えや、playoutの質が上がるほど効果は下がっていく
  - CrazyStoneではUCBに組み込む形では使っていない。Nomitanも使っていない。Ayaは使ってる。
  - 絶対に必須、というわけでもない？

# RAVEの他の使い道

- いい手には 高い初期値を与えて選ばれやすくする
  - `rave_rate = 1.0;`
  - `rave_games= 30;`
- RAVEの高い手をProgressive Wideningで優先的に追加する
  - 三目中手、などを見つけてくれる



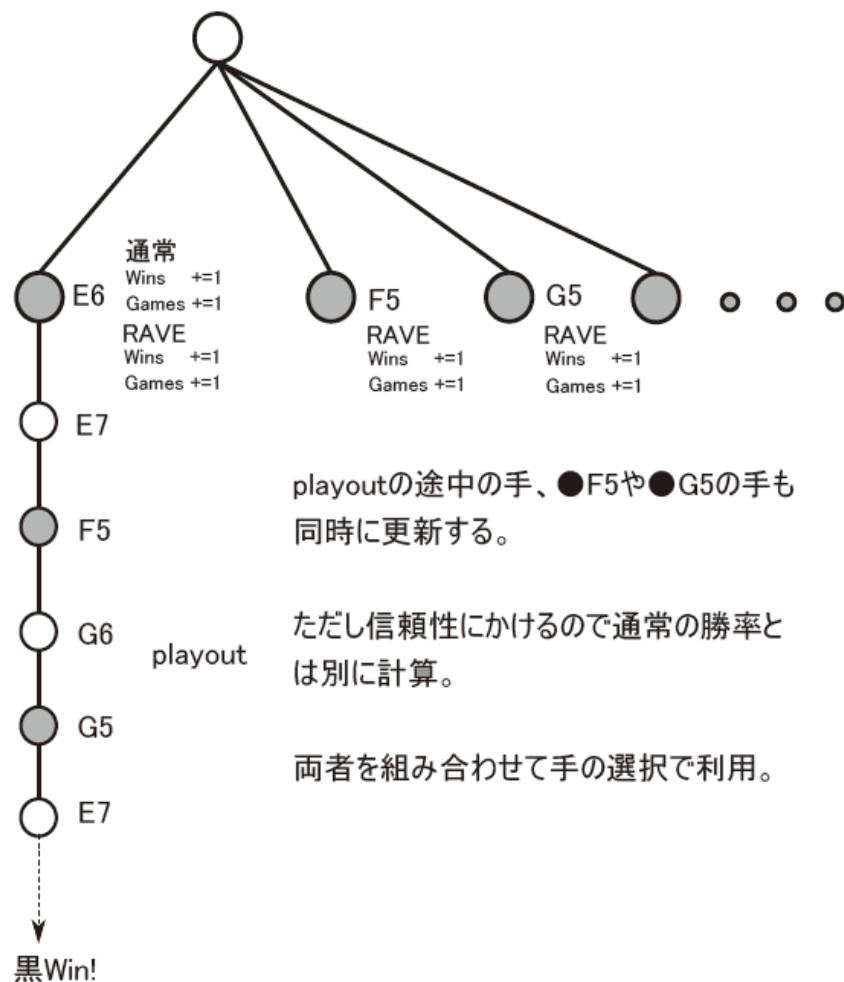
# RAVEの更新イメージ



1回のplayoutで100手近くかかる。

その結果、更新されるのは最初  
選ばれた1手の勝率のみ。

もっとplayoutの途中の手の情報を  
利用できないか？

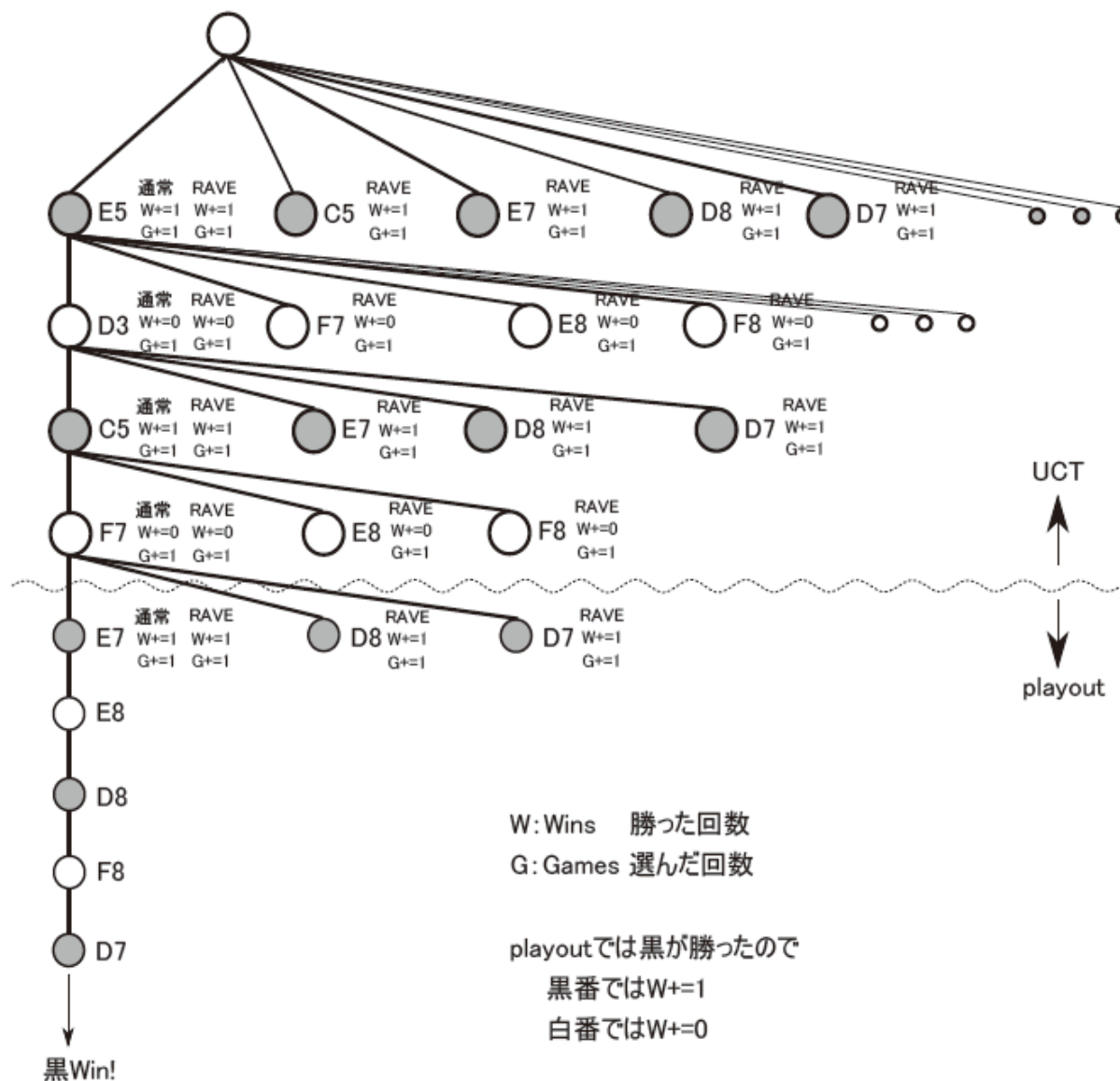


playoutの途中の手、●F5や●G5の手も  
同時に更新する。

ただし信頼性にかけるので通常の勝率と  
は別に計算。

両者を組み合わせて手の選択で利用。

# UCT+playoutでの更新イメージ

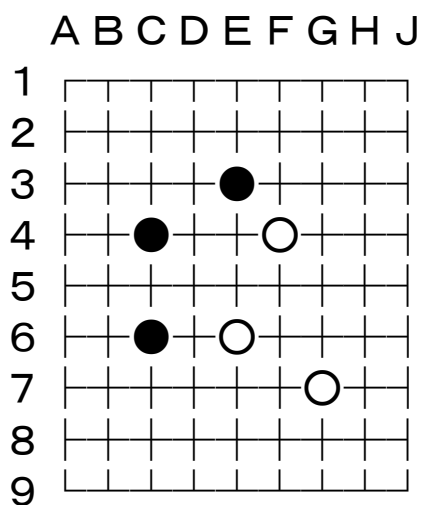
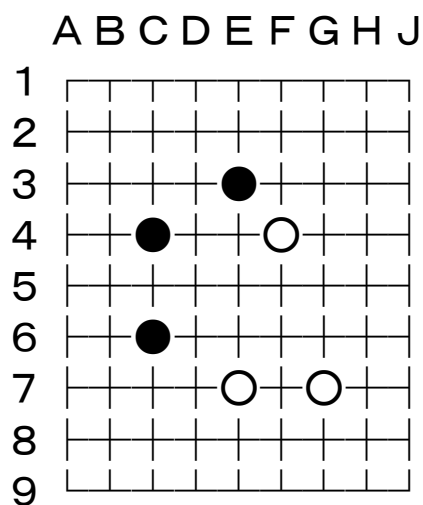


# ハッシュ法による同一局面判定 (go12.c)

## ■ 二つの局面が同一か調べたい

- 81回比較する必要がある

- 1回の比較で済ませる方法(ハッシュ法)



ここの白石が違ってる

# XOR(排他的論理和)

## ■ ビット演算の一つ

- $0 \text{ xor } 1 = 1, 1 \text{ xor } 0 = 1$
- $0 \text{ xor } 0 = 0, 1 \text{ xor } 1 = 0$
- 違うbitは1、同じbitは0になる。
- $111000 \text{ xor } 001100 = 110100$
- $110100 \text{ xor } 001100 = 111000$
- 2回、同じ数値と xor をすると元に戻る

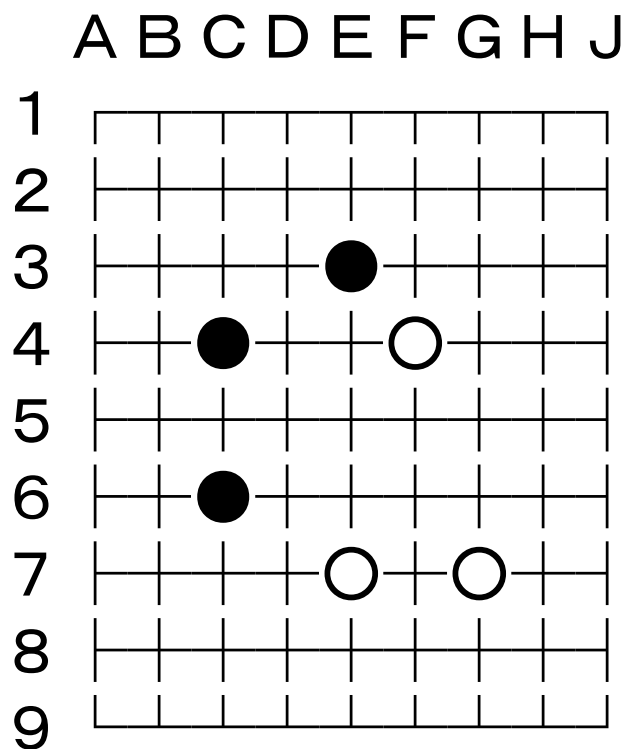
# 石を置くたびに乱数とXOR

- ある場所に黒、白の石が存在する状態に一つの乱数を割り当て、すべてのXORを取る
- 黒石を E3 に置けば
  - $E3(\text{黒}) = 0x5ea51507$
  - E3の黒石を取ったら、再度  $0x5ea51507$  とXORを取る。→ 元に戻る。
- 白石を E3 に置けば
  - $E3(\text{白}) = 0x9c928ce4$

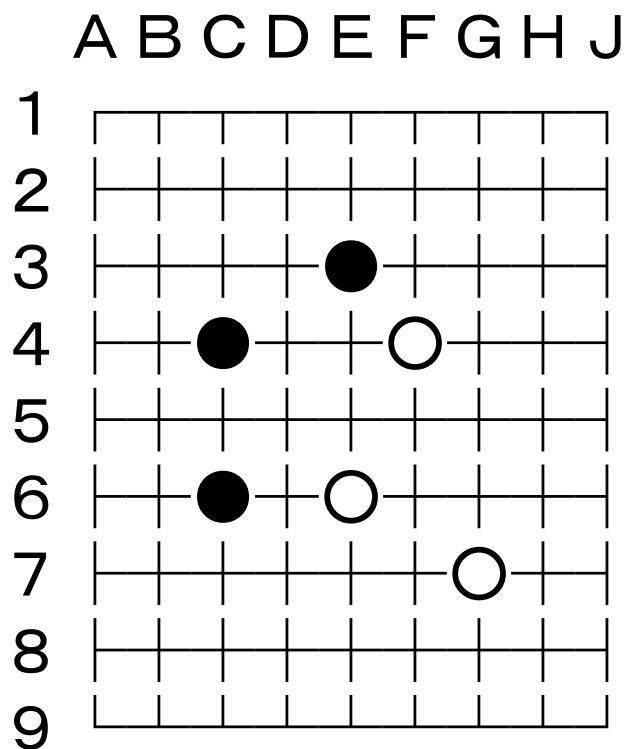
# 石を置くたびにハッシュ値を更新

- 石を置くたびに xor
- 石を取っても xor
- コウの場所も専用の値で xor
  - 1手前がコウなら xor で消す
- パスしたら手番が変わるのでビット反転
- 石を置いたら手番も変わるのでビット反転も
- 同一局面（石の配置、手番、コウの位置）

# 値が違うので別の局面と分かる



0xf905cdbf

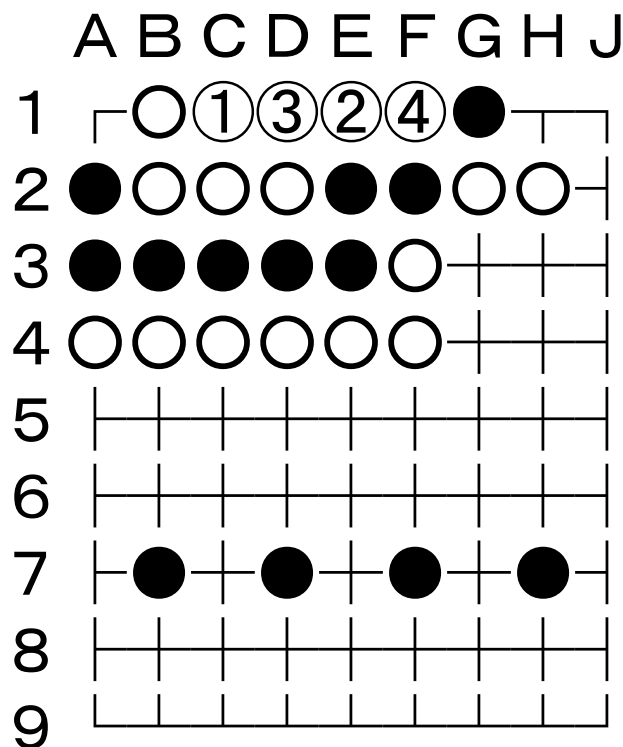
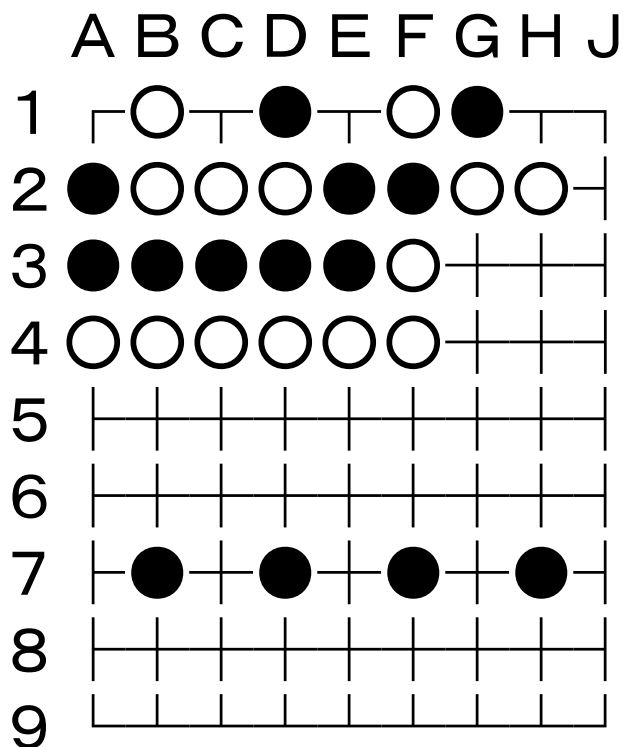


0xd613e4f7

# 同一局面で同じ値を確認(**go12.c**)

## ■ 4手一組で元に戻る

■ 長生(ちょうせい)と呼ばれる。極めてまれに出現



黒から①  
に打つ。



## 3コウの場合 **test\_3kou()**

- OF8で同じ形に戻るがコウの場所が違うので●B7で同一局面になる（6手一組）

	A	B	C	D	E	F	G	H	J	現在	4704085c33b7f224	コウ	なし
9	●	●	●	●	●	●	●	○	○				
8	●	○	●	+	●	○	●	○	○	●B7,	54c6938a17d1624e	コウ	B8
7	○	+	○	●	○	+	○	○	○	○D8,	134ac5f5094ca42e	コウ	D7
6	○	○	○	○	○	○	○	○	○	●F7,	09d94c00dc8b983f	コウ	F8
5	●	●	●	●	●	●	●	●	●	○B8,	2d0b3f6d0c6b044e	コウ	B7
4										●D7,	2796832bdf2615e6	コウ	D8
3										○F8,	440a679163b01cb7	コウ	F7
2										●B7,	54c6938a17d1624e	コウ	B8
1													

# 手番でも変わる



- `clear_board`
- `play b pass`
- `play w pass`
- でハッシュ値が 0 に戻るのを確認

# 碁盤の表現に必要なビット数

- 9路盤の情報量は  $3^{81} = 2^{129}$
- 129ビットは必要なので32ビットでは表現不可能
  - 19路は  $3^{361} = 2^{571}$  (571ビット)
- 探索中に出現する局面の数を考慮すると、異なる局面に同じハッシュ値が割り当てられる可能性を無視できる。(衝突する、という)

# 32ビットだと危ないので64ビットで

- 32ビットでは $2^{16}$ 局面の探索で衝突が一度でも発生する確率が50%になる。
  - $2^{16} = 65536$ 局面はすぐ行く
- 64ビットだと
  - $2^{32}$ 局面の探索で衝突が50%で発生
- 半分のビット数の探索で衝突が無視できなくなる
  - 8ビット(256)だと半分の4ビット(16)局面の探索で

# 乱数でrand()はなるべく使わない

- VC++のrand()の下位5bitから(x,y,z,w)という一つの座標を作った時に乱数性はほとんどない。
- サンプルでは XorShift を利用
- メルセンヌツイスター、M系列などがお勧め

# 地になる確率を表示(**go13.c**)


- playoutが正しく局面を認識しているか、が分かる。
- board\_area\_sum[] を追加
- count\_score()関数で黒地(黒石＋黒空白)は+1、白地は-1

# どこを地だと思っているか

■ +100で100%黒地、-100で100%白地

	A	B	C	D	E	F	G	H	J										
9										-14	-14	-18	-20	-23	-26	-23	-20	-18	
8										-14	-13	-13	-17	-26	-29	-25	-18	-16	
7					○	○				-12	-10	-15	-8	-49	-49	-18	-13	-12	
6		○		●	●					-6	-9	-34	12	51	16	2	-6	-5	
5										-0	2	4	25	27	20	11	1	-1	
4				●	★					9	10	22	71	71	26	14	5	3	
3										10	15	24	34	33	23	15	10	5	
2										13	14	22	25	21	18	14	11	10	
1										13	13	17	18	18	18	14	11	11	

# Criticality(**go14.c**)



- 勝敗を決定する重要な位置を求める
- 黒勝ちで、黒石が残っている場所を+1
- 白勝ちで、白石が残っている場所を+1



# 計算式

$$c(x) = \frac{v(x)}{N} - \left( \frac{w(x)}{N} \times \frac{W}{N} + \frac{b(x)}{N} \times \frac{B}{N} \right)$$

$v(x)$  プレイアウトで勝った方がこの位置に置かれていた回数

$w(x)$  この位置に白が置かれていた回数

$b(x)$  この位置に黒が置かれていた回数

$N$  プレイアウト全体の回数

$W$  白がプレイアウトで勝った回数

$B$  黒がプレイアウトで勝った回数


$c(x)$  の値は最小で 0、最大で 0.5

# 探索木の手の選択に使える

## ■ 攻め合いになってる石の周辺が高い

	A	B	C	D	E	F	G	H	J	
1			○	+	●	○	+	●		0.18 0.19 0.26 0.32 0.38 0.37 0.32 0.27 0.14
2		+	○	+	●	○	+	●		0.21 0.21 0.27 0.32 0.42 0.41 0.33 0.27 0.19
3		+	○	+	●	○	+	●		0.20 0.21 0.26 0.34 0.41 0.41 0.33 0.27 0.19
4		+	○	+	●	○	+	●		0.21 0.21 0.27 0.32 0.42 0.40 0.33 0.27 0.18
5		+	○	+	●	○	+	●		0.21 0.22 0.26 0.32 0.41 0.41 0.33 0.26 0.21
6		+	○	○	●	○	●	+		0.21 0.21 0.27 0.26 0.36 0.40 0.27 0.21 0.22
7		+	○	●	○	○	●	+		0.21 0.22 0.26 0.29 0.36 0.41 0.27 0.23 0.23
8		+	○	●	●	●	+	+		0.23 0.21 0.26 0.31 0.31 0.29 0.25 0.26 0.24
9										0.17 0.21 0.25 0.27 0.23 0.25 0.26 0.24 0.20

# 局面ごとに持たせる




- サンプルではCriticalityのデータはグローバルだが、実際は NODE 構造体の中に入れて、局面ごとに持たせる。

# より詳しくは



ご静聴ありがとうございました

A thick, horizontal yellow brushstroke underline that spans the width of the text above it, with a slightly textured, hand-painted appearance.