囲碁プログラミング講習会

～　マニュアル　～

平成１６年５月１５日(土)
平成１６年５月１６日(日)

目次

5月15日(土)

5月16日(日)

囲碁プログラミング講演会

日程：２００４年５月１５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

講演会スケジュール

- 13:00 - 14:00 人工知能とゲームプログラミング
- 14:10 - 14:40 過去の大会の報告と特選譜解説
- 14:50 - 15:40 コンピュータ囲碁の構造
- 15:50 - 16:00 総合質疑

第1部 人工知能とゲームプログラミング

第1部の内容

- 人工知能とは？
- ゲームプログラミングの意義
- ゲームプログラミング研究の具体例
  ゲームのプログラム開発の歴史
- 「人工知能」はどこまで人間の知性に近づいたか？

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

ゲームの分類

- ゲームの分類（使用するものによる分類）
  コンピュータゲーム、ネットワークゲーム、ボードゲーム、カードゲーム、ワードゲーム、、、
- ゲームの分類（内容による分類）
  思考ゲーム、シミュレーションゲーム、ロールプレイングゲーム、シューティングゲーム、レーシングゲーム、、、
- 思考ゲームの分類
  囲碁、将棋、チェス、コントラクトブリッジ、パズル、、、

囲碁の性質

- 二人ゲーム
- 零和ゲーム
  一方が勝ったら、もう一方が負け
- 有限ゲーム
  いつかは必ず終わるゲーム（囲碁はルール次第）
- 確定ゲーム
  （サイコロのような）偶然的な要素が入らない
- 完全情報ゲーム
  全ての情報が与えられている

二人零和有限確定完全情報ゲームには、必勝法が存在する

コンピュータと囲碁

- 対局プログラム
- 囲碁を解く（詰碁、ヨセ、小路盤）
- e-Learningの教材（講座、問題集）
- 棋譜データベース、棋譜管理
- インターネット碁会所
- その他（棋譜OCR、棋譜自動記録など）

ゲームプログラムの歴史

- 1950年、Shannonによるチェスプログラムの論文
- 1962年、Remusによる囲碁プログラムの論文
- 1990年、日本でコンピュータ将棋選手権が始まる（2004年5月に第14回大会）
- 1997年、チェスプログラムDeepBlueが、世界チャンピオンG.Kasparovに勝利
- 2001年、囲碁プログラム「手談対局4」が初段に認定される

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、6×6オセロ
- 世界チャンピオンと同等以上
  - チェッカー
  - リバーシ（オセロ）
  - チェス

ゲームプログラムの強さ

- まだ人間の方が強いゲーム
  - 将棋：アマチュア4、5段程度
    人間の名人まで、あと10～15年？
  - 囲碁：アマチュア初段程度
    人間の名人を破るのは2048年と予想
    （囲碁プログラム開発者、研究者の予測の平均値：2003年1月4日、朝日新聞夕刊より）

メジャーなゲームの中では、囲碁は最も難しい！？

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

プログラムの難しさ（探索空間）

- 探索空間
  - ある局面でN通りの候補があり、終局までにM手かかるならば、探索空間は

$N \times N \times N \times \ldots = N^M$

ゲーム	オセロ	チェス	将棋	囲碁
探索空間	10^{60}	10^{120}	10^{200}	10^{300}

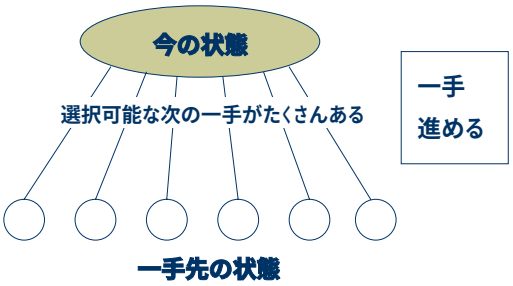
プログラムの難しさ（ヒューリスティクス）

- 知識や判断基準の定義と優先順位の決定を正確に行なうことがとても難しい（相反する場合、過ぎたるは及ばざるがごとしなんてことも多い）
- 人間をお手本にしたくても、何をどのような順番で考えているか、全てはわからない

コンピュータの得意分野

- ゲームプログラミングの分野で一番成功しているのは、しらみつぶしの探索
  全部の手を調べちゃいましょう！！
- 将棋や囲碁でも、探索が有効に働く部分はかなり成功している
  詰将棋：プロ棋士以上
  詰碁：高段者並（狭い領域の問題はほぼ完璧）

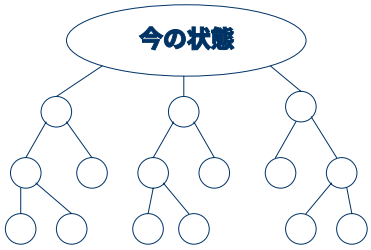
先読みと局面評価



先読みと局面評価

- プログラムは、今の状態（局面）から一手進めてみて、自分と相手の有利不利を評価する
  （ゲームの定義、ヒューリスティクスから決めた「評価関数」を使う）
- 選択可能な次の一手すべてについて、一手進めて、評価するという手順を繰り返す
- 自分に一番有利になる手を次の一手として選択する
  （本当は一手だけではなく、可能な限り先まで進めて評価する）

先読みと局面評価



自分も相手も、それぞれにとって一番有利になる手を選んでくると仮定する

先読みと局面評価

- どれだけ先まで読むか（速さ）
- どれだけ正確な評価をするか（正確さ）

これらが優れていれば強いプログラムになる！

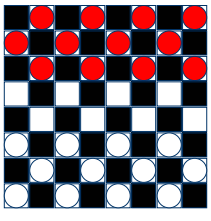
囲碁の難しさ

- 探索空間
  有名なゲームではもっとも複雑？
- ヒューリスティクス
  盤上は黒、白、空白の3つの状態のみ
  しかし、そこから多様なパターンが現れる！
  何を基準に形勢判断すれば良いのか、将棋等と比べても定式化が難しい
  （なぜ人間は正確に判断できるんだ？？？）

さまざまなゲームのプログラム

- さまざまなゲームのプログラムは、どのようにして強くなってきたのか？
  - チェッカー
  - チェス
  - 将棋
  - 囲碁

チェッカー



駒を斜めに動かしながら相手の駒を取っていき、先に駒を全部取られた方が負けとなる

チェッカー

- 1950年代 Samuelのプログラム
  マスターレベルのプログラム
  その後、研究者の興味はチェスの方に向く
- CHINOOK（1989ー）
  J. Schaefferら
- 人間の最強のチャンピオン M. Tinsley
  (1950年 ー 1992年で負けたのはたった5回)

チェッカー

- 1992年 マン・マシン世界選手権
  - 当初、全米チェッカー協会とイギリスチェッカー協会は対戦を認めず
  - Tinsleyは世界チャンピオン位を返上してもCHINOOKとの対戦を望む（「人間とプレイしても、誰も相手にならないから退屈である」）
  - 正式なタイトル戦が実現
  - 4勝2敗33引き分けでTinsleyの勝利

チェッカー

- 1994年 CHINOOK対Tinsley再戦
  - 0勝0敗6引き分け
  - 病気のため、Tinsleyが棄権。CHINOOKがチャンピオンになる

チェッカー

- CHINOOKの仕組み
  - 序盤：データベース
  - 中盤：探索と評価関数
  - 終盤：完全な先読みとデータベース
  - （駒数が少なくなるので）

チェス

- ゲームプログラミングの研究対象としては、長い間チェスが主役だった
- 1950年 Shannonの論文
  - ミニマックス法によるゲーム木の探索
  - 評価関数を用いた局面の評価
  - 現在でも使用されている基本的なアイディアの提案
- 1950年代後半 チェスをプレイするプログラム誕生
  - ハードウェアの制約により全ての可能な手を探索することは非現実的で、知識主導型のアルゴリズムが中心

チェス

- 1970年代 探索主導型アルゴリズム
  - ハードウェアの進歩に伴い、探索主導型のアルゴリズムが中心となる（KAISSA, Chess 4.6等）
- 1980年代 チェス専用ハードウェア
  - チェス専用のハードウェアを利用したプログラムの出現（Deep Thought；F. Hsuら）
  - トップ100のレベルに達したと言われている

チェス

- 1996年 世界チャンピオンとの対戦
  - Deep Blue（Deep Thoughtの後継プログラム）が世界チャンピオンのG. Kasparovと対戦
  - 3勝1敗2引き分けでKasparovが勝利
- 1997年 Deep Blue世界チャンピオンに勝つ
  - 2勝1敗3引き分けでDeep Blueが勝利

チェス

- Deep Blueの仕組み
  - 序盤：データベース
  - 中盤：探索と評価関数
  - 終盤：正確に読み切る
  - チェス専用ハードウェアで、探索の速度が非常に速かった
    （1秒間に数億手を読むことができる）

将棋

- 将棋の難しさ（チェスとの比較）
  - 探索空間が大きいので、しらみつぶし探索が難しい
    良さそうな手だけ選択して先読みするから、読み抜けが起きることがある
  - 良い手を逃さずに先読みすること、評価関数をさらに改良するためにさまざまな工夫が試みられている
  - 詰め将棋等、終盤に将棋独自の工夫が数多く研究されている

将棋

- 1979年 初めてコンピュータ将棋のプログラム同士の対局が行なわれる
- 1980年代 パソコン用市販プログラムの登場
- 1990年 コンピュータ将棋選手権始まる
  （第14回：2004年5月2 - 4日）

将棋

- 詰め将棋解答プログラムは1990年代に急速に進歩した
  1997年には最長手数の詰め将棋「ミクロコスモス」（1525手詰め）を解いた
- 将棋のトッププログラムはアマチュア高段者レベル（終盤は相当強い）
- プロレベルまでもう少しです！

囲碁

- 1969年、Zobristによる初めての対局囲碁プログラム（強さは約38級）
- 1979年、ReitmanとWilcoxの囲碁プログラム（約15級）
- 1984年、初めてのコンピュータ囲碁大会
- 1995年、日本棋院が初めて囲碁プログラムに級位を認定（5級）
- 2001年、囲碁プログラムが初めて初段に認定

現在は4つのプログラムが初段に認定されている

探索以外の方法

- 探索以外の方法も長年研究されている
  - 探索以外の技法のプログラムへの取込み
    機械学習、ニューラルネットワーク、推論等
  - 人間の思考の過程を調べる研究
    （例）アイカメラを使った研究
    人間の強豪プレイヤは、盤面を見た瞬間に重要な部分を瞬時に判断して、そこに視線を向ける
  - 将来はさまざまな技法が探索と組み合わせて利用されるようになるのではないか

これからのテーマ

- 将棋、囲碁
- 不完全情報ゲーム
- もっと現実の社会に近い問題
  ロボカップサッカー、ロボカップレスキュー、社会科学、市場問題等々

コンピュータと知性

- コンピュータは知性を持ったのか？
- 人間の思考を再現して強くなったわけではない
- 特定の問題に最適化したプログラムを作っただけではないのか？
- プログラムを賢く作るのも、人間がやったこと

あえて言うなら

- プログラム
  ある種の知性的に見える機能は実現した
  まだまだやるべきことは多い！
- 人間
  それでもやはり人間の能力は素晴らしい
  すべてを計算せずに、認知、判断できる

まとめ

- 人工知能研究の例題として、ゲームプログラミングが用いられてきた
- 特に探索の分野で成果を上げ、人工知能、コンピュータサイエンスに貢献した
- 現状：量で質をカバーして、問題を解決している
  （囲碁ではあまりうまく働いていない。。。）
- 頑張って賢いプログラムを作りましょう！

第2部 過去の大会の報告と特選譜解説

- コンピュータ囲碁大会について
- 特選譜解説（岐阜チャレンジ２００３より）

コンピュータ囲碁大会

- 囲碁プログラムどうしを対戦させ、一番強いプログラムを決める大会
- 大会会場に自作のプログラムとコンピュータを持ち込み、通信ケーブルをつないで着手の位置を送りあう
- この講習会の目的は、コンピュータ囲碁の世界大会「岐阜チャレンジ２００４」（１０月２、３日に開催）へ参加できるプログラムの作成

コンピュータ囲碁世界大会

- **現在も続いている世界大会**
  (1) 岐阜チャレンジ（2003年～）：初段認定されている4つのプログラムが上位4位までを独占した、最もレベルの高い大会
  (2) 21世紀杯（2001年～）：開催場所が毎回変わる
  (3) Computer Olympiad（1989年～）：他のゲームの大会も同時開催
- **過去に行われた主な世界大会**
  (1) IngCup（応氏杯）（1985年～2000年）：初めての世界大会。プロ棋士に勝ったら2億円の賞金
  (2) FOST杯（1995年～1999年）：日本で開催された世界大会。最も多いときは38プログラムが参加

岐阜チャレンジ2003

- 2003年8月2日、3日
- 会場：ソフトピアジャパン・センタービル
- 主催：コンピュータ囲碁フォーラム
- 賞金：優勝30万円（6位まで賞金）
- 参加プログラム：17（日本、中国、韓国、北朝鮮、アメリカ、イギリス、オランダ）

※ソフトピアジャパン3Fソピアホールにて

※昨年度の大会風景

※解説をまじえながら・・・

※放送局からの取材もありました

岐阜チャレンジ2003（結果）

- 優勝：KCC囲碁（KCC囲碁開発チーム、北朝鮮）
  8勝1敗、商品名：銀星囲碁
- 2位：HARUKA（河龍一、日本）
  8勝1敗、商品名：最高峰
- 3位：Go++（Michael Reiss、イギリス）
  7勝2敗、商品名：最強の囲碁
- 4位：Goemate（陳志行、中国）
  7勝2敗、商品名：手談対局
- 5位：Many Faces of Go（David Fotland、アメリカ）
  5勝4敗、商品名：AI囲碁

岐阜チャレンジ2003（棋譜紹介）

- 黒：Goemate（4位） ー 白：HARUKA（2位）
  - 白27目半勝ち
  - 下辺の黒攻めに失敗しても、代償を得る強さ
  - 左上104の切りからの攻め合い
- 黒：GNU Go（6位） ー 白：Go Intellect（9位）
  - 黒14目半勝ち
  - 左下、右上、中央と死活の連続

第3部 コンピュータ囲碁の構造

- どうすれば強いプログラムができる？
- 囲碁プログラムの処理の流れ
  （3段階モデル）
- 囲碁プログラムのデータ構造
- 開発に適した言語・コンピュータ

どうすれば強いプログラムができる？

- 囲碁の必勝法は、まだ見つかっていない
- コンピュータに「力ずく」で先読みをさせようとしても、盤が広すぎて時間が足らない
- プロの棋譜を全部覚えて、同じ手を打とうとしても、10万局の棋譜じゃ対応しきれない

↓

人と同じように考えるプログラムを作ろう

碁を打つとき、人は何を考える？

- この石は生きているのかな
- この模様は何目ぐらいの大きさだろう
- この定石だと、次の手は確かここだったよな
- もし自分がここに打つと相手がここに打って、次に自分がここに打つと…
- この手は後手3目だから、あっちにある後手2目の手より先に打とう
- 残り時間があと5分しかないから、もっと早く打たなきゃ
- こいつにだけは絶対に負けられない
- 今日の晩御飯のおかずは何かな

囲碁プログラムの処理の流れ

生死、地、模様など

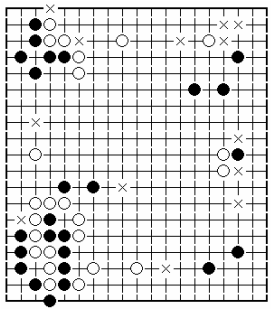
定石、死活、ヨセなど

どの候補手が一番良いか

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

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

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

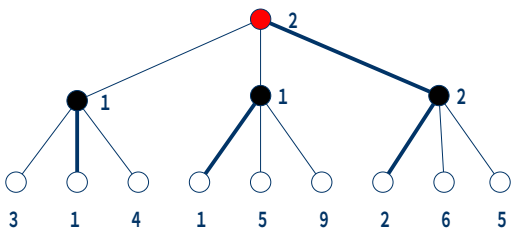
評価

- 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
- 目数に換算できない手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が重要である
  - 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

着手選択

- 探索方式
  各座標に仮に打ってみて盤面の状態を認識する。最も有利になる手を選択。何手か先まで読むこともある。
- 優先順位方式
  A候補手があればAを採用。A候補手がないとき、B候補手があればB。B候補手もないとき、C候補手があれば、、、
- 加点方式
  A候補手はa点。B候補手はb点。C候補手は、、、
  各座標ごとに点を加算していき、最高点の手を選択する。
  （例：ヒラキとツメを兼ねると点が高くなる）

探索（2手読みの場合）



- 相手は、相手が有利になる（こちらが不利になる）手を選ぶ
- 深く読むほど盤面認識の回数が多く、処理時間が長くなる

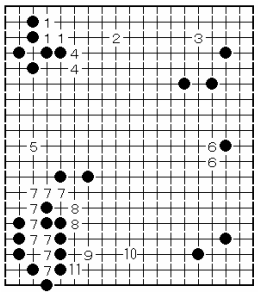
囲碁プログラムのデータ構造

- 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

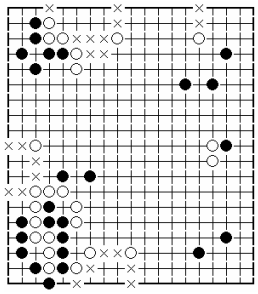
典型的なデータ構造の例

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

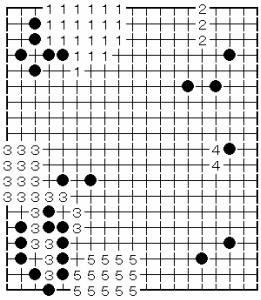
データ構造（連）



データ構造（結線）



データ構造（群）



1の群の属性：

座標は（4、1）、、、
色は白
地は10目
黒に包囲されていない
近くに同色の群は無い
強い
隣接する敵の群は強い ・・・・・

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
  - 地が増やせない＆近くに味方がいるなら、連結する手を生成する
- 評価
  - 地が計算できるので、地が増える手の評価値が高くなる
  - 連結した後の群の形状がわかるので、地や強さを計算できる

囲碁向きの言語･コンピュータ

- 囲碁プログラムの開発に向いている言語が存在するのかどうか、不明
  囲碁プログラムの作者は、本人が使い慣れた言語を使用している
- 囲碁プログラムを実行するには、速いコンピュータの方が良い
  やりたい処理はたくさんある
  速さを使いこなせなくては強くなれない（盤面認識の精度を上げるためには、評価関数間のバランスが重要）

囲碁プログラミング講習会

日程：2004年5月16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

講義スケジュール

- 5月15日：講演会（オリエンテーション）
- 5月16日：講義（囲碁プログラムの概要）
- 6月12日：講義（盤面認識、候補手生成）
- 6月13日：講義（候補手生成、着手選択）
- 7月24、25日：実習
- 8月18～20日：実習

本日のスケジュール

- 10:00 - 11:00 自己紹介
  囲碁プログラムの概要説明
- 11:00 - 12:00 碁盤プログラムの作成
  （石を取る、合法手、終局の判断）
- 13:00 - 14:30 地の認識の説明
  （地とは何か、どのように認識するか）
- 14:30 - 15:30 データ構造の説明
  （囲碁用のデータ構造、連）
- 15:30 - 16:00 質疑

自己紹介

- 氏名
- 学校名、学年
- 囲碁の棋力
- プログラミング経験
- プログラミング環境（自宅／学校、言語）
- 講習会に参加した目的と目標

囲碁プログラムとは

- コンピュータの画面を碁盤と碁石の代わりにする。
- プログラムが次の一手を考える。
- 強い囲碁プログラムの開発は、人工知能の研究における究極の課題の一つである
  （現在は4つのプログラムが初段に認定されている）

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 ⇒ 結線
- 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、判断できると有利

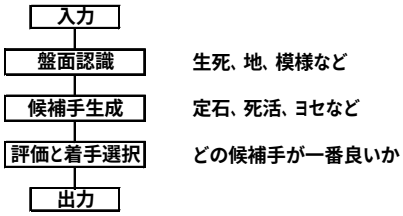
石の生死の判断

- 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
- 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）と有利

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

囲碁プログラムの処理の流れ



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

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）

参考文献

- **将棋とコンピュータ**
  松原仁、共立出版、1994年
  メインテーマは将棋だが、ゲームプログラミング全般に共通する話題もたくさん書かれている
- **ゲームプログラミング**
  松原仁・竹内郁雄共編、共立出版、1998年
  各種のゲームプログラミングに関する当時の最新情報が書かれている
- **人工知能と人間**
  長尾真、岩波新書、1992年
  人工知能全般についてわかりやすく書かれている

参考URL

- **勝也のページ**
  http://homepage1.nifty.com/Ike/katsunari/index.html
- YSSと彩のページ
  http://www32.ocn.ne.jp/~yss/index_j.html
- GNU Goのページ
  http://www.gnu.org/software/gnugo/
- CGFのページ
  http://www.computer-go.jp/indexj.html
- ICGAのページ
  http://www.cs.unimaas.nl/icga/

第2部 碁盤プログラムの作成

碁盤プログラムの作成

最初はコンピュータを碁盤と碁石の代わりにした、碁盤プログラムを作ります。

碁盤プログラムの機能

- 盤面を表示する(出力)
- 石を打つことができる(着手位置の入力)
- 石を取る処理ができる
- 合法手(ルールに違反しない手)の判断ができる
- 交互に着手できる
- 終局できる

以上の機能が実現できれば、人対人の対局ができます。また、思考プログラムを作成して人の代わりにすれば、人対プログラムの対局ができるようになります。

碁盤の表現(データ構造)

碁盤は二次元配列を使うのがわかりやすいでしょう。碁盤は19×19なのですが、周囲を一回り大きくして21×21の配列を用意します。実際に石を置くのは21×21の内側の19×19の範囲です。理由はいろいろありますが、一番の理由は分かりやすさです。だって、座標が0から始まるよりも1から始まる方が分かりやすいでしょう？

碁盤の各点には、黒／白／空点(石が置かれていない状態)の3つのうちのどれかを設定します。ついでに盤外という状態もあると便利です。21×21の一番外側には常に盤外を設定しておきましょう。

なお、この講習会ではC言語を使って説明します。

```
#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             */

/* 碁盤 */
int board[BOARD_SIZE][BOARD_SIZE];

/* 碁盤の初期化 */
void InitializeBoard()
{
  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;
  }
}
```

右上隅の星、つまり座標(16,4)に黒石を置くときは、次のようになります。
(X座標とY座標の順番をわざと逆にしていますので注意してください。)

```
board[4][16] = BLACK;
```

碁盤の表示

次は碁盤の表示です。凝ったグラフィックで綺麗な画面を書き、操作しやすいユーザーインタフェースを書きたいところですが、ここではそこまでしません。囲碁の対局に最低限必要な部分だけにとどめます。

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

/* 碁盤を表示する */
void DisplayBoard()
{
  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" );
  }
}
```

右上隅の星(16,4)に黒、左上隅小目(3,4)に白があるとき、次のように表示されます。

```
    [ 1 2 3 4 5 6 7 8 9101112131415161718 19]
[ 1]  + + + + + + + + + + + + + + + + + + +
[ 2]  + + + + + + + + + + + + + + + + + + +
[ 3]  + + + + + + + + + + + + + + + + + + +
[ 4]  + + O + + + + + + + + + + * + + + +
[ 5]  + + + + + + + + + + + + + + + + + + +
[ 6]  + + + + + + + + + + + + + + + + + + +
[ 7]  + + + + + + + + + + + + + + + + + + +
[ 8]  + + + + + + + + + + + + + + + + + + +
[ 9]  + + + + + + + + + + + + + + + + + + +
[10]  + + + + + + + + + + + + + + + + + + +
[11]  + + + + + + + + + + + + + + + + + + +
[12]  + + + + + + + + + + + + + + + + + + +
[13]  + + + + + + + + + + + + + + + + + + +
[14]  + + + + + + + + + + + + + + + + + + +
[15]  + + + + + + + + + + + + + + + + + + +
[16]  + + + + + + + + + + + + + + + + + + +
[17]  + + + + + + + + + + + + + + + + + + +
[18]  + + + + + + + + + + + + + + + + + + +
[19]  + + + + + + + + + + + + + + + + + + +
```

対局の流れ

本来は、日本の囲碁ルールである日本囲碁規約の通りに対局すべきなのですが、ここでは簡略化したルールを使います。(あなたは、対局の停止と終局の違いを知っていますか？ルールを完全に知らなくても、誰でも対局ができるのです。)

ここでは、黒と白の両者が連続してパスをしたときに終局としましょう。

```
/*--------------------*/
/* 終局するまでループ */
/*--------------------*/

while( 1 ){

  /*------*/
  /* 黒番 */
  /*------*/

  /* (1) 黒の着手位置を入力 */

  /* (2) 黒が投了なら白の勝ち */
  /*     このループを抜ける   */

  /* (3) 黒白ともにパスなら地を数えて勝者を表示 */
  /*     このループを抜ける                     */
```

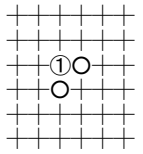
```
  /*（4）碁盤に黒石を置く */

  /*------*/
  /* 白番 */
  /*------*/

  /*（1）白の着手位置を入力 */

  /*（2）白が投了なら黒の勝ち */
  /*       このループを抜ける    */

  /*（3）黒白ともにパスなら地を数えて勝者を表示 */
  /*       このループを抜ける                    */

  /*（4）碁盤に白石を置く */
}
```

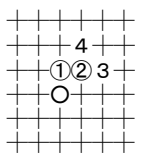
石を取る

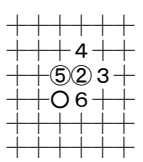
四方を囲まれたら、囲まれた石を取り除く処理です。
ある座標(x,y)の石が囲まれているかどうか調べるということは、ある座標の石の隣の4点(x-1,y),(x,y-1),(x+1,y),(x,y+1)に空点がある(ダメがある)かどうかを探すことです。もしも空点があれば、その石は囲まれていないので、石を取ることはできません。もしも空点が一つもなかったら、その石は碁盤から取り除きます。

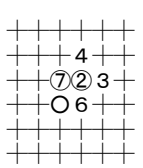
調べようとしている座標(x,y)の隣(例えば(x+1,y))に同じ色の石があったら、その石の四方も調べないといけません。その石(x+1,y)の隣(x+2,y)も同じ色の石があったら、その石の四方も調べないといけません。同じ色の石が連続している間は、隣、隣、、、と調べ続けます。もしも、その石の隣に相手の石があるか、隣が盤外ならば、方向を変えて(例えば((x,y-1))同じように空点があるかどうか調べていきます。

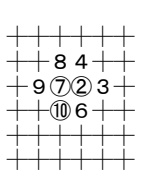
（例）白３子がつながっている場合において、空点があるかどうかを調べる順番

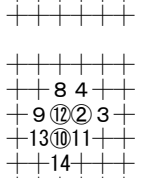

①の石から調べはじめる場合


右が同じ色なので、さらに右へと進み
③が空点なので、方向転換して上へ


上も空点なので、また方向転換して⑤へ
⑤は既に調査済み(①)なので、方向転換して下へ


②の石の四方を全て調べたので、①の位置へ戻り(⑦)


⑦から見て、残りの三方向を調べる


⑩の四方向を調べたら、全ての石の四方向を調べ終わったので終了

このように、同じ処理を繰り返すときは、再帰的アルゴリズムを使うと良いでしょう。

【再帰的アルゴリズム】

```
「その定義の中に、更にその定義されるべきものが簡単化されて含まれているとき」
それを再帰的という。

再帰の例（階乗計算）
    0！＝１
    K ＞ 0ならば、K！＝K × (K−1)！

プログラムで書くと、
    F( K: integer )
        if( K ＞ 0 ){
            return( K × F(K-1) )
        }else{
            return( 1 )
        }
```

空点があるかどうかを調べる処理を、再帰の形で書き表すと次のようになります。

```
( x, y )が空点ならば、ダメがある
          敵石ならば、ダメがない
          盤外ならば、ダメがない
          味方ならば、隣の座標を調べるために再帰
```

隣の座標は4つあること、同じ座標を何度も調べないようにすること、を考慮して、プログラムの形で書くと次のようになります。

```
/*---------------------------------------------*/
/* 座標(x,y)の石が囲まれているか調べる関数       */
/* 囲まれていない（ダメがある）ならばFALSEを返す */
/*---------------------------------------------*/

Function( x, y )
{
  /* (1) (x,y)が既に調べた点ならTRUEを返す                              */
  /*     まだ調べてない点なら調べたことを覚えておく                     */
  /*     覚えておかないと、同じ場所を何度も調べることになってしまうから */

  /* (2) (x,y)が空点ならばFALSEを返す */

  /* (3) (x,y)が同じ色の石ならばその石の四方を調べるために */
  /*     Functionを再帰呼び出し                           */
  /* (3.1)  rtn = Function( x+1, y );                     */
  /*        rtnがFALSEならばFALSEを返す                   */
  /* (3.2)  rtn = Function( x, y-1 );                     */
  /*        rtnがFALSEならばFALSEを返す                   */
  /* (3.3)  rtn = Function( x-1, y );                     */
  /*        rtnがFALSEならばFALSEを返す                   */
  /* (3.4)  rtn = Function( x, y+1 );                     */
  /*        rtnがFALSEならばFALSEを返す                   */

  /* (4) (x,y)が相手の色の石ならばTRUEを返す */

  /* (5) (x,y)が盤外ならばTRUEを返す */
}
```

次は石を取り除く関数です。囲まれているかどうかを調べるのと同じように再帰的アルゴリズムを使って書くことができます。

```
( x, y )が空点ならば、何もしないで終了
          敵石ならば、何もしないで終了
          盤外ならば、何もしないで終了
          味方ならば、( x, y )に空点を設定し、アゲハマを１つカウントし、
          隣の座標を調べるために再帰
```

なお、石を取り除く関数を呼ぶ前に、ダメがあるかどうかを調べる関数を呼んでおきましょう。ダメが無いことがわかったら、石を取り除く関数を呼びます。

合法手

合法手とは、ルールで認められた手のことです。ルールで認められていない手とは、次の3種類です。

1. **既に石があるところ**
2. **四方が相手の石に囲まれているところ**
3. **劫だてを打たなければいけないときの劫**

この3種類のどれかにあてはまるかどうかを調べます。

(1) 既に石があるところ

これは調べるのは簡単です。

```
if( board[y][x] != SPACE ){
  /* 空点じゃないので打てません */
}
```

(2) 四方が相手の石に囲まれているところ

これは実際に碁盤の上に置いてみて、その石が囲まれているかどうかを調べるのが簡単でしょう。石が囲まれているかどうかを調べる処理は前章で書いた関数を使います。しかし、囲まれていても、相手の石を取れるなら打っても良いので、打った石の隣に相手の石があるならば、その石が打った石と同じ色の石で囲まれているかどうかも調べなければいけません。

```
/* (1) 仮に石を置く                                    */
/*                                                     */
/* (2) その石は相手に囲まれているか調べる              */
/*                                                     */
/* (3) if( 囲まれている )                              */
/*         その石の隣の相手の石が取れるかどうか調べる  */
/*         if( 相手の石が取れる )                      */
/*             合法手                                  */
/*         else                                        */
/*             合法手ではない                          */
/*     else                                            */
/*         囲まれていないので合法手                    */
/*                                                     */
/* (4) 仮に置いた石をはがす                            */
```

(3) 劫だてを打たなければいけないときの劫

劫の場所に打ってよいかどうか調べるには、直前の相手の着手位置がわからないといけません。直前に相手が劫の場所に打ったのなら、自分はそこに打てないからです。そのためには、対局のときに1手目から順番にどこに打ったのかを覚えておきましょう。

劫かどうかを調べるにはどうすればよいのでしょう。実は、劫の形かどうかを調べなくても良いのです。それから、劫による着手禁止かどうかは、手を打ったときに求めて覚えておく方が簡単です。これから打とうとする座標が、覚えた座標と同じならば、着手できないというように処理します。

では、手を打ったときに劫による着手禁止の求め方です。

- 置いた石の隣に同じ色の石があれば劫ではない
- 置いた石により周囲の相手の石を取れるとき、取った石数が1ならば、抜き跡が劫による着手禁止点になる

実は、この二つの処理では、劫でないところも劫と判定してしまいます。しかし劫ではなくても、四方が相手の石に囲まれているので、置くことができません。ですから、劫だから着手禁止点と判定しても何の問題も起きないのです。

(A) 置いた石の隣に同じ色の石があるので劫ではない例

```
＋●○＋      ＋●○○＋
●○×○      ●○×●○
＋●●＋      ＋●○○＋
```

(B) 取った石数が１ではないので劫ではない例

```
＋＋●＋      ＋●●○＋
＋●○●      ●○○×○
●○×○      ＋●●○＋
＋●○＋
```

(C) 劫ではないのに劫と判定してしまう例
しかし抜き跡に白は打てないので問題ない

```
＋●○＋
●○×○
＋●＋＋
```

最も単純な思考ルーチン(ランダムに次の一手を求める)

碁盤プログラムができたら、人の代わりに思考プログラムを作ります。思考プログラムの最も単純なのは、ランダムに次の一手を求めるプログラムでしょう。(ランダムじゃ、思考しているとは言えないかも知れませんけど)

1. 乱数関数を使うことによりランダムに数を生成する
2. 数を19×19の碁盤の座標に割り当てる

(例) 生成された数を361で割った余り＝19×Y＋XとなるX,Yを求める
3. 割り当てた座標が着手禁止点かどうか調べる
4. 着手禁止点ならば、乱数関数を再び呼ぶ
   着手禁止点でなくなるまで呼び、着手禁止点でなければそれを次の一手の座標とする

人間プレーヤーの代わりにこの関数を呼べば、人対コンピュータの対局ができます。ただし、弱いだけじゃなく、着手禁止点がなくなるまで打ち続けるので、相手をしてると嫌になっちゃいます。でも、これからあなたが作るプログラムのテスト対局の相手にはなるでしょう。

碁盤プログラムのサンプル

```
/*----------------------------------------------------------------------*/
/* コンピュータ上で人間どうしが対局するプログラム                       */
/*----------------------------------------------------------------------*/

#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 */

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

/*----------------------------------------------------------------------*/
/* 構造体                                                               */
/*----------------------------------------------------------------------*/

/* 棋譜 */
typedef struct {
  int color;
  int x;
  int y;
} type_log;

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

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

/* 碁盤の初期化 */
static void InitializeBoard( void );

/* 碁盤を表示する */
static void DisplayBoard( void );

/* 着手位置の入力 */
static void ThinkMove( int color, int *x, int *y );

/* 画面から数値を入力 */
static void InputCoordinate( int color, int *x, int *y );

/* 合法手がどうか調べる */
static int  CheckLegal( int color, int x, int y );

/* 自殺手かどうか調べる */
static int  CheckSuicide( int color, int x, int y );

/* チェック用の碁盤をクリア */
static void ClearCheckBoard( void );

/* 相手に囲まれているか調べる */
static int  DoCheckRemoveStone( int color, int x, int y );

/* 勝敗の判定 */
static int  CountScore( void );

/* 碁盤に石を置く*/
static void SetStone( int color, int x, int y );

/* 囲まれた石を取り除く */
static int  RemoveStone( int color, int x, int y );
```

```
/* 囲まれた石を取り除く */
static int  DoRemoveStone( int color, int x, int y, int prisoner );

/* 棋譜に記録 */
static void RecordMove( int color, int x, int y );

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

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

/* 碁盤 */
static int board[BOARD_SIZE][BOARD_SIZE];

/* 手数 */
static int move;

/* アゲハマ */
static int black_prisoner;
static int white_prisoner;

/* 劫の位置 */
static int ko_x;
static int ko_y;

/* 劫が発生した手数 */
static int ko_num;

/* 棋譜(499手まで覚える) */
static type_log logData[500];

/* 合法手かどうか調べるのに使う */
static int checkBoard[BOARD_SIZE][BOARD_SIZE];

/*--------------------------------------------------------------------*/
/* メイン関数                                                         */
/*--------------------------------------------------------------------*/
int
main( int argc, char *argv[] )
{
  int xBlack, yBlack;  /* 黒の着手位置 */
  int xWhite, yWhite;  /* 白の着手位置 */
  int score;           /* 黒地－白地   */

  /* 碁盤の初期化 */
  InitializeBoard();

  printf( "Sample Program Start¥n" );

  xBlack = yBlack = 999;  /* 最初は0以外の数字を書いておく */
  xWhite = yWhite = 999;  /* 最初は0以外の数字を書いておく */

  /* アゲハマ */
  black_prisoner = 0;
  white_prisoner = 0;

  /* 手数 */
  move = 1;

  /* 終局するまでループ */
  while( 1 ){

    /*------*/
    /* 黒番 */
    /*------*/

    /* 碁盤の表示 */
    DisplayBoard();

    /* 黒の着手位置を入力 */
    ThinkMove( BLACK, &xBlack, &yBlack );

    /* 黒が投了なら白の勝ち */
    if( xBlack >= (BOARD_SIZE-1) || yBlack >= (BOARD_SIZE-1) ){
      printf( "Black Resign. White Win.¥n" );
      break;
    }

    /* 黒白ともにパスなら地を数えて勝者を表示 */
    if( ( xBlack < 1 || yBlack < 1 ) && ( xWhite < 1 || yWhite < 1 ) ){
      score = CountScore();
      if( score > 0 ){
            /* 黒が多ければ黒の勝ち */
```

```
            printf( "Black Win¥n" );
        }else if( score < 0 ){
            /* 黒が少なければ白の勝ち */
            printf( "White Win¥n" );
        }else{
            /* 同数ならば引分け */
            printf( "Draw¥n" );
        }
        break;
    }

    /* 座標が1～19なら碁盤に石を置く */
    SetStone( BLACK, xBlack, yBlack );

    /* 棋譜に記録 */
    RecordMove( BLACK, xBlack, yBlack );

    /* 手数が1増える */
    move++;

    /*------*/
    /* 白番 */
    /*------*/

    /* 碁盤の表示 */
    DisplayBoard();

    /* 白の着手位置を入力 */
    ThinkMove( WHITE, &xWhite, &yWhite );

    /* 白が投了なら黒の勝ち */
    if( xWhite >= (BOARD_SIZE-1) || yWhite >= (BOARD_SIZE-1) ){
        printf( "White Resign. Black Win.¥n" );
        break;
    }

    /* 黒白ともにパスなら地を数えて勝者を表示 */
    if( ( xBlack < 1 || yBlack < 1 ) && ( xWhite < 1 || yWhite < 1 ) ){
        score = CountScore();
        if( score > 0 ){
            /* 黒が多ければ黒の勝ち */
            printf( "Black Win¥n" );
        }else if( score < 0 ){
            /* 黒が少なければ白の勝ち */
            printf( "White Win¥n" );
        }else{
            /* 同数ならば引分け */
            printf( "Draw¥n" );
        }
        break;
    }

    /* 座標が1～19なら碁盤に石を置く */
    SetStone( WHITE, xWhite, yWhite );

    /* 棋譜に記録 */
    RecordMove( WHITE, xWhite, yWhite );

    /* 手数が1増える */
    move++;
  }

  printf( "Sample Program End¥n" );
}

/*---------------------------------------------------------------------*/
/* 碁盤の初期化                                                        */
/*---------------------------------------------------------------------*/
static void
InitializeBoard( void )
{
  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( void )
{
  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" );
  }
}

/*---------------------------------------------------------------------*/
/* 着手位置の入力                                                      */
/* 今は画面から入力するようになっていますが、ここをコンピュータが考 */
/* えて着手位置を返すようにすれば、コンピュータ対人間のプログラムに */
/* なります                                                            */
/*---------------------------------------------------------------------*/
static void
ThinkMove( int color, int *x, int *y )
{
  int inputX, inputY;

  while( 1 ){
    /* 着手位置の入力 */
    InputCoordinate( color, &inputX, &inputY );

    if( inputX > 0 && inputX < (BOARD_SIZE-1) &&
        inputY > 0 && inputY < (BOARD_SIZE-1) ){
      /* 座標が1～19ならば合法手がどうか調べる */
      if( CheckLegal( color, inputX, inputY ) == TRUE ){
            break;
      }
    }else{
      break;
    }
  }

  *x = inputX;
  *y = inputY;
}

/*---------------------------------------------------------------------*/
/* 画面から数値を入力                                                  */
/* X座標が1～19ならばY座標も入力する                                   */
/* X座標が1より小さければパスとみなし、Y座標は入力しない              */
/* X座標が19より大きければ投了とみなし、Y座標は入力しない              */
/*---------------------------------------------------------------------*/
static void
InputCoordinate( int color, int *x, int *y )
{
  int val;

  printf( "¥n" );
  if( color == BLACK ){
    printf( "Please Input Black Coordinates.¥n" );
  }else{
    printf( "Please Input White Coordinates.¥n" );
  }
  printf( " Pass -> 0, Quit -> 20¥n" );

  /* X座標の入力 */
  while( 1 ){
    printf( "InputX:" );
    fflush( stdout );
    val = scanf( "%d", x );
    if( val == 1 ){
      break;
    }
  }

  if( *x >= 1 && *x < (BOARD_SIZE-1) ){
    /* X座標が1～19なのでY座標の入力 */
    while( 1 ){
      printf( "InputY:" );
      fflush( stdout );
      val = scanf( "%d", y );
```

```
      if( val == 1 ){
            break;
      }
    }
  }else if( *x < 1 ){
    /* X座標が1より小さいのでパスとみなし、Y座標は入力しない */
    *y = 0;
  }else{
    /* X座標が19より大きいので投了とみなし、Y座標は入力しない */
    *y = 20;
  }
}

/*------------------------------------------------------------------------*/
/* 合法手かどうか調べる                                                   */
/*------------------------------------------------------------------------*/
static int
CheckLegal( int color, int x, int y )
{
  /* 空点じゃないと置けません */
  if( board[y][x] != SPACE ){
    return( FALSE );
  }

  /* 一手前に劫を取られていたら置けません */
  if( move > 1 ){
    if( ko_x == x && ko_y == y && ko_num == (move-1) ){
      return( FALSE );
    }
  }

  /* 自殺手なら置けません */
  if( CheckSuicide( color, x, y ) == TRUE ){
    return( FALSE );
  }

  /* 以上のチェックをすべてクリアできたので置けます */
  return( TRUE );
}

/*------------------------------------------------------------------------*/
/* 自殺手かどうか調べる                                                   */
/*------------------------------------------------------------------------*/
static int
CheckSuicide( int color, int x, int y )
{
  int rtnVal;
  int opponent;  /* 相手の色 */

  /* 仮に石を置く */
  board[y][x] = color;

  /* マークのクリア */
  ClearCheckBoard();

  /* その石は相手に囲まれているか調べる */
  rtnVal = DoCheckRemoveStone( color, x, y );

  /* 囲まれているならば自殺手の可能性あり */
  if( rtnVal == TRUE ){

    /* 相手の色を求める */
    opponent = reverseColor( color );

    /* その石を置いたことにより、隣の相手の石が取れるなら自殺手ではない */
    if( x > 1 ){
      /* 隣は相手？ */
      if( board[y][x-1] == opponent ){
            /* マークのクリア */
            ClearCheckBoard();
            /* 相手の石は囲まれているか？ */
        rtnVal = DoCheckRemoveStone( opponent, x-1, y );
            /* 相手の石を取れるので自殺手ではない */
        if( rtnVal == TRUE ){
              /* 盤を元に戻す */
              board[y][x] = SPACE;
          return( FALSE );
        }
      }
    }
    if( y > 1 ){
      /* 隣は相手？ */
      if( board[y-1][x] == opponent ){
            /* マークのクリア */
            ClearCheckBoard();
```

```
            /* 相手の石は囲まれているか？ */
        rtnVal = DoCheckRemoveStone( opponent, x, y-1 );
            /* 相手の石を取れるので自殺手ではない */
        if( rtnVal == TRUE ){
              /* 盤を元に戻す */
              board[y][x] = SPACE;
          return( FALSE );
        }
      }
    }
    if( x < (BOARD_SIZE-2) ){
      /* 隣は相手？ */
      if( board[y][x+1] == opponent ){
            /* マークのクリア */
            ClearCheckBoard();
            /* 相手の石は囲まれているか？ */
        rtnVal = DoCheckRemoveStone( opponent, x+1, y );
            /* 相手の石を取れるので自殺手ではない */
        if( rtnVal == TRUE ){
              /* 盤を元に戻す */
              board[y][x] = SPACE;
          return( FALSE );
        }
      }
    }
    if( y < (BOARD_SIZE-2) ){
      /* 隣は相手？ */
      if( board[y+1][x] == opponent ){
            /* マークのクリア */
            ClearCheckBoard();
            /* 相手の石は囲まれているか？ */
        rtnVal = DoCheckRemoveStone( opponent, x, y+1 );
            /* 相手の石を取れるので自殺手ではない */
        if( rtnVal == TRUE ){
              /* 盤を元に戻す */
              board[y][x] = SPACE;
          return( FALSE );
        }
      }
    }

    /* 盤を元に戻す */
    board[y][x] = SPACE;

    /* 相手の石を取れないなら自殺手 */
    return( TRUE );

  }else{

    /* 盤を元に戻す */
    board[y][x] = SPACE;

    /* 囲まれていないので自殺手ではない */
    return( FALSE );
  }
}

/*------------------------------------------------------------------*/
/* チェック用の碁盤をクリア                                         */
/*------------------------------------------------------------------*/
static void
ClearCheckBoard( void )
{
  int x, y;

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

/*------------------------------------------------------------------*/
/* 座標(x,y)にあるcolor石が相手に囲まれているか調べる               */
/*------------------------------------------------------------------*/
static int     /* 空点があればFALSEを返し、空点がなければTRUEを返す */
DoCheckRemoveStone( int color, int x, int y )
{
  int rtn;

  /* その場所は既に調べた点ならおしまい */
  if( checkBoard[y][x] == TRUE ){
    return( TRUE );
  }
```

```
  /* 調べたことをマークする */
  checkBoard[y][x] = TRUE;

  /* 何も置かれていないならばおしまい */
  if( board[y][x] == SPACE ){
    return( FALSE );
  }

  /* 同じ色の石ならばその石の隣も調べる */
  if( board[y][x] == color ){
    /* その石の左(x-1,y)を調べる */
    if( x > 1 ){
      rtn = DoCheckRemoveStone( color, x-1, y );
      if( rtn == FALSE ){
            return( FALSE );
      }
    }
    /* その石の上(x,y-1)を調べる */
    if( y > 1 ){
      rtn = DoCheckRemoveStone( color, x, y-1 );
      if( rtn == FALSE ){
            return( FALSE );
      }
    }
    /* その石の右(x+1,y)を調べる */
    if( x < (BOARD_SIZE-2) ){
      rtn = DoCheckRemoveStone( color, x+1, y );
      if( rtn == FALSE ){
            return( FALSE );
      }
    }
    /* その石の下(x,y+1)を調べる */
    if( y < (BOARD_SIZE-2) ){
      rtn = DoCheckRemoveStone( color, x, y+1 );
      if( rtn == FALSE ){
            return( FALSE );
      }
    }
  }

  /* 相手の色の石があった */
  return( TRUE );
}

/*------------------------------------------------------------------------*/
/* 勝敗の判定                                                             */
/* このプログラムでは地を数えていません                                   */
/* アゲハマの多い方を勝ちと判定しています                                 */
/*------------------------------------------------------------------------*/
static int
CountScore()
{
  int black_score;
  int white_score;

  /* 碁盤上の地を数えるべきところだが省略 */
  black_score = 0;
  white_score = 0;

  /* アゲハマを加える */
  black_score += black_prisoner;
  white_score += white_prisoner;

  /* 黒－白を返す */
  return( black_score - white_score );
}

/*------------------------------------------------------------------------*/
/* 碁盤に石を置く                                                         */
/*------------------------------------------------------------------------*/
static void
SetStone( int color, int x, int y )
{
  int prisonerN;    /* 取り除かれた石の数（上） */
  int prisonerE;    /* 取り除かれた石の数（右） */
  int prisonerS;    /* 取り除かれた石の数（下） */
  int prisonerW;    /* 取り除かれた石の数（左） */
  int prisonerAll;  /* 取り除かれた石の総数 */
  int koFlag;       /* 劫かどうか */

  /* 座標(x,y)に石を置く */
  board[y][x] = color;

  /* 置いた石の隣に同じ色の石はあるか？ */
  if( board[y+1][x] != color &&
```

```
      board[y-1][x] != color &&
      board[y][x+1] != color &&
      board[y][x-1] != color ){
    /* 同じ色の石がないならば劫かもしれない */
    koFlag = TRUE;
  }else{
    /* 同じ色の石があるならば劫ではない */
    koFlag = FALSE;
  }

  /* 取り除かれた石の数 */
  prisonerN = prisonerE = prisonerS = prisonerW = 0;

  /* 置いた石の周囲の相手の石が死んでいれば碁盤から取り除く */
  if( y > 1 ){
    prisonerN = RemoveStone( color, x, y-1 );
  }
  if( x > 1 ){
    prisonerW = RemoveStone( color, x-1, y );
  }
  if( y < (BOARD_SIZE-2) ){
    prisonerS = RemoveStone( color, x, y+1 );
  }
  if( x < (BOARD_SIZE-2) ){
    prisonerE = RemoveStone( color, x+1, y );
  }

  /* 取り除かれた石の総数 */
  prisonerAll = prisonerN + prisonerE + prisonerS + prisonerW;

  /* 置いた石の隣に同じ色の石がなく、取り除かれた石も１つならば劫 */
  if( koFlag == TRUE && prisonerAll == 1 ){
    /* 劫の発生した手数を覚える */
    ko_num = move;
    /* 劫の座標を覚える */
    if( prisonerE == 1 ){
      /* 取り除かれた石が右 */
      ko_x = x+1;
      ko_y = y;
    }else if( prisonerS == 1 ){
      /* 取り除かれた石が下 */
      ko_x = x;
      ko_y = y+1;
    }else if( prisonerW == 1 ){
      /* 取り除かれた石が左 */
      ko_x = x-1;
      ko_y = y;
    }else if( prisonerN == 1 ){
      /* 取り除かれた石が上 */
      ko_x = x;
      ko_y = y-1;
    }
  }

  /* アゲハマの更新 */
  if( prisonerAll > 0 ){
    if( color == BLACK ){
      black_prisoner += prisonerAll;
    }else if( color == WHITE ){
      white_prisoner += prisonerAll;
    }
  }
}

/*------------------------------------------------------------------*/
/* 座標(x,y)の石が死んでいれば碁盤から取り除く                      */
/*------------------------------------------------------------------*/
static int              /* 碁盤から取り除かれた石数 */
RemoveStone( int color, int x, int y )
{
  int prisoner;  /* 取り除かれた石数 */

  /* 置いた石と同じ色なら取らない */
  if( board[y][x] == color ){
    return( 0 );
  }

  /* 空点なら取らない */
  if( board[y][x] == SPACE ){
    return( 0 );
  }

  /* マークのクリア */
  ClearCheckBoard();
```

```
  /* 囲まれているなら取る */
  if( DoCheckRemoveStone( board[y][x], x, y ) == TRUE ){
    prisoner = DoRemoveStone( board[y][x], x, y, 0 );
    return( prisoner );
  }

  return( 0 );
}

/*--------------------------------------------------------------------*/
/* 座標(x,y)のcolor石を碁盤から取り除き、取った石の数を返す           */
/*--------------------------------------------------------------------*/
static int   /* アゲハマ */
DoRemoveStone( int color, int x, int y, int prisoner )
{
  /* 取り除かれる石と同じ色ならば石を取る */
  if( board[y][x] == color ){

    /* 取った石の数を１つ増やす */
    prisoner++;

    /* その座標に空点を置く */
    board[y][x] = SPACE;

    /* 左を調べる */
    if( x > 1 ){
      prisoner = DoRemoveStone( color, x-1, y, prisoner );
    }
    /* 上を調べる */
    if( y > 1 ){
      prisoner = DoRemoveStone( color, x, y-1, prisoner );
    }
    /* 右を調べる */
    if( x < (BOARD_SIZE-2) ){
      prisoner = DoRemoveStone( color, x+1, y, prisoner );
    }
    /* 下を調べる */
    if( y < (BOARD_SIZE-2) ){
      prisoner = DoRemoveStone( color, x, y+1, prisoner );
    }
  }

  /* 取った石の数を返す */
  return( prisoner );
}

/*--------------------------------------------------------------------*/
/* 棋譜に記録                                                         */
/*--------------------------------------------------------------------*/
static void
RecordMove( int color, int x, int y )
{
  if( move < 500 ){
    logData[move].color = color;
    logData[move].x     = x;
    logData[move].y     = y;
  }
}

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

第3部 地の認識

地の認識

「地」とは何でしょうか？

「地」、「勢力」、「模様」、「ダメ場」。関連する言葉がたくさんありますが、それぞれの定義は何でしょうか？

ある盤面で、人によって地（勢力／模様／ダメ場）の数え方が違うことはありませんか？

そうです。地とはとても曖昧な概念なのです。

仮に「地」を定義できたとしても、序盤や中盤で地を正確に認識できるでしょうか？（終盤ならば、全ての石の生死がはっきりしていて、境界も決まっているので、簡単に認識できるでしょうけど）

「地」の定義、そして「地」を正しく認識する方法については、囲碁プログラムの作者がいろいろな試行錯誤を繰り返しながら、その人なりのアルゴリズムを開発しています。そのアルゴリズムが公開されていることはほとんどありません。もし公開されても、人によって「地」の定義が違うので、あなたにとっては違和感を感じるかもしれません。そこで、ここで幾つか実験をしてみましょう。

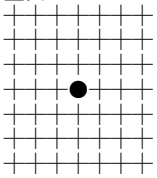
地の認識の実験(その1)

どこが黒地でどこが白地なのかを認識する実験をします。

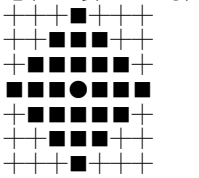
まず最初に、各石から距離3以内(二間トビまたはケイマまで)を、その石の色の地と定義してみます。

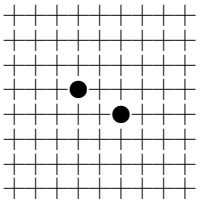
（例）

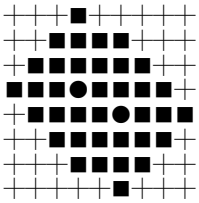
盤面



地（■で表している）

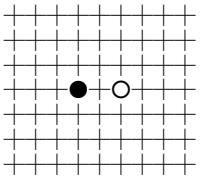


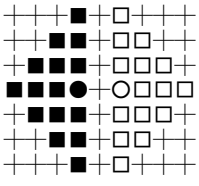




さらに、相手の石が近くにある場合は、勢力が打ち消しあって、どちらの地でもないと定義してみます。

（例）



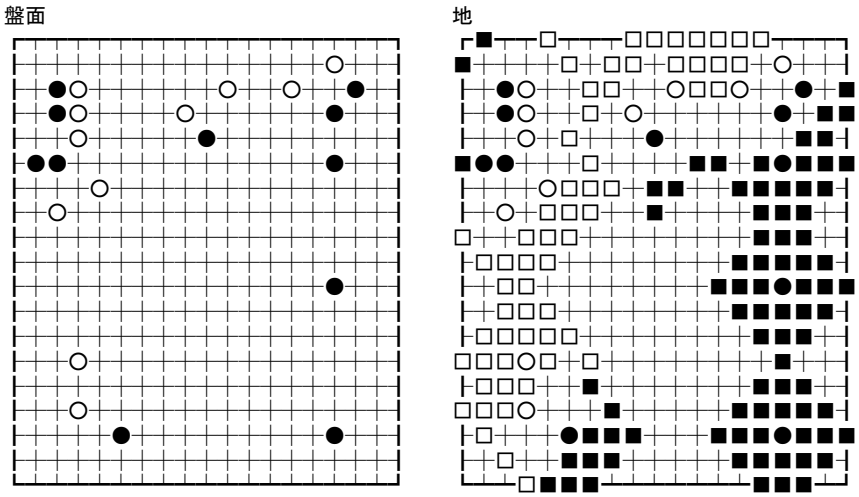


これをプログラムで書くと、次のようになります。

```
/*--------------------------------------------------------------------*/
/* 盤面の評価                                                         */
/*--------------------------------------------------------------------*/
static int
EvalBoard( int board[][BOARD_SIZE] )
{
  int blackBoard[BOARD_SIZE][BOARD_SIZE];  /* 黒石の影響範囲 */
  int whiteBoard[BOARD_SIZE][BOARD_SIZE];  /* 白石の影響範囲 */
  int totalBoard[BOARD_SIZE][BOARD_SIZE];  /* 地 */
  int x, y, value;

  /* 初期化 */
  for( y=1; y < (BOARD_SIZE-1); y++ ){
    for( x=1; x < (BOARD_SIZE-1); x++ ){
      blackBoard[y][x] = FALSE;
      whiteBoard[y][x] = FALSE;
      totalBoard[y][x] = SPACE;
    }
  }

  /* 距離3以内なら地とみなす */
  for( y=1; y < (BOARD_SIZE-1); y++ ){
    for( x=1; x < (BOARD_SIZE-1); x++ ){
      if( board[y][x] == BLACK ){
        SetValue( blackBoard, x-3, y   );
        SetValue( blackBoard, x-2, y-1 );
        SetValue( blackBoard, x-2, y   );
        SetValue( blackBoard, x-2, y+1 );
        SetValue( blackBoard, x-1, y-2 );
        SetValue( blackBoard, x-1, y-1 );
        SetValue( blackBoard, x-1, y   );
        SetValue( blackBoard, x-1, y+1 );
        SetValue( blackBoard, x-1, y+2 );
        SetValue( blackBoard, x,   y-3 );
        SetValue( blackBoard, x,   y-2 );
        SetValue( blackBoard, x,   y-1 );
        SetValue( blackBoard, x,   y   );
        SetValue( blackBoard, x,   y+1 );
        SetValue( blackBoard, x,   y+2 );
        SetValue( blackBoard, x,   y+3 );
        SetValue( blackBoard, x+1, y-2 );
        SetValue( blackBoard, x+1, y-1 );
        SetValue( blackBoard, x+1, y   );
        SetValue( blackBoard, x+1, y+1 );
        SetValue( blackBoard, x+1, y+2 );
        SetValue( blackBoard, x+2, y-1 );
        SetValue( blackBoard, x+2, y   );
        SetValue( blackBoard, x+2, y+1 );
        SetValue( blackBoard, x+3, y   );
      }else if( board[y][x] == WHITE ){
        SetValue( whiteBoard, x-3, y   );
        SetValue( whiteBoard, x-2, y-1 );
        SetValue( whiteBoard, x-2, y   );
        SetValue( whiteBoard, x-2, y+1 );
        SetValue( whiteBoard, x-1, y-2 );
        SetValue( whiteBoard, x-1, y-1 );
        SetValue( whiteBoard, x-1, y   );
        SetValue( whiteBoard, x-1, y+1 );
        SetValue( whiteBoard, x-1, y+2 );
        SetValue( whiteBoard, x,   y-3 );
        SetValue( whiteBoard, x,   y-2 );
        SetValue( whiteBoard, x,   y-1 );
        SetValue( whiteBoard, x,   y   );
        SetValue( whiteBoard, x,   y+1 );
        SetValue( whiteBoard, x,   y+2 );
        SetValue( whiteBoard, x,   y+3 );
        SetValue( whiteBoard, x+1, y-2 );
        SetValue( whiteBoard, x+1, y-1 );
        SetValue( whiteBoard, x+1, y   );
        SetValue( whiteBoard, x+1, y+1 );
        SetValue( whiteBoard, x+1, y+2 );
        SetValue( whiteBoard, x+2, y-1 );
        SetValue( whiteBoard, x+2, y   );
        SetValue( whiteBoard, x+2, y+1 );
        SetValue( whiteBoard, x+3, y   );
      }
    }
  }

  value = 0;
  for( y=1; y < (BOARD_SIZE-1); y++ ){
    for( x=1; x < (BOARD_SIZE-1); x++ ){
      if( board[y][x] == BLACK ){
```

```
        totalBoard[y][x] = BLACK;
      }else if( board[y][x] == WHITE ){
        totalBoard[y][x] = WHITE;
      }else{
        if( blackBoard[y][x] == TRUE ){
          if( whiteBoard[y][x] == TRUE ){
            /* 黒地と白地が同じ点ならば相殺される */
            continue;
          }else{
            /* 黒地としてカウント */
            totalBoard[y][x] = BLACK;
              value++;
          }
        }else{
          if( whiteBoard[y][x] == TRUE ){
            /* 白地としてカウント */
            totalBoard[y][x] = WHITE;
            value--;
          }
        }
      }
    }
  }

  return( value );
}

/*------------------------------------------------------------------*/
/* 地の設定                                                         */
/*------------------------------------------------------------------*/
static void
SetValue( int board[][BOARD_SIZE], int x, int y )
{
  if( x > 0 && x < (BOARD_SIZE-1) && y > 0 && y < (BOARD_SIZE-1) ){
    board[y][x] = TRUE;
  }
}
```

このプログラムで次の盤面を認識させてみると、次のような結果になりました。



地の認識の実験(その2)

実験の結果は、あなたが思う地と一致していますか？特に左上隅がほとんど黒地と認識されていないのは変ですよね。まだ認識の方法が稚拙なようですので、改良しましょう。

こんどは、各石から距離3以内(二間トビまたはケイマまで)を、その石の影響範囲と考えます。そして、各点において、その点に対して影響力の強い方 (その点に影響を及ぼす黒石の数と白石の数の多い方)の色の地と認識します。

これをプログラムで書くと、次のようになります。

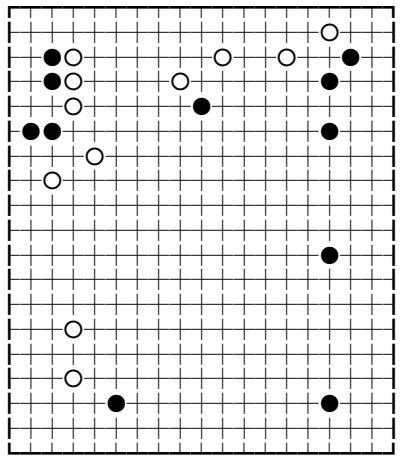
```
/*------------------------------------------------------------------*/
/* 盤面の評価                                                       */
/*------------------------------------------------------------------*/
static int
EvalBoard( int board[][BOARD_SIZE] )
{
```

```
int blackBoard[BOARD_SIZE][BOARD_SIZE];  /* 黒石の影響範囲 */
int whiteBoard[BOARD_SIZE][BOARD_SIZE];  /* 白石の影響範囲 */
int totalBoard[BOARD_SIZE][BOARD_SIZE];  /* 地 */
int x, y, value;

/* 初期化 */
for( y=1; y < (BOARD_SIZE-1); y++ ){
  for( x=1; x < (BOARD_SIZE-1); x++ ){
    blackBoard[y][x] = 0;
    whiteBoard[y][x] = 0;
    totalBoard[y][x] = SPACE;
  }
}

/* 距離3以内なら地とみなす */
for( y=1; y < (BOARD_SIZE-1); y++ ){
  for( x=1; x < (BOARD_SIZE-1); x++ ){
    if( board[y][x] == BLACK ){
      SetValue( blackBoard, x-3, y   );
      SetValue( blackBoard, x-2, y-1 );
      SetValue( blackBoard, x-2, y   );
      SetValue( blackBoard, x-2, y+1 );
      SetValue( blackBoard, x-1, y-2 );
      SetValue( blackBoard, x-1, y-1 );
      SetValue( blackBoard, x-1, y   );
      SetValue( blackBoard, x-1, y+1 );
      SetValue( blackBoard, x-1, y+2 );
      SetValue( blackBoard, x,   y-3 );
      SetValue( blackBoard, x,   y-2 );
      SetValue( blackBoard, x,   y-1 );
      SetValue( blackBoard, x,   y   );
      SetValue( blackBoard, x,   y+1 );
      SetValue( blackBoard, x,   y+2 );
      SetValue( blackBoard, x,   y+3 );
      SetValue( blackBoard, x+1, y-2 );
      SetValue( blackBoard, x+1, y-1 );
      SetValue( blackBoard, x+1, y   );
      SetValue( blackBoard, x+1, y+1 );
      SetValue( blackBoard, x+1, y+2 );
      SetValue( blackBoard, x+2, y-1 );
      SetValue( blackBoard, x+2, y   );
      SetValue( blackBoard, x+2, y+1 );
      SetValue( blackBoard, x+3, y   );
    }else if( board[y][x] == WHITE ){
      SetValue( whiteBoard, x-3, y   );
      SetValue( whiteBoard, x-2, y-1 );
      SetValue( whiteBoard, x-2, y   );
      SetValue( whiteBoard, x-2, y+1 );
      SetValue( whiteBoard, x-1, y-2 );
      SetValue( whiteBoard, x-1, y-1 );
      SetValue( whiteBoard, x-1, y   );
      SetValue( whiteBoard, x-1, y+1 );
      SetValue( whiteBoard, x-1, y+2 );
      SetValue( whiteBoard, x,   y-3 );
      SetValue( whiteBoard, x,   y-2 );
      SetValue( whiteBoard, x,   y-1 );
      SetValue( whiteBoard, x,   y   );
      SetValue( whiteBoard, x,   y+1 );
      SetValue( whiteBoard, x,   y+2 );
      SetValue( whiteBoard, x,   y+3 );
      SetValue( whiteBoard, x+1, y-2 );
      SetValue( whiteBoard, x+1, y-1 );
      SetValue( whiteBoard, x+1, y   );
      SetValue( whiteBoard, x+1, y+1 );
      SetValue( whiteBoard, x+1, y+2 );
      SetValue( whiteBoard, x+2, y-1 );
      SetValue( whiteBoard, x+2, y   );
      SetValue( whiteBoard, x+2, y+1 );
      SetValue( whiteBoard, x+3, y   );
    }
  }
}

value = 0;
for( y=1; y < (BOARD_SIZE-1); y++ ){
  for( x=1; x < (BOARD_SIZE-1); x++ ){
    if( board[y][x] == BLACK ){
      totalBoard[y][x] = BLACK;
    }else if( board[y][x] == WHITE ){
      totalBoard[y][x] = WHITE;
    }else{
      if( blackBoard[y][x] > whiteBoard[y][x] ){
        /* 黒地としてカウント */
        totalBoard[y][x] = BLACK;
        value++;
```

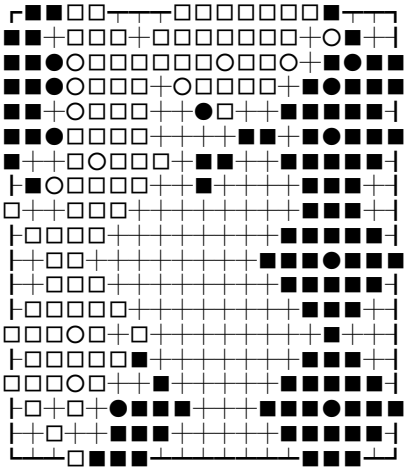
```
        }else if( blackBoard[y][x] < whiteBoard[y][x] ){
          /* 白地としてカウント */
          totalBoard[y][x] = WHITE;
          value--;
        }
      }
    }
  }

  return( value );
}

/*------------------------------------------------------------------*/
/* 地の設定                                                         */
/*------------------------------------------------------------------*/
static void
SetValue( int board[][BOARD_SIZE], int x, int y )
{
  if( x > 0 && x < (BOARD_SIZE-1) && y > 0 && y < (BOARD_SIZE-1) ){
    board[y][x] += 1;
  }
}
```

この改良をしたプログラムでの認識です。

盤面



地



左上隅は良くなったと思いますが、それでもまだまだ認識結果に不満が残りませんか。

- 相手の石の向こう側に、自分の石の影響が及ぶのは変である。
- 石から遠く離れていても、囲まれているなら地とするべきである。
- 石からの影響力が、隣と二間とで同じなのはおかしい。遠いと影響力も低いはず。
- 弱い石や死んでいる石からも、影響があるのは変である。

他にも、認識結果がおかしくなる理由はあるはずです。
ここから先の改良は、みなさんが行なって下さい。前にも書きましたが、正しい認識の方法はありません。みなさんが自由に考えて作ってください。

地の認識結果を評価値としたプログラム

地の認識ができたら、それを評価値としたプログラムを作ることができます。

- 何らかの方法で候補手を生成する
  (全ての空点を候補手としても良い)
- 着手禁止点のチェックをする
  これで合法手だけが残る
- 地の認識をし、形勢を求める
- 全ての候補手の評価値を比較する
- 形勢が最も良くなる手を、次の一手とする
  1目も良くならないときはパスする
- 両者がパスしたら終局する

```
/* 現在の盤面の評価値を求める */
initValue = EvalBoard( 現在の盤面 );
```

```
/* すべての候補手を評価する */
for( すべての候補手 ){

  /* (1) その手は合法手でないならば次の候補手の評価へ */
  if( 合法手でない ){
    continue;
  }

  /* (2) その手を評価する */
  value = EvalBoard( その手を仮に打った盤面 );

  /* (3) 評価が最高のものを残す */
  if( maxValue < value ){
    maxValue = value;
    move_x = x;
    move_y = y;
  }
}

/* 現在の盤面の評価値と候補手を仮に打った盤面の評価値とを比較 */
if( maxValue > initValue ){
  /* 候補手を打った方が評価値が高いのならば( move_x, move_y )が次の一手 */
}else{
  /* 候補手を打つと現在よりも悪くなるならばパスが次の一手 */
}
```

第2部で作った「ランダムに打つプログラム」よりも、かなり囲碁らしく、強いプログラムができたはずです。

地の認識結果を評価値としたプログラムの実験

前章で作った関数を使って、候補手の評価値を見てみましょう。地の認識の実験で使った盤面で、黒番とします。評価値は、地の数です。

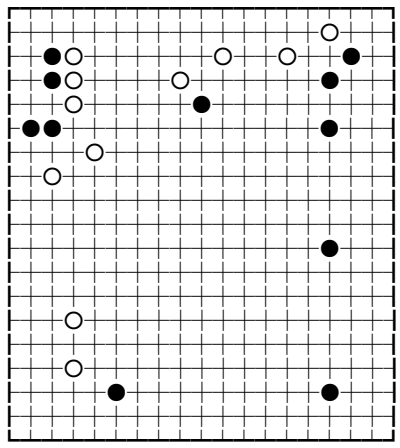
実験結果は次の通りです。数字は、

```
（その点に候補手を打ったときの地 － 現在の地）
```

を表します。

この候補手の評価の実験では、中央付近に打つ手が最善(24目の手)という結果が出ました。

盤面



評価値（地の増分）

```
 2  4  6 10 12 12 11 12 13 13 10  7  7  8  7  6  5  4  4
 2  4  8 10 12 13 15 16 16 13 13 12 11  8  5 ○  6  5  5
 1  4 ●  ○ 11 14 16 18 16 14 ○ 16 12 ○  3  4 ●  6  4
 1  2 ●  ○ 12 14 17 15 ○ 16 16 13 11  7  5 ●  4  3  4
 2  2  4 ○ 11 15 15 16 17 ● 15 13 10  8  6  4  0  3  3
 4 ●  ●  8 10 13 17 18 18 17 16 14 13 10  5 ●  3  4  4
 5  7  8 10 ○ 15 18 20 19 19 17 16 14  9  5  4  3  5  6
 8  9 ○ 14 15 17 19 20 20 20 20 17 13 10  7  5  5  5  6
11 13 15 16 18 19 20 21 22 21 21 18 15 10  6  5  5  5  6
13 18 19 18 19 20 22 24 23 22 20 20 16 10  4  3  3  6  7
15 19 20 20 19 22 24 24 24 22 22 19 16 11  5 ●  4  7  7
15 17 20 19 21 22 23 24 24 24 22 20 16 12  5  5  4  8  7
12 16 16 19 20 21 21 22 24 24 24 21 18 11  9  7  8  7  9
12 13 16 ○ 18 17 20 20 21 24 24 24 17 13  9  9  8  9  8
10 15 16 16 14 15 16 18 20 21 24 21 18 11  9  7  8  7  9
12 15 18 ○ 13 12 13 16 18 21 20 20 16 12  5  5  4  8  7
12 16 15 15 12 ● 12 13 16 17 19 17 15 10  5 ●  4  6  6
12 14 15 12 11 11  8 10 12 16 16 16 12  8  3  3  2  4  4
 9 11 11 11  9  7  6  5  9 12 15 12  9  5  4  3  3  2  3
```

地の認識結果を評価値として次の一手を求めるプログラム

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

++++++++++++++++++
+++++++++++++++0+++
+++*0++++++0++0+++*+++
+++*0++++0++++++*+++
+++0+++++*+++++++++
+**++++++++++++*+++
```

```
++++O++++++++++++
++O+++++++++++++++
++++++++++++++++++
++++++++++++++++++
+++++++++++++++*+++
++++++++++++++++++
++++++++++++++++++
+++O+++++++++++++++
++++++++++++++++++
+++O+++++++++++++++
+++++*+++++++++*+++
++++++++++++++++++
++++++++++++++++++
```

```
/*---------------------------------------------------------------------*/
/* ファイルから盤面図を読み込み、黒番の次の一手を返すプログラム      */
/*---------------------------------------------------------------------*/

#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 */

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

/*---------------------------------------------------------------------*/
/* 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] );

/* 次の一手を求める */
static void ThinkMove( int board[][BOARD_SIZE], int color, int *x, int *y );

/* 盤面の評価 */
static int  EvalBoard( int board[][BOARD_SIZE] );

/* 地の設定 */
static void SetValue( int board[][BOARD_SIZE], int x, int y );

/* 合法手がどうか調べる */
static int  CheckLegal( int board[][BOARD_SIZE], int color, int x, int y );

/* 自殺手かどうか調べる */
static int  CheckSuicide( int board[][BOARD_SIZE], int color, int x, int y );

/* チェック用の碁盤をクリア */
static void ClearCheckBoard( void );

/* 相手に囲まれているか調べる */
static int  DoCheckRemoveStone( int board[][BOARD_SIZE], int color, int x, int y );

/* 碁盤を複写 */
static void CopyBoard( int source[][BOARD_SIZE], int copied[][BOARD_SIZE] );

/* 碁盤に石を置く*/
static void SetStone( int board[][BOARD_SIZE], int color, int x, int y );

/* 囲まれた石を取り除く */
static int  RemoveStone( int board[][BOARD_SIZE], int color, int x, int y );

/* 囲まれた石を取り除く */
static int  DoRemoveStone( int board[][BOARD_SIZE], int color, int x, int y, int prisoner );
```

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

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

/* 合法手かどうか調べるのに使う */
static int checkBoard[BOARD_SIZE][BOARD_SIZE];

/*--------------------------------------------------------------------*/
/* メイン関数                                                         */
/*--------------------------------------------------------------------*/
int
main( int argc, char *argv[] )
{
  int board[BOARD_SIZE][BOARD_SIZE];  /* 碁盤 */
  int xBlack, yBlack;                 /* 黒の着手位置 */

  printf( "Sample Program Start¥n" );

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

  /* 黒の次の一手を求める */
  ThinkMove( board, BLACK, &xBlack, &yBlack );

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

  /* 結果の表示 */
  printf( "Computer -> (%2d,%2d)¥n", xBlack, yBlack);

  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             */
/* 劫による着手禁止点はないものとする                                 */
/*--------------------------------------------------------------------*/
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" );
  }
}

/*------------------------------------------------------------------------*/
/* 次の一手を求める                                                       */
/*------------------------------------------------------------------------*/
static void
ThinkMove( int board[][BOARD_SIZE], int color, int *xBlack, int *yBlack )
{
  int tmpBoard[BOARD_SIZE][BOARD_SIZE];  /* 複写用碁盤 */
  int evalBoard[BOARD_SIZE][BOARD_SIZE]; /* 評価値格納用碁盤 */
  int resultX, resultY;
  int x, y;
  int eval, init_eval, max_eval;

  /* 評価値格納用碁盤の初期化 */
  for( y=1; y < (BOARD_SIZE-1); y++ ){
    for( x=1; x < (BOARD_SIZE-1); x++ ){
      evalBoard[y][x] = 0;
    }
  }

  /* 盤面の評価 */
  init_eval = EvalBoard( board );

  /* 候補手を仮りに盤上に打って評価 */
  for( y=1; y < (BOARD_SIZE-1); y++ ){
    for( x=1; x < (BOARD_SIZE-1); x++ ){
      /* 着手禁止点なら次の点へ */
      if( CheckLegal( board, color, x, y ) == FALSE ){
        continue;
      }
      /* 碁盤を複写 */
      CopyBoard( board, tmpBoard );
      /* 候補手を盤上に打つ */
      SetStone( tmpBoard, color, x, y );
      /* 着手後の盤面の評価 */
      eval = EvalBoard( tmpBoard );
      /* 着手前よりどのくらい良くなったかを評価値格納用碁盤に記録 */
      evalBoard[y][x] = eval - init_eval;
    }
  }

  /* 評価値が最高のものを選ぶ */
  max_eval = 0;
  resultX = resultY = 0;
  for( y=1; y < (BOARD_SIZE-1); y++ ){
    for( x=1; x < (BOARD_SIZE-1); x++ ){
      if( max_eval < evalBoard[y][x] ){
        max_eval = evalBoard[y][x];
        resultX = x;
        resultY = y;
      }
    }
```

```
  }

  *xBlack = resultX;
  *yBlack = resultY;
}

/*------------------------------------------------------------------------*/
/* 盤面の評価                                                             */
/*------------------------------------------------------------------------*/
static int
EvalBoard( int board[][BOARD_SIZE] )
{
  int blackBoard[BOARD_SIZE][BOARD_SIZE];  /* 黒石の影響範囲 */
  int whiteBoard[BOARD_SIZE][BOARD_SIZE];  /* 白石の影響範囲 */
  int totalBoard[BOARD_SIZE][BOARD_SIZE];  /* 地 */
  int x, y, value;

  /* 初期化 */
  for( y=1; y < (BOARD_SIZE-1); y++ ){
    for( x=1; x < (BOARD_SIZE-1); x++ ){
      blackBoard[y][x] = 0;
      whiteBoard[y][x] = 0;
      totalBoard[y][x] = SPACE;
    }
  }

  /* 距離3以内なら地とみなす */
  for( y=1; y < (BOARD_SIZE-1); y++ ){
    for( x=1; x < (BOARD_SIZE-1); x++ ){
      if( board[y][x] == BLACK ){
        SetValue( blackBoard, x-3, y   );
        SetValue( blackBoard, x-2, y-1 );
        SetValue( blackBoard, x-2, y   );
        SetValue( blackBoard, x-2, y+1 );
        SetValue( blackBoard, x-1, y-2 );
        SetValue( blackBoard, x-1, y-1 );
        SetValue( blackBoard, x-1, y   );
        SetValue( blackBoard, x-1, y+1 );
        SetValue( blackBoard, x-1, y+2 );
        SetValue( blackBoard, x,   y-3 );
        SetValue( blackBoard, x,   y-2 );
        SetValue( blackBoard, x,   y-1 );
        SetValue( blackBoard, x,   y   );
        SetValue( blackBoard, x,   y+1 );
        SetValue( blackBoard, x,   y+2 );
        SetValue( blackBoard, x,   y+3 );
        SetValue( blackBoard, x+1, y-2 );
        SetValue( blackBoard, x+1, y-1 );
        SetValue( blackBoard, x+1, y   );
        SetValue( blackBoard, x+1, y+1 );
        SetValue( blackBoard, x+1, y+2 );
        SetValue( blackBoard, x+2, y-1 );
        SetValue( blackBoard, x+2, y   );
        SetValue( blackBoard, x+2, y+1 );
        SetValue( blackBoard, x+3, y   );
      }else if( board[y][x] == WHITE ){
        SetValue( whiteBoard, x-3, y   );
        SetValue( whiteBoard, x-2, y-1 );
        SetValue( whiteBoard, x-2, y   );
        SetValue( whiteBoard, x-2, y+1 );
        SetValue( whiteBoard, x-1, y-2 );
        SetValue( whiteBoard, x-1, y-1 );
        SetValue( whiteBoard, x-1, y   );
        SetValue( whiteBoard, x-1, y+1 );
        SetValue( whiteBoard, x-1, y+2 );
        SetValue( whiteBoard, x,   y-3 );
        SetValue( whiteBoard, x,   y-2 );
        SetValue( whiteBoard, x,   y-1 );
        SetValue( whiteBoard, x,   y   );
        SetValue( whiteBoard, x,   y+1 );
        SetValue( whiteBoard, x,   y+2 );
        SetValue( whiteBoard, x,   y+3 );
        SetValue( whiteBoard, x+1, y-2 );
        SetValue( whiteBoard, x+1, y-1 );
        SetValue( whiteBoard, x+1, y   );
        SetValue( whiteBoard, x+1, y+1 );
        SetValue( whiteBoard, x+1, y+2 );
        SetValue( whiteBoard, x+2, y-1 );
        SetValue( whiteBoard, x+2, y   );
        SetValue( whiteBoard, x+2, y+1 );
        SetValue( whiteBoard, x+3, y   );
      }
    }
  }
```

```
  value = 0;
  for( y=1; y < (BOARD_SIZE-1); y++ ){
    for( x=1; x < (BOARD_SIZE-1); x++ ){
      if( board[y][x] == BLACK ){
        totalBoard[y][x] = BLACK;
      }else if( board[y][x] == WHITE ){
        totalBoard[y][x] = WHITE;
      }else{
        if( blackBoard[y][x] > whiteBoard[y][x] ){
          /* 黒地としてカウント */
          totalBoard[y][x] = BLACK;
          value++;
        }else if( blackBoard[y][x] < whiteBoard[y][x] ){
          /* 白地としてカウント */
          totalBoard[y][x] = WHITE;
          value--;
        }
      }
    }
  }

  return( value );
}

/*--------------------------------------------------------------------*/
/* 地の設定                                                           */
/*--------------------------------------------------------------------*/
static void
SetValue( int board[][BOARD_SIZE], int x, int y )
{
  if( x > 0 && x < (BOARD_SIZE-1) && y > 0 && y < (BOARD_SIZE-1) ){
    board[y][x] += 1;
  }
}

/*--------------------------------------------------------------------*/
/* 合法手かどうか調べる                                               */
/* このサンプルプログラムでは劫のチェックは省略                       */
/*--------------------------------------------------------------------*/
static int
CheckLegal( int board[][BOARD_SIZE], int color, int x, int y )
{
  /* 空点じゃないと置けません */
  if( board[y][x] != SPACE ){
    return( FALSE );
  }

  /* 自殺手なら置けません */
  if( CheckSuicide( board, color, x, y ) == TRUE ){
    return( FALSE );
  }

  /* 以上のチェックをすべてクリアできたので置けます */
  return( TRUE );
}

/*--------------------------------------------------------------------*/
/* 自殺手かどうか調べる                                               */
/*--------------------------------------------------------------------*/
static int
CheckSuicide( int board[][BOARD_SIZE], int color, int x, int y )
{
  int rtnVal;
  int opponent;  /* 相手の色 */

  /* 仮に石を置く */
  board[y][x] = color;

  /* マークのクリア */
  ClearCheckBoard();

  /* その石は相手に囲まれているか調べる */
  rtnVal = DoCheckRemoveStone( board, color, x, y );

  /* 囲まれているならば自殺手の可能性あり */
  if( rtnVal == TRUE ){

    /* 相手の色を求める */
    opponent = reverseColor( color );

    /* その石を置いたことにより、隣の相手の石が取れるなら自殺手ではない */
    if( x > 1 ){
      /* 隣は相手？ */
      if( board[y][x-1] == opponent ){
        /* マークのクリア */
```

```
        ClearCheckBoard();
        /* 相手の石は囲まれているか？ */
        rtnVal = DoCheckRemoveStone( board, opponent, x-1, y );
        /* 相手の石を取れるので自殺手ではない */
        if( rtnVal == TRUE ){
          /* 盤を元に戻す */
          board[y][x] = SPACE;
          return( FALSE );
        }
      }
    }
    if( y > 1 ){
      /* 隣は相手？ */
      if( board[y-1][x] == opponent ){
        /* マークのクリア */
        ClearCheckBoard();
        /* 相手の石は囲まれているか？ */
        rtnVal = DoCheckRemoveStone( board, opponent, x, y-1 );
        /* 相手の石を取れるので自殺手ではない */
        if( rtnVal == TRUE ){
          /* 盤を元に戻す */
          board[y][x] = SPACE;
          return( FALSE );
        }
      }
    }
    if( x < (BOARD_SIZE-2) ){
      /* 隣は相手？ */
      if( board[y][x+1] == opponent ){
        /* マークのクリア */
        ClearCheckBoard();
        /* 相手の石は囲まれているか？ */
        rtnVal = DoCheckRemoveStone( board, opponent, x+1, y );
        /* 相手の石を取れるので自殺手ではない */
        if( rtnVal == TRUE ){
          /* 盤を元に戻す */
          board[y][x] = SPACE;
          return( FALSE );
        }
      }
    }
    if( y < (BOARD_SIZE-2) ){
      /* 隣は相手？ */
      if( board[y+1][x] == opponent ){
        /* マークのクリア */
        ClearCheckBoard();
        /* 相手の石は囲まれているか？ */
        rtnVal = DoCheckRemoveStone( board, opponent, x, y+1 );
        /* 相手の石を取れるので自殺手ではない */
        if( rtnVal == TRUE ){
          /* 盤を元に戻す */
          board[y][x] = SPACE;
          return( FALSE );
        }
      }
    }

    /* 盤を元に戻す */
    board[y][x] = SPACE;

    /* 相手の石を取れないなら自殺手 */
    return( TRUE );

  }else{

    /* 盤を元に戻す */
    board[y][x] = SPACE;

    /* 囲まれていないので自殺手ではない */
    return( FALSE );
  }
}

/*---------------------------------------------------------------*/
/* チェック用の碁盤をクリア                                      */
/*---------------------------------------------------------------*/
static void
ClearCheckBoard( void )
{
  int x, y;

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

```
  }
}

/*--------------------------------------------------------------------*/
/* 座標(x,y)にあるcolor石が相手に囲まれているか調べる                 */
/*--------------------------------------------------------------------*/
static int     /* 空点があればFALSEを返し、空点がなければTRUEを返す */
DoCheckRemoveStone( int board[][BOARD_SIZE], int color, int x, int y )
{
  int rtn;

  /* その場所は既に調べた点ならおしまい */
  if( checkBoard[y][x] == TRUE ){
    return( TRUE );
  }

  /* 調べたことをマークする */
  checkBoard[y][x] = TRUE;

  /* 何も置かれていないならばおしまい */
  if( board[y][x] == SPACE ){
    return( FALSE );
  }

  /* 同じ色の石ならばその石の隣も調べる */
  if( board[y][x] == color ){
    /* その石の左(x-1,y)を調べる */
    if( x > 1 ){
      rtn = DoCheckRemoveStone( board, color, x-1, y );
      if( rtn == FALSE ){
        return( FALSE );
      }
    }
    /* その石の上(x,y-1)を調べる */
    if( y > 1 ){
      rtn = DoCheckRemoveStone( board, color, x, y-1 );
      if( rtn == FALSE ){
        return( FALSE );
      }
    }
    /* その石の右(x+1,y)を調べる */
    if( x < (BOARD_SIZE-2) ){
      rtn = DoCheckRemoveStone( board, color, x+1, y );
      if( rtn == FALSE ){
        return( FALSE );
      }
    }
    /* その石の下(x,y+1)を調べる */
    if( y < (BOARD_SIZE-2) ){
      rtn = DoCheckRemoveStone( board, color, x, y+1 );
      if( rtn == FALSE ){
        return( FALSE );
      }
    }
  }

  /* 相手の色の石があった */
  return( TRUE );
}

/*--------------------------------------------------------------------*/
/* 碁盤を複写                                                         */
/*--------------------------------------------------------------------*/
static void
CopyBoard( int source[][BOARD_SIZE], int copied[][BOARD_SIZE] )
{
  int x, y;

  for( y=0; y < BOARD_SIZE; y++ ){
    for( x=0; x < BOARD_SIZE; x++ ){
      copied[y][x] = source[y][x];
    }
  }
}

/*--------------------------------------------------------------------*/
/* 碁盤に石を置く                                                     */
/* このサンプルプログラムでは取った石数は使わない                     */
/*--------------------------------------------------------------------*/
static void
SetStone( int board[][BOARD_SIZE], int color, int x, int y )
{
  int prisonerN;    /* 取り除かれた石の数（上） */
  int prisonerE;    /* 取り除かれた石の数（右） */
  int prisonerS;    /* 取り除かれた石の数（下） */
```

```
  int prisonerW;    /* 取り除かれた石の数（左） */
  int prisonerAll;  /* 取り除かれた石の総数 */

  /* 座標(x,y)に石を置く */
  board[y][x] = color;

  /* 取り除かれた石の数 */
  prisonerN = prisonerE = prisonerS = prisonerW = 0;

  /* 置いた石の周囲の相手の石が死んでいれば碁盤から取り除く */
  if( y > 1 ){
    prisonerN = RemoveStone( board, color, x, y-1 );
  }
  if( x > 1 ){
    prisonerE = RemoveStone( board, color, x-1, y );
  }
  if( y < (BOARD_SIZE-1) ){
    prisonerS = RemoveStone( board, color, x, y+1 );
  }
  if( x < (BOARD_SIZE-1) ){
    prisonerW = RemoveStone( board, color, x+1, y );
  }

  /* 取り除かれた石の総数 */
  prisonerAll = prisonerN + prisonerE + prisonerS + prisonerW;
}

/*---------------------------------------------------------------------*/
/* 座標(x,y)の石が死んでいれば碁盤から取り除く                         */
/*---------------------------------------------------------------------*/
static int    /* 碁盤から取り除かれた石数 */
RemoveStone( int board[][BOARD_SIZE], int color, int x, int y )
{
  int prisoner;  /* 取り除かれた石数 */

  /* 置いた石と同じ色なら取らない */
  if( board[y][x] == color ){
    return( 0 );
  }

  /* 空点なら取らない */
  if( board[y][x] == SPACE ){
    return( 0 );
  }

  /* マークのクリア */
  ClearCheckBoard();

  /* 囲まれているなら取る */
  if( DoCheckRemoveStone( board, board[y][x], x, y ) == TRUE ){
    prisoner = DoRemoveStone( board, board[y][x], x, y, 0 );
    return( prisoner );
  }

  return( 0 );
}

/*---------------------------------------------------------------------*/
/* 座標(x,y)のcolor石を碁盤から取り除き、取った石の数を返す            */
/*---------------------------------------------------------------------*/
static int   /* アゲハマ */
DoRemoveStone( int board[][BOARD_SIZE], int color, int x, int y, int prisoner )
{
  /* 取り除かれる石と同じ色ならば石を取る */
  if( board[y][x] == color ){

    /* 取った石の数を１つ増やす */
    prisoner++;

    /* その座標に空点を置く */
    board[y][x] = SPACE;

    /* 左を調べる */
    if( x > 1 ){
      prisoner = DoRemoveStone( board, color, x-1, y, prisoner );
    }
    /* 上を調べる */
    if( y > 1 ){
      prisoner = DoRemoveStone( board, color, x, y-1, prisoner );
    }
    /* 右を調べる */
    if( x < (BOARD_SIZE-2) ){
      prisoner = DoRemoveStone( board, color, x+1, y, prisoner );
    }
    /* 下を調べる */
```

```
    if( y < (BOARD_SIZE-2) ){
      prisoner = DoRemoveStone( board, color, x, y+1, prisoner );
    }
  }

  /* 取った石の数を返す */
  return( prisoner );
}

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

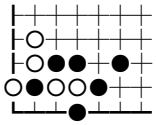
第4部 盤面認識のデータ構造

囲碁プログラミング講習会(5月16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。プログラムにも同じことをさせるためのデータ構造を考えましょう。



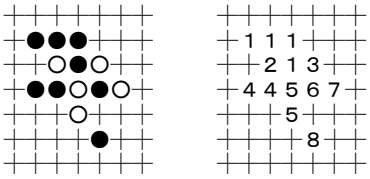
人間は、アタリになっている白の２子を
一つの単位として捉えている

最も基本的な単位(連)

石の生死を考えるとき、最も基本的な単位は何でしょう。それは、縦横につながった同じ色の石の集合です。別の言い方をすると、相手の石によって囲まれたら盤上から取り上げられる石の集合です。囲碁用語には、この「縦横につながった同じ色の石の集合」を表現する言葉はありませんが、多くの囲碁プログラムの作者は「連」と呼んでいるので、ここでも連と呼ぶことにします(海外では英語でstring(ストリング)と呼ばれることが多い)。

以下にその例を描きます。左は盤面で、右は連ごとに同じ番号を付けたものです。この盤面には8つの連が存在していることがわかります。

盤面　　連の番号



連のデータ構造の例

それぞれの連について、どんな情報を認識するとよいのでしょうか？残念ながら、その答はわかりません。ここでは、多くの囲碁プログラムに共通して認識されている情報を挙げておきます。

- 連の位置(構成要素となる各石の座標)
- 色
- サイズ(いくつの石がつながっているか)
- ダメ数
- 生死の状態
- その連に隣接する相手の連

生死の状態は、生き/死に/中立(自分から先に打てば生きだが相手から先に打つと死に)のどれかです。もしも中立ならば、どこに打てば生きてどこに打てば死ぬかという情報も一緒に覚えておいた方が良いでしょう。そうすれば、その情報から生きるための手/殺すための手を、候補手として生成できるからです。

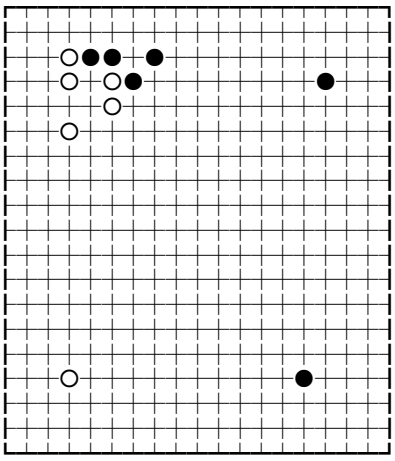
連のデータ構造のプログラム例

それぞれの連に関する情報を蓄えるために構造体を定義します。

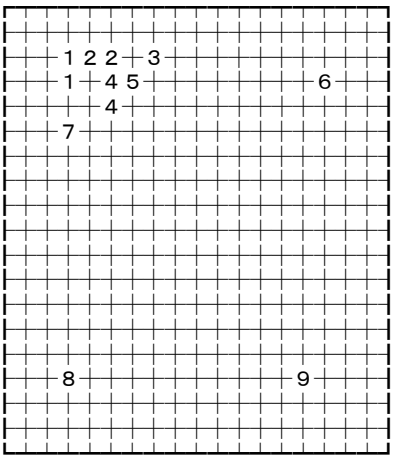
```
/* 碁盤の大きさ */
#define BOARD_SIZE 21

/* 色 */
#define BLACK 1            /* 黒   */
#define WHITE 2            /* 白   */

/* 生死 */
#define ALIVE     1        /* 生   */
#define DEAD      2        /* 死   */
#define NEUTRAL   3        /* 中立 */

/* 連 */
struct go_string {
  int id;                      /* 識別番号                    */
  int color;                   /* 色 (BLACK/WHITE)            */
  int size;                    /* サイズ                      */
  int dame;                    /* ダメ数                      */
  int status;                  /* 生死 (ALIVE/DEAD/NEUTRAL)   */
  int offense_x;               /* 取る手のＸ座標              */
  int offense_y;               /* 取る手のＹ座標              */
  int defense_x;               /* 逃げる手のＸ座標            */
  int defense_y;               /* 逃げる手のＹ座標            */
  struct go_string *neighbors; /* 隣接する連                  */
  struct go_string *next;      /* 次の連                      */
};

/* 連識別番号(ID) */
static int stringIdBoard[BOARD_SIZE][BOARD_SIZE];
```

連の構造体の要素に、連を構成する各石の座標を格納してもよいのですが、ここでは連の識別番号(ID)を格納することにします。そして、連の識別番号(ID)を格納するための2次元配列を、別に用意しました。
下の図において、IDが4番の連ならば、色はWHITE、サイズは2、ダメ数は4、生死はALIVE、隣接する連は2番と5番の連になります。座標は、識別番号格納用の配列から4番を探して、(6,4)と(6,5)であることがわかります。

盤面



連の識別番号(ID)



連の生死を調べる(探索)

連の生死は先読みで(探索)によって求めます。黒から先に打った場合と、白から先に打った場合の2通りの探索を行ないます。

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

```
/*-------------------------*/
/* ある黒の連の生死を調べる */
/*-------------------------*/

/* (1) 白から先に打ったときに、黒石を取れるか調べる */
```

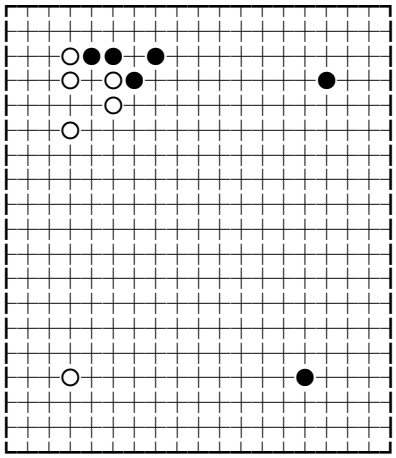
```
status = SearchStatus( 生死を調べる黒石, 次は白が打つ );

/* (2) 結果によって分類 */
if( status == ALIVE ){
  /* 黒石を取れないならば、その黒石は生き */
  return( ALIVE );
}else if( status == DEAD ){
  /* 黒石が取れるならば、黒から先に打ったときに、黒石が取られないか調べる */
  status = SearchStatus( 生死を調べる黒石, 次は黒が打つ );

  /* (3) 結果によって分類 */
  if( status == ALIVE ){
    /* 黒石が取られないならば、その黒石は中立 */
    return( NEUTRAL );
  }else if( status == DEAD ){
    /* 黒石が取られるならば、その黒石は死に */
    return( DEAD );
  }
}
```

以上のように、黒から先に打った場合と、白から先に打った場合の結果がわかれば、連の生死の判定は簡単にわかるのですが、実はここには大きな問題が隠れています。

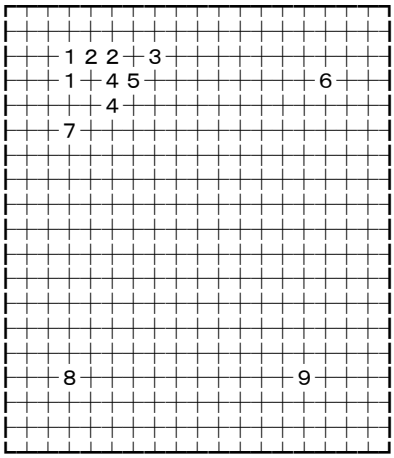
それは、「どのような状態の時、連を取れないと判断するのか？」です。

もしも、取るのはダメが無くなった時、取れないのは合法手がなくなった時、と定義したら、どうなるでしょう。以下の図の4番の連が取れないことがわかるまでにいったい何手先を読むことになるでしょう。きっと、10手や20手じゃわからないでしょうね。

盤面



連の識別番号(ID)



思考時間は無限ではありません。ですから、「多分この連は取れないだろう。」と判断して、読みを打ち切らなければいけません。人間も、ある程度読んだところで、「多分この連は取れないだろう。」と判断しています。
多くのプログラムは、「多分この連は取れないだろう。」の判断基準に、次のどれかを使用しています。

- 読みの深さが、ある値よりも大きくなった時
- 読んだ手数が、ある値よりも大きくなった時
- 連のダメ数が、ある値よりも大きくなった時

探索は、とても時間がかかります。石を一つ置いた盤面を作り、石が取れるかどうか(ダメがあるかどうか)を調べる処理を、毎回行うからです。しかも、盤上に連がたくさんあったら、連の数を掛けた数だけ盤面を作ることになります。もしもある連を取る手に5つの候補があり、5手先まで探索したら、5×5×5×5×5＝3125もの盤面を作ることになります。
実は、探索には効率良く行うためのアルゴリズムがたくさん研究されています。3125も盤面を作らなくても、生死がわかるのです。探索アルゴリズムについては、次回の講習会で説明します。

連以外のデータ構造

最も基本的な単位である連のデータ構造について説明しました。しかし、連だけでは、強いプログラムは作れません。多くの囲碁プログラムでは、連が幾つか集まってできる「群」という集合を扱っています。また、プログラムによっては、連と群の中間的なデータ構造を定義して扱っているものもあります。さらに、連や群とは全く異なるデータ構造を定義しているプログラムもあります。連以外のデータ構造についても、次回の講習会で説明します。

連の認識をするプログラム(連の識別番号付け)

このプログラムは、"figure"という名前のファイルを読んで、連の識別番号を表示するプログラムです。

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

```
/*---------------------------------------------------------------------*/
/* ファイルから盤面図を読み込み、連の識別番号を出力するプログラム    */
/*---------------------------------------------------------------------*/

#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 */

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

/*---------------------------------------------------------------------*/
/* 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] );

/* 連の認識 */
static void MakeString( int board[][BOARD_SIZE] );

/* 連の認識 */
static int  ClassificationString( int board[][BOARD_SIZE], int x, int y,
                                  int color, int number );

/* 連の表示 */
static void DisplayString( void );

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

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

/* 連識別番号 */
static int stringIdBoard[BOARD_SIZE][BOARD_SIZE];

/*---------------------------------------------------------------------*/
/* メイン関数                                                          */
/*---------------------------------------------------------------------*/
```

```
int
main( int argc, char *argv[] )
{
  int board[BOARD_SIZE][BOARD_SIZE];  /* 碁盤 */

  printf( "Sample Program Start¥n" );

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

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

  /* 連の認識 */
  MakeString( board );

  /* 連の表示 */
  DisplayString();

  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" );
  }
}

/*------------------------------------------------------------------*/
/* 連の認識                                                         */
/*------------------------------------------------------------------*/
static void
MakeString( int board[][BOARD_SIZE] )
{
  int x, y;
  int stringId;

  for( y=0; y < BOARD_SIZE; y++ ){
    for( x=0; x < BOARD_SIZE; x++ ){
      stringIdBoard[y][x] = 0;
    }
  }

  /* string id */
  stringId = 1;
  for( y=1; y<(BOARD_SIZE-1); y++ ){
    for( x=1; x<(BOARD_SIZE-1); x++ ){
      if( board[y][x] != SPACE ){
        if( ClassificationString( board, x, y, board[y][x], stringId ) == TRUE ){
              stringId++;
        }
      }
    }
  }

  /* 連情報を登録 */
}

/*------------------------------------------------------------------*/
/* 連を区別する                                                     */
/*------------------------------------------------------------------*/
static int
ClassificationString( int board[][BOARD_SIZE], int x, int y, int color, int number )
{
  /* 既にIDが付いているので対象外 */
  if( stringIdBoard[y][x] != 0 ){
    return( FALSE );
  }

  if( board[y][x] == color ){
    /* 同じ色なので同じ連である */
    stringIdBoard[y][x] = number;
    if( x > 1 ){
      ClassificationString( board, x-1, y, color, number );
    }
    if( x < (BOARD_SIZE-2) ){
      ClassificationString( board, x+1, y, color, number );
    }
    if( y > 1 ){
      ClassificationString( board, x, y-1, color, number );
    }
    if( y < (BOARD_SIZE-2) ){
      ClassificationString( board, x, y+1, color, number );
    }
    return( TRUE );
  }else{
    /* 同じ色ではないので終了 */
    return( FALSE );
  }
}

/*------------------------------------------------------------------*/
/* 連の表示                                                         */
/*------------------------------------------------------------------*/
```

```
static void
DisplayString( void )
{
  int x, y;

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

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