囲碁プログラミング講習会

～ マニュアル(その２)～

平成１６年６月１２日(土)
平成１６年６月１３日(日)

目次

６月１２日(土)

６月１３日(日)

囲碁プログラミング講習会

日程：2004年6月12日（土）
場所：ソフトピアジャパン・ドリームコア
講師：清愼一、佐々木宣介
（コンピュータ囲碁フォーラム）
主催：岐阜チャレンジ2004実行委員会

本日のスケジュール

- 第1部　前回のおさらい
- 第2部　探索（先読み）
- 第3部　盤面認識のデータ構造
- 第4部　候補手生成の概要

第1部　前回のおさらい

- 人工知能と囲碁
- 強い囲碁プログラムを作るには
- 囲碁プログラムの構造

人工知能とは

- 「コンピュータが、あたかも知能があるかのように振舞っていると感じる」時、それを人工知能と言う
- 「人間の知的活動をコンピュータ（プログラム）で実現させようとする」のが人工知能の研究目的である

ゲーム研究の貢献

- 特にヨーロッパでは、チェスは知性の象徴とされてきた
- 強いゲームプログラムの実現（ゲームプログラミング）は、人工知能の黎明期から盛んに研究され続けている

Chess is the Drosophila of Artificial Intelligence.
(by Alexander Kronrod)

チェスは人工知能のショウジョウバエである

ゲームプログラミングの特徴

- 研究の題材として扱いやすい
  - ルールが明確である
  - 勝ち負けがはっきりつく
  - ゲーム自体がおもしろい
- 人工知能の様々な要素技術を含んでいる

機械学習、知識表現、推論、探索、ニューラルネット、ファジー推論、知識ベース、分散・協調、マルチエージェント、感性情報、認知科学、教育など

ゲームプログラムの強さ

- 解かれたゲーム（必勝法解明）
  - 五目並べ、オセロ（6×6）
- 世界チャンピオンと同等以上
  - チェッカー、オセロ、チェス
- まだ人間の方が強いゲーム
  - 将棋：アマチュア4、5段程度
  - 囲碁：アマチュア初段程度

思考ゲームの中では、囲碁は最も難しい！？

ゲームプログラムの難しさ

- ゲームプログラムの手法は主に2つ
  - **探索**：読みのこと、試行錯誤的に全ての可能性を調べる
  - **ヒューリスティクス**：定理や法則など、問題を解くのに役に立つ知識や経験を利用する
- どちらかの手法だけで解けるゲームは少ない。適当に組み合わせて作る。

囲碁とは

- 囲碁とは、地を取り合うゲーム
- 大きい地（地の合計）を取った方が勝ちだが
- 相手の石を自分の石で囲うと、相手の石を盤上から取り除くことができる
  - 完全に囲わなくても、2眼作るスペースが無ければ、最終的には取られる
- 相手の石を取りながら（取るぞと脅しながら）、地を作る（地を大きく囲う）複雑なゲームである

囲碁が強くなるには

- 囲碁が強くなるには（強いプログラムを作るには）次の2つが必要

（1）地の認識ができる
（2）石の生死の判断ができる

地の認識

- 地の認識をするために、地の定義を決める
  - 同じ色の石で囲われた場所
  - 同じ色の石によって囲われているとみなすことができる（同じ色の石がつながっているとき、石が無い点も石があると考える）場所
- 今は地で無くても、何手が後には自分の地になりそうな場所（模様、勢力）も、判断できると、なお良い

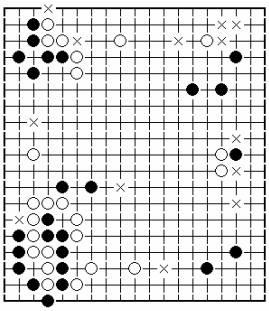
石の生死の判断

- 石の生死の判断をするために、石の生死の単位を定義する
- 群の生死の判断をするために、死（または生き）を定義する
  - 眼を2つ持っているか
  - 眼を2つ作る大きさを持っているか
  - 生きている味方の石につながることができるか
  - 敵の包囲網を突破して逃げることができるか
  - 包囲している敵の石を取ることができるか
- 早い（石が少ない）段階で生死の判断ができる（生きやすさ）と、なお良い

囲碁が強くなるには (2)

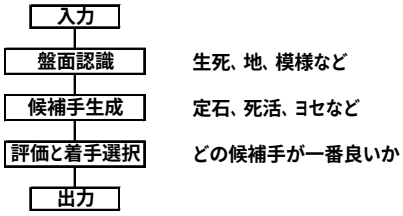
- 長時間考えることができれば誰でも良い手を打てるが、持ち時間には制限がある
- 短時間で良い手を打つには候補手生成の知識が必要
  - たくさんの定石や手筋を知っている
  - 良い形 (スキが無い、効率が良い) を知っている
  - たくさんの格言 (一間トビに悪手なし、2目の頭見ずハネよ) を知っている

黒は次にどこに打つ？



- 人間 (上級者) は、盤面を見ただけで、打ちたい場所がすぐに思いつく
- 無意識のうちに多くの処理を行っている
- しかし、これをプログラムで実現するのは難しい

囲碁プログラムの処理の流れ



盤面認識

- 現在の状況を正確に認識する
  - どこに黒石／白石が置かれている？
  - どの石が生きている／死んでいる？
  - どの石がつながっている／切れている？
  - どこが地／勢力／ダメ場？
  - どの石が強い／弱い？
  - どちらが何目勝っている？
- 盤面認識の結果は
  - 候補手生成の基になる
  - 最善手を選ぶときの評価基準になる

候補手生成

- 盤面認識の結果から、戦略をたてる
  - どこに地を作ろう
  - どの石から攻めよう
  - 流れにまかせよう
- 認識結果と戦略から、複数の候補手を生成する
  - 布石、定石、手筋
  - 死活 (取る／逃げる)、攻め／守り、連結／切断
  - 地を囲う／侵略する
  - ハネ／ノビ／一間トビ／ケイマ、、、
  - 形の良い手

評価と着手選択

- 候補手を仮に打ってみたときの盤面の状態を認識し、どちらが何目有利か調べる
- 最も有利になる候補手を、着手として選択する
  - 目数に換算しにくい手 (模様や勢力、石の強さ) もあるので注意が必要
  - 定石や手筋の途中ならば、その手順通り続けることも重要である
  - 捨て石や劫材などの一見無駄に見える手が、重要な手になることもある

碁盤プログラムの作成

- 盤面を表示する（出力）
- 石を打つことができる（入力）
- 石を取る処理ができる
- 合法手の判断ができる
- 交互に着手できる
- 終局できる

地の認識の実験

- 地とは何か？
- 地の定義の実験（石の周囲は地である）
- 実験結果から、より人間の認識結果に近づくように改良
- 地を評価値にした思考部を作成

囲碁プログラムのデータ構造

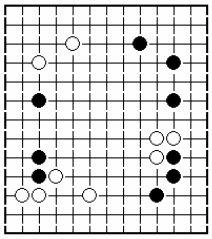
- データ構造とは、「データをコンピュータで効果的に扱うための形式」
- 囲碁の場合、黒石／白石／空点（石が置かれていない点）を表現できれば十分なはずだが、、、
- 人間は、複数の同色の石を一塊と見ているので、その集団を表現するデータ構造が必要
- しかし、人によってその集団の定義（どの石を一塊としているか）は、ばらばら

典型的なデータ構造の例

点(point)	一つ一つの座標ごとの状態（黒、白、空点）
連(string)	縦横に連結している同色の石
群(group)	つながっていると考えられる同色の石の集団
結線(linkage)	石と石の連結を表す仮想的な線 群を定義するときに使う

データ構造（点）

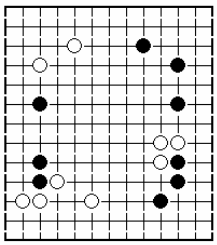
空点：0、黒：1、白：2



```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 2 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 2 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 2 2 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 2 1 0 0 0 0 0 0 0
0 0 1 2 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 2 2 0 0 2 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

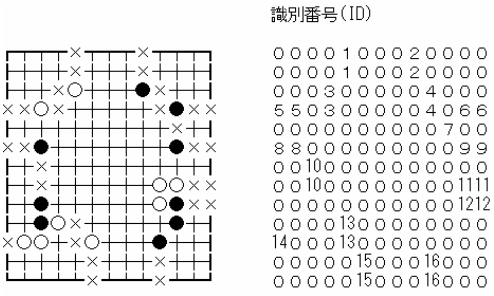
データ構造（連）

識別番号（ID）

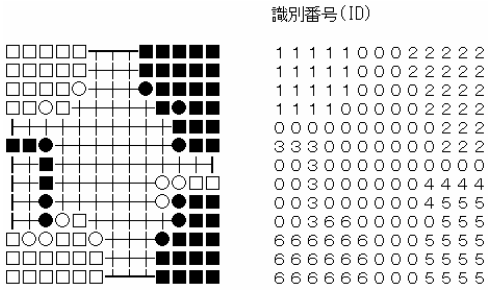


```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 2 0 0 0 0 0 0 0 0 0 0
0 0 3 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 5 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 7 7 0 0 0 0 0 0 0 0 0
0 0 8 0 0 0 0 0 0 7 9 0 0 0 0 0 0 0 0
0 0 8 10 0 0 0 0 0 0 9 0 0 0 0 0 0 0 0
0 1 1 1 1 0 0 12 0 0 0 13 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

データ構造 (結線)



データ構造 (群)



群の使いみち

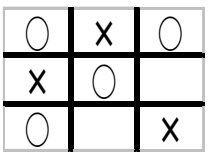
- 候補手生成
  - 地が小さいなら、地を増やす手を生成する
  - 近くに味方がいるなら、連結する手を生成する
- 評価
  - 地が計算できる
  - 群の形状がわかるので、強さを計算できる

第2部 探索 (先読み)

- 探索とは
- 探索アルゴリズム
- 囲碁プログラムへの応用

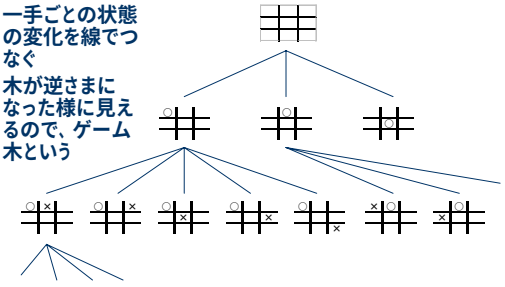
3目並べ (Tic-Tac-Toe)

- 3×3のマス目に、○と×を交互に置いていき、縦または横または斜めに3つ並んだ方が勝ち、というゲーム

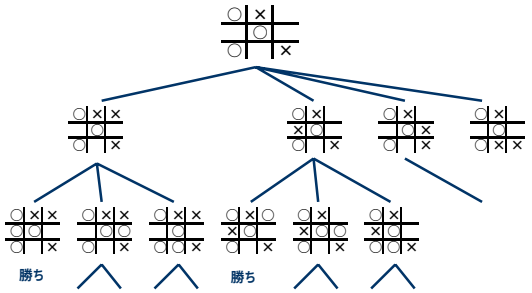


ゲーム木

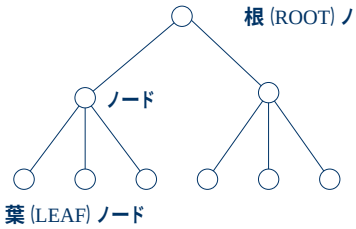
- 一手ごとの状態の変化を線でつなぐ
- 木が逆さまになった様に見えるので、ゲーム木という



ゲーム木



ゲーム木

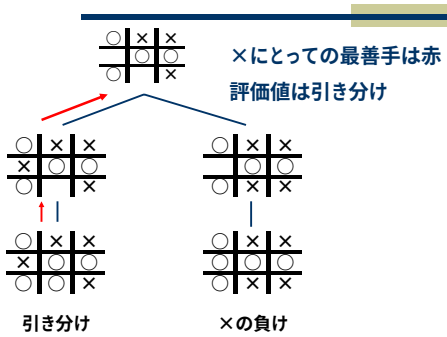


根から深くなるに従って、自分の番と、相手の番が交互に現れる

ゲーム木探索とは

- ゲーム木探索とは、ゲーム木をたどることによって、(1) 最善手につながる枝と (2) 最善手の評価値を見つけること

ゲーム木探索の簡単な例

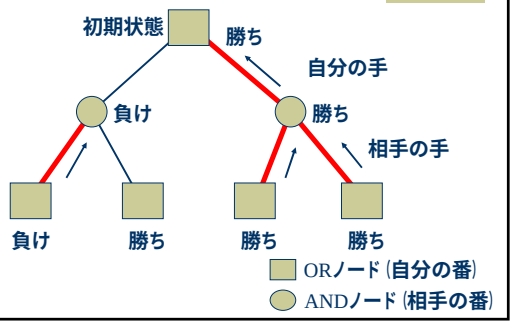


ゲーム木探索の性質

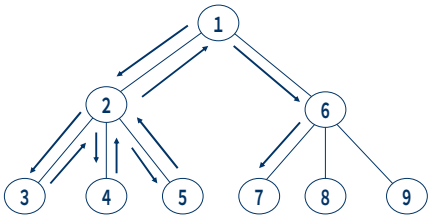
- 自分の番では、「負け」が幾つあっても、「勝ち」が一つあれば「勝ち」になる。
- 相手の番では、「勝ち」が幾つあっても、「負け」が一つあれば「負け」になる。全部「勝ち」でなければ、「勝ち」にならない。

注意：相手は必ず最善の手を選ぶと仮定する

AND／OR木



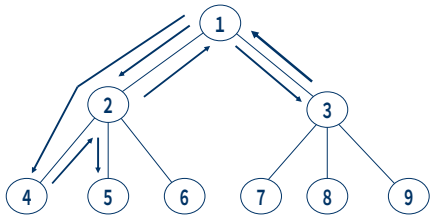
深さ優先探索



最初の枝を末端まで探索し、その後、次の枝へと移っていく

プログラムが簡単に書ける

幅優先探索



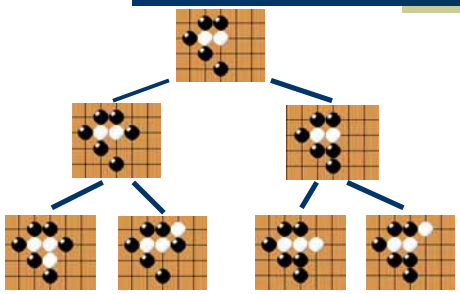
浅い部分から探索し、その後、深い枝へと移っていく

最短経路（最短の深さ）が求められる

囲碁におけるAND／OR木

- 石の生死を調べる
- 石と石の連結を調べる
- 先手かどうか調べる

石の生死を調べる探索



深さ優先探索（ANDノード）

```
int SearchAnd( 現在の盤面 )
{
    if( この盤面が終端の場合 )  return ( 結果 );
    可能な手を全て生成する
    for( それぞれの手に対して ){
        その手を打った盤面を作る
        Score = SearchOr( 打った後の盤面 );  /* ORノードの処理へ */
        if( Score = 負け )  return( 負け );
        手を打つ前の盤面に戻す
    }
    return( 勝ち );       /* ORノードから負けが一つも返ってこない */
}
```

深さ優先探索（ORノード）

```
int SearchOr( 現在の盤面 )
{
    if( この盤面が終端の場合 )  return ( 結果 );
    可能な手を全て生成する
    for( それぞれの手に対して ){
        その手を打った盤面を作る */
        Score = SearchAnd( 打った後の盤面 );  /* ANDノードの処理へ */
        if( Score = 勝ち )  return( 勝ち );
        手を打つ前の盤面に戻す */
    }
    return( 負け );       /* ANDノードから勝ちが一つも返ってこない */
}
```

石の生死を調べる場合

- 「この盤面が終端の場合」とは、「石のダメが0＝死に」になるか「合法手が無い＝生き」である
  - 実際は、ダメが0になるまで(合法手が無くなるまで)探索させたらノード数が多すぎて終わらない
  - ある深さまで到達したら、合法手があっても終端(生き)とする
  - ダメ数がある数以上ならば、終端(生き)とする

石の生死を調べる場合

- 「可能な手を全て生成する」とは、「石の周囲の座標を生成する」こと
- 可能な手を生成せずに、「候補手としてふさわしいならば、処理を進める」やり方もある

```
for( 全ての座標に対して ){
    if( 候補手としてふさわしい ){
        その手を打った盤面を作る
        Score = SearchOr( 打った後の盤面 );  /* ORノードの処理へ */
        if( Score = 負け )  return( 負け );
        手を打つ前の盤面に戻す
    }
}
```

石の生死を調べる場合

- 「その手を打つ前の盤面に戻す」処理は2通りの方法がある
  - 打った石をはがす(もしも、打ったことによって石を取っているなら、その石を盤面に戻す)
  - その手を打つ前の盤面をコピーして保存しておき、保存した盤面を使う

石の生死の判定

- 黒から打った場合と、白から打った場合の2回の探索を行う
- 探索結果と以下の表から、生死が求まる

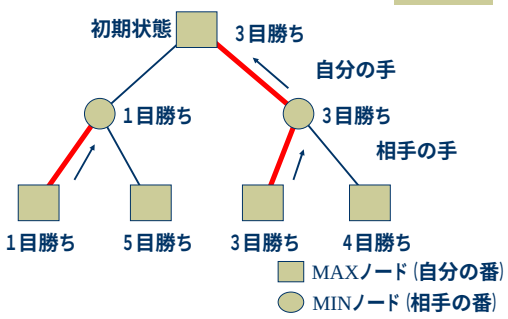
黒の連の生死

黒番＼白番	生き	死に
生き	生き	中立
死に	生き	死に

ミニマックス法

- 終端の評価値が2値(「勝ち」「勝ちではない」)でなくても良い
- ゲーム木の中で、最も良い手(最も評価値の高い手)を求めることができる
  - ゲーム木の終端まで記述できないとき(ゲーム木が巨大なときなど)は、仮の終端ノードで何らかの方法で評価値を算出する

ミニマックス法



囲碁におけるミニマックス法

- 詰碁（生き／死に／コウ（取り番、ヨセコウ））
- 盤面全体で、最も良い手を選択する

ミニマックス法（MINノード）

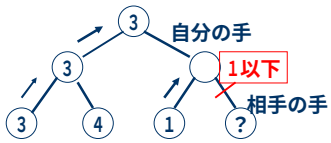
```
int SearchMin( 現在の盤面 )
{
    if( この盤面が終端の場合 )  return ( 結果 );

    ScoreMinの初期化(大きな値を設定)

    可能な手を全て生成する

    for( それぞれの手に対して ){

        その手を打った盤面を作る

        Score = SearchMax( 打った後の盤面 );  /* MAXノードの処理へ */

        if( Score < ScoreMin ){
            ScoreMin = Score;              /* より悪い手が見つかった */
        }

        手を打つ前の盤面に戻す
    }

    return( ScoreMin );  /* 最小値が返る */
}
```

ミニマックス法（MAXノード）

```
int SearchMax( 現在の盤面 )
{
    if( この盤面が終端の場合 )  return ( 結果 );

    ScoreMaxの初期化(小さな値を設定)

    可能な手を全て生成する

    for( それぞれの手に対して ){

        その手を打った盤面を作る

        Score = SearchMin( 打った後の盤面 );  /* MINノードの処理へ */

        if( Score > ScoreMax ){
            ScoreMax = Score;              /* より良い手が見つかった */
        }

        手を打つ前の盤面に戻す
    }

    return( ScoreMax );  /* 最大値が返る */
}
```

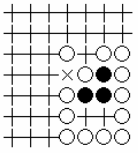
枝刈り

- 全ての枝を探索しなくても、最善手を求めることができる場合がある
- このとき、探索しない枝を刈るという
- αβ法などが有名



水平線効果（地平線効果）

- 探索の深さを制限しているときに起きる現象
- 嫌な結果を先延ばしにするために、少し損をする手を選ぶこと
- 嫌な結果が深さ制限の先に行くので、最善手を選んでいるように見えてしまう



白から２手で黒は取られるが

黒が×に打つとアタリなので黒３子を取るまでの手数が１手延びる

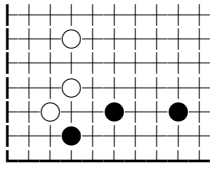
しかし死に石が増えるので損

第3部 盤面認識のデータ構造

囲碁プログラミング講習会(6月12日)

盤面の認識

強いプログラムを作るために最も重要なことは、盤面の状況を詳しく正確に認識することです。盤面の状況と言っても、黒石や白石がどこにあるかということではありません。どの石が生きていてどの石が死んでいるのか、どの石が重要な石なのか、などといった視覚情報だけではわからない（考えて初めてわかる）情報です。このような情報を正確に認識できないと、どの手が良いのか悪いのかわかりません。
今、「どの石が」と言いましたが、これは一つ一つの石を指しているのではなく、複数の石からなる集合を指しています。人間は、複数の石からなる集合を一つの単位として捉えていて、その集合ごとに情報を認識しています。一般的な囲碁プログラムでは、データ構造として「点」「連」「群」「結線」を持っています。



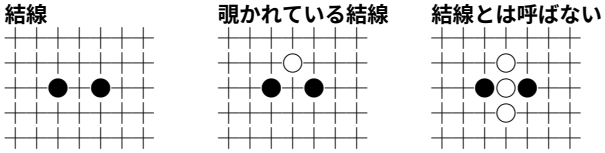
人間は、６つの石を別々に捉えていない

黒３つを一つのグループ、白３つを一つのグループとして捉えている

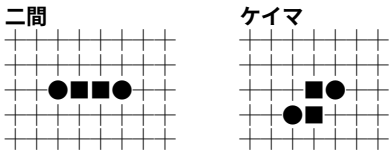
石と石との接続関係(結線)

石と石との接続関係について考えましょう。多くの囲碁プログラムは、石と石との間に仮想的な線を引いて、石どうしを結んでいる「結線（けっせん）」というデータ構造をもっています。（海外では英語でlinkage（リンケージ）と呼ばれることが多い）。石と石が接続していれば、その石を一つの集合として捉えることができるので、石と石との接続関係(結線)を認識するのです。

以下にその例を描きます。左図の黒(一間)は、周囲に敵の白石が無いので接続していると考えられるでしょう。そこでこの二つの石の間には結線があると言います。中図の黒は白に覗かれている（白から切断可能）ので結線と呼びません（覗かれているという情報を持った結線と呼ぶプログラムもあるようです）。右図の黒は白によって完全に接続が絶たれているので、この状態で結線と呼ぶプログラムは無いようです。



以下は、結線を■で表した図です。

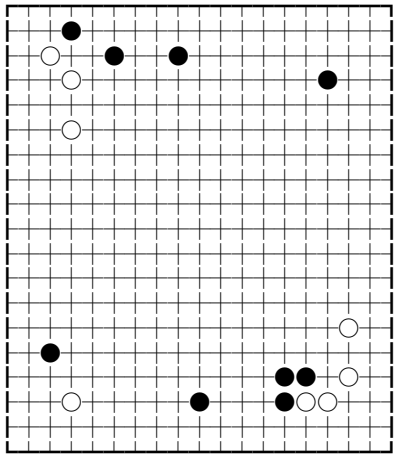


結線を考えるときの石と石の位置関係は、一間、二間、コスミやケイマなどです。あまり石と石が大きく離れている場合は考える必要はないでしょう（プログラムによっては、大きく離れていても結線を考えるものもあるようです）。

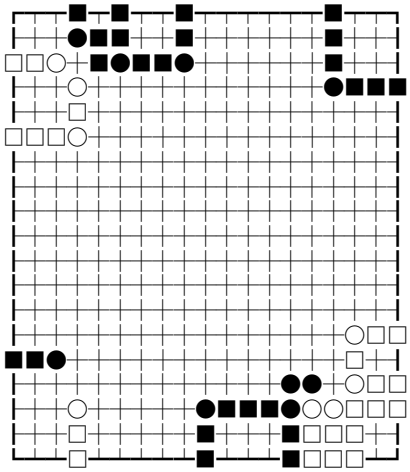
また、石と石の間だけではなく、石と盤端も結線として認識しています。以下にその例を描きます。

盤面全体では、以下のようになります。
結線の定義はプログラムによって異なるようです。例えば、右下の(14,17)の黒と(15,17)の白のように隣あっている場合、図では三線と認識している場合が描かれていますが、「相手からハネる手があるから結線ではない」と処理するプログラムもあります。

盤面



結線



結線のデータ構造の例

それぞれの結線について、どんな情報を認識するとよいのでしょうか？ここでは、多くの囲碁プログラムが認識している情報を挙げておきます。

- 結線の両端の位置(二つの石の座標)
- 結線の位置(石と石の間の座標)
- 色
- 位置関係(一間/コスミ/ケイマなど)
- 接続の度合(つながっている/切れている/先手ならつながる)

接続の度合いの調べ方

結線の接続の度合いを調べる方法はいろいろあり、プログラムによって異なるようです。主な調べ方は次の3つです。

- パターン
- 探索
- ポテンシャル（影響度）

パターンとは、各座標に対して、黒石があるのか、白石があるのか、空点なのかを書き記したものです。そのパターンと盤面とが一致すれば、「つながっている」と認識します。

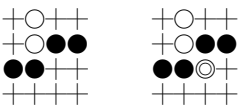
一間（座標(x1,y1)と座標(x1+2,y1)）がつながっている例

```
/* 座標(x1,y1)と座標(x1+2,y1)の連結の度合いについて */

if( board[y1][x1] == BLACK && board[y1][x1+2] == BLACK ){

  /* パターンに書かれている座標の状態と一致するか調べる */

  if( board[y1-1][x1  ] == SPACE &&
      board[y1-1][x1+1] == SPACE &&
      board[y1-1][x1+2] == SPACE &&
      board[y1  ][x1+1] == SPACE &&
      board[y1+1][x1  ] == SPACE &&
      board[y1+1][x1+1] == SPACE &&
      board[y1+1][x1+2] == SPACE ) {

    /* 一致した */
    return( つながっている );
  }
```

```
}
```

探索では、仮に相手の石を置いた盤面を作り、その置いた石を取れるかどうかでつながっているかどうかを判定します。

盤面　　相手の石を仮に置いた盤面

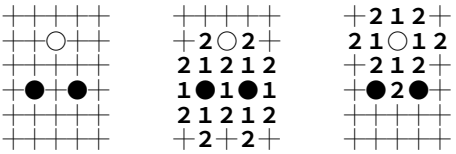


この盤面で、黒番で◎を取ることができるか探索する

```
/* 座標(x1,y1)と座標(x1+1,y1-1)の連結の度合いについて */

/* (1) 仮に石を置いてみる */
SetStone( x1+1, y1, WHITE );

/* (2) 黒番で探索 */
status = Search( x1+1, y1, 黒番 );

if( status == DEAD ){

  /* 白石が死んでいる＝白が切っても黒に取られる、ので「つながっている」 */
  return( つながっている );

}else{

  /* 白石は死なない＝切った白石が黒に取られない、ので「切れている」 */
  return( 切れている );

}
```

ポテンシャルでは、結線の座標に対する各石からの影響度を考えます。

盤面　　黒からの影響度　白からの影響度



一間の中間の座標は、黒からの影響度が白からの影響度よりも大きい
（黒石からの距離が白石からの距離よりも小さいから）ので、つながっていると認識する

結線の使いみち

結線の認識ができると、次のような利点があります。

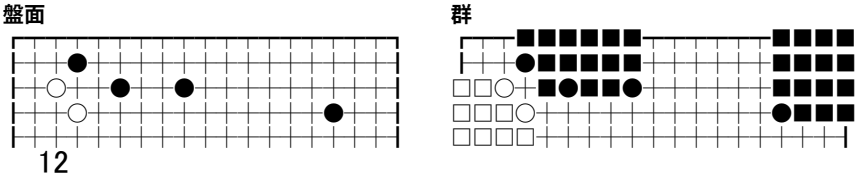
- 石と石をつなぐ手/切る手を生成することができる
- 離れた石を一つの塊とみなすことができるので、群を作るための前処理になる
- 地（領域）の境界を考えることができる

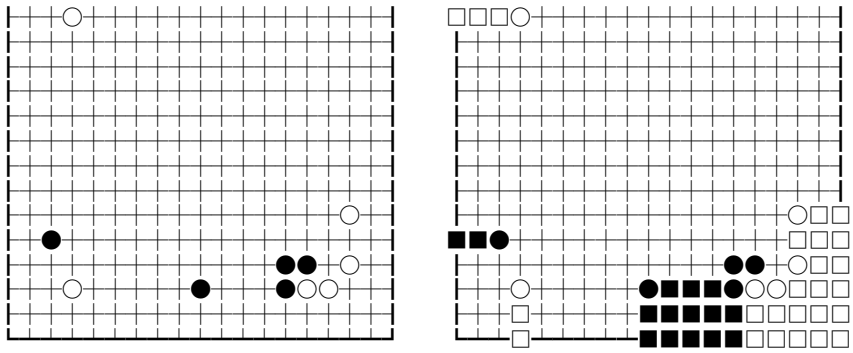
群

人間が囲碁を打っているときに、普通は各石ごとの生死は考えていません。縦横にはつながっていないが、非常に近くにある石をひとまとまりの単位として、生死を考えています。それを多くの囲碁プログラムの作者は「群」と呼んでいます（海外では英語でgroup（グループ）と呼ばれることが多い）。

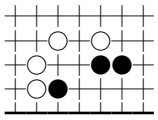
群の定義に、決定的なものはありません。囲碁プログラムの作者によって、いろいろな定義をしています。群の定義によく使われるのは、結線です。次の図のように、群は、石と結線によって囲まれた部分と、囲んでいる石と結線から構成されると定義とするのが、最も人間の考え方に近いでしょう。

盤面　　群





この定義は、石が少ないうちは人間の認識結果に近いのですが、石が混みあってくるといろいろと問題が出てきます。盤面の認識のうち、群の定義は非常に難しい問題なのです。この群の定義と、群の生死の認識がどのくらい正確かによって、プログラムの強さが決まると言ってもいいくらいです。



ケイマの関係にある黒石だが、
左の黒石は、白番では死んでいるので、つながっていないと認識される。

しかし、３つの黒石を一塊で考えているプログラムもある。

石と結線で囲まれた領域を認識するアルゴリズムは、碁盤プログラムで石が取られるかどうか（ダメが無いかどうか）を調べるアルゴリズムとよく似ています。碁盤プログラムでは、相手の石または盤外で全て囲まれていたらダメが無い（つまり取られる）と判断しましたが、領域の認識では、同色の結線または連で囲まれていたら、その色の群の領域とするのです。

```
/*----------*/
/* 群の認識 */
/*----------*/

/* (1) 結線の認識処理 */

/* (2) 全ての空点について、どちらの色の石と結線に囲まれているかを調べる */

  for( 全ての座標 ){

    if( 空点 ){

      /* (2.1) その座標がどちらの色の石に囲まれているか調べる */

      if( 黒に囲まれている ){

        /* (2.1.1) その座標は黒で囲まれていることを記録する */

      }else if( 白に囲まれている ){

        /* (2.1.2) その座標は白で囲まれていることを記録する */

      }else{

        /* (2.1.3) その座標は黒と白の両方に囲まれているので、両方の色に囲まれていることを記録 */

      }

    }
  }

/* (3) 群の識別番号を付与する処理 */

  for( 全ての座標 ){

    if( 石がある ){

      /* (3.1) その点に群の識別番号を付与する */

      /* (3.2) その点の四方について                                  */
      /*       同じ色の石ならば、同じ識別番号を付与する              */
      /*       同じ色の石で囲まれている領域ならば、同じ識別番号を付与する */

    }else if( 白または黒で囲まれている点 ){

      /* (3.1) その点に群の識別番号を付与する */

      /* (3.2) その点の四方について                                  */
      /*       同じ色の石ならば、同じ識別番号を付与する              */
      /*       同じ色の石で囲まれている領域ならば、同じ識別番号を付与する */
```

```
    }else{
      /* 両方の色の石に囲まれている点は、群にならないので、何もしない */
    }
  }
```

群の使いみち

群の認識ができると、次のような利点があります。

- 候補手生成
  - 地が小さいなら、地を増やす手を生成する
  - 近くに味方がいるなら、連結する手を生成する
- 評価
  - 地が計算できる
  - 群の形状がわかるので、強さを計算できる

群のデータ構造の例

それぞれの群について、どんな情報を認識するとよいのでしょうか？ここでは、多くの囲碁プログラムが認識している情報を挙げておきます。

- 群の位置
- 群の色
- 群の強さ(または生死)
- 群の眼数
- 群の領域(地)
- 群の包囲のされ具合
- 近くにいる敵の群
- 近くにいる味方の群

群の強さの調べ方

群の強さは、どのように調べると良いでしょうか。残念ながら群の強さの調べ方も、特に確立された方法はありません。多くのプログラムでは、以下の要素について調べ、それらの関係から総合的に判断しているようです。

- 群の眼数
- 群の領域(地)
- 群の包囲のされ具合
- 近くにいる敵の群
- 近くにいる味方の群
- 対局の進行度(序盤/中盤/終盤)
- 群の大きさ

例えば、地が無くても敵に包囲されていなければ「強い」と認識したり、眼は1.5眼しか無くても、近くに味方石があれば、(2眼作る手と味方につながる手が見合いだから)「強い」と認識したりします。

眼数

眼数の調べ方については、いくつかの論文で紹介されています。ここでは、そのうち二つを紹介します。

論文"Static Analysis of Life and Death in the game of Go"では、包囲する石数によって眼数を求めています。

包囲する石数と眼数の関係

包囲する石数	眼数	例
6以下	1.0	図1
7	1.5	図2
8(正方形)	1.0	図3
8(2回曲がり)	2.0	図4
8(その他)	1.5	図5
9(正方形含む)	1.5	図6
9(正方形含まず)	2.0	図7
10以上	2.0	図8

図１

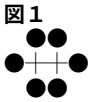


図２

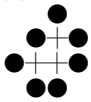


図３

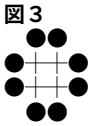


図４

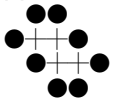


図５

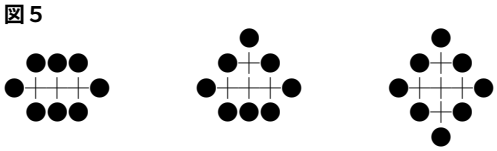


図６

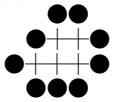


図７

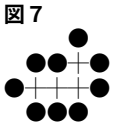
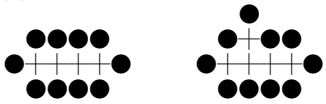


図８



論文"棋士システム「碁世代」"では、領域のそれぞれの点について、その点が同じ領域と幾つ接しているかを調べ (次数コードと呼んでいます)、その接し方から眼数を調べています。

領域のサイズと眼数の関係

領域のサイズ	次数コード	眼数	例
2以下	1,1	1.0	図1
3	2,1,1	1.5	図2
4	2,2,1,1	2.0	図3
4	3,1,1,1	1.5	図4
4	2,2,2,2	1.0	図5
5	3,2,1,1,1	2.0	図6
5	3,2,2,2,1	1.5	図7
5	4,1,1,1,1	1.5	図8
5	2,2,2,1,1	2.0	図9

※領域のサイズ7まで記述されているが、ここでは省略する。

図１

次数コード

急所

図2

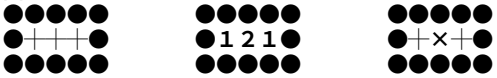


図3

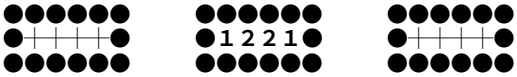


図4

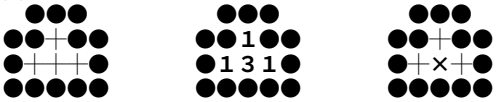


図5

図6

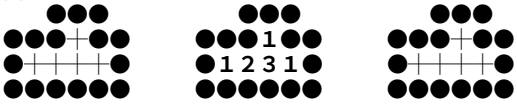


図7

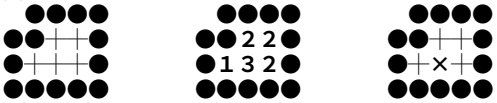


図8

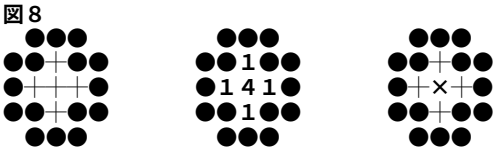
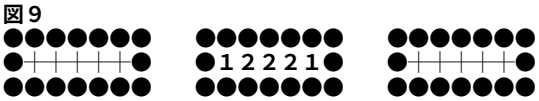


図9



ただし、どちらの論文にも、「例外があるので注意が必要」と書かれています。例えば、隅の曲がり四目のように領域が隅にある場合や、領域内に相手の石が存在する場合などです。

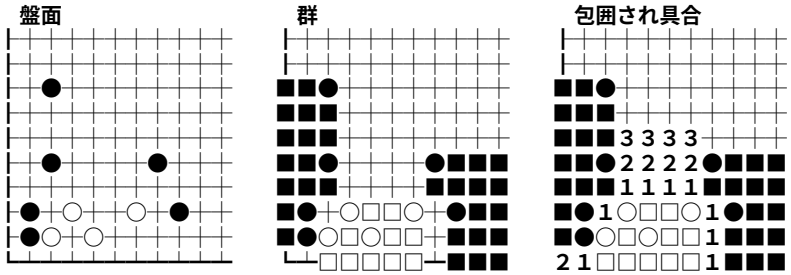
領域 (地)

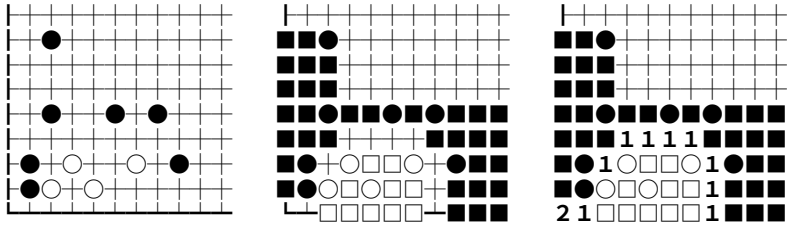
領域 (地) は、群のうちの空点の数になります。

包囲のされ具合

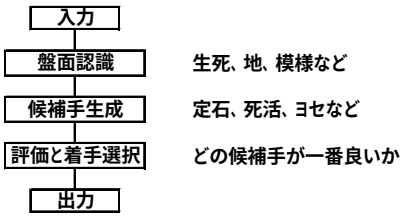
プログラムによって方法は異なりますが、一般的な方法は次のようになります。

- 群の一番外側から1路ずつ、外側に手を伸ばします。このとき、相手の群にぶつかったら、手を伸ばすのを止めます。
- 3手(ぐらい)まで手を伸ばします。
- 距離3(ぐらい)の点の数が多いほど、包囲度が小さいとみなします。





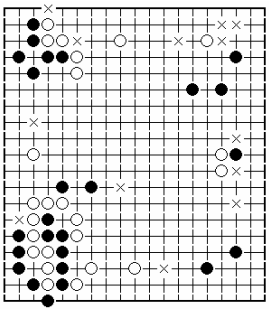
第4部　候補手生成



候補手生成の意味

- 着手可能な点は全て候補手として良い
- しかし、全ての候補手を評価するのは時間がかかり過ぎる
- 明らかに候補手として相応しくない手は、最初から生成しない方が良い
- そこで、良い手しか思い浮かばない（候補手としない）ようなプログラムを作る

黒は次にどこに打つ？



- 人間（上級者）は、盤面を見ただけで、打ちたい場所がすぐに思いつく
- 無意識のうちに多くの処理を行っている
- しかし、これをプログラムで実現するのは難しい

候補手生成

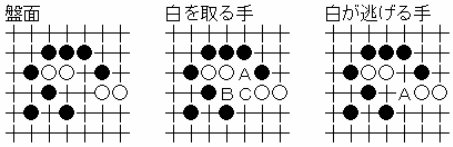
- 盤面認識の結果から、戦略をたてる
  - どこに地を作ろう
  - どの石から攻めよう
  - 流れにまかせよう
- 認識結果と戦略から、複数の候補手を生成する
  - 布石、定石、手筋
  - 死活（取る／逃げる）、攻め／守り、連結／切断
  - 地を囲う／侵略する
  - ハネ／ノビ／一間トビ／ケイマ、、、
  - 形の良い手

候補手の生成方法による分類

- 局所的な探索の結果から
  - 石を取る手や取られないように逃げる手、石と石がつながる手や切る手は、局所的な探索の結果から生成する
- パターン（データベース）から
  - 定石や手筋の手を生成するために、石の配置と候補手の位置を記述したデータベースを用意する
- その他の方法から
  - 人間は、布石の手や模様の手などは、石の強さや距離をもとに候補手の位置を決めている。このような手を求めるためのアルゴリズムを用意する。

候補手の生成方法による分類

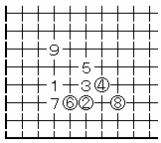
- 局所的な探索の結果から（盤面認識の結果から）
  - 石を取る
  - 石と石をつなぐ



候補手の生成方法による分類

- パターンから
  - データベース（プログラムにとっての定石事典）
  - 単純なパターン（一間トビ、ハネなど）

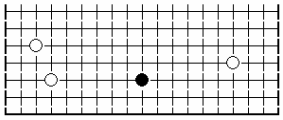
ツケノビ定石



候補手の生成方法による分類

- その他の方法から
  - 割打ち（相手の石と石の真ん中の位置を計算）
  - 消し（石の強さを見ながら最適な位置を計算）

割打ち（左右いずれかへのヒラキを見る）



Many Faces of Goの候補手

- 布石
- 辺での大きい動き
- 中央での戦い
- 優勢局面での安全な打ち方
- 劣勢局面での＜もがき＞
- パターン・マッチング
- 弱い群を助ける
- 危ない連を助ける
- 取れそうな連を捕獲する
- 弱い群を殺す
- キリとツナギ
- 接近戦
- コウ争い
- ダメを詰める

「Knowledge Representation in The Many Faces of Go」, 1993年より

碁世代の候補手

- 群の攻防（死活、攻め合い、包囲・脱出、連絡・分断）
- 定石
- 辺点
- ダメ点（ハネ、ノビ、オシなど）
- 族の分離・連絡
- 模様手
- 結線の連結・切断
- 打ち込み
- ヨセ

「棋士システム『碁世代』」,1993年より

HARUKAの候補手

- 優先手（直前石回りのツナギ、アタリ石の逃げ、二線オサエなど）
- 両アタリ位置、捕獲位置、同逃げ位置
- 攻め合い手
- 弱い石（重要度の高い石優先）の攻め手、守り手
- 接触、非接触点評価値の大きい位置
- コウダテ

「HARUKAのプログラム構造」,2003年より

GNU Goの候補手

- Attacking and defending moves
- Threats to Attack or Defend
- Multiple attack or defense moves
- Cutting and connecting moves
- Semeai winning moves
- Making or destroying eyes
- Antisuji moves
- Territorial moves
- Attacking and Defending Dragons
- Combination Attacks

「GNU Go 3.4 Documentation」, 2003年より

勝也の候補手

- パターン（布石、定石、手筋、好形）
- 連の死活
- 眼形

2003年版より

候補手の優先順位

- 候補手の中でも、優先的に打たせたい手がある
  - 定石の途中なら、定石の手を続けるなど
- 候補手がたくさん生成されると、全ての候補手について評価していたら時間がかかり過ぎる

↓

- 候補手生成時に、予め優先順位を考慮すると効率が良い
  - 「Aの候補手があれば、必ずAを打つ」
  - 「Bの候補手がなければCの候補手を生成する」

候補手のボーナス点

- 優先的に打たせたい手に対し、ボーナス点を与え、評価値を高くする
- 異なる候補手知識の間の関係ではなく、単純に候補手に「＋A点」を与える方式（「×A」のように係数を与えるやり方もある）

囲碁プログラミング講習会

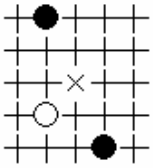
日程：2004年6月13日（日）
場所：ソフトピアジャパン・ドリームコア
講師：清愼一、佐々木宣介
（コンピュータ囲碁フォーラム）
主催：岐阜チャレンジ2004実行委員会

本日のスケジュール

- 第1部　パターンによる候補手
- 第2部　その他の候補手
- 第3部　評価と着手選択
- 第4部　囲碁プログラムの設計
- 第5部　その他（大会規約など）

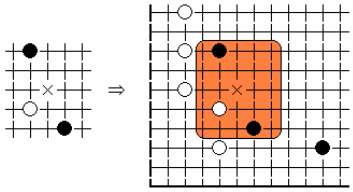
第1部　パターンによる候補手

- パターンとは、石の配置と候補手の位置を表したもの
- 定石、手筋、好形などの表現に使われる



パターンによる候補手

- 対局中に、常に盤面とパターンと比べて、同じ配置が現れたら、そのパターンに書かれている手の位置を候補手とする

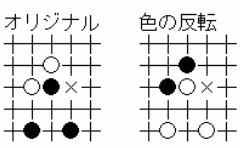


パターンの形と大きさ

- パターンの形と大きさは、設計者が自由に決めて良い
- パターンの用途や、プログラムの作りやすさなどを考慮して、決定する
  （例）局所的なパターンならば5×5
  定石ならば10×12
  布石ならば19×19
- 用途ごとに異なる形や大きさのパターンを作っている囲碁プログラムが多い

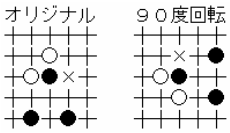
色の反転

- パターンの総数を減らすため、（普通は）色だけが異なるパターンは記述しない
- 盤面と比較するときに、色を反転したパターンを作り、盤面と比較する

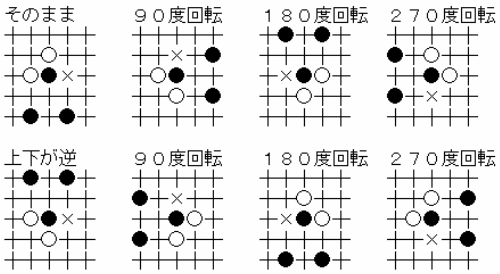


対称形の処理

- パターンの総数を減らすため、(普通は) 対称形のパターンは記述しない
- 盤面と比較するときに、対称形のパターンを作り、盤面と比較する



８種類の対称形



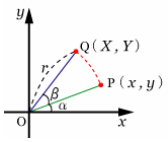
パターンとの比較アルゴリズム

```
for( すべての座標 ){
    /* (1) 盤面から(x,y)を左上とする５×５の形を切り出す */
    /* (2) 白番ならば白黒を反転する（黒番ならばそのまま） */
    for( i=0; i < データ数; i++ ){
        /* (3) データから８種類の対称形を作る */
        /* (4) データと切り出した５×５を比較する */
        /* (4.1) 一致したらデータに書かれている候補手位置を覚える */
    }
}
```

対称形の作り方

- 回転した形を作るには、行列演算を行う

$$\begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$



対称形の作り方

点 P(x,y) を β 回転させたものを点 Q(X,Y) とすると

$x = r\cos\alpha$　　$y = r\sin\alpha$

$X = r\cos(\alpha+\beta)$　　$Y = r\sin(\alpha+\beta)$

加法定理より

$X = r(\cos\alpha\cos\beta - \sin\alpha\sin\beta)$

$Y = r(\sin\alpha\cos\beta + \cos\alpha\sin\beta)$

x, y を代入して

$X = x\cos\beta - y\sin\beta$

$Y = y\cos\beta + x\sin\beta)$

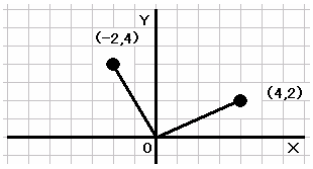
行列で表すと

$$\begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

９０度回転 (対称形)

$\cos 90度 = 0, \sin 90度 = 1$

$$\begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} \cos 90度 & -\sin 90度 \\ \sin 90度 & \cos 90度 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -y \\ x \end{pmatrix}$$



１８０度、２７０度回転

$\cos 180度 = -1, \sin 180度 = 0$

$$\begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} \cos 180度 & -\sin 180度 \\ \sin 180度 & \cos 180度 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -x \\ -y \end{pmatrix}$$

$\cos 270度 = 0, \sin 270度 = -1$

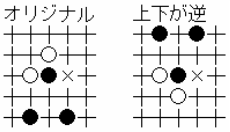
$$\begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} \cos 270度 & -\sin 270度 \\ \sin 270度 & \cos 270度 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} y \\ -x \end{pmatrix}$$

上下が逆

- 縦軸（Y座標）を反転させるだけ

$$\begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} x \\ -y \end{pmatrix}$$

オリジナル　上下が逆



上下が逆の形を９０度回転

$\cos 90度 = 0, \sin 90度 = 1$

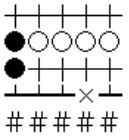
$$\begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} \cos 90度 & -\sin 90度 \\ \sin 90度 & \cos 90度 \end{pmatrix} \begin{pmatrix} x \\ -y \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ -y \end{pmatrix} = \begin{pmatrix} y \\ x \end{pmatrix}$$

８種類の対称形

	対称形のX座標	対称形のY座標
そのまま	X	Y
９０度回転	-Y	X
１８０度回転	-X	-Y
２７０度回転	Y	-X
上下が逆	X	-Y
上下が逆で９０度回転	Y	X
上下が逆で１８０度回転	-X	Y
上下が逆で２７０度回転	-Y	-X

パターンの記述

- 黒、白、空点、盤外
- 候補手の位置
- 手番



×：候補手

＃：盤外

パターンのデータ構造

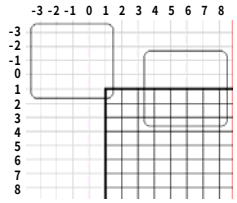
```
/* パターンのデータ構造の定義 */
struct go_data {
    int x;                  /* 候補手のＸ座標 */
    int y;                  /* 候補手のＹ座標 */
    int data[5][5];         /* 石の配置       */
};

/* パターン */
static struct pattern_data patternData[] =
{
  [ 3,  /* 候補手のＸ座標 */
    2,  /* 候補手のＹ座標 */
    [ [ SPACE, SPACE, SPACE, SPACE, SPACE ],   /*  */
      [ SPACE, SPACE, WHITE, SPACE, SPACE ],   /*  */
      [ SPACE, WHITE, BLACK, SPACE, SPACE ],   /*  */
      [ SPACE, SPACE, SPACE, SPACE, SPACE ],   /*  */
      [ SPACE, BLACK, SPACE, BLACK, SPACE ] ]  /*  */
  ]
};
```

盤面とパターンのマッチング

```
/* 二つのパターンを比較する */
static int
ComparePattern( struct pattern_data *p1, struct pattern_data *p2 )
{
  int x, y;

  for( y=0; y < PATTERN_SIZE; y++ ) {
    for( x=0; x < PATTERN_SIZE; x++ ) {
      if( p1->data[y][x] != p2->data[y][x] ) {
        /* 一致していなかったら -1 を返す */
        return( -1 );
      }
    }
  }

  /* 一致していたら 0 を返す */
  return( 0 );
}
```

盤面の切り出し

- 盤を1点でも含むように、盤の外側から、パターンの大きさ分だけ切り出す

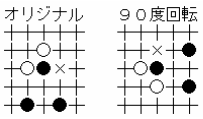


盤面の切り出し

```
/* 盤面切り出し関数を呼ぶ */
for( y=1-PATTERN_SIZE+1; y <= PURE_BOARD_SIZE; y++ ){
   for( x=1-PATTERN_SIZE+1; x <= PURE_BOARD_SIZE; x++ ){
      GetPatternFromBoard( board, &cutData, x, y );
   }
}

/* 盤面切り出し関数の中の処理 */

/* 指定された座標(x,y)から切り出す */
for ( y1=0,i=y; i < y+PATTERN_SIZE; i++,y1++ ) {
    for ( x1=0,j=x; j < x+PATTERN_SIZE; j++,x1++ ) {
        if( i < 1 || i > PURE_BOARD_SIZE || j < 1 || j > PURE_BOARD_SIZE ) {
            /* 盤外 */
            pattern->data[y1][x1] = OUT;
        }else{
            pattern->data[y1][x1] = board[i][j];
        }
    }
}
```

データ構造を考慮した回転

- 前に説明した座標の変換は、原点(0, 0)を中心とした回転だった
- プログラムでは、配列を使用してパターンを表現しているので、前の説明のままの変換式では使えない



例えば(3,2)は(2,1)に移動する

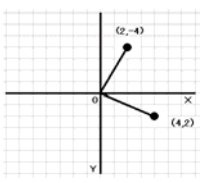
前に説明した式のままだと
(3,2)は(-2,3)になってしまう
(∵(X,Y)=(-y,x)だったから)

データ構造を考慮した回転

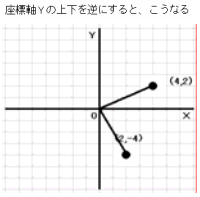
(1) 回転を行う
(2) 座標系の違いを考慮して修正する
　配列を使用しているので、Y軸の＋－が逆

データ構造を考慮した回転

- 座標軸の向きが異なっているので、９０度回転したつもりでも、実は２７０度回転している

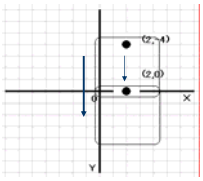


座標軸Yの上下を逆にすると、こうなる



データ構造を考慮した回転

- 回転した後の座標は、Yがマイナスになっているので、(パターンのサイズー1) だけ足して、平行移動させる



パターンを座標０～４
（４＝パターンのサイズ－１）
に収めるために、平行移動させる

データ構造を考慮した回転

- 整理すると、９０度回転した対称形を作るには、「２７０度回転させた後に、Y座標に (パターンのサイズー1) を足す」

$$\cos 270度 = 0, \sin 270度 = -1$$

$$\begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} 0 \\ Size-1 \end{pmatrix} + \begin{pmatrix} \cos 270度 & -\sin 270度 \\ \sin 270度 & \cos 270度 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} y \\ Size-1-x \end{pmatrix}$$

データ構造を考慮した回転

```
/* パターンの各座標の回転 */
for( y=0; y < PATTERN_SIZE; y++ ) {
    for( x=0; x < PATTERN_SIZE; x++ ) {
        to->data[y][x] = from->data[x][PATTERN_SIZE-y-1];
    }
}
/* 候補手の位置の回転 */
to->x = from->y;
to->y = PATTERN_SIZE - from->x - 1;
```

データ構造を考慮した回転

	対称形のX座標	対称形のY座標
そのまま	X	Y
９０度回転	Y	Size-1-X
１８０度回転	Size-1-X	Size-1-Y
２７０度回転	Size-1-Y	X
上下が逆	X	Size-1-Y
上下が逆で９０度回転	Size-1-Y	Size-1-X
上下が逆で１８０度回転	Size-1-X	Y
上下が逆で２７０度回転	Y	X

高速化

- パターン作成の高速化
  - 一つのパターンから、色の反転、対称形を、自分の手番ごとに作るのは時間がかかる
  - 対局開始時に、パターンから、色の反転、対称形を作っておくと良い
- パターンマッチングの高速化
  - パターンを分類できれば、同じカテゴリーの中のパターンとだけ比較すれば良い
  - ハッシュ関数を使ってカテゴリーに分類する

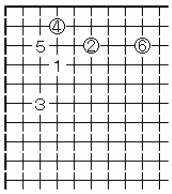
パターンの入力の問題点

- パターンの選定
  - 具体的にどんなパターンを作るかを考える
- パターンの入力
  - 手で一つ一つパターンを記述するか、棋譜から自動的に集めるか
- 重複のチェック
  - 同じパターンを既に入力していないか
- 網羅性のチェック
  - 記述したいパターンを全て入力したか

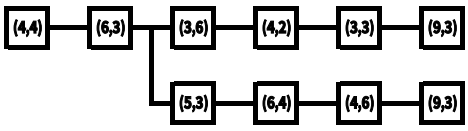
定石パターン

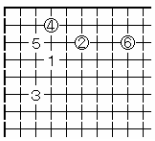
- 一手一手を異なるパターンとして記述する
  - 手順前後や、途中で他のところに打っても、定石を打てる
  - 検索に時間がかかる
- 木構造で表す
  - 手順前後に対応することが難しい
  - 木構造をたどるだけなので、検索時間は短い

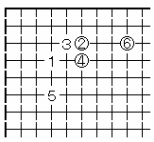
定石の木構造



定石の木構造

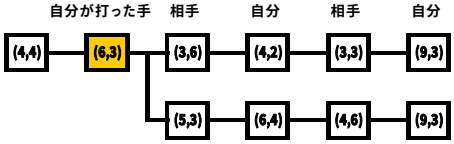






定石の検索

- 現在の盤面が、定石のどの手まで打った盤面なのかを覚えておく
- 相手が、その手の次の手を打ったら、自分はさらに一手先に書かれている手を打つ



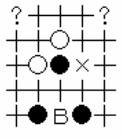
単純なパターン

- 小さいパターンならば、構造体を使ってパターンを記述しなくても、if文のたくさん書くことでもパターンを記述できる

```
if( board[y-1][x-1] == SPACE &&
    board[y-1][x  ] == WHITE &&
    board[y-1][x+1] == SPACE &&
    board[y  ][x-1] == BLACK &&    /* ┼○┼ */
    board[y  ][x  ] == SPACE &&    /* ●×● */
    board[y  ][x+1] == BLACK &&    /* ┼┼┼ */
    board[y+1][x-1] == SPACE &&
    board[y+1][x  ] == SPACE &&
    board[y+1][x+1] == SPACE ){

  /* (x,y)が候補手 */

}
```

パターンの記述（あると便利）

- Don‘tCare（何でも良い）
- 黒または空点、白または空点



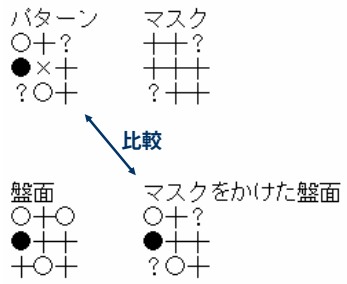
？：Don’tCare(何でも良い)

Ｂ：黒または空点

Don’tCareの実現方法

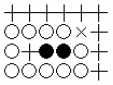
- 本当のデータ（石の配置のパターン）以外に、マスクデータ（Don‘tCareの位置だけを書いたパターン）を用意する
- マスクとは、本当のデータを覆い隠すためのデータのこと
- 盤面から切り出したデータにマスクをかける（マスクに置き換えることによって隠す）
- マスクをかけたデータと、パターン（これもマスクをかけておく）とを比較する

Don’tCareの実現方法



パターンの記述（高度な記述）

- 石の生死（生き／死に／中立）
- 石のダメ数



黒２子が死んでいて、白が
全て生きているならば、×は候補手にならない

白のダメが０ならば×で死に

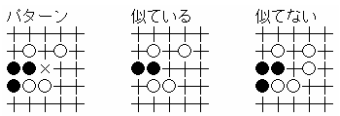
白のダメが１以上ならば×は候補手にならない

パターンの候補手にも情報

- 候補手の位置だけじゃなく、評価値（絶対値、優先順位、重み付け）に関する情報も記述する
- 候補手を評価する際に、その情報も考慮することができる

類似パターン

- 完全に一致するパターンではなく、異なる部分が少しだけならば、「類似しているから候補にする」という処理に使う
- パターンと盤面を比較する際に、異なる部分の場所や数から、類似しているかどうかを判定する



パターンの候補手生成以外の用途

- 石と石の連結の度合いを調べる
  - 石の配置と、連結度合いを記述したパターン
- ２眼かどうか調べる
  - 石の配置と、眼数を記述したパターン

第1部 パターンによる候補手

パターンから候補手を生成するプログラム

このプログラムは、"figure"という名前のファイルを読んで、次の一手を返すプログラムです。 "figure"というファイルには、次の一手を求めたい盤面を描いておきます。

```
"figure"の例
++++++++++++++++++
++++++++++++++++++
+O**++++*++++++++++
++O+*+++++++++*+*+++
+O++++++++++++++++++
+++O++++++++++O+O++
++++++++++++++++++
+++++++++++++++O*++
++++++++++++++++*+++
++O++++++++++++++++
++++++++++++++++++
++O+*++++++++++++++
++++++++++++++++++
++O++++++++++++++++
++++O++++++++++++*++
++++++*++++++++++++
++++O+++++*++++*+++
++++++++++++++++++
++++++++++++++++++
```

```
/*---------------------------------------*/
/* パターンから候補手を生成するプログラム */
/*---------------------------------------*/

#include <stdio.h>
#include <string.h>

#define TRUE 1
#define FALSE 0

#define SPACE 0            /* 空点(石が置かれていない) */
#define BLACK 1            /* 黒   */
#define WHITE 2            /* 白   */
#define OUT   3            /* 盤外 */

#define BOARD_SIZE 21                       /* 碁盤の大きさ */
#define PURE_BOARD_SIZE (BOARD_SIZE-2)      /* 本当の碁盤の大きさ */


#define PATTERN_SIZE 5  /* パターンのサイズ */

struct pattern_data {
  int x;                   /* 候補手のＸ座標 */
  int y;                   /* 候補手のＹ座標 */
  int data[PATTERN_SIZE][PATTERN_SIZE];  /* 石の配置       */
};

/* 相手の色 */
#define reverseColor( color ) ( color == BLACK ? WHITE : ( color == WHITE ? BLACK : color ) )

/* 回転方向 */
#define ROTATE_000  0          /* そのまま             */
#define ROTATE_090  1          /* ９０度回転           */
#define ROTATE_180  2          /* １８０度回転         */
#define ROTATE_270  3          /* ２７０度回転         */
#define ROTATE_M000 4          /* 上下が逆             */
#define ROTATE_M090 5          /* 上下が逆 ９０度回転  */
#define ROTATE_M180 6          /* 上下が逆 １８０度回転 */
#define ROTATE_M270 7          /* 上下が逆 ２７０度回転 */
```

```
/*---------------------------------------------------------------------*/
/* 関数                                                                */
/*---------------------------------------------------------------------*/

/* メイン関数 */
int main( int argc, char *argv[] );

/* 盤面図入力 */
static void InputFigure( int board[][BOARD_SIZE] );

/* 碁盤の初期化 */
static void InitializeBoard( int board[][BOARD_SIZE] );

/* 碁盤を表示する */
static void DisplayBoard( int board[][BOARD_SIZE] );

/* データベースから候補手の生成 */
static void MakePatternCandidates( int turn, int board[][BOARD_SIZE], int evalBoard[][BOARD_SIZE] );

/* ボードからパターンを切り出す */
static void GetPatternFromBoard( int board[][BOARD_SIZE], struct pattern_data *pattern, int x, int y );

/* パターンの白黒反転 */
static void ReverseColorPattern(struct pattern_data *pattern);

/* パターンの回転 */
static void RotatePattern(struct pattern_data *from, struct pattern_data *to, int type);

/* 候補手の認識 */
static int  ClassificationPatternCandidates( int board[][BOARD_SIZE], int x, int y,
                                             int color, int number );

/* 候補手の表示 */
static void DisplayPatternCandidates( int evalBoard[][BOARD_SIZE] );

/*---------------------------------------------------------------------*/
/* 変数                                                                */
/*---------------------------------------------------------------------*/

/* 碁盤の表示のための文字 */
static char stone[] = { '+', '*', 'O', '?' };

/* パターン */
static struct pattern_data patternData[] =
{
{ 3, 2,                                     /* 候補手の座標 */
 {{ SPACE, SPACE, SPACE, SPACE, SPACE },    /* ┼┼┼┼┼ */
  { SPACE, SPACE, WHITE, SPACE, SPACE },    /* ┼┼○┼┼ */
  { SPACE, WHITE, BLACK, SPACE, SPACE },    /* ┼○●×┼ */
  { SPACE, SPACE, SPACE, SPACE, SPACE },    /* ┼┼┼┼┼ */
  { SPACE, BLACK, SPACE, BLACK, SPACE }},}, /* ┼●┼●┼ */
{ 2, 2,                                     /* 候補手の座標 */
 {{ SPACE, BLACK, SPACE, SPACE, SPACE },    /* ┼●┼┼┼ */
  { SPACE, SPACE, SPACE, SPACE, SPACE },    /* ┼┼┼┼┼ */
  { SPACE, SPACE, SPACE, SPACE, SPACE },    /* ┼┼×┼┼ */
  { SPACE, WHITE, SPACE, SPACE, SPACE },    /* ┼○┼┼┼ */
  { SPACE, SPACE, SPACE, BLACK, SPACE }},}  /* ┼┼┼●┼ */
};

/* パターンの数 */
int patternNumber = sizeof(patternData)/sizeof(struct pattern_data);

/*---------------------------------------------------------------------*/
/* メイン関数                                                          */
/*---------------------------------------------------------------------*/
int
main( int argc, char *argv[] )
{
  int board[BOARD_SIZE][BOARD_SIZE];      /* 碁盤 */
  int evalBoard[BOARD_SIZE][BOARD_SIZE];  /* 評価値格納碁盤 */

  printf( "Sample Program Start¥n" );

  /* 盤面図の入力 */
  InputFigure( board );

  /* 碁盤の表示 */
  DisplayBoard( board );

  /* 黒番の候補手の生成 */
  MakePatternCandidates( BLACK, board, evalBoard );

  /* 候補手の表示 */
```

```
  DisplayPatternCandidates( evalBoard );

  /* 白番の候補手の生成 */
  MakePatternCandidates( WHITE, board, evalBoard );

  /* 候補手の表示 */
  DisplayPatternCandidates( evalBoard );

  printf( "Sample Program End¥n" );
}

/*--------------------------------------------------------------------*/
/* 盤面図入力                                                         */
/* ファイル名"figure"というファイルに盤面図が書かれている             */
/*--------------------------------------------------------------------*/
static void
InputFigure( int board[][BOARD_SIZE] )
{
  FILE *fp;
  char buf[256];
  int  x, y, i;

  /* ファイルを開く */
  if( (fp = fopen( "figure", "r" )) == NULL ){
    fprintf( stderr, "ERROR: file(figure) open fail¥n" );
    exit(1);
  }

  /* 碁盤の初期化 */
  InitializeBoard( board );

  /* 各座標を読む */
  for( y=1; y < (BOARD_SIZE-1); y++ ){
    buf[0] = '¥0';
    fgets( buf, 256, fp );
    for( x=1, i=0; x < (BOARD_SIZE-1); x++ ){
      if( buf[i] == 'O' ){
        board[y][x] = WHITE;
      }else if( buf[i] == '*' ){
        board[y][x] = BLACK;
      }else if( buf[i] == '+' ){
        board[y][x] = SPACE;
      }else{
        board[y][x] = SPACE;
      }
      i++;
    }
  }

  /* ファイルを閉じる */
  fclose( fp );
}

/*--------------------------------------------------------------------*/
/* 碁盤の初期化                                                       */
/*--------------------------------------------------------------------*/
static void
InitializeBoard( int board[][BOARD_SIZE] )
{
  int x, y;

  for( y=1; y < (BOARD_SIZE-1); y++ ){
    for( x=1; x < (BOARD_SIZE-1); x++ ){
      board[y][x] = SPACE;
    }
  }

  for( y=0; y < BOARD_SIZE; y++ ){
    board[y][0]= OUT;
    board[y][BOARD_SIZE-1] = OUT;
    board[0][y] = OUT;
    board[BOARD_SIZE-1][y] = OUT;
  }
}

/*--------------------------------------------------------------------*/
/* 碁盤を表示する                                                     */
/*--------------------------------------------------------------------*/
static void
DisplayBoard( int board[][BOARD_SIZE] )
{
  int x, y;

  printf( "    [ 1 2 3 4 5 6 7 8 9101112131415161718 19]¥n" );
  for( y=1; y < (BOARD_SIZE-1); y++ ){
```

```
    printf( "[%2d] ", y );
    for( x=1; x < (BOARD_SIZE-1); x++ ){
      printf( " %c", stone[board[y][x]] );
    }
    printf( "¥n" );
  }
}

/*------------------------------------------------------------------*/
/* データベースから候補手の生成                                     */
/*------------------------------------------------------------------*/
static void
MakePatternCandidates( int turn, int board[][BOARD_SIZE], int evalBoard[][BOARD_SIZE] )
{
  int x, y;
  int n, r;
  struct pattern_data cutData;
  struct pattern_data rotateData;

  /* 評価用のボードの初期化 */
  for( y=0; y < BOARD_SIZE; y++ ){
    for( x=0; x < BOARD_SIZE; x++ ){
      evalBoard[y][x] = -1;
    }
  }

  for( y=1-PATTERN_SIZE+1; y <= PURE_BOARD_SIZE; y++ ){
    for( x=1-PATTERN_SIZE+1; x <= PURE_BOARD_SIZE; x++ ){
      /* 盤面の切りだし */
      GetPatternFromBoard( board, &cutData, x, y );

      if( turn == WHITE ) {
        /* 白番なら白黒を反転 */
          ReverseColorPattern( &cutData );
      }

      /* パターンと比較 */
      for ( n = 0; n < patternNumber; n++) {

        /* そのまま比較する */
          RotatePattern( patternData + n, &rotateData, ROTATE_000);
          if( ComparePattern( &rotateData, &cutData ) == 0 ) {
            printf("(%2d, %2d) pattern data found (%d)¥n", x+rotateData.x, y+rotateData.y, n);
            evalBoard[y+rotateData.y][x+rotateData.x] = n;
          }

        /* パターンを 90度回転して比較する */
          RotatePattern( patternData + n, &rotateData, ROTATE_090);
          if( ComparePattern( &rotateData, &cutData ) == 0 ) {
            printf("(%2d, %2d) pattern data found (%d)¥n", x+rotateData.x, y+rotateData.y, n);
            evalBoard[y+rotateData.y][x+rotateData.x] = n;
          }

        /* パターンを 180度回転して比較する */
          RotatePattern( patternData + n, &rotateData, ROTATE_180);
          if( ComparePattern( &rotateData, &cutData ) == 0 ) {
            printf("(%2d, %2d) pattern data found (%d)¥n", x+rotateData.x, y+rotateData.y, n);
            evalBoard[y+rotateData.y][x+rotateData.x] = n;
          }

        /* パターンを 270度回転して比較する */
          RotatePattern( patternData + n, &rotateData, ROTATE_270);
          if( ComparePattern( &rotateData, &cutData ) == 0 ) {
            printf("(%2d, %2d) pattern data found (%d)¥n", x+rotateData.x, y+rotateData.y, n);
            evalBoard[y+rotateData.y][x+rotateData.x] = n;
          }

        /* パターンを 上下が逆にして比較する */
          RotatePattern( patternData + n, &rotateData, ROTATE_M000);
          if( ComparePattern( &rotateData, &cutData ) == 0 ) {
            printf("(%2d, %2d) pattern data found (%d)¥n", x+rotateData.x, y+rotateData.y, n);
            evalBoard[y+rotateData.y][x+rotateData.x] = n;
          }

        /* パターンを 上下が逆にして 90度回転して比較する */
          RotatePattern( patternData + n, &rotateData, ROTATE_M090);
          if( ComparePattern( &rotateData, &cutData ) == 0 ) {
            printf("(%2d, %2d) pattern data found (%d)¥n", x+rotateData.x, y+rotateData.y, n);
            evalBoard[y+rotateData.y][x+rotateData.x] = n;
          }

        /* パターンを 上下が逆にして 180度回転して比較する */
          RotatePattern( patternData + n, &rotateData, ROTATE_M180);
          if( ComparePattern( &rotateData, &cutData ) == 0 ) {
            printf("(%2d, %2d) pattern data found (%d)¥n", x+rotateData.x, y+rotateData.y, n);
```

```
            evalBoard[y+rotateData.y][x+rotateData.x] = n;
          }

        /* パターンを 上下が逆にして 270度回転して比較する */
          RotatePattern( patternData + n, &rotateData, ROTATE_M270);
          if( ComparePattern( &rotateData, &cutData ) == 0 ) {
            printf("(%2d, %2d) pattern data found (%d)¥n", x+rotateData.x, y+rotateData.y, n);
            evalBoard[y+rotateData.y][x+rotateData.x] = n;
          }
      }
    }
  }
}

/*------------------------------------------------------------------*/
/* Board から patternを切り出す                                     */
/*------------------------------------------------------------------*/
static void
GetPatternFromBoard( int board[][BOARD_SIZE], struct pattern_data *pattern, int x, int y )
{
  int i, j;
  int x1,y1;

  /* 値をかえす変数をクリア */
  memset( pattern, 0, sizeof(struct pattern_data) );

  /* 指定された座標から PATTERN_SIZEだけ 切り出す */
  for ( y1=0,i=y; i < y+PATTERN_SIZE; i++,y1++ ) {
    for ( x1=0,j=x; j < x+PATTERN_SIZE; j++,x1++ ) {
      if( i < 1 || i > PURE_BOARD_SIZE || j < 1 || j > PURE_BOARD_SIZE ) {
        /* 盤外 */
          pattern->data[y1][x1] = OUT;
      } else {
          pattern->data[y1][x1] = board[i][j];
      }
    }
  }
}

/*------------------------------------------------------------------*/
/* pattern を白黒反転させる                                         */
/*------------------------------------------------------------------*/
static void
ReverseColorPattern( struct pattern_data *pattern )
{
  int x, y;

  for( y=0; y < PATTERN_SIZE; y++ ) {
    for( x=0; x < PATTERN_SIZE; x++ ) {
      /* 白黒を反転させる */
      pattern->data[y][x] = reverseColor( pattern->data[y][x] );
    }
  }
}

/*------------------------------------------------------------------*/
/* pattern を回転させる                                             */
/*------------------------------------------------------------------*/
static void
RotatePattern( struct pattern_data *from, struct pattern_data *to, int type )
{
  int x, y;
  memset( to, 0, sizeof(struct pattern_data) );

  switch(type) {
  case ROTATE_000:
    /* そのまま                    */
    /* ┼┼┼┼┼      ┼┼┼┼┼ */
    /* ┼┼○┼┼      ┼┼○┼┼ */
    /* ┼○●×┼ ===> ┼○●×┼ */
    /* ┼┼┼┼┼      ┼┼┼┼┼ */
    /* ┼●┼●┼      ┼●┼●┼ */
    *to = *from;
    break;

  case ROTATE_090:
    /* パターンを 90度回転する   */
    /* ┼┼┼┼┼      ┼┼┼┼┼ */
    /* ┼┼○┼┼      ┼┼×┼● */
    /* ┼○●×┼ ===> ┼○●┼┼ */
    /* ┼┼┼┼┼      ┼┼○┼● */
    /* ┼●┼●┼      ┼┼┼┼┼ */

    for( y=0; y < PATTERN_SIZE; y++ ) {
      for( x=0; x < PATTERN_SIZE; x++ ) {
```

```
            to->data[y][x] = from->data[x][PATTERN_SIZE-y-1];
      }
    }
    to->x = from->y;
    to->y = PATTERN_SIZE - from->x - 1;
    break;

  case ROTATE_180:
    /* パターンを 180度回転する   */
    /* ┼┼┼┼┼      ┼●┼●┼ */
    /* ┼┼○┼┼      ┼┼┼┼┼ */
    /* ┼○●×┼ ===> ┼×●○┼ */
    /* ┼┼┼┼┼      ┼┼○┼┼ */
    /* ┼●┼●┼      ┼┼┼┼┼ */

    for( y=0; y < PATTERN_SIZE; y++ ) {
      for( x=0; x < PATTERN_SIZE; x++ ) {
            to->data[y][x] = from->data[PATTERN_SIZE-y-1][PATTERN_SIZE-x-1];
      }
    }
    to->x = PATTERN_SIZE - from->x - 1;
    to->y = PATTERN_SIZE - from->y - 1;
    break;

  case ROTATE_270:
    /* パターンを 270度回転する   */
    /* ┼┼┼┼┼      ┼┼┼┼┼ */
    /* ┼┼○┼┼      ●┼○┼┼ */
    /* ┼○●×┼ ===> ┼┼●○┼ */
    /* ┼┼┼┼┼      ●┼×┼┼ */
    /* ┼●┼●┼      ┼┼┼┼┼ */

    for( y=0; y < PATTERN_SIZE; y++ ) {
      for( x=0; x < PATTERN_SIZE; x++ ) {
            to->data[y][x] = from->data[PATTERN_SIZE-x-1][y];
      }
    }
    to->x = PATTERN_SIZE - from->y - 1;
    to->y = from->x;
    break;

  case ROTATE_M000:
    /* パターンの 上下を逆にする  */
    /* ┼┼┼┼┼      ┼●┼●┼ */
    /* ┼┼○┼┼      ┼┼┼┼┼ */
    /* ┼○●×┼ ===> ┼○●×┼ */
    /* ┼┼┼┼┼      ┼┼○┼┼ */
    /* ┼●┼●┼      ┼┼┼┼┼ */

    for( y=0; y < PATTERN_SIZE; y++ ) {
      for( x=0; x < PATTERN_SIZE; x++ ) {
            to->data[y][x] = from->data[PATTERN_SIZE-y-1][x];
      }
    }
    to->x = from->x;
    to->y = PATTERN_SIZE - from->y - 1;
    break;

  case ROTATE_M090:
    /* パターンの 上下を逆にして 90度回転する */
    /* ┼┼┼┼┼      ┼┼┼┼┼ */
    /* ┼┼○┼┼      ●┼×┼┼ */
    /* ┼○●×┼ ===> ┼┼●○┼ */
    /* ┼┼┼┼┼      ●┼○┼┼ */
    /* ┼●┼●┼      ┼┼┼┼┼ */

    for( y=0; y < PATTERN_SIZE; y++ ) {
      for( x=0; x < PATTERN_SIZE; x++ ) {
            to->data[y][x] = from->data[PATTERN_SIZE-x-1][PATTERN_SIZE-y-1];
      }
    }
    to->x = PATTERN_SIZE - from->y - 1;
    to->y = PATTERN_SIZE - from->x - 1;
    break;

  case ROTATE_M180:
    /* パターンの 上下を逆にして 180度回転する */
    /* ┼┼┼┼┼      ┼┼┼┼┼ */
    /* ┼┼○┼┼      ┼┼○┼┼ */
    /* ┼○●×┼ ===> ┼×●○┼ */
    /* ┼┼┼┼┼      ┼┼┼┼┼ */
    /* ┼●┼●┼      ┼●┼●┼ */

    for( y=0; y < PATTERN_SIZE; y++ ) {
      for( x=0; x < PATTERN_SIZE; x++ ) {
```

```
            to->data[y][x] = from->data[y][PATTERN_SIZE-x-1];
      }
    }
    to->x = PATTERN_SIZE - from->x - 1;
    to->y = from->y;
    break;

  case ROTATE_M270:
    /* パターンの 上下を逆にして 270度回転する */
    /* ┼┼┼┼┼      ┼┼┼┼┼ */
    /* ┼┼○┼┼      ┼┼○┼● */
    /* ┼○●×┼ ===> ┼○●┼┼ */
    /* ┼┼┼┼┼      ┼┼×┼● */
    /* ┼●┼●┼      ┼┼┼┼┼ */

    for( y=0; y < PATTERN_SIZE; y++ ) {
      for( x=0; x < PATTERN_SIZE; x++ ) {
            to->data[y][x] = from->data[x][y];
      }
    }
    to->x = from->y;
    to->y = from->x;
    break;
  }
}

/*------------------------------------------------------------------*/
/* pattern を比較する                                               */
/*------------------------------------------------------------------*/
static int
ComparePattern( struct pattern_data *p1, struct pattern_data *p2 )
{
  int x, y;

  for( y=0; y < PATTERN_SIZE; y++ ) {
    for( x=0; x < PATTERN_SIZE; x++ ) {
      if( p1->data[y][x] != p2->data[y][x] ) {
        /* 一致していなかったら -1 を返す */
            return( -1 );
      }
    }
  }

  /* 一致していたら 0 を返す */
  return( 0 );
}

/*------------------------------------------------------------------*/
/* 候補手の表示                                                     */
/*------------------------------------------------------------------*/
static void
DisplayPatternCandidates( int evalBoard[][BOARD_SIZE] )
{
  int x, y;

  printf( "### Pattern ID ###¥n" );
  for( y=1; y < (BOARD_SIZE-1); y++ ){
    for( x=1; x < (BOARD_SIZE-1); x++ ){
      if( evalBoard[y][x] < 0 ){
            printf( "  +" );
      }else{
            printf( "%3d", evalBoard[y][x] );
      }
    }
    printf( "¥n" );
  }
}

/*---------------------- < end of program > ------------------------*/
```

第2部　その他の候補手

- 局所的な探索の結果から
  - 石を取る手や取られないように逃げる手、石と石がつながる手や切る手は、局所的な探索の結果から生成する
- 計算から
  - 人間は、布石の手や模様の手などは、石の強さや距離をもとに候補手の位置を決めている。このような手を求めるためのアルゴリズムを用意する

石を取る手／逃げる手

- 盤面認識における「連のデータ構造」を認識する際に、連の生死を求めている
- 連が「中立」（攻め側が先手ならば死に、守り側が先手ならば生き）のとき、「石を取る手/逃げる手」が生成される
- 候補手の座標は、連の属性として記憶しておけば、それを取り出すだけで済む

石と石をつなぐ手／切る手

- 盤面認識における「結線のデータ構造」を認識する際に、結線の接続度合いを求めている
- 結線の接続度合いが「中立」（結線の色の側が先手ならば連結、相手側が先手ならば切断）のとき「石と石をつなぐ手／切る手」が生成される
- 候補手の座標は、結線の属性として記憶しておけば、それを取り出すだけで済む

群を殺す手／生きる手

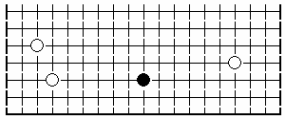
- 盤面認識における「群のデータ構造」を認識する際に、群の生死を求めている
- 群が「中立」（完全には生きていない、完全には死んでいない）のとき、群を殺す手／生きる手が生成される
  - 味方の群とつながる手
  - 眼を二つに分割する手（中手）
  - 相手の群を包囲し逃げ道をなくす手
- 候補手の座標は、群の属性として記憶しておけば、それを取り出すだけで済む

割り打ち

- 相手の石と石の真ん中に打つ手

X＝（X1＋X2）／2，Y＝17

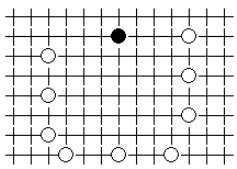
割打ち（左右いずれかへのヒラキを見る）



消し

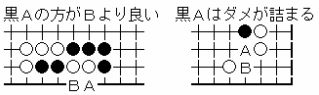
- 相手の模様の先端と先端を結んだ線の上の位置を計算

X＝（X1＋X2）／2，Y＝（Y1＋Y2）／2



負の候補手

- ここには絶対に打たせたくない手を「負の候補手」という（通常の候補手は「正の候補手」）
- 正の候補手と負の候補手が同じ座標だったら、その手は抑制（打たない）
  - 水平線効果（地平線効果）のような無駄な手
  - 意味の無いキカシ
  - 効率の悪い手



第3部　評価と着手選択

- 候補手を生成したら、各候補手について評価値を求める
- 評価値の最も高い手を「次の一手」として選択する
- もしも、最も高い評価値が、現状より良くならないならば、パスをする（このときは終局に違いない）

評価の方法

- 方法1）候補手を仮に打ってみたときの盤面の状態を認識し、どちらが何目有利か調べる
- 方法2）幾つかの評価基準を決め、それらの評価基準に照らし合わせて評価をする。それぞれの評価値に適当な重み付けをし、それらの関数として総合的な評価値を算出する。

総合評価値 ＝ A×a ＋ B×b ＋ C×c ＋…..

A，B，C：重み付け

a，b，c：それぞれの基準での評価値

評価基準の例

- 地（目数）
- 模様（地ではないが、将来地になりそうな点）
- 石の強さ（石の生きやすさ）
- その手は定石や手筋の途中か？
- その手は捨て石か？
- その手はコウ材か？
- その手は先手か？

重み付けの例

- 先手ならば評価値を2倍（後手ならば1倍）
- 絶対に打たせたい手（ノゾキにつなぐ手や、定石の途中など）は10000倍
  （係数でなく10000目の固定した値でも良い）

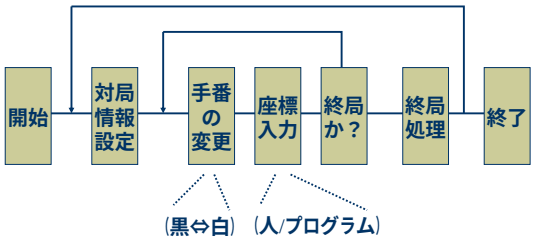
課題

- 先手か後手かの判断
- 小さくても先手の手を打つか、大きい後手の手を打つか
- コウの評価値の計算方法
- 優勢のときに安全策、劣勢のときに逆転可能な大胆な手
- 進行度（序盤／中盤／終盤）による調整
- 異なる評価基準の間の重み付けのバランス
- 捨石のように、何手か先に利益のある手

第4部　囲碁プログラムの設計

- メイン部
  - プログラムの開始と終了
  - インターフェース部と思考部の呼び出し
- インターフェース部
  - 画面
  - 通信
  - 棋譜ファイル
- 思考部
  - 盤面認識
  - 候補手生成
  - 候補手の評価
  - 着手選択
- その他（データベースファイル）

処理フロー



対局情報設定

- 黒番（白番）は、コンピュータ／人間／通信
- 盤の大きさは、十九路／十三路／九路
- ハンデは、互先／定先／二子／三子／…
- ルールは、日本ルール／中国ルール／…
- 持ち時間は、何分
- 対局日時、対局者名、対局場所、コミ、など

画面表示

- 碁盤と碁石
- 対局者名
- アゲハマ
- 消費時間
- どちらの番か
- 今は何手目か
- 最後に打たれた石のはどこか

着手位置の入力

- カーソルを使って移動し、Enterキーで決定
- マウスで移動し、クリックで決定
- 座標を数値で入力

終局処理

- 勝敗の判定（目数の計算）
- 勝敗の表示（どちらが何目勝ったか）
- 盤面の状態の表示（石の生死、地の表示）
- 棋譜ファイルの出力

デバッグ用の機能

- 対局中の1手ごとに盤面認識状況や、評価値を表示する
- 表示の方法は、ファイルへの出力、別ウインドウを開いて表示など

デバッグ用プログラム

- 任意の盤面を入力し、その盤面での認識状況や評価値を表示するプログラム
- 入力する任意の盤面は、盤面図（石を任意の場所に配置した盤面）や棋譜

あると便利なプログラム

- データベース入力プログラム
  - データ量が多いと、入力しやすい方がいい
  - 既に入力したデータと重複していないか調べる
- 問題集と問題回答プログラム
  - 現在の強さを調べたり、改良による悪影響がないかを調べるための問題集
  - 1問ずつ人間が与えるのではなく、自動で全問回答させる
  - 正解数や正解率を集計させる

あると便利なプログラム

- 自動対局プログラム
  - 他の囲碁プログラムと対局させて棋譜を残し、後で解析して弱点を見つける
  - 自動で何局も続けて対戦させ、勝率を計算させる

改良の手順

- 仕様を決めてプログラムを作っても、思っていたほど強いプログラムはできない
  - 想定外の局面が、必ず現れる
  - 異なる評価基準の間の重み付けのバランスをとるのが難しい
- 試しに作ってみては、テスト（対局や問題集）する、の繰り返し：トライアルアンドエラー（trial-and-error）
- コンピュータの能力は毎年上昇するので、設計からやり直すことも：スクラップアンドビルド（scrap-and-build）

開発の注意事項

- 設計書、仕様書を作る
  - アイデアや方針をはっきりさせるため
  - 作った内容が、後で見てわかるようにするため
  - 文書や図を記述すると、考えが整理され、内容が洗練されていくという効果もある
- プログラムには、たくさんコメントを入れる

講習会で説明しきれなかった課題

- トランスポジションテーブル（同じ盤面が出現したときに、以前に覚えた情報をそのまま使うことができる）
- インクリメンタル更新（盤面の一部分が以前に覚えた情報と同じ場合、異なる部分だけ情報の更新を行えば済む）
- 時間制御（深く考えれば良い手がみつかりそうな盤面とそうでない盤面がある、残り時間によって考える内容を変える）

第5部 コンピュータ囲碁大会の大会規約(岐阜チャレンジ2003)

■大会規約

(注意)規約の言語について
本規約には日本語版と英語版とがあります。疑義ある時は日本語版を以て正本とします。

1.重要な日程

省略

2.登録

(1)参加資格
a) 参加者の国籍、性別、年齢は問いません。
b) 個人、チーム、団体のいずれも参加できます。ただし、同一あるいはほとんど同一のプログラムは、参加者名の如何に関わらず、本大会に1つしか参加できません。(プログラムの同一性は、思考部分のアルゴリズムにおける独自性の有無で、大会実行委員長が判断します)
あるチームが、2つ以上の異なったプログラムを共に参加させたい場合には、その旨を記し、それらの相違点を記して、実行委員会に参加の許諾を求めることができます。実行委員会は、その説明を判断し、複数参加の許諾を与えることがあります。許諾にあたっては、各プログラムを提出していただいた上で判断することもあります。
c) 代表者は、プログラム開発者、または開発者から許可を得た者に限ります。
d) 参加プログラムは、他者のプログラムからの無断コピー等の不正行為によらない、参加者独自のものに限ります。(コピーが許可されているプログラムを部分的に利用することはできます)
e) 参加プログラムは、シリアルポート(RS232C)による通信対局機能を備えていなければなりません。(実装にあたっては、DLLの提供などのサポートがあります)
f) 参加プログラムは、SGF形式での棋譜記録機能を備えていなければなりません。

(2)参加申し込み
付録A「参加申込書」に記入して頂いたものを、下記申込先への電子メールで受け付けます。
日本語と英語2種類の参加者用メーリングリスト(以下「参加者ML」)が用意されます。参加申し込み時に、どちらか一方もしくは両方に登録して頂きます。
(参加者MLは原則として大会終了後1年間保持されます。この期間中にアドレスを変更した時には、大会事務局に速やかにご連絡下さい)

(3)申込先(大会事務局)
省略

(4)申込の取消
参加を取り消す場合は速やかに、遅くとも大会開始1週間前の日本時間17:00までに、電子メールで連絡しなければなりません。(取消の連絡を受信した場合は、事務局から2日以内に取消確認のメールを返信し、参加者MLから削除します)

(5)問い合わせ
省略

3.機材と確認

(1) 機材
必要な機材は参加者が用意するものとします。原則として大会事務局は機材の貸し出しを行いません。(計算機本体に加え、通信対戦に使用するケーブルなど、必要な機材は必ず持参して下さい。必要ならば変換コネクタなども持参してください)
使用する計算機に対する制限は設けません。ただし、リモートアクセスをして対戦することは認めません。また、消費電力等で物理的に設置が不可能な、極端に大規模な計算機については、大会事務局の判断でお断りすることがあります。

(2) 通信対局
対局は原則として、RS232Cによる直接通信対局方式によります。この大会で使用される通信規約は、SGMP(Standard Go Modem Protocol)の簡易版です(詳細は今後公開する資料を参照してください)。通信対局機能を実装していないプログラムが参加することはできません。通信プログラムとその実装例は大会Webページで公開します。参加者は事前に通信のチェックをしておかなければなりません。
大会においては、通信トラブル等で大会実行委員会がやむを得ないと認めた場合に限り、手入力で対局を行うことができます。この場合、手入力にかかる時間も消費時間に加えます。

4.対局時の操作

(1)参加プログラムは、原則として参加申込者が操作しなければなりません。参加申込者が操作できない場合は、操作の代理人を大会受付時までに申告するものとします。

(2)一局の間での人によるプログラムの修正は許されません。不必要なキータッチ等の操作は、計算機に情報を与えたものとみなされます。審判の許可なくキータッチ等の操作を行うことは許されません。

(3)大会実行委員会の許可を得て手入力で対局する場合には、参加者は大会実行委員会が用意する通信機能付きの入力プログラムに自分のプログラムの着手を入力します。対局中、勝敗の決定前に着手の入力間違いが発見された場合には、自動的に負けになります。

5. 対局

対局は以下の方式で行います。組合せ方法や順位決定方法の詳細は、別途お知らせします。

(1) 手合割
　互先、先番6目半コミ出し。

(2) 先後
　対局ごとに操作者のにぎりで決定します。

(3) 持ち時間
　各40分とします。消費時間は双方のプログラムで計測します。

(4) 遅刻
　交通機関の遅延などやむを得ない場合を除き、1局目の組合せに遅刻した場合は、大会参加資格を失います。大会実行委員会がやむを得ない事情と判断した場合には、1局目を不戦敗とし、2局目からの参加を認めます。2局目開始時刻にも遅刻した場合は、いかなる理由でも大会参加資格を失います。
　1局目の組合せは受付時間終了直後に行います。受付時間は大会2週間前までに大会Webページに掲載します。

6.終局と勝敗の決定

(1) 対局の停止
　次のいずれかの時点で対局を停止します。
　a) プログラムの一方が着手を放棄し、次いで相手方も着手を放棄した時点
　b) プログラムの一方が投了の意思表示をした時点
　c) 操作者の一方が投了の意思表示をした時点
　d) 一方の持ち時間が無くなった時点
　e) 400手目の着手完了または着手放棄時点（着手放棄も1手に数えます。プログラムあるいはプログラム操作者が対局を停止させます）
　f) 反則があった時点。

(2) 終局
　6(1)a)の場合には、対局の停止後、以下を行うことにより対局を終了し、これを終局とします。プログラムあるいはプログラム操作者が対局の再開を要請することはできません。双方のプログラムは、死石を明確にモニター上に表示します。双方のプログラムが地の計算を行います。双方のプログラムの死活情報および地の計算が目数まで一致した場合、それを結果とします。一致しなければ、プログラム操作者同士が、石の死活および地を確認し合意することにより対局を終了し、最後に審判が確認して勝敗を判定します。一方あるいは双方のプログラム操作者に判定が困難な場合、あるいは双方で勝敗に合意出来ない場合は、審判が勝敗を判定します。

(3) 投了の場合の処理
　一方のプログラムまたは操作者が投了の意思表示をした場合には、投了した方の負け、相手方の中押勝とします。

(4) 時間切れの場合の処理
　持ち時間を使い切った場合には、使い切った方の時間切れ負けとします。

(5) 400手の場合の処理
　審判が石の死活および地を確認することにより対局を終了し勝敗を判定します。

(6) 反則の場合の処理
　反則した方を負けとします。相手方のプログラムまたは操作者が反則を見逃し、対局が続行した場合でも、双方の勝敗合意前かつ審判の勝敗判定前ならば、操作者あるいは審判が反則に気付けば、反則した方を負けとします。双方の勝敗合意後あるいは審判の勝敗判定後は、勝敗は覆りません。

7.対局ルール

　ここに定めのない対局ルールについては、日本囲碁規約によります。ルールの解釈については、審判の解釈に従って頂きます。

8.異常事態の処置

(1) 同一盤面再現
　対局中に同一盤面が再現した場合には、審判が、対局を中断し、無勝負の裁定あるいは勝敗の判定をします。無勝負は双方0.5勝とします。

(2) プログラムの異常終了
　プログラムの異常終了に際しては、審判の判定によって、途中からの対局再開、再対局、中断局面での審判による勝敗の判定のいずれかを行います。異常終了の収拾に要した時間は、審判の判断で持ち時間から差し引きます。審判が、一方あるいは双方の計算機システムの動作環境に問題があると認めた場合には、それを修正して実行を再開することができます。その際に、対局プログラムのアルゴリズムやデータベースの変更は許されません。また審判が、異常終了の原因が通信プロトコルの問題であると認めた場合には、手入力への切り替えができます。参加者は大会実行委員会が用意する通信機能付きの入力プログラムに自分のプログラムの着手を入力します。対局中、勝敗の決定前に着手の入力間違いが発見された場合には、自動的に入力間違いの時点での負けになります。

(3) 電源事故
　双方もしくは片方の計算機に電源事故などが発生し対局が中断した場合には、審判の判定によって、途中からの対局再開、再対局、中断局面での審判による勝敗の判定のいずれかを行います。

(4) 引き分け
終局に至らない局面で対局が停止し、審判が判定する時には、引き分けとすることがあります。引き分けは双方0.5勝とします。

(5) 不正な行為
不正な行為が発覚した場合には、不正な行為が行われた時点でその対局は不正を行った方の負けとなり、その後の対局は失格となります。また懲罰の対象となることがあります。

(6) その他
ここに記載されていない原因による異常事態が対局中に生じた場合には、審判が裁定を行います。審判で判断がつかない場合には、大会実行委員長が裁定を下します。その他、対局結果の取り扱いなどで本規定に定めのない事態が生じた場合にも、審判又は大会実行委員長の裁定に従って頂きます。いかなる場合についても大会実行委員長の裁定を最終決定とします。

9.表彰

上位参加者を表彰し、賞金および賞状、記念品を授与します。

10.提出物

以下の4点を提出することを大会参加者の義務とします。提出がない場合には、失格となることがあります。

(1) 参加プログラム
a) 盗作プログラム等の不正なプログラムの参加を防止するため、嫌疑がかけられた時の調査資料として、参加プログラム（実行形式）のコピーを提出して頂きます。実行時にプログラムが利用するファイルが別にある場合は、それらのコピーも提出して頂きます。プログラムは大会受付時に提出して頂きます。
b) プログラム等は大会事務局が厳重に保管し、不正プログラムの嫌疑がかけられた場合の調査以外では使用しません。
c) 大会終了後1年を経過しても不正プログラムの指摘等がなく、かつ問題がない場合、プログラム等は大会事務局が責任を持って廃棄いたします。

(2) 勝敗記録用紙
対局終了毎に勝敗結果を大会事務局に提出して頂きます。対局開始時に、大会事務局より勝敗の記録用紙をお渡しします。対局終了後、その用紙に勝敗結果および両対局者の署名を記入し、勝者が提出します。審判が判定あるいは裁定した場合には、審判の署名も必要です。

(3) 棋譜ファイル
対局終了毎に棋譜ファイルを大会事務局に提出して頂きます。対局開始時に、大会事務局よりフロッピーディスク（FD）をお渡しします。対局終了後、そのFDに棋譜ファイルをSGF形式で書き込み、双方が提出します。

(4) 誓約書
参加申込者には、参加プログラムが参加者独自のものであり、また参加者が本大会規約ならびに大会参加に関わる大会実行委員長による決定に全て従う旨の誓約書（付録B）を、大会受付時に提出していただきます。

11.その他

(1) 不正の扱い
参加プログラムが不正なプログラムであることが明らかになった場合には、大会参加資格を取り消したり、該当プログラムの成績を大会記録から抹消の上、賞金の返還等を求めたりすることがあります。

(2) 大会運営の円滑化をはかるために、試合の前日まで、ルールなどの事項の小規模な変更を行うことがあります。この場合は全参加者に同時にかつ速やかに知らされるようにします。

(3) 本大会規約に含まれない事態が対局以外で発生した場合は、大会実行委員長が決定し、この決定を最終決定とします。

(4) 当日の日程
詳細発表は大会Webページで行います。

(5) 著作権
大会中に対局を記録した著作物の著作権は主催者に帰属しますが、非営利の目的には自由に使用できます。

(6) 関係するWebページ・電子メールアドレス一覧最新の情報やこの本規約の補足情報などは、大会Webページに掲示します。問い合わせは大会実行委員会までお願いします。

12.大会役員

省略

■ 通信規格資料日本語版

省略

■ 棋譜ファイルの書式日本語版

Smart Game Format (SGF)

ＳＧＦは、( ) の中に、;で始まるノードの列で表されます。最初のノードはルートノードと呼ばれ、一局全体にかかわるプロパティを含んでいます。ルートノードの次のノードが第一手を表します。

(;
GM[1]　　/GMはゲームの種類を表す。囲碁は1
FF[1]　　/FFはSGFのバージョン(1～4)を表す。
SZ[19]　　/盤のサイズ。19路盤
PB[player black]　　/黒番の名前
PW[player white]　　/白番の名前
DT[date]　　/対局日
PC[place]　　/対局場所
KM[komi]　　/コミ
TM[time for each player]　　/持ち時間（分）
RU[rule]　　/使用するルール。Japanese
RE[result]　　/結果　B+10.5は黒番１０．５目勝ち。B+Rは黒中押し勝ち
EV[event]　　/イベント名
GN[game name]　　/ゲーム名。第x局
;B[aa];W[bb];B[cc]・・・　　/黒番から見た左上をaa,右上をsaとする。１０手ごとに改行。
;B[tt];W[tt]　　/パスはtt。
)

具体例
(;
GM[1]
FF[1]
SZ[19]
PB[Black Program]
PW[White Program]
DT[2003-08-02]
PC[Gifu]
KM[6.5]
TM[40]
RU[Japanese]
RE[B+0.5]
EV[Gifu Challenge 2003]
GN[1st round]
;B[dd];W[pd]・・・
;B[tt];W[tt]
)

次回までの宿題

- 各自（グループでも可）テーマを決めて、プログラムを作り始めてください。
  （例）対局プログラム
  棋譜再現プログラム
  詰碁プログラム
  任意の盤面を与え、次の一手を求めるプログラム
- 完成していなくても良いので、できたところまでのプログラムとドキュメントを持ってきて、説明してください。