L12. Search for Decisions in Games
ゲーム木の探索

- ゲーム木の探索について
- ミニマックス法のアルゴリズム
- **アルファベータ法のアルゴリズ**
- 三目並べゲームの例

ゲーム

TicTacToe
Othello
Chess
……
Let us find game and play!

三目並べ http://perfecttictactoe.herokuapp.com/

オセロ http://atohi.com/osg/default.aspx

将棋

ゲーム木の探索問題

ゲームはAIにおける、最も古くから力を入れられている分野の1つです。限られた時間の中で答えを見つけることはとても難しい

ゲーム木は人工知能で重要であり、最良の手はゲーム木を探索することで得られ、ミニマックス法などのアルゴリズムを使用する。三目並べのゲーム木は小さいので探索も容易だが、チェスなどの完全ゲーム木は大きすぎて全体を探索することができない。その場合は代わりに部分ゲーム木を使う.

チェスゲームでは

35通りの置き方があり、レイヤーは50回移動することができる。

つまり、探索木には合わせると35^{100}もの葉が存在することになります。

このような複雑なゲームにはある程度の場所で妥協する必要があります。これは情報がないのではなく、いかなる動きに対する正確な結果を計算する時間がないからです。

この点で、ゲームは標準的な検索問題よりも現実の世界に似ています。

まずはじめに三目並べを例に、理論上最高の手を見つける方法を分析することから始めましょう。

2人のゲームの探索

ゲーム木は人工知能で重要であり、最良の手はゲーム木を探索することで得られ、ミニマックス法などのアルゴリズムを使用する。

問題は以下の要素から構成されます:

- **初期状態**
- **定められた動き**
- **終了状態**
- **評価関数**: +1(+10, +99), -1 (-10, -99),

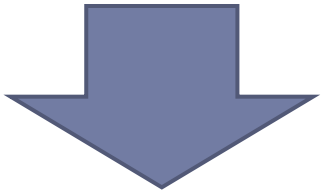
  または 0のように数値(評価値)を与えます.

コンピュータは相手が取りうる動き(手)の中で正しい動き(手)を含む戦略を見つけなければなりません。そして、最終状態の勝者を導き出し、その流れの最初の一手を選びます。

重要: 評価関数 はどの動きが最高の動きであるかを決定するじゅうような構成要素です。

ミニマックス法

- 考え方
  - 自分の手では局面が最良になる手を選びたい
  - 相手は（自分にとって）局面が最悪となる手を選ぶだろう



相手が自分にとってまずい手を打ってきても、
そのまずさ(Min)があまり悪くない手(Max)を
打とう。

ミニマックス法

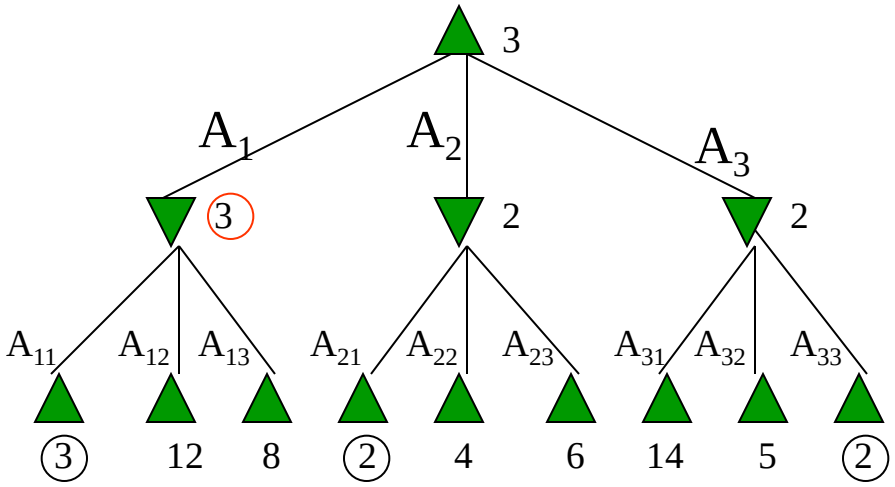
- もう少し詳しく
  - 今、自分が打てる複数の候補手を考えた時、それぞれの手に関して、相手の対抗手（これもまた複数）を探索する。この時、対抗手の評価値を全て計算する。
  - その中で最小の評価値（相手にとっては最高の評価値）hを求める。
  - 自分の候補手の中で、最も高い評価値である手を最適なものとする。

自分の手では評価値がmax, 相手の手では評価値がminとなる手を選択

5つのステップから構成されます:

- ゲーム木の全体を生成する.
- 木の末端ノードに評価値を得るために効用関数を適用する.
- この評価値を探索木の1つ上のレベルのノードの評価値を得るために　　使う(どちらの手番かによってどのノードの評価値を選ぶかは変わる).
- これを末端から根まで繰り返す.
- コンピュータは最も評価値の高いものを選ぶ.

```
minimax(節点 n, 深さ d)
{
    if(d==0 || n が終端節点) return(n の評価値);
    n の子節点を n_i とする
    if(n が Max 節点) /* (先手番) */
      return max_i{minimax(n_i, d − 1)};
    else  /* n が Min 節点 (後手番) */
      return min_i{minimax(n_i, d − 1)};
}
```



L1 (return maximum in L2**)**

L2 (return minimum in L3**)**

L3(return 評価値)

ゲーム木の全体を生成する

完全木

例：三目並べゲーム木

9

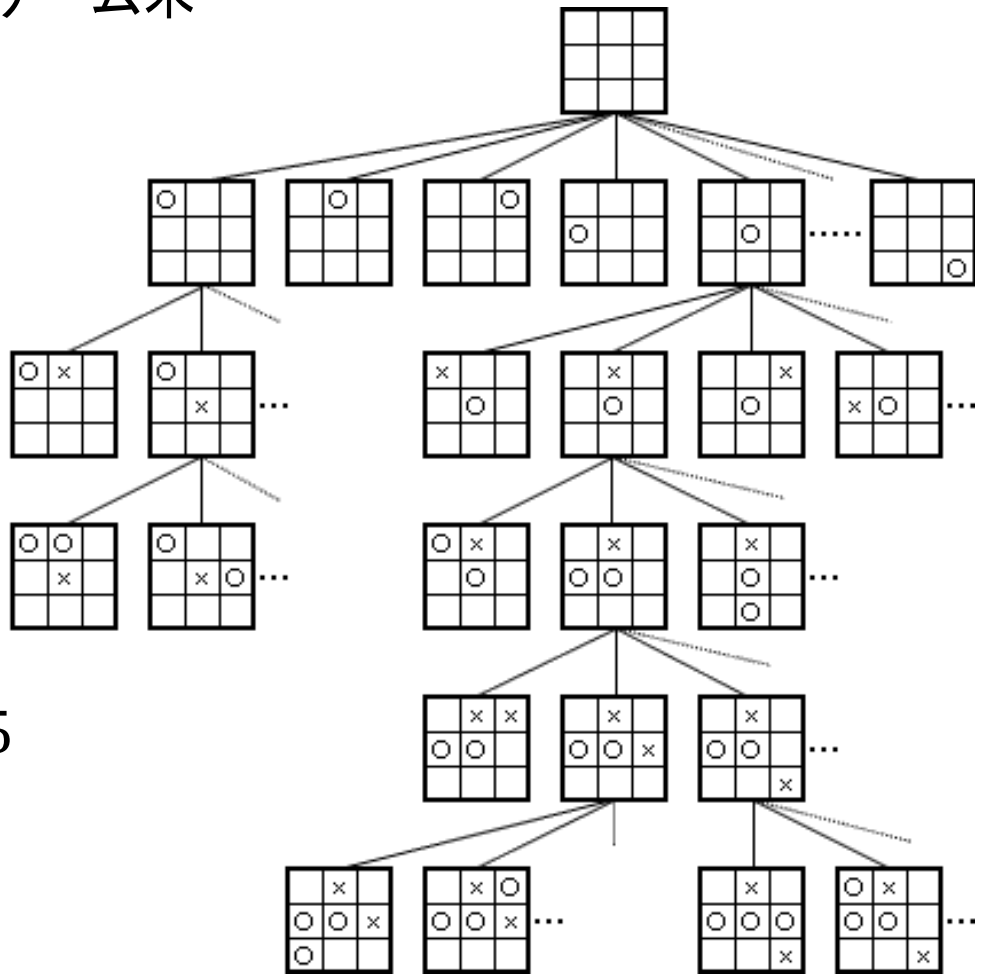
9*8

9*8*7

9*8*7*6

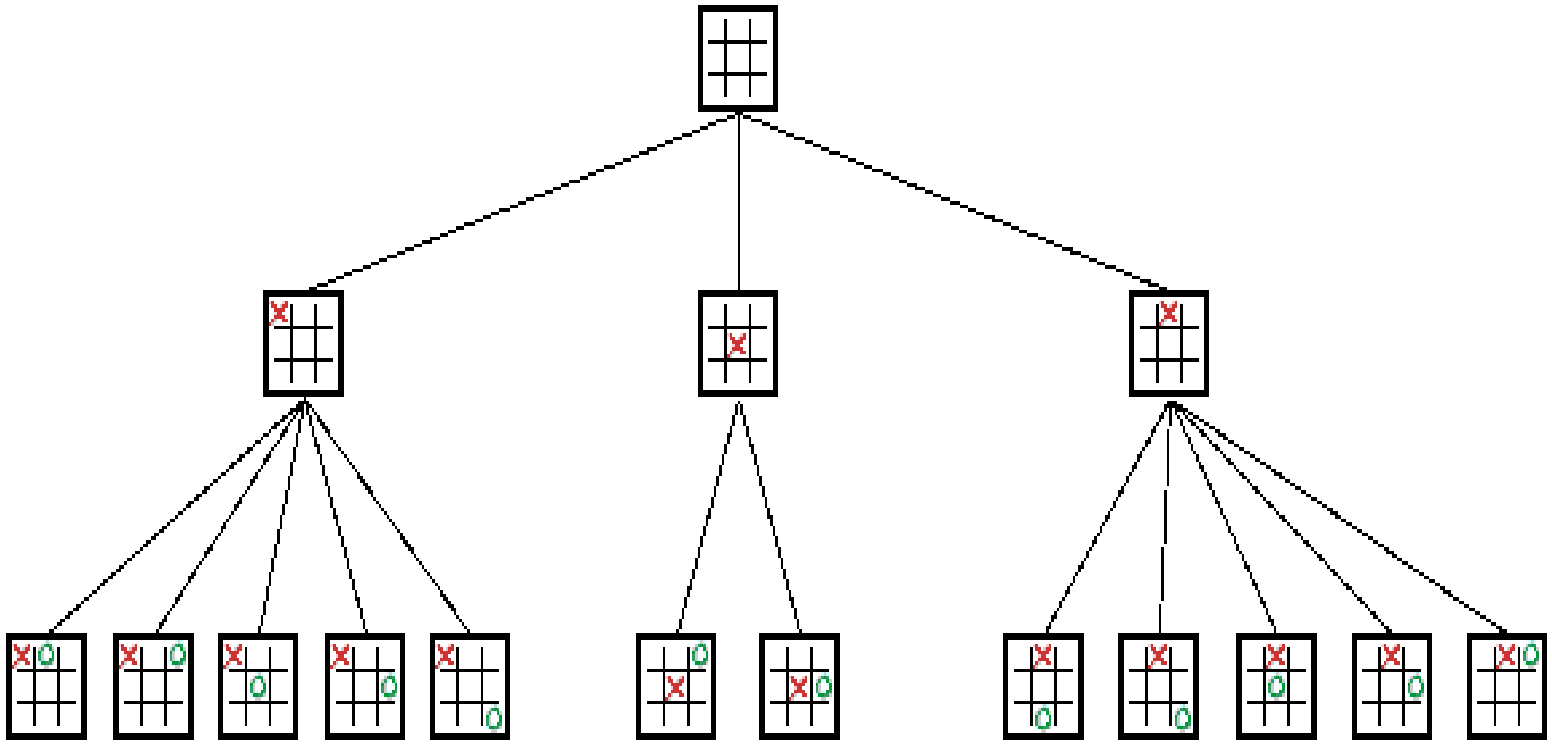
9*8*7*6*5

……

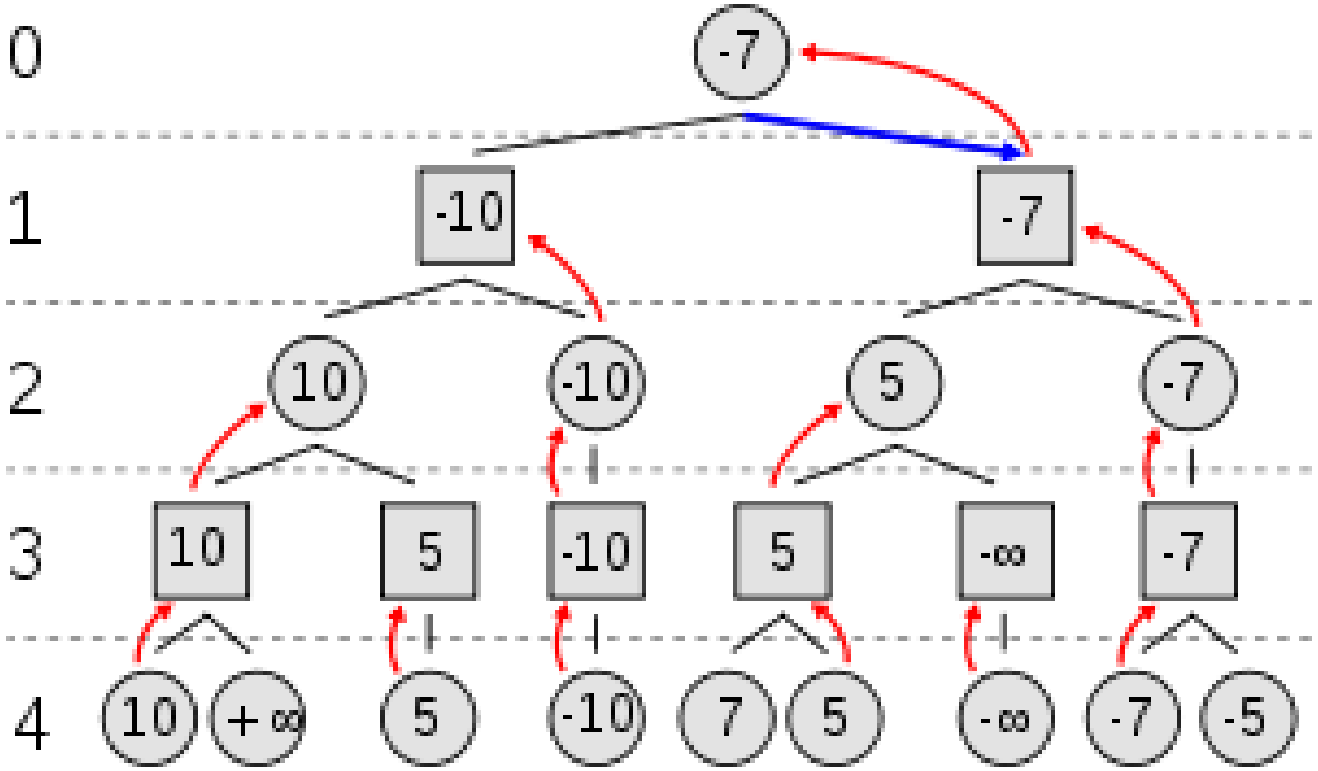


MinMax - Searching tree

- Ex: Tic-Tac-Toe (symmetrical positions removed)
- 对称性

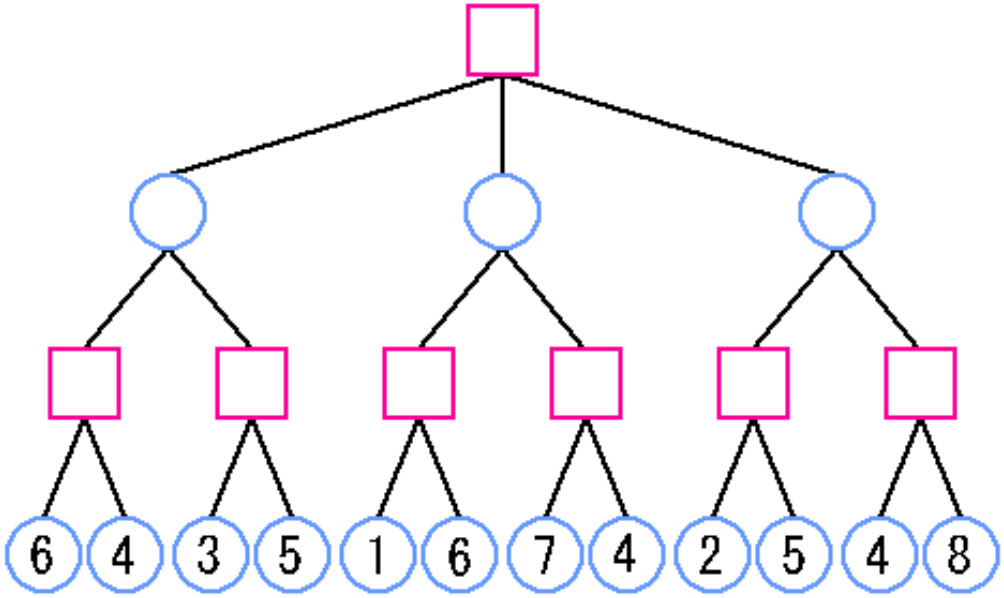


- Min-Max 探索における評価値の動きの例



演習1

Fill in the return value at each blank node



どのように実装するか

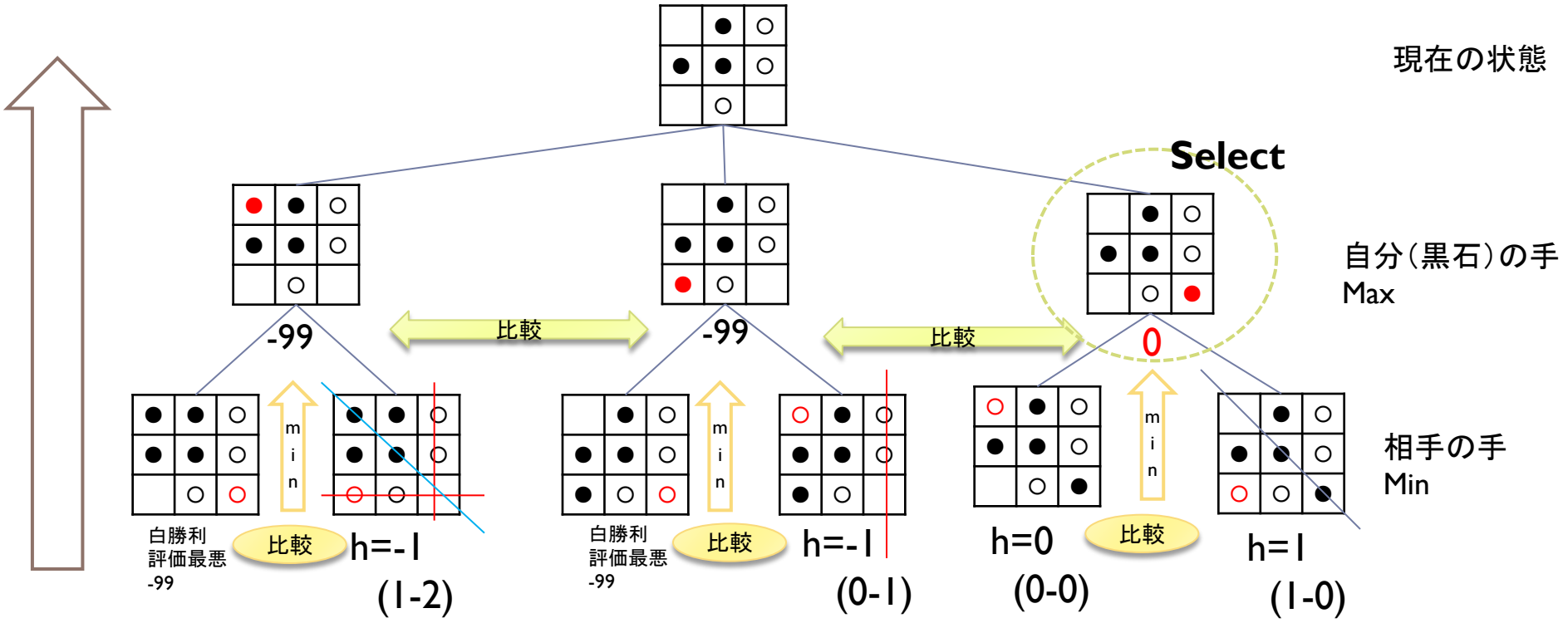
1. 自分が打てる複数の候補手を考えた時、それぞれの手に関して、相手の対抗手を探索する。
    1. これを深読みレベルまで繰り返す
    2. 木の末端まで来た時、評価値を計算する

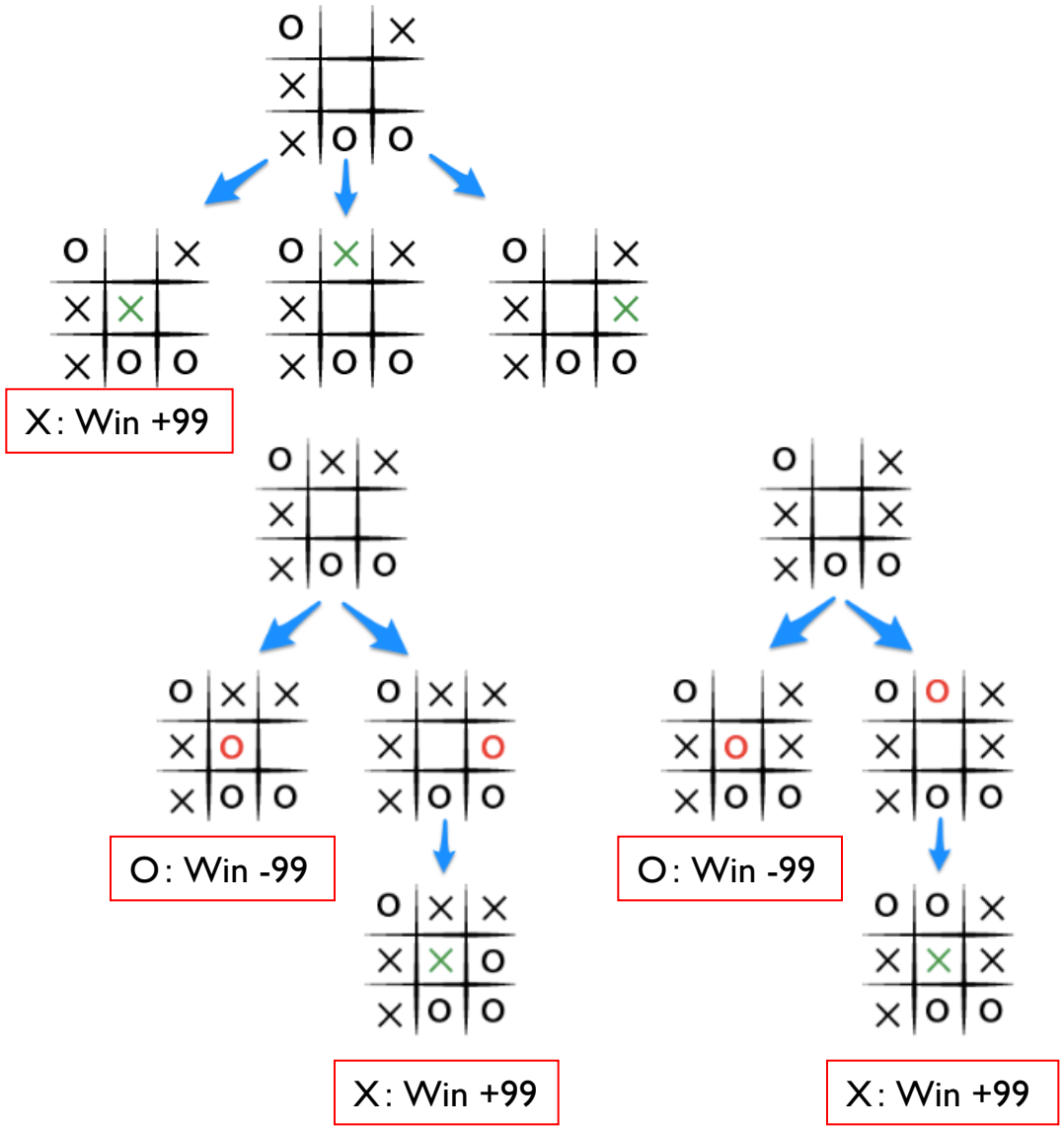
2. 子ノードから返された評価値比較する
    1. AI（自分）のターンなら最大を選択する
    2. プレイヤー（相手）のターンなら最小を選択する

3. 評価値を親ノードへと返す
    1. 自分自身が子ノードの場合、選ばれた評価値を親ノードへ返す
    2. 根の場合、最適な場所を返す

三目並べの例

- 評価値h = h0 – h1
  - h0: 自分の石だけが置かれている縦、横、斜めの列数
  - h1: 相手の石だけが置かれている縦、横、斜めの列数





http://postd.cc/tic-tac-toe-understanding-the-minimax-algorithm/

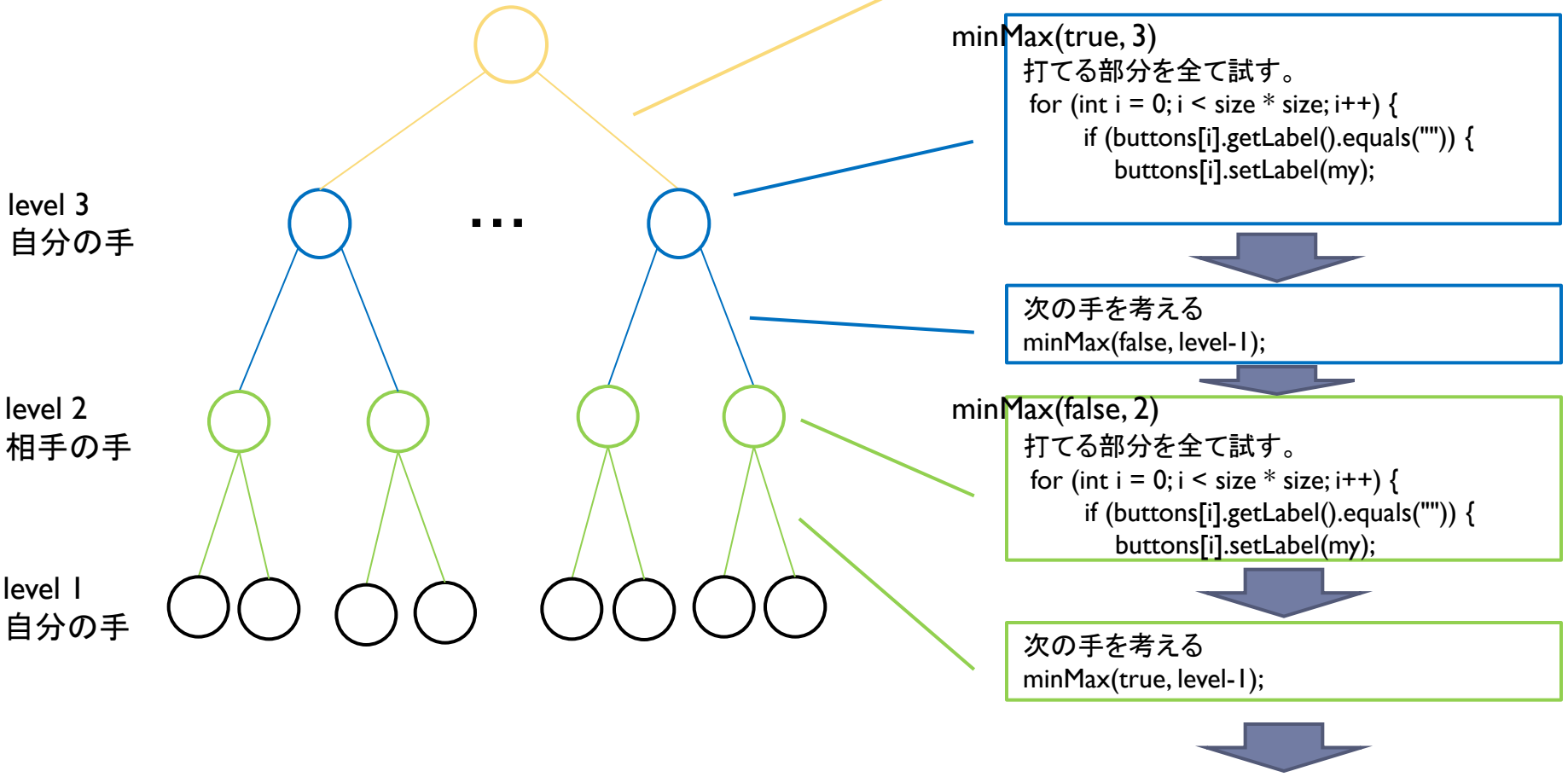
コード部分

flag :AIの手のときtrue、プレイヤーの手のときfalse。
level : 先読みのレベル。

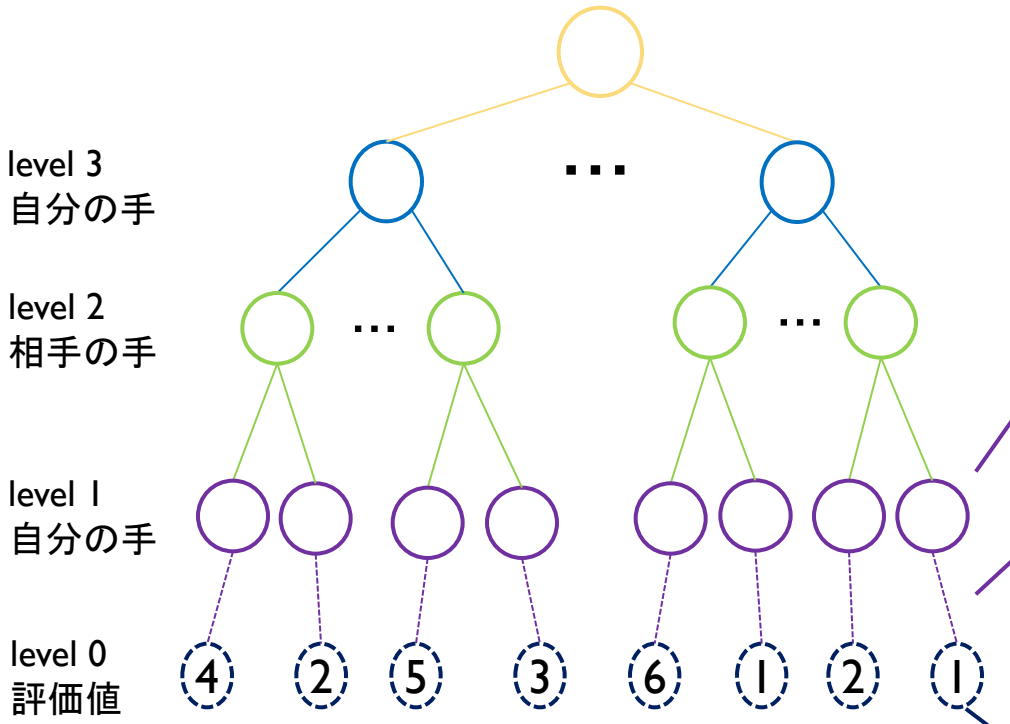
```
public int minMax(boolean flag, int level) {
    String my; // 現在の手
    int value;        // 評価値

    int child;        // 子ノードからくる評価値
    // うつ場所
    int bestPosition = -99; // 仮

1.2 if (level == 0) {
        return evaluation(flag);        //評価値を計算する
    }

    if (flag) { // AIターンでの初期化
        value = -999;
        my = "O";
    } else { // プレイヤーターンでの初期化
        value = 999;
        my = "X";
    }

1   for (int i = 0; i < size * size; i++) {
        if (buttons[i].getLabel().equals("")) {
            buttons[i].setLabel(my);
            if (check(flag)) {
                buttons[i].setLabel("");
                if (!flag)
                    return -1000;
                else
                    return i;
            }
            // 次の手に移動、ここで子ノードの評価値が帰ってくる
1.1         child = minMax(!flag, level - 1);
```

```
2   if (flag) {           // AIならノードの中で最大の評価値を選ぶ
2.1         if (child >= value) {
                value = child;
                bestPosition = i;
            }
        } else {   // プレイヤーならノードの中で最小の評価値を選ぶ
2.2         if (child <= value) {
                value = child;
                bestPosition = i;
            }
        }
        buttons[i].setLabel("");
    }
}

3   if (level == SEARCH_LEVEL) { // 根
        if (bestPosition == -99) { // CPの負けが確実
            JOptionPane.showMessageDialog(null, "まいりました。");
3.2         System.exit(0);
        }
        return bestPosition;
    } else { // 子ノード
3.1     return value; // 評価値を返す
    }
}
```

イメージ1


minMax(true, SEARCH_LEVEL);
minMax(true, 3)
打てる部分を全て試す。
for (int i = 0; i < size * size; i++) {
if (buttons[i].getLabel().equals("")) {
buttons[i].setLabel(my);
次の手を考える
minMax(false, level-1);
minMax(false, 2)
打てる部分を全て試す。
for (int i = 0; i < size * size; i++) {
if (buttons[i].getLabel().equals("")) {
buttons[i].setLabel(my);
次の手を考える
minMax(true, level-1);
level 3
自分の手
level 2
相手の手
level 1
自分の手

イメージ2



minMax(ture, 1)

打てる部分を全て試す。

```
for (int i = 0; i < size * size; i++) {
    if (buttons[i].getLabel().equals("")) {
        buttons[i].setLabel(my);
```

次の手を考える

```
minMax(false, level-1);
```

minMax(false, 0)

レベルが0なので親ノード

```
if (level == 0) {
    return evaluation(flag);   //評価値を計算する
}
```

evaluation(false)

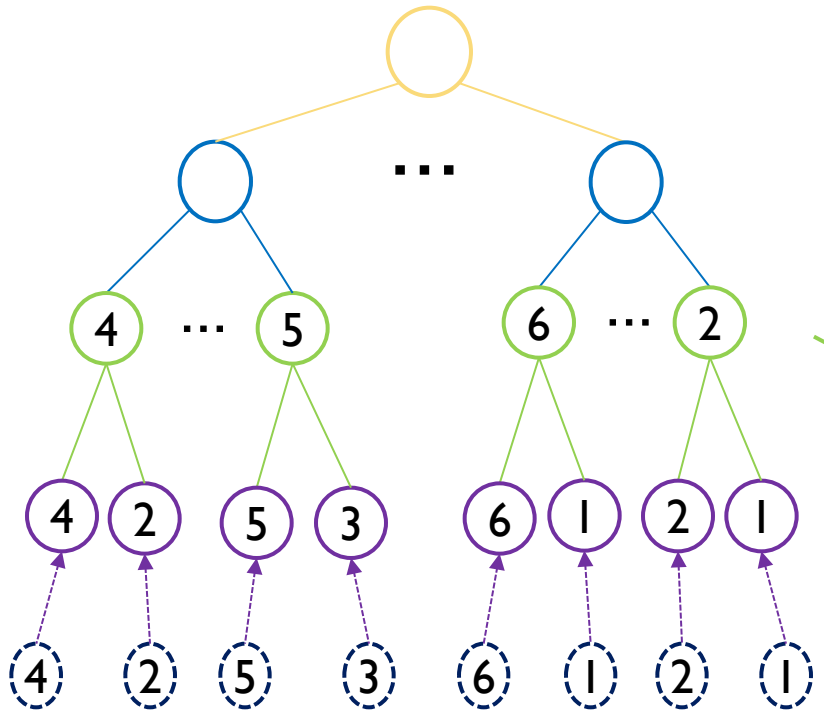
評価値を計算して返す

イメージ3

level 3
自分の手
Max

level 2
相手の手
Min

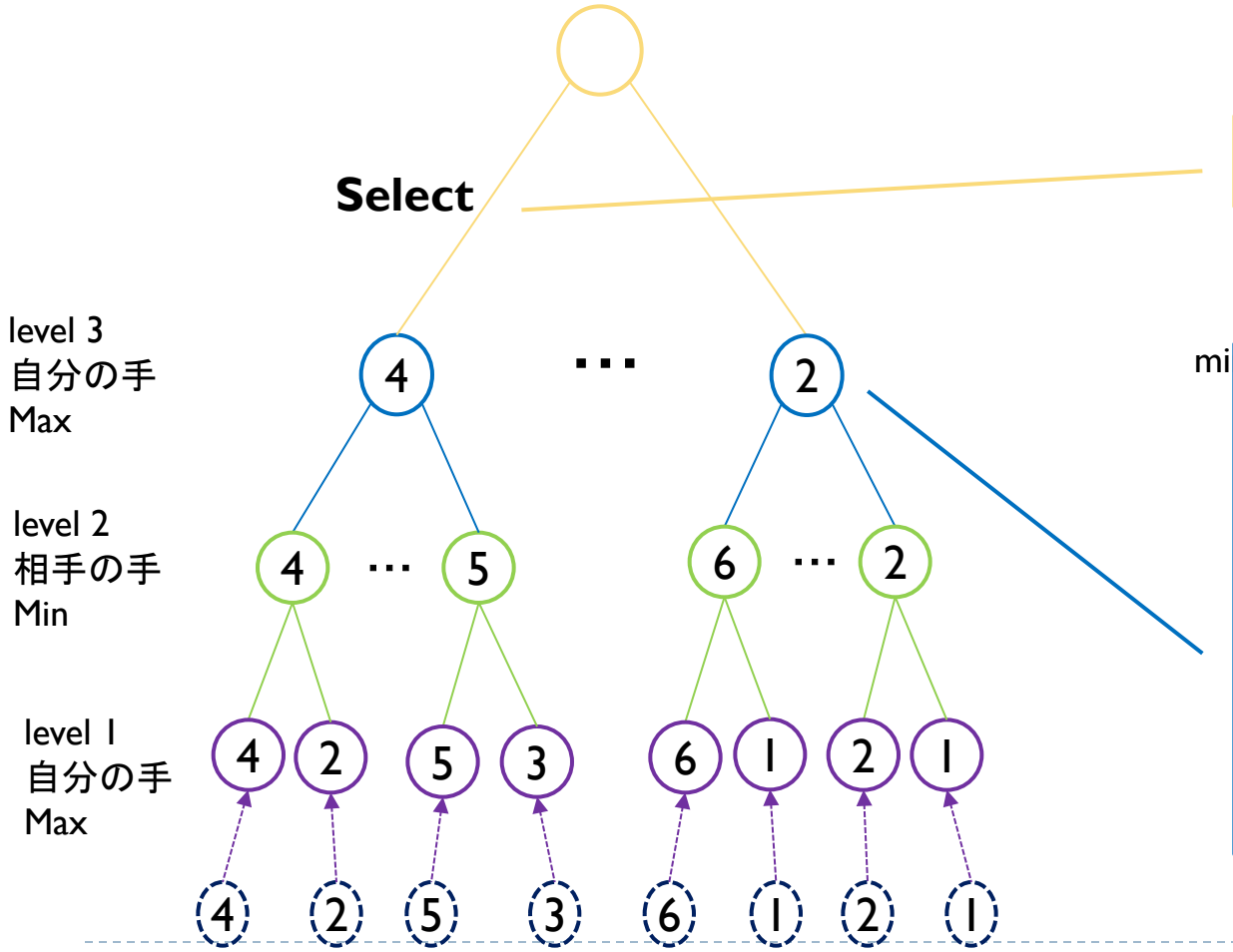
level 1
自分の手
Max




```
minMax(false, 2)
  else {
    if (child <= value) {
    value = child;
    bestPosition = i;
    }
  }
  ・・・
  Levelは2なので親ノードに返す
  return value;
```

```
minMax(ture, 1)
  if (flag) {
    if (child >= value) {
      value = child;
      bestPosition = i;
   }
  ・・・
  Levelは1なので親ノードに返す
  return value;
```

イメージ4



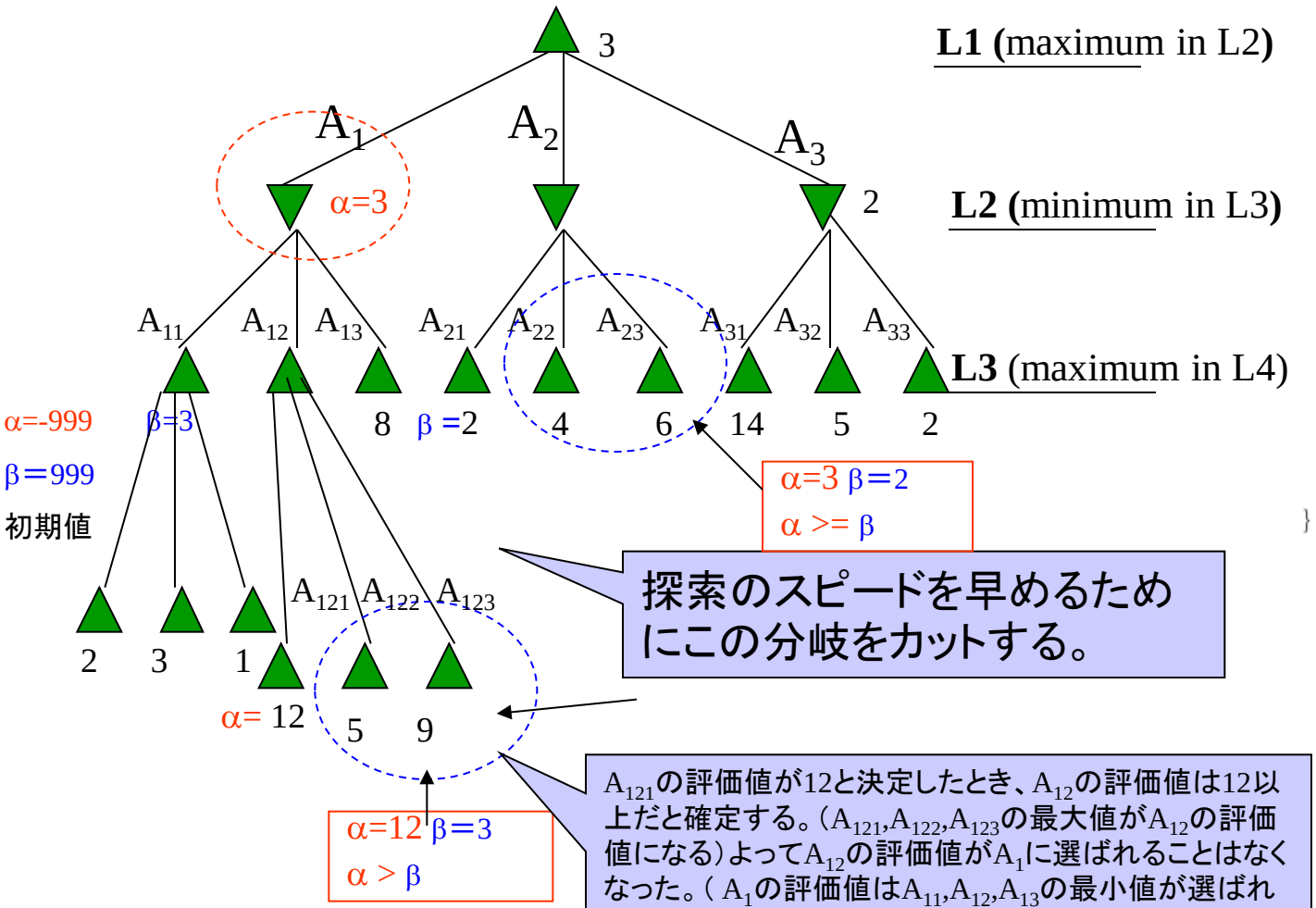
このノードを選択する

```
minMax(ture, 3)
 if (flag) {
    if (child >= value) {
       value = child;
       bestPosition = i;
 }
・・・
Levelは3なので親ノードに返す
if (level == SEARCH_LEVEL) {
・・・
    return bestPosition;
}
```

まとめると:

コンピュータは最後に一番高い評価値を選ぶ.
しかし相手プレイヤーもまた自分にとって良い手を選びます。したがって相手プレイヤーはコンピュータの評価値の最小を選びます。(コンピュータにとって評価値が低いということは自分にとって有利であるから)

α-β pruning (Alpha-Beta法)



L1 (maximum in L2)

L2 (minimum in L3)

L3 (maximum in L4)

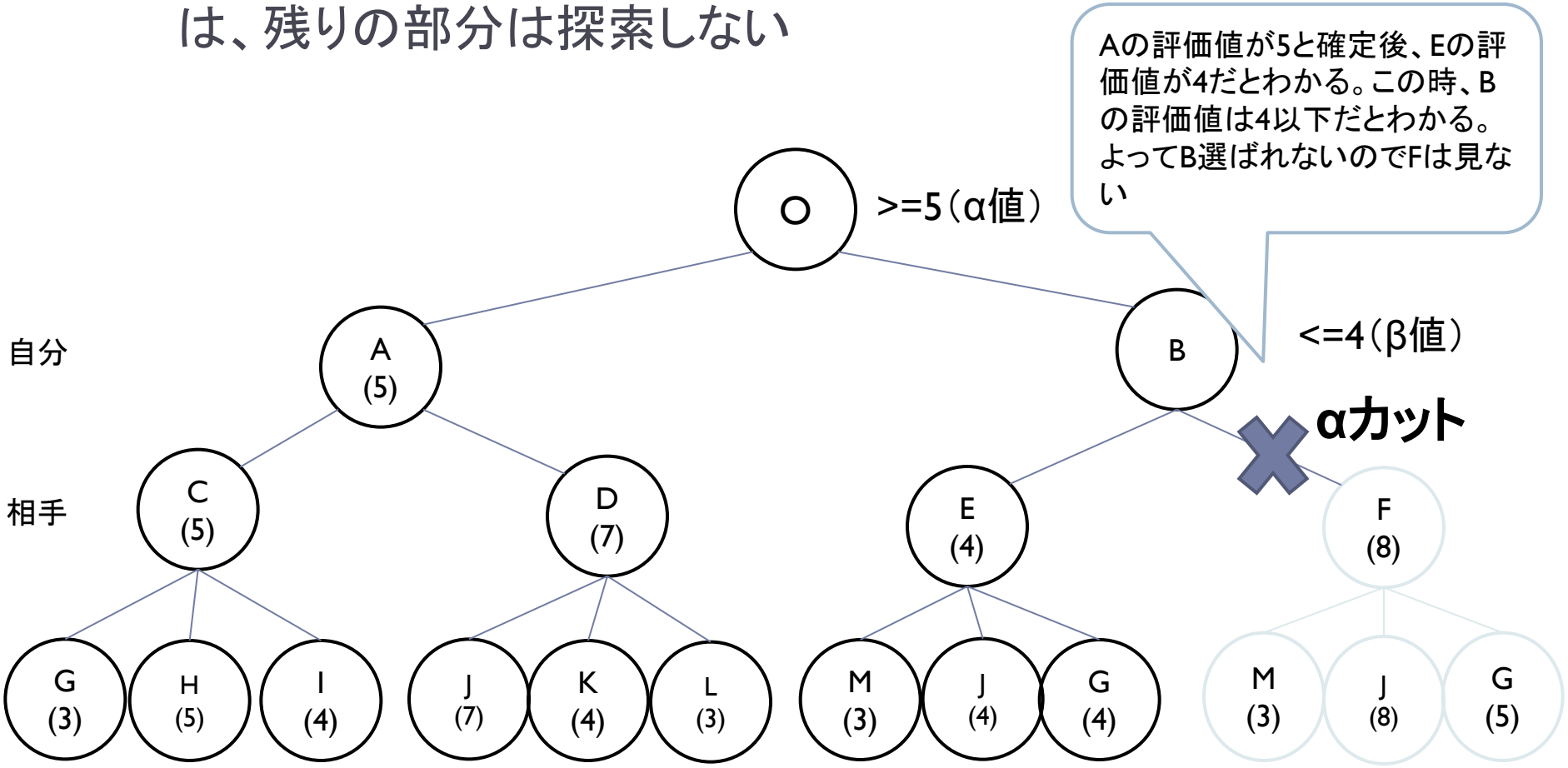
探索のスピードを早めるためにこの分岐をカットする。

A_{121}の評価値が12と決定したとき、A_{12}の評価値は12以上だと確定する。(A_{121},A_{122},A_{123}の最大値がA_{12}の評価値になる)よってA_{12}の評価値がA_1に選ばれることはなくなった。(A_1の評価値はA_{11},A_{12},A_{13}の最小値が選ばれる。)ので、A_{122},A_{123}の評価値を計算する必要がないのでカットする。

```
α β (節点 n, 深さ d, α, β)
{
    if(d==0 || n が終端節点) return(n の評価値);
    α_n=−∞; β_n=+∞;
    if(n が Max 節点) {
        for(全ての n の子節点 n_i) {
            v=α β (n_i, d−1, max{α_n,α}, β);
            α_n=max{α_n,v};
            if(α_n>=β) return α_n;
        }
        return α_n;
    }
    else { // n が Min 節点
        for(全ての n の子節点 n_i) {
            v=α β (n_i, d−1, α, min{β_n,β});
            β_n=min{β_n,v};
            if(β_n<=α) return β_n;
        }
        return β_n;
    }
}
```

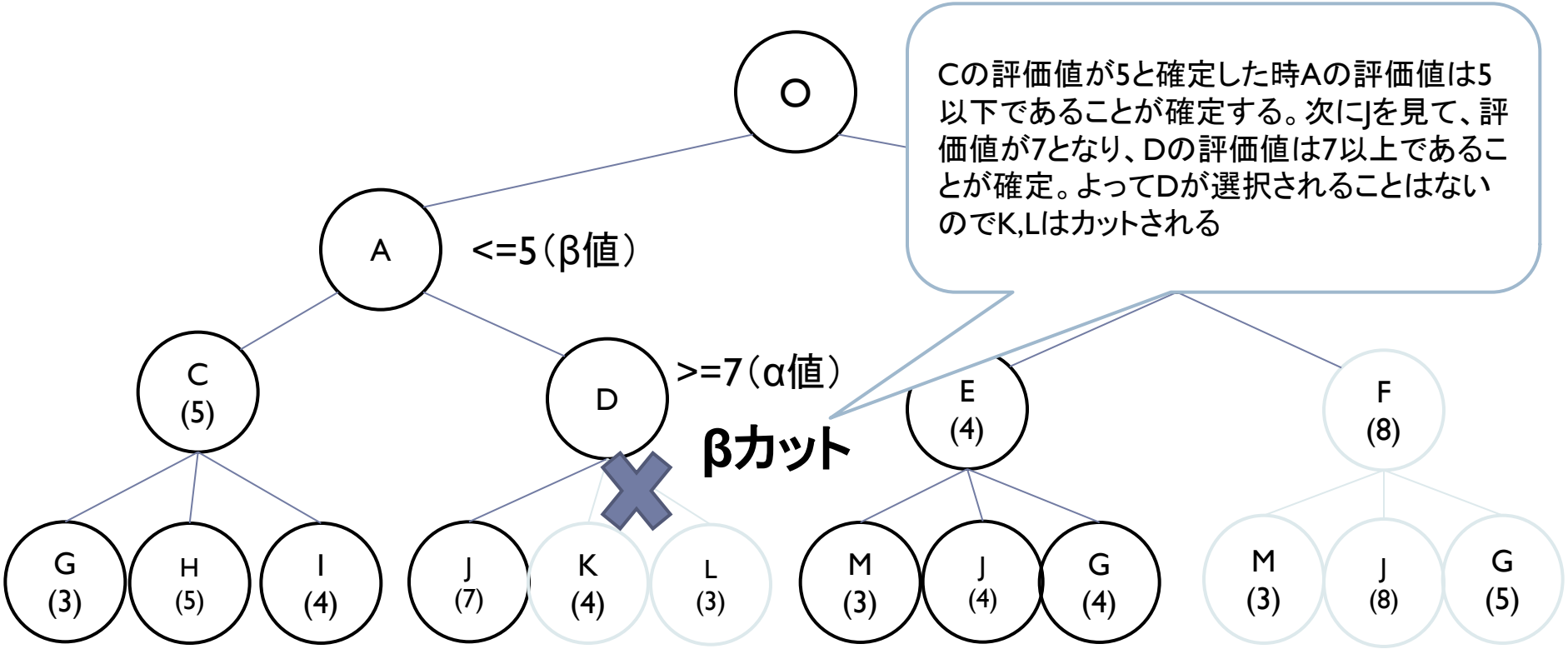
αカット

- 最大の評価値を取ろうとするときの下限（α）以下になる場合は、残りの部分は探索しない



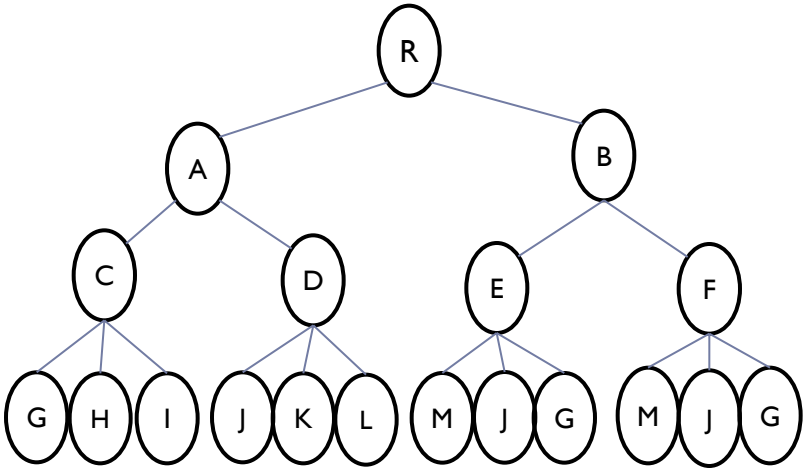
βカット

- 最小の評価値を取ろうとするときの上限（β）以上になる場合は、残りの部分は探索しない

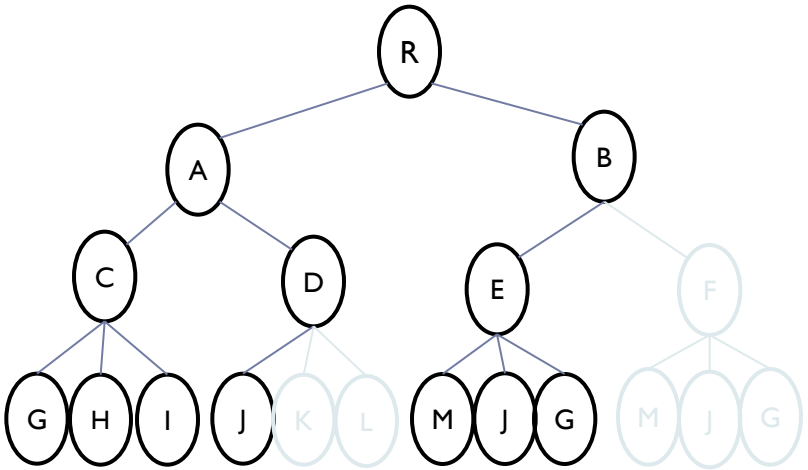


計算回数

- ミニマックス法
  - 12回

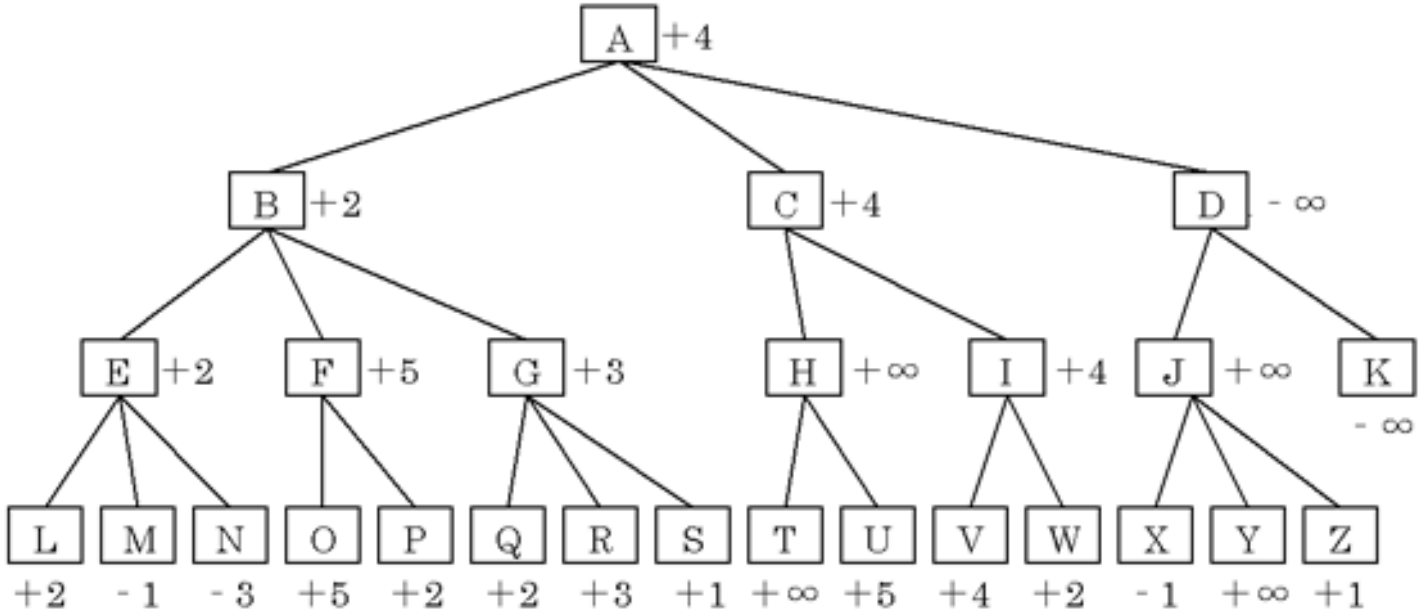


- αβ法
  - 7回



演習２

Draw the branches which can be cut



擬似コード

flag : AIの手のときtrue、プレイヤーの手のときfalse。
level : 先読みのレベル。
alpha : α値。
beta : β値。

演習３:
complete the codes

```
public int alphaBeta(boolean flag, int level, int alpha, int beta) {
 String my; // 現在の手
    int value;       // 評価値
    int child;       // 子ノードからくる評価値
    // うつ場所
    int bestPosition = -99; // 仮

    if (level == 0) {
        return evaluation(flag);
    }
    if (flag) { // AIターンでの初期化
        value = -999;
        my = "O";
    } else { // プレイヤーターンでの初期化
        value = 999;
        my = "X";
    }
    for (int i = 0; i < size * size; i++) {
        if (buttons[i].getLabel().equals("")) {
            buttons[i].setLabel(my);
            if (check(flag)) {
                buttons[i].setLabel("");
                if (!flag)
                    return -1000;
                else
                    return i;
            }
            child = alphaBeta(!flag, level - 1, alpha, beta);
```

1

```
if (flag) {
    if (childValue > value) {
        1.1を書く

    }
        1.2を書く
```

2

```
} else {
    if (childValue < value) {
        2.1を書く

    }
        2.2を書く
}
```

まとめ

ゲームは、自分にとっては最も有利な手を自分が打ち（max）、次に相手が自分にとって最も不利な手を打ち（min）、それらが交互に繰り返されることによって成り立ちます。

<αーβ　法（刈）>

Minimaxを改良したもの。枝刈りを行うことでMinimaxより評価するノードを抑えている

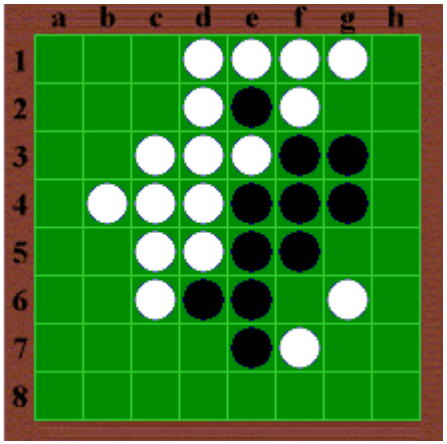
<Minimax　algorithmとαーβ　algorithmの違い>

Minimax法ではすべてを探索し最良の手を選択するのに対して、α-β法は、minimax法で採用されないと判断された手については、そこから先を探索しないことで無駄な探索に費やす時間をカットしている。また、α-β法による結果はminimax法での結果と同じになる。

枝刈りを行うことにより探索がminimax法より早く終わるのでα-β法のほうが効率的である。

オセロゲームの評価方法について

http://uguisu.skr.jp/othello/5-1.html



	a	b	c	d	e	f	g	h
1	120	-20	20	5	5	20	-20	120
2	-20	-40	-5	-5	-5	-5	-40	-20
3	20	-5	15	3	3	15	-5	20
4	5	-5	3	3	3	3	-5	5
5	5	-5	3	3	3	3	-5	5
6	20	-5	15	3	3	15	-5	20
7	-20	-40	-5	-5	-5	-5	-40	-20
8	120	-20	20	5	5	20	-20	120

- この図では隅に重みを上げ、隅の周りのマスの重みを下げていることが分かります。
- この評価値の付け方だと「隅を取ると有利になる」ことしか満たしていません。

例えば、左の局面を上記の評価値を用いて計算すると、
白：＋33
黒：＋13
となります。

したがって、この局面では白が評価値20ほど有利だと言えます。
しかしながらこれは誤りです。
この場面では黒が有利と判定されなければなりません。

石の位置による評価を重視しての方法。

各マスの評価点は、中央の方が点数が高くなるように、
また基本的に負の値に設定する方法です。

	a	b	c	d	e	f	g	h
1	30	-12	0	-1	-1	0	-12	30
2	-12	-15	-3	-3	-3	-3	-15	-12
3	0	-3	0	-1	-1	0	-3	0
4	-1	-3	-1	-1	-1	-1	-3	-1
5	-1	-3	-1	-1	-1	-1	-3	-1
6	0	-3	0	-1	-1	0	-3	0
7	-12	-15	-3	-3	-3	-3	-15	-12
8	30	-12	0	-1	-1	0	-12	30

- こうすると、自分の石が多いほど合計点が小さくなることは分かります。負にすることで何のメリットがあるのでしょうか？
- 実はこの方法だと、「相手に囲ませる」、「石を多くとらない」といった、「リバーシ」の必勝通りの戦法になります。

例えば、左のような重み付けをしたとすると、先ほどの局面は
白：－23
黒：－18
となり、黒が優勢となります。

ヒント:

「序盤は隅が非常に重要だが、終盤はそれほど重要でない」など、
石の位置の重みも変化してくるので、この評価値を「序盤」「中盤」
「終盤」などに分けて数パターン作成しておきます。

(Optional) 課題をするにあたってやってほしいこと:
(1) TicTacToe_MinMax Java

TicTacToe_alphabeta.java

プログラムをダウンロードする

(2) 実行する
(3) ソースコードを読み理解する
(4) “ゲームAI"をキーワードにして検索し、出てきた記事を読む

課題:
自分の言葉で以下のアルゴリズムを説明しなさい
(1) MinMax
(2) α－β

参考サイト:

Java
http://fuktommy.com/java/

In python
http://www.geocities.jp/m_hiroi/func/abcscm43.html

Othello Game (Min-Max):
http://uguisu.skr.jp/othello/minimax.html
http://www.net.c.dendai.ac.jp/~ksuzuki/

α－β
http://hp.vector.co.jp/authors/VA015468/platina/algo/2_3.html