March 11, 1985

Smart Go Board:
Algorithms for the Tactical Calculator

Anders Kierulf
Informatik, ETH, CH-8092 Zürich

Contents

1. Overview

This report explains the algorithms for the tactical calculator integrated in the Smart Go Board described in [Kierulf / Nievergelt 85]. These algorithms solve all ladder problems, many loose ladders and some life and death problems. Prerequisite for reading is knowledge of go tactics and computer programming.

Of the various tactical problems in go (e.g. cuts, connections, endgame), captures were selected, because they represent the most basic tactic. For example, after a cut, one important question is whether the cutting stone can be captured. In the endgame you keep sente only if there is some threat behind your move, e.g. the threat to capture. A second reason for programming captures first is that the ladder, which is a method to capture a block, is the simplest tactic and provides a good starting point.

The tactical calculator is designed as a tool for human players and not as an automatic problem solver. The most important consequence is that exception handling is left to the user. For example, if the search tree explodes, it's the user's responsibility to limit it. A problem must also be well-defined: "Capture that block" is not enough, the computer must be told, how tightly it shall be captured, that is, how many liberties the block may at most have during search.

The search is programmed with a backtracking algorithm. This may lead to unlimited growth of the search tree, if the moves are not sorted well. If a poor choice is made somewhere, the likelyhood of another poor choice is increased: the tree explodes. Because pruning a branch of the search tree results in an exponential saving of the number of nodes examined, while cutting down the time spent at each node results in a linear saving, much effort should be invested in finding the best continuation from each node. For example, the algorithm should not rely on estimates for the number of liberties after a move: it should execute the move, count the number of liberties and undo the move to get the exact number. Though this takes more time at each node, that will be more than offset by the smaller number of nodes examined.

Liberties play a central role in the algorithms. A brief section about the data structure representing the board and the algorithm to find the liberties of a block is therefore included.

The term *hunter* is used to designate the player who tries to capture, and *prey* is the player (and the block of stones) trying to escape. *Hunter blocks* are blocks of stones adjacent to the prey.

2. Normal Ladders

Ladders are so simple, because the search tree is essentially unbranched. There is not much choice of where to play: the prey is always in atari, and in most cases the hunter has only one move to prevent the prey from obtaining two liberties.

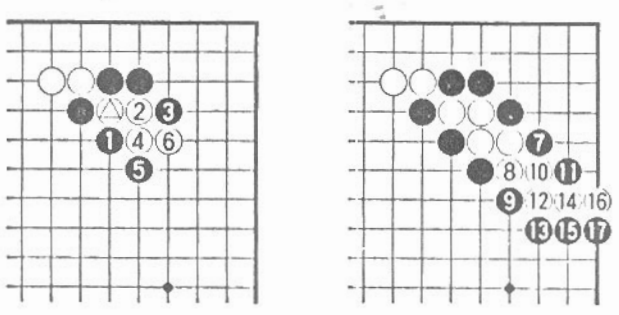


Figure 2-1: A simple ladder

Figure 2-1 shows an example. The sequence continues until the edge of the board is reached and White is captured. White must always be kept in atari - if White has time to play an atari against Black, the ladder will crumble, and White will escape.

Ladders have the following characteristics:

Prey to play:

- prey has one liberty: ladder goes on;
- prey has two or more liberties: ladder has failed, prey can escape.

Hunter to play:

- prey has one liberty: ladder was a success, prey can be killed with the next move;
- prey has two liberties: ladder goes on;
- prey has three or more liberties: prey can escape.

If the prey can escape from a normal ladder, it may still be possible to catch it - but there is no way to catch it without ever giving it more than two liberties.

The two players don't have much choice of where to play. The hunter has to play at one of the two liberties of the prey to put it in atari. The prey must somehow increase its number of liberties, either by playing at its liberty or by capturing an adjacent hunter block.

"Ladders should be the school that teaches you to read patiently, move by move" [Fundamentals, p. 14] - and it's also where our algorithm will learn to read. I haven't implemented any shortcut, such as sighting along the diagonal and conclude that if the ladder runs into a stone of the prey, it will escape, otherwise it will be captured. This simple rule doesn't apply in the interesting cases, such as Figure 2-2.

A recursive algorithm is used to solve the problem of whether a block can be caught or not, because after executing a move, the same problem is present, just with the block one stone larger. The recursion comes to an end when either the prey can be killed or has escaped.

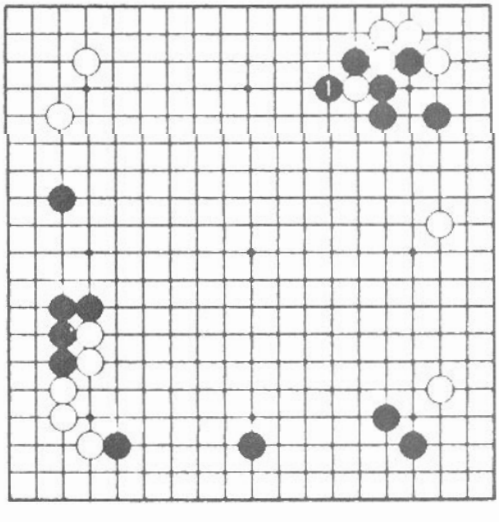


Figure 2-2: A ladder where rules of thumb fail - reading is required

2.1. A Simple Ladder Algorithm

An informal ladder algorithm can now be formulated in a somewhat computerized English.

Prey to play:

If the prey is not in atari, then it has escaped,
otherwise find all moves which increase the number of liberties of the prey.
For each of these moves, do the following:
- execute the move
- see if the prey can now be caught in a ladder
- undo the move.

If there was a move which led to escape, then the prey can escape, else it will be killed.

Hunter to play:

If the prey is in atari, then it can be killed,
if the prey has three or more liberties, then it has escaped,
otherwise find all moves which take away a liberty of the prey.
For each of these moves, do the following:
- execute the move
- see if the prey can now be caught in a ladder
- undo the move.

If there was a move which led to capture, then the prey will be killed, otherwise it can escape.

To be more precise, I present the algorithms in a Modula-2-like language [Wirth 82]. I define the following types, variables and procedures:

```
TYPE
  PointType; (* A variable of PointType designates a point (an intersection) on the board. *)
             (* Blocks are identified by any point in the block (see chapter 5). *)
  ListType;  (* A list is a set of elements, possibly sorted after some criteria. *)
             (* The operation '+' on two lists is a union, '-' is the difference of two lists. *)
  StatusType = (Dead, Alive); (* Dead: prey can be killed - Alive: prey can escape. *)
  ColorType  = (Empty, Black, White); (* The different states of a point on the board. *)
  BlackWhite = (Black, White);


VAR
  ToPlay : BlackWhite; (* The player whose turn it is to play. (Read-only variable) *)


PROCEDURE Color (Point : PointType) : ColorType;
(* Returns the color of a point on the board. *)

PROCEDURE MoveIsLegal (Move  : PointType) : BOOLEAN;
(* Whether a move by ToPlay is legal or not. Point occupied, ko at that point or suicide, are possible reasons
for illegality. *)

PROCEDURE ExecuteMove (Move  : PointType);
(* Execute a legal move by ToPlay on the board. *)

PROCEDURE UndoMove;
(* Undo the last move executed, that is, restore the board as it was before that move. *)

PROCEDURE Liberties (Block : PointType) : ListType;
(* Returns a list containing the liberties of a block. *)

PROCEDURE NuLiberties (Block : PointType) : CARDINAL;
(* Returns the number of liberties of the Block. *)
```

```
PROCEDURE Atari (Block : PointType) : BOOLEAN;
(* Is TRUE if the block is in atari, that is, NuLiberties(Block) = 1 *)

PROCEDURE AdjacentBlocks (Block : PointType;
                          MaxLib: CARDINAL) : ListType;
(* Returns a list containing the blocks adjacent to Block having at most MaxLib liberties. *)
```

Some operators and control structures are defined in a way which is not possible in Modula-2. The differences are:

- Not all variables are declared (for example, if the name implies of what type the variable is).
- I have lists as the return value of functions, although Modula-2 doesn't allow records as return values.
- The procedure `Neighbors` is used both with one (point) and two (point and color) parameters.
- The operators '+' and '-' are used for list union and difference.
- The statement `p1, p2 := List` is used to set p1 and p2 to the first two elements of the list.
- Lists have to be created before use and disposed of afterwards; I ignore this.
- The statement `WHEN RETURN DO ... END` is new. It executes some statements whenever returning from a procedure.
- The control structure

```
FOR EACH legal Move IN MoveList DO
  .....
END(*FOR*);
```

is to be understood as a macro notation for the Modula-2 statements:

```
WHILE NOT IsEmpty(MoveList) DO
  Move := First(MoveList);  MoveList := MoveList - Move;
  IF MoveIsLegal(Move) THEN
    .....
  END(*IF*);
END(*WHILE*);
```

The algorithm *CatchInLadder* (on the following page) solves all ladder problems correctly. It never leads to a combinatorial explosion, even though a move such as Black 3 at 4 in Figure 2-1 is tried, because the next move of the prey will already lead to its escape. - So this algorithm always terminates.

In sections 2.2 to 2.5, this algorithm is improved in several directions.

- It is possible to limit and interrupt the search.
- The solution of the problem is stored so that the user can look at it.
- The moves are given priorities - this will become much more important with the loose ladder.
- A ko occuring during search is taken into account.

```
PROCEDURE CatchInLadder
  (    Prey   : PointType;    (* one stone of the prey block *)
   VAR Result : StatusType); (* whether the prey can be killed or not *)
CONST
  MaxLibPrey   = 1; (* maximal number of prey liberties when it's the prey's turn to play *)
  MaxLibHunter = 2; (* dito when it's the hunter to play *)
VAR
  MoveList : ListType; (* the moves considered at the current level of recursion *)

  PROCEDURE SelectPreyMoves (VAR MoveList : ListType); (* find all moves for prey *)
  BEGIN
    MoveList := Liberties(Prey);
    FOR EACH HunterBlock IN AdjacentBlocks(Prey, 1) DO (* the blocks adjacent to the prey *)
      MoveList := MoveList + Liberties(HunterBlock); (* liberties of hunter blocks in atari *)
    END(*FOR*);
  END SelectPreyMoves;

  PROCEDURE SelectHunterMoves (VAR MoveList : ListType); (* find all moves for hunter *)
  BEGIN
    MoveList := Liberties(Prey);
  END SelectHunterMoves;

BEGIN (* CatchInLadder *)
  IF ToPlay = Color(Prey) THEN
    IF NuLiberties(Prey) > MaxLibPrey THEN Result := Alive; RETURN END;
    SelectPreyMoves(MoveList);
    FOR EACH legal Move IN MoveList DO
      ExecuteMove(Move);
      CatchInLadder(Prey, Result);
      UndoMove;
      IF Result = Alive THEN RETURN END; (* found a way to escape *)
    END(*FOR*);
    Result := Dead; RETURN; (* no move which leads to an escape, so prey is dead *)
  ELSE (* hunter to play *)
    IF Atari(Prey)                        THEN Result := Dead;  RETURN END;
    IF NuLiberties(Prey) > MaxLibHunter THEN Result := Alive; RETURN END;
    SelectHunterMoves(MoveList);
    FOR EACH legal Move IN MoveList DO
      ExecuteMove(Move);
      CatchInLadder(Prey, Result);
      UndoMove;
      IF Result = Dead THEN RETURN END; (* found a way to kill prey *)
    END(*FOR*);
    Result := Alive; RETURN; (* no move which kills prey, so prey is alive *)
  END(*IF*);
END CatchInLadder;
```

CatchInLadder: A simple ladder algorithm

The parameters `Prey` and `Result` could also be declared as global variables (the prey is constant; the earlier result need not be saved during the search), but were kept as parameters for sake of clarity.

2.2. Limit and Interrupt the Search

Obvious ways to limit the search tree are to give a maximal search depth and a maximal number of nodes. To implement this, StatusType must be extended and some lines must be inserted in Algorithm 1.

```
TYPE
  StatusType = (Dead, Alive, Aborted); (* Result of search is not known if it is aborted. *)

VAR
  NuNodes, MaxNodes : CARDINAL; (* number of nodes searched *)
  Depth,   MaxDepth : CARDINAL; (* search depth *)
(* Before the search, NuNodes and Depth must be set to 0; MaxNodes and MaxDepth should be specified by
the user or set to some default value. *)

BEGIN (* CatchInLadder *)
  INC(Depth);
  IF NuNodes > MaxNodes THEN Result := Aborted; RETURN END;
  IF Depth   > MaxDepth THEN Result := Alive;   RETURN END;
  ...
  IF (Result = Alive) OR (Result = Aborted) THEN RETURN END;
  ...
  IF (Result = Dead)  OR (Result = Aborted) THEN RETURN END;
  ...
  WHEN RETURN DO DEC(Depth) END; (* Before returning from the procedure, decrement the depth. *)
END CatchInLadder;
```

To let the user influence the search, the following procedure is called at every node. When tracing, it shows the current board and the number of nodes searched and waits for a keypress. It interprets commands such as *stop*, *abort* and setting MaxDepth. In addition, the user can tell the computer that, in the currently shown position, the prey is clearly dead resp. alive - the computer will not continue on that branch.

```
PROCEDURE UserIntervention (VAR Result : StatusType) : BOOLEAN;
(* Gives the user the possibility to control the search. TRUE is only returned if the user executed one of the
commands abort, dead or alive. *)

BEGIN (* CatchInLadder *)
  INC(Depth);
  IF UserIntervention(Result) THEN RETURN END;
```

Another way to limit the search is to specify a boundary to reduce the search area. The hunter is not allowed to play on the points of that boundary, while the prey may play there. This has the effect that once the boundary is reached, the prey will gain enough liberties to escape.

```
  VAR
    SearchBoundary : ListType; (* points where the hunter is not allowed to play *)
  ...
  ELSE (* hunter to play *)
    ...
    FOR EACH legal Move IN MoveList AND NOT IN SearchBoundary DO
    ...
  END(*IF*);
```

2.3. Store the Optimal Sequence

When the computer tells that "the prey can be killed in 11 moves", the user wants to know how. The computer shows an answer diagram similar to those given in go books, showing that even the strongest defense fails. A closer look at this strongest defense reveals that, at any node in the game, the strongest move for the prey is the one which leads to its escape (regardless how far away it is), or the one which resists capture as long as possible. For the hunter, the strongest move either leads to capture or lets the prey escape as late as possible.

```
CONST
  MaxSequence = 16; (* the first 16 moves of the solution will be stored *)

TYPE
  MoveSequence = ARRAY [1..MaxSequence] OF PointType;

VAR (* global *)
  BestMoves : ARRAY [1..MaxSequence] OF MoveSequence;
    (* the best continuations for depth d, only BestMoves[d, d..MaxSequence] is used *)
  SolutionDepth : CARDINAL; (* the depth of the solution on the currently examined branch *)
VAR (* local to CatchInLadder *)
  HighestDepth : CARDINAL; (* the currently highest depth of the solution *)

PROCEDURE StoreNewBestSequence (CurrentMove : PointType);
BEGIN
  IF Depth <= MaxSequence THEN (* else not stored *)
    IF Depth < MaxSequence THEN
      BestMoves[Depth] := BestMoves[Depth+1]; (* copy whole best sequence from higher level *)
    ELSE (* Depth = MaxSequence *)
      (* no sequence from higher level to copy *)
    END(*IF*);
    BestMoves[Depth, Depth] := CurrentMove; (* new best move at current depth *)
  END(*IF*);
END StoreNewBestSequence;

(* CatchInLadder *)
  HighestDepth  := 0;
  SolutionDepth := Depth; (* depth of solution for terminal nodes *)
  ...
    IF (SolutionDepth > HighestDepth) OR (Result = Alive) THEN
      StoreNewBestSequence(Move);
    END(*IF*);
    IF (Result = Alive) OR (Result = Aborted) THEN RETURN END;
    IF SolutionDepth > HighestDepth THEN HighestDepth := SolutionDepth END;
  END(*IF*);
END(*WHILE*);
Result := Dead; SolutionDepth := HighestDepth; RETURN;
```

The code for the hunter is similar - see algorithm *Catch*. After the search, the optimal sequence is stored in `BestMoves[1, 1..SolutionDepth]`.

2.4. Move Priority

Though the order in which moves are tried is not very critical for the ordinary ladder, I will introduce the move sorting procedures at this point, because they are extremely important for the loose ladder.

To get the moves at each node sorted, the priority list is used. As soon as the value of a move is known, it can be put into the priority list. The moves in that list are sorted, that is, moves with a high value are given high priority and come first in the list. Later, only the order of the moves is important and not their values, so the priority list can be assigned to a normal move list, and the priority list can be used again at the next node.

```
PROCEDURE IncludeInPriorityList (Move  : PointType;
                                 Value : CARDINAL);
(* Assign a value to a move. If the move is already in the list, the value will be set to the maximum of the old
and the new value. *)

PROCEDURE InitPriorityList; (* Clear the priority list before using it. *)

PROCEDURE PriorityList() : ListType; (* Assign the priority list to a normal list. *)
```

Because the values of moves are sometimes derived by multiplication with a constant, octal numbers are used to get both simple numbers and powers of two (fast multiplication). Octal values are denoted by a "b" after the number, e.g. 10b = 8, 100b = 64, 1000b = 512.

The additional parameter ImmediateEscape is TRUE if there is a move which lets the prey escape - then there is no need to look at the other moves. Not only does this save one level of recursion, it avoids going far down the wrong branch when another branch would have led to escape at once.

```
PROCEDURE SelectPreyMoves (VAR MoveList        : ListType; (* find all moves for prey *)
                           VAR ImmediateEscape : BOOLEAN);
BEGIN
  (* Move selection is as before, but now the moves in MoveList are sorted afterwards. *)
  InitPriorityList;
  FOR EACH legal Move IN MoveList DO
    ExecuteMove(Move);
    IF NuLiberties(Prey) > MaxLibPrey THEN
      ImmediateEscape := TRUE;  UndoMove;  RETURN; (* return at once if escape is possible *)
    END(*IF*);
    Value := NuStonesKilled(); (* kill hunter stones rather than set at liberty *)
    IncludeInPriorityList(Move, Value);
    UndoMove;
  END(*FOR*);
  MoveList := PriorityList(); (* get the sorted moves *)
  ImmediateEscape := FALSE;
END SelectPreyMoves;
```

```
PROCEDURE SelectHunterMoves (VAR MoveList : ListType); (* find all moves for hunter *)
VAR
  Value : CARDINAL;
BEGIN
  MoveList := Liberties(Prey); (* as before *)
  InitPriorityList;
  FOR EACH legal Move IN MoveList DO
    ExecuteMove(Pass); (* let the prey play to evaluate the hunter moves *)
    IF MoveIsLegal(Move) THEN
      ExecuteMove(Move);          (* The more liberties the prey could get, ... *)
      Value := NuLiberties(Prey); (* ... the better the move is for the hunter. *)
      UndoMove;
    ELSE
      Value := 0; (* Prey cannot play there, so hunter need not play there at once. *)
    END(*IF*);
    IncludeInPriorityList(Move, Value);
    UndoMove; (* undo the pass *)
  END(*FOR*);
  MoveList := PriorityList(); (* get the sorted moves *)
END SelectHunterMoves;

...
SelectPreyMoves(MoveList, ImmediateEscape);
...
IF ImmediateEscape THEN Result := Alive; RETURN END;
```

2.5. Ko During the Search

If the user thinks that Black has enough ko threats to win any ko which might arise during the search, he can set `PlayerWhoCanWinKo` to Black. Assuming that Black could have played a ko threat, the ko is then taken back by Black at once, ignoring the ko restriction. If nothing is known about who could win a ko, only strictly local ko threats are played during the search.

Ko during the search is implemented by replacing `MoveIsLegal` by `MoveIsLegalOrKo` everywhere in the program. ExecuteMove must allow the execution of moves which are illegal because of ko.

```
VAR
  PlayerWhoCanWinKo : ColorType; (* is Empty if nothing about the ko threats is known *)

PROCEDURE ReasonForIllegality() : (IsOccupied, IsKo, IsSuicide);
(* Tells why the last parameter to MoveIsLegal was an illegal move. *)

PROCEDURE MoveIsLegalOrKo (Point : PointType) : BOOLEAN;
(* Either the move is legal, or it would be illegal because of ko - but the player to play can win any ko, so the
ko restriction is ignored. *)
BEGIN
  RETURN MoveIsLegal(Point)
      OR (ReasonForIllegality() = IsKo) AND (ToPlay = PlayerWhoCanWinKo);
END MoveIsLegalOrKo;
```

3. Loose Ladders

An example of a loose ladder is shown in Figure 3-1 (taken from [Tesuji, p. 17]). The normal ladder starting with *a* doesn't work, so Black plays 1 (the loose ladder tesuji) and guides White firmly to the edge of the board with 3 and 5. If White plays *a* in the third diagram, Black plays *b* and puts him into atari.

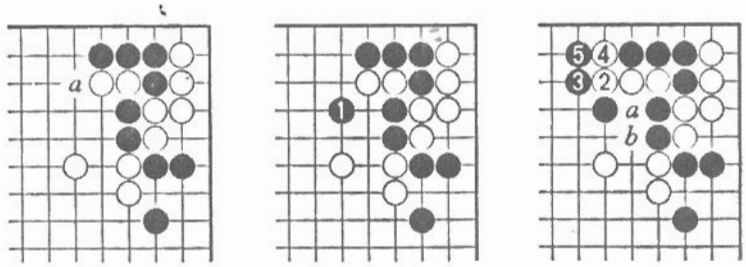


Figure 3-1: A simple loose ladder

The difference to a real ladder is that the prey is not always in atari. The extra liberty at *a* prohibits the zig-zag pursuit of the normal ladder, because the prey could break out with a double atari. So the loose ladder is a ladder with an extra liberty somewhere and has the following characteristics:

Prey to play:

- prey has one or two liberties: ladder goes on;
- prey has more than two liberties: ladder has failed, prey can escape.

Hunter to play:

- prey has one liberty: ladder was a success, prey can be killed;
- prey has two or three liberties: ladder goes on;
- prey has more than three liberties: prey can escape.

The framework of the ladder algorithm can be left unchanged, except that `MaxLibPrey` and `MaxLibHunter` must be set to 2 resp. 3. The move selection has to be expanded - in contrast to the ladder, the two players now have many moves to choose from. To avoid a combinatorial explosion, the moves which most probably lead to success must be tried first.

3.1. Move Selection

In two cases move selection for the loose ladder is just the same as for the ladder. (1) If the prey is in atari, it must escape by playing at its liberty or capture an adjacent hunter block. (2) If it's the hunter to play and the prey has three liberties, then the hunter has to take away a liberty of the prey (else it still has three liberties after the hunter has played and thus escapes).

In addition to the moves selected in the ladder algorithm, the prey has time to play moves which endanger a hunter block, and escape moves, such as the one-point jump or the diagonal move. The hunter will have to consider moves which protect endangered blocks, and points adjacent to the liberties of the prey, e.g. loose ladder and slapping tesuji.

In some situations moves which are not even adjacent to liberties of the prey should be tried, e.g. the knight's move and the cross-cut tesuji. To avoid a combinatorial explosion, such moves should only be considered when they have a real chance of leading to success.

Currently, only two tesuji are recognized: the loose ladder and the slapping tesuji - problems involving other tesuji will not be solved well. However, because the tactical calculator is designed as a tool for good human players and not as an automatic problem solver, the human player will often recognize the tesuji.

Loose Ladder Tesuji

The loose ladder tesuji appears often enough in the middle of a problem to be of general interest. The tesuji move doesn't take away a liberty at once, but, because it is adjacent to two liberties, takes away two potential liberties.

Whether the move at * in Figure 3-2 is a tesuji or not depends on the stones nearby. The move is good, if both *A* and *B* are occupied by the hunter - the prey is then caught in a net. If only *A* is occupied, then the intention of the hunter is to chase the prey in the direction of *B* and *C*. Whether he will succeed depends on the neighbors of *B* and *C*. If, for example, *C* is empty and has four empty neighbors, the prey will get four liberties (three around *C* and one at *Lib2*) and escape.

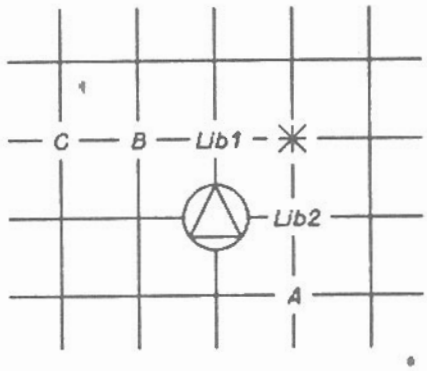


Figure 3-2: Loose ladder tesuji

Slapping Tesuji

The important elements of the slapping tesuji are shown in Figure 3-3. It is less common than the loose ladder tesuji, however, it also leads to a loose ladder. The threat is to play *Lib2* next, capturing the prey, so the prey must connect at *Lib2*. The hunter will then play *D* to start a loose ladder.

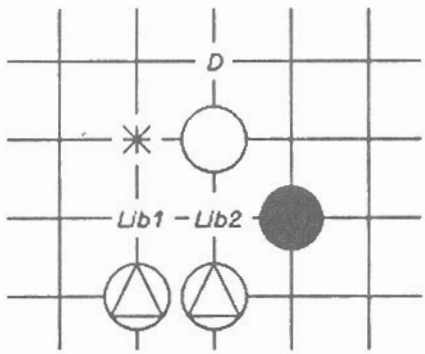


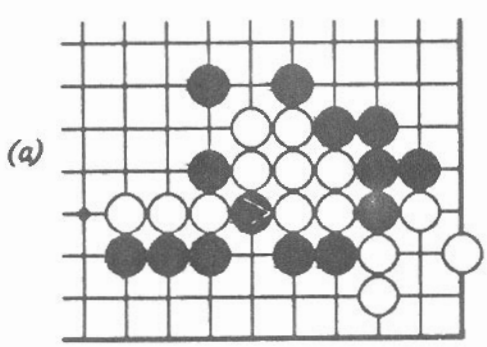
Figure 3-3: Slapping tesuji

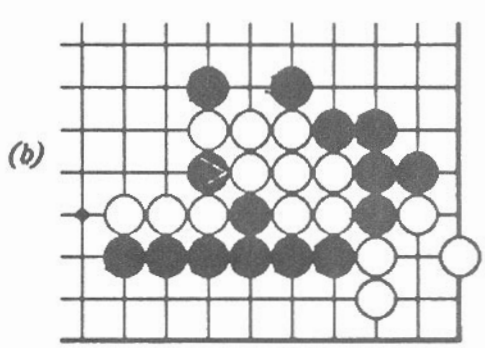
3.2. Move Priority

The most important criteria for sorting moves is the number of liberties of the prey. The prey tries to increase its number of liberties as much as possible, the hunter tries to limit the liberties of the prey. But in some situations there are more important moves, e.g. when the prey threatens to kill a hunter block which should be defended. The following paragraph shows that whether a stone shall be defended or sacrificed is again a question of liberties.

Sacrifice or Defend Stones

How can the program recognize whether a hunter block which is in atari shall be sacrificed or defended ? Consider the example of Figure 3-4. The triangled stone in (a) must be defended, because if it is captured, the prey can connect with his three stones on the left side. In addition, the two black stones below come under attack. In (b), the triangled stone should be sacrificed. White can still connect with his three stones on the left, but now this does not give him enough additional liberties.





Thus the rule is as follows: count the liberties of the prey after the prey has captured the block and connected (while the hunter passes). In the first example, this gives the prey 6 liberties, in the second only 4. Because the prey will also play in that time, the prey will finally get 4 resp. 2 liberties. Therefore sacrifice those stones which don't give the prey more than 4 liberties.

Defending a block which is in atari is best done by capturing an adjacent block. If that is not possible, the hunter block must get out of atari with a move at its liberty.

Priorities

Prey:

1. Highest priority to moves which put hunter blocks in double atari without putting the just played stone into atari (to escape from a normal ladder pursuit).
2. Maximize the number of liberties of the prey after the move.
3. Capture as many stones as possible.
4. Maximize the number of hunter stones which are in atari.

Hunter:

1. Moves which nearly kill the prey, that is, the prey is in atari after the move, no hunter block is in atari and after moving at its liberty, the prey still stays in atari. This avoids missing 'obvious' captures during the search.
2. Protect hunter block which must be defended, preferably by capturing an adjacent block, otherwise by playing at its liberty.
3. Loose ladder or slapping tesuji.
4. Play at the liberties of the prey, where the moves are sorted after which would give most new liberties to the prey if it could play there. - As a second sorting criteria I use the heuristic that the hunter should leave as few cutting points as possible. I do this by prefering moves which are adjacent to as many single-stone hunter blocks as possible (the move would connect these together).
5. At a liberty of the prey, where the hunter is in atari after the move (could be a throw-in).
6. Protect a hunter block which is near atari by playing at one of its liberties.

3.3. Loose Ladder Algorithm

In addition to the procedures already defined, the following procedures are used:

```
PROCEDURE Exchange (VAR Point1, Point2);
(* Exchange two points. *)

PROCEDURE NuStonesKilled() : CARDINAL;
(* The number of stones killed by the last ExecuteMove. *)

PROCEDURE Clear (List : ListType);
(* Clear a list. *)

PROCEDURE Cardinality (List : ListType) : CARDINAL;
(* The number of elements in a list. *)

PROCEDURE IsEmpty (List : ListType) : BOOLEAN;
(* TRUE if Cardinality(List) = 0 *)

PROCEDURE Occupied (Point : PointType) : BOOLEAN;
(* Whether a point is occupied or not. *)
```

```
PROCEDURE OnBoard (Point : PointType) : BOOLEAN;
 (* Whether a point is on the board or outside. *)

PROCEDURE Line (Point : PointType) : CARDINAL;
(* Return the distance of the point from the edge, e.g. Line(Edge) = 1, Line(Hoshi) = 4. *)

PROCEDURE NuStones (Block : PointType) : CARDINAL;
(* The number of stones of a block. *)

PROCEDURE StoneWithLib (Block  : PointType;
                        MinLib : CARDINAL) : PointType;
(* Return a stone of Block which has at least MinLib empty neighbors. If there is no such stone, NoPoint is
returned. *)

PROCEDURE ALiberty (Block : PointType) : PointType;
(* Return one liberty of the block. *)

PROCEDURE Adjacent (Point1, Point2 : PointType) : BOOLEAN;
(* TRUE if the points are direct (horizontal or vertical) neighbors. *)

Neighbors, NuNeighbors, EmptyNeighbors, NuEmptyNeighbors: (* Return the direct neighbors of
a point, either all, only those of a given color or only the number of such neighbors. *)

PROCEDURE Neighbor (Point : PointType;
                    Color : ColorType) : PointType;
(* Return one neighbor of that color, or NoPoint if there is no such neighbor. *)


PROCEDURE MoveNotLegalOrAtari (Move : PointType) : BOOLEAN;
(* TRUE if the move is not legal, or if, after the move, the block which the move is a part of is in atari. *)
VAR
  IsInAtari : BOOLEAN;
BEGIN
  IF NOT MoveIsLegal(Move) THEN RETURN TRUE END;
  ExecuteMove((Move);  IsInAtari := Atari(Move);  UndoMove;
  RETURN IsInAtari;
END MoveNotLegalOrAtari;


PROCEDURE MoveLegalAndNotAtari (Move : PointType) : BOOLEAN;
(* TRUE if the move is legal and the block is not in atari after the move. Move...AndNot... is exactly
inverse to MoveNot...Or.... *)
BEGIN
  RETURN NOT MoveNotLegalOrAtari;
END MoveLegalAndNotAtari;


PROCEDURE PreyIsNearlyKilled() : BOOLEAN;
(* If it's the prey to play, and the prey is in atari, and there is no adjacent hunter block to capture and the prey
doesn't get out of atari by playing at its liberty, then the prey has no way to escape. *)
BEGIN
  RETURN Atari(Prey)                                (* prey is in atari *)
     AND IsEmpty(AdjacentBlocks(Prey, 1))           (* and no hunter block in atari *)
     AND MoveNotLegalOrAtari(ALiberty(Prey));       (* and still in atari after next move *)
END PreyIsNearlyKilled;
```

```
PROCEDURE SelectPreyMoves (VAR MoveList        : ListType;
                           VAR ImmediateEscape : BOOLEAN);
BEGIN
  InitPriorityList;  Clear(MoveList);
  (* Find threats to the hunter. Goal: after the move, a hunter block shall have less liberties than the prey. *)
  FOR EACH HunterBlock IN AdjacentBlocks(Prey, NuLiberties(Prey)) DO
    FOR EACH Liberty IN Liberties(HunterBlock) DO
      IF Liberty IN MoveList AND MoveLegalAndNotAtari(Liberty) THEN
        IncludeInPriorityList(Liberty, 1000b); (* double-threat to the hunter *)
      ELSE
        MoveList := MoveList + Liberty;
      END(*IF*);
    END(*FOR*);
  END(*FOR*);
  MoveList := MoveList + Liberties(Prey); (* All liberties of the prey are good moves. *)
  IF NOT Atari(Prey) THEN
    MoveList := MoveList + Pass; (* A pass may be best if a living position is reached. *)
    (* Empty neighbors of the liberties of the prey may be good escape moves (e.g. one-point jump and
diagonal move). But there is no sense in trying moves when, after the prey has played and the hunter moved
inbetween, the prey is nearly killed. *)
    FOR EACH Liberty IN Liberties(Prey) DO
      FOR EACH Neighbor IN EmptyNeighbors(Liberty) AND NOT IN MoveList DO
        IF MoveIsLegal(Neighbor) THEN (* prey plays a one-point jump or diagonal move *)
          ExecuteMove(Neighbor);
          IF MoveIsLegal(Liberty) THEN (* hunter plays inbetween *)
            ExecuteMove(Liberty);
            IF NOT PreyIsNearlyKilled() THEN (* ok if prey cannot be killed at once *)
              IncludeInPriorityList(Neighbor, 70b * NuLiberties(Prey));
            END(*IF*);
            UndoMove;
          ELSE (* ok if the hunter is not allowed to play inbetween *)
            IncludeInPriorityList(Neighbor, 70b * NuLiberties(Prey));
          END(*IF*);
          UndoMove;
        END(*IF*);
      END(*FOR*);
    END(*FOR*);
  END(*IF*);
  FOR EACH legal Move IN MoveList DO (* evaluate the moves *)
    ExecuteMove(Move);  NuKilled := NuStonesKilled();
    IF NuLiberties(Prey) > MaxLibHunter THEN
      UndoMove;  ImmediateEscape := TRUE;  RETURN;
    END(*IF*);
    NuHunterStonesInAtari := 0;
    FOR EACH HunterBlock IN AdjacentBlocks(Prey, 1) DO
      INC(NuHunterStonesInAtari, NuStones(HunterBlock));
    END(*FOR*);
    IncludeInPriorityList(Move, 100b * NuLiberties(Prey)       (* maximize liberties *)
                              + 10b * NuKilled                 (* kill hunter stones *)
                              +       NuHunterStonesInAtari); (* attack hunter *)
    UndoMove;
  END(*FOR*);
  MoveList := PriorityList();  ImmediateEscape := FALSE;
END SelectPreyMoves;
```

```
PROCEDURE LooseLadderTesuji (VAR Tesuji : PointType) : BOOLEAN;
VAR
  PreyStone, Lib1, Lib2, A, B, C : PointType;
BEGIN
  PreyStone := StoneWithLib(Prey, 2); (* at most one stone to examine *)
  IF PreyStone = NoPoint THEN RETURN FALSE END; (* no stone with 2 empty neighbors *)
  Lib1, Lib2 := EmptyNeighbors(PreyStone);
  Tesuji := Lib1 + Lib2 - PreyStone; (* see explanation in chapter 5 *)
  IF Occupied(Tesuji)  THEN RETURN FALSE END;
  IF Line(Tesuji) <= 2 THEN RETURN FALSE END; (* A normal ladder is ... *)
  A := 2*Lib1 - Tesuji; (* critical points *)              (* ... better if too near the edge. *)
  B := 2*Lib2 - Tesuji; (* Lib 2 + (Lib2 - Tesuji); see chapter 5 *)
  IF Color(A) = Empty THEN Exchange(A, B) END; (* if a point is empty, then let it be B *)
  IF   (Color(A) = Color(Prey))
    OR (Color(B) = Color(Prey))
    OR ((Color(A) = Empty) AND (Color(B) = Empty))
  THEN
    RETURN FALSE; (* no tesuji, some precondition not satisfied *)
  ELSE
    IF NOT OnBoard(B) OR Occupied(B) OR (NuEmptyNeighbors(B) <= 2) THEN
      RETURN TRUE; (* ok if both points occupied by hunter or border, or if B has few empty neighbors *)
    ELSE
      C := (3*B - Tesuji) DIV 2; (* else test for some obstacle in that direction *)
      RETURN NuEmptyNeighbors(C) < 4; (* ok if C has few neighbors *)
    END(*IF*);
  END(*IF*);
END LooseLadderTesuji;


PROCEDURE SlappingTesuji (VAR Tesuji : PointType) : BOOLEAN;
VAR
  Lib1, Lib2 : PointType;
BEGIN
  IF NuLiberties(Prey) <> 2 THEN RETURN FALSE END;
  Lib1, Lib2 := Liberties(Prey);
  IF NOT Adjacent(Lib1, Lib2) THEN RETURN FALSE END;
  IF NuEmptyNeighbors(Lib1) = 1 THEN Exchange(Lib1, Lib2) END;
  IF   (NuEmptyNeighbors(Lib1) <> 3)
    OR (NuEmptyNeighbors(Lib2) <> 1)
    OR (NuNeighbors(Lib2, Color(Prey)) <> 2)
    OR (Board[2*Lib2 - Lib1] <> Color(Hunter))
  THEN
    RETURN FALSE; (* some precondition not satisfied *)
  ELSE
    Tesuji := 2*Lib1 - Neighbor(Lib1, Color(Prey));
    RETURN TRUE;
  END(*IF*);
END SlappingTesuji;
```

```
PROCEDURE MustBeProtected (     Hunter : PointType;
                           VAR Moves  : ListType) : BOOLEAN;
(* The moves returned are the moves which help to protect the endangered hunter block. The hunter block
must only be protected if a capture and connection would give the prey too many liberties. *)
VAR
  Protect : BOOLEAN;
BEGIN
  Clear(Moves);
  FOR EACH Move IN Liberties(Hunter) DO
    IF MoveLegalAndNotAtari(Move) THEN Moves := Moves + Move END;
  END(*FOR*);
  FOR EACH PreyBlock IN AdjacentBlocks(Hunter, 1) DO
    Moves := Moves + ALiberty(PreyBlock); (* capture a block to get more liberties *)
  END(*FOR*);
  IF Atari(Hunter) THEN
    IF NuStones(Hunter) >= 2 THEN
      Protect := TRUE; (* always protect blocks with several stones *)
    ELSE
      ExecuteMove(Pass); ExecuteMove(ALiberty(Hunter)); (* prey captures... *)
      ExecuteMove(Pass); ExecuteMove(Hunter);           (* ... and connects *)
      Protect := (NuLiberties(Prey) >= 6); (* protect if too many liberties *)
      UndoMove; UndoMove; UndoMove; UndoMove;
    END(*IF*);
  ELSE
    Protect := FALSE; (* Defensive action is not urgent when not in atari. *)
  END(*IF*);
  RETURN Protect;
END MustBeProtected;


PROCEDURE SelectHunterMoves;
VAR
  NuLibertiesAfterMove, SingleStoneNeighbors : CARDINAL;
  NuHunterBlocksInAtari, Value : CARDINAL;
  DefensiveMoves : ListType;
BEGIN
  InitPriorityList;
  FOR EACH Move IN MoveList DO
    ExecuteMove(Pass); (* imagine that prey could move now *)
    IF MoveIsLegal(Move) THEN
      ExecuteMove(Move);
      NuLibertiesAfterMove  := NuLiberties(Prey);
      NuHunterBlocksInAtari := Cardinality(AdjacentBlocks(Prey, 1));
      UndoMove;
    ELSE
      NuLibertiesAfterMove  := 0; (* hunter should not hurry to play... *)
      NuHunterBlocksInAtari := 0; (* ... where the prey is not allowed to play *)
    END(*IF*);
    UndoMove; (* undo the pass *)

    SingleStoneNeighbors := 0;
    FOR EACH Neighbor IN Neighbors(Move, Color(Hunter)) DO
      IF NuStones(Neighbor) = 1 THEN INC(SingleStoneNeighbors) END;
    END(*FOR*);
```

```
    IF MoveIsLegal(Move) THEN
      ExecuteMove(Move);
      IF Atari(Move) THEN
        IncludeInPriorityList(Move,   50b); (* bad: hunter is in atari after move *)
      ELSIF PreyIsNearlyKilled() THEN
        IncludeInPriorityList(Move, 2000b); (* super: prey will surely be killed *)
      ELSE
        IncludeInPriorityList(Move,  200b
                                + 100b * NuLibertiesAfterMove
                                +  10b * NuHunterBlocksInAtari
                                +   2b * SingleStoneNeighbors
                                +        NuEmptyNeighbors(Move) );
      END(*IF*);
      UndoMove;
    END(*IF*);
  END(*FOR*);

  IF NuLiberties(Prey) <= MaxLibPrey THEN
    (* Need not take away liberty at once; has time to protect hunter blocks with few liberties. *)
    FOR EACH HunterBlock IN AdjacentBlocks(Prey, MaxLibPrey) DO
      IF MustBeProtected(HunterBlock, DefensiveMoves) THEN
        Value := 640b; (* protect important stones *)
      ELSE
        Value :=  40b; (* sacrifice squeeze stones *)
      END(*IF*);
      FOR EACH Move IN DefensiveMoves DO IncludeInPriorityList(Move, Value) END;
    END(*FOR*);
    (* Tesuji: Good moves which are not at the liberties of the prey. *)
    IF LooseLadderTesuji(Tesuji) OR SlappingTesuji(Tesuji) THEN (* can never be both *)
      IncludeInPriorityList(Tesuji, 1000b);
    END(*IF*);
  END(*IF*);
  MoveList := PriorityList();
END SelectHunterMoves;


PROCEDURE Catch
  (    Prey   : PointType;   (* one stone of the prey block *)
   VAR Result : StatusType); (* whether the prey can be killed or not *)
VAR
  MoveList     : ListType; (* the moves considered at the current level of recursion *)
  HighestDepth : CARDINAL; (* see section 2.3 *)
BEGIN (* Catch *)
  INC(Depth);
  HighestDepth  := 0;
  SolutionDepth := Depth; (* depth of solution for terminal nodes *)
  IF UserIntervention(Result) THEN RETURN END; (* as in section 2.2 *)
  IF NuNodes > MaxNodes THEN Result := Aborted; RETURN END;
  IF Depth   > MaxDepth THEN Result := Alive;   RETURN END;
```

```
  IF ToPlay = Color(Prey) THEN
    IF NuLiberties(Prey) > MaxLibPrey THEN Result := Alive; RETURN END;
    SelectPreyMoves(MoveList, ImmediateEscape);
    IF ImmediateEscape THEN Result := Alive; RETURN END;
    FOR EACH legal Move IN MoveList DO
      ExecuteMove(Move);
      Catch(Prey, Result);
      UndoMove;
      IF (SolutionDepth > HighestDepth) OR (Result = Alive) THEN
        StoreNewBestSequence(Move); (* as in section 2.3 *)
      END(*IF*);
      IF (Result = Alive) OR (Result = Aborted) THEN RETURN END;
      IF SolutionDepth > HighestDepth THEN HighestDepth := SolutionDepth END;
    END(*FOR*);
    (* no move which leads to an escape, so prey is dead *)
    Result := Dead; SolutionDepth := HighestDepth; RETURN;
  ELSE (* hunter to play *)
    IF Atari(Prey) THEN Result := Dead;  RETURN END;
    IF NuLiberties(Prey) > MaxLibHunter THEN Result := Alive; RETURN END;
    SelectHunterMoves(MoveList);
    FOR EACH legal Move IN MoveList AND NOT IN SearchBoundary DO (* see 2.2 *)
      ExecuteMove(Move);
      Catch(Prey, Result);
      UndoMove;
      IF (SolutionDepth > HighestDepth) OR (Result = Dead) THEN
        StoreNewBestSequence(Move);
      END(*IF*);
      IF (Result = Dead)  OR (Result = Aborted) THEN RETURN END;
      IF SolutionDepth > HighestDepth THEN HighestDepth := SolutionDepth END;
    END(*FOR*);
    (* no move which kills prey, so prey is alive *)
    Result := Alive; SolutionDepth := HighestDepth; RETURN;
  END(*IF*);
  WHEN RETURN DO DEC(Depth) END; (* Before returning from the procedure, decrement the depth. *)
END Catch;
```

Catch: A general ladder and loose ladder algorithm

The normal and the loose ladder differ only in the values of `MaxLibPrey` and `MaxLibHunter`, that is, the maximal number of liberties when it's the prey resp. the hunter to play:

```
Normal Ladder:  MaxLibPrey   = 1;
                MaxLibHunter = 2;
Loose Ladder:   MaxLibPrey   = 2;
                MaxLibHunter = 3;
```

Some promising experiments with other values for `MaxLibPrey` and `MaxLibHunter` have been made. If both are 2 or both are 3, the algorithm catches stones in a net. For higher values, the danger of an explosion is much greater, and additional tesuji should be recognised because both players have more room for complicated tactics.

4. Life and Death

Not enough effort has been invested in the life and death algorithm to make it useful. Therefore I will only sketch the algorithm and mention some problems.

Before the search, the user selects the blocks he wants killed and the area where the prey shall try to make two eyes. This fix area for the search makes move selection easy: the prey moves in the selected area; the hunter moves in the selected area and at the outside liberties of the prey blocks.

The prey is *killed* when one of the prey blocks is killed. The prey is *alive* when each prey block has at least two liberties where the hunter is not allowed to play, even when all outside liberties are filled in. This is a very primitive concept of two eyes. - A seki is alive, because there is no way to kill it.

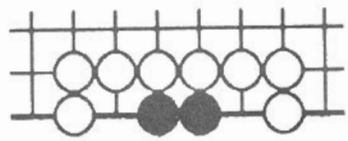
Highest priority is given to the *prey* moves which create illegal moves for the hunter, that is, make eyes. Next come moves which kill hunter stones. If that is not possible, the prey sets at a key point (a point which has several empty neighbors). - The *hunter* tries to maximize his liberties and to kill as many stones as possible. Outside liberties are filled in last. At a node, each move is tried out to see, if it at once leads to a dead (alive) group.

With this algorithm, 170 nodes were examined to find out, that the bulky-five shape is dead - with three outside liberties, 2900 nodes were examined! This observation led to the conclusion, that some dead shapes must be recognised, independent of the number of outside liberties of the prey.

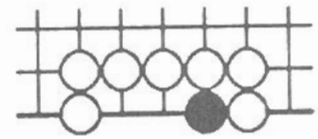
To determine whether a shape is unconditionally dead or not, the number of hunter stones and the number of empty points in the selected area (the area where the prey shall make two eyes) are counted.

(1) If there is one empty point, this is at most one eye. To make two eyes, the prey must kill hunter stones by playing at that sole empty point. If less than four hunter stones are killed, the resulting shape is dead.

(2) If there are two empty points, this may be two eyes. The shape is dead, if these points are connected by one or two hunter stones. For example, the shape eye-hunter-hunter-eye is dead, but not eye-eye-hunter. (If there were three hunter stones, it could be a seki.)



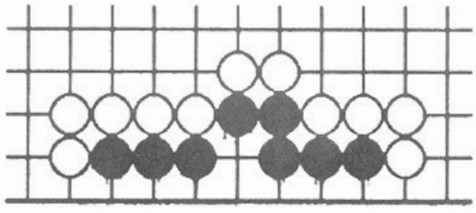
dead



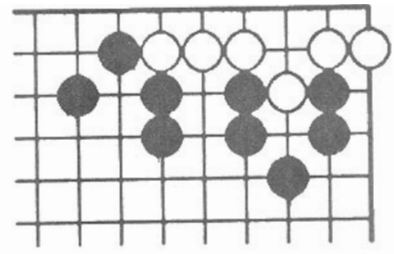
not dead

With these new criteria for dead groups, the bulky-five problem is solved with 16 nodes (31 in the case of three outside liberties) - a factor 10 resp. 100 of improvement. The extra work at each node slowed the search down from 40 to 30 nodes per second, which is insignificant in relation to the improvement.

An additional problem are boundary effects. There are rare situations, where a move outside the selected area is necessary. Enlarging the area is not a solution, because (1) that additional area must then be taken into account at every move, which needs much time, and (2) the same problem may occur with a new boundary.



(a) 1000 nodes, 45 seconds
[Life & Death, p. 30, Dia. 1]



(b) 18000 nodes, 10 minutes
[Fundamentals, p. 119, Dia. 1]

Figure 4-1: Two life and death problems which can be solved.

5. Board Data Structure

The internal representation of the go board should be a good trade-off between execution time, memory requirement and complexity of programming. There are two extremes:

(a) Compute: Only store the board, compute everything else when it is needed. This approach has the advantages, that very little storage is needed, and that moves can easily be undone (there is no redundant information which must be updated). The disadvantage is, that the same things are computed over and over again.

(b) Store: Store much more information about the board (e.g. blocks and liberties). The advantage is, that information is only computed once; however, much memory is needed, and when undoing moves, all information about the earlier state of the board must be restored.

Approach *(a)* is potentially very fast, when only small blocks are on the board. To find the stones and liberties of a block, the procedure explained in [Kierulf / Merkle 84] can be used. Approach *(b)* is appropriate, when much about the moves must be known and moves are seldom taken back.

I have choosen a data structure which is between these two extremes, because neither is ideal for tactical search: (a) Blocks may become quite large (e.g. the prey which is chased in a ladder), and (b) moves must be taken back as often as they are executed.

The main points of the data structure are:

- The stones of a block are linked together in a double linked list. That list is sorted in such a way, that stones which are adjacent to liberties come first in the list. This speeds up the search for liberties when the blocks are large and have few liberties.

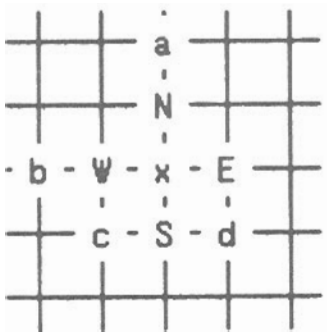
- The liberties are not stored; however, the number of liberties is stored whenever the liberties have been computed. The number of liberties may have the value Unknown, which tells that the liberties must be recomputed. During tactical search, the number of liberties is known between 95 and 98 % of the time.

- For each point, the number of empty neighbors is stored - a kind of local liberties. This allows to keep the stone lists sorted efficiently: Each time a stone is placed on or removed from the board, the adjacent stones must be checked to see if they lost their last or got their first liberty (then they must possible be put at another place in the stone list).

- The number of stones of a block is seldom needed, but it is stored to speed up internal operations on the data structure: When two blocks are joined by a move, the smaller block should be joined to the larger.

- For two reasons, the board itself is represented as a one-dimensional array: (1) Each access to a two-dimensional array needs a (time-consuming) multiplication with 19 because the internal representation is one-dimensional. (2) It's simpler to have each point represented by a single value instead of a pair of row and column. - A line outside the board is included to avoid range checks when finding the neighbors of a point.

- Finding the neighbors of a point is as easy as for the two-dimensional array: just add a constant for each direction. Relationships between points can be expressed independent of direction.




```
N = x - 20;      a = N - (x - N) = 2N - x;
W = x -  1;      b = 2W - x;
E = x +  1;      c = W + S - x;
S = x + 20;      d = E + S - x;
```

Figure 5-1: Relationships between points on the board

References

[Fundamentals] Kageyama, Toshiro.
Lessons in the Fundamentals of Go.
Ishi Press, Tokyo, 1978.

[Kierulf / Merkle 84] Kierulf, Anders / Merkle, Marcello.
A Smart Go Board.
Unpublished paper, Term Project ETH Zürich, March & April, 1984.

[Kierulf / Nievergelt 85] Kierulf, Anders / Nievergelt, Jurg.
Computer Go: A Smart Board and its Applications.
Unpublished paper, ETH Zürich, February, 1985.

[Kierulf 85] Kierulf, Anders.
Computer Go Bibliography.
Unpublished paper, ETH Zürich, March, 1985.

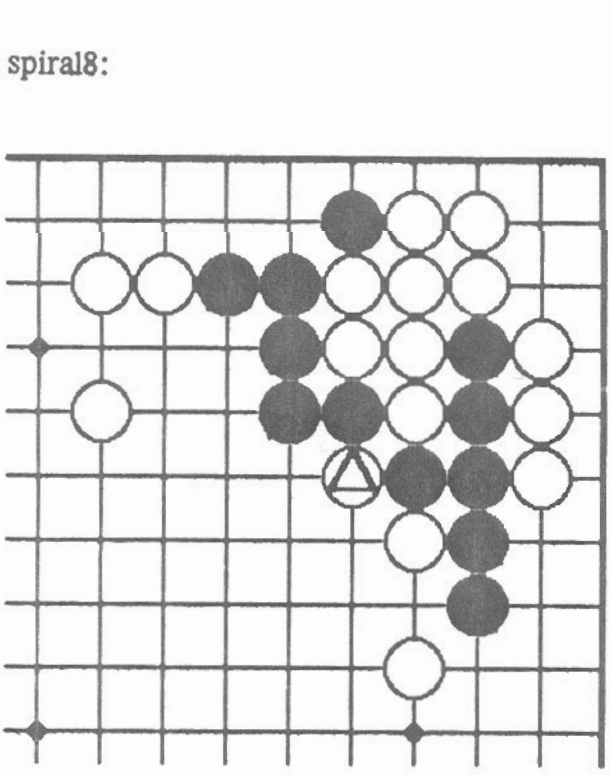
[Life & Death] Davies, James.
Life and Death.
Ishi Press, Tokyo, 1975.

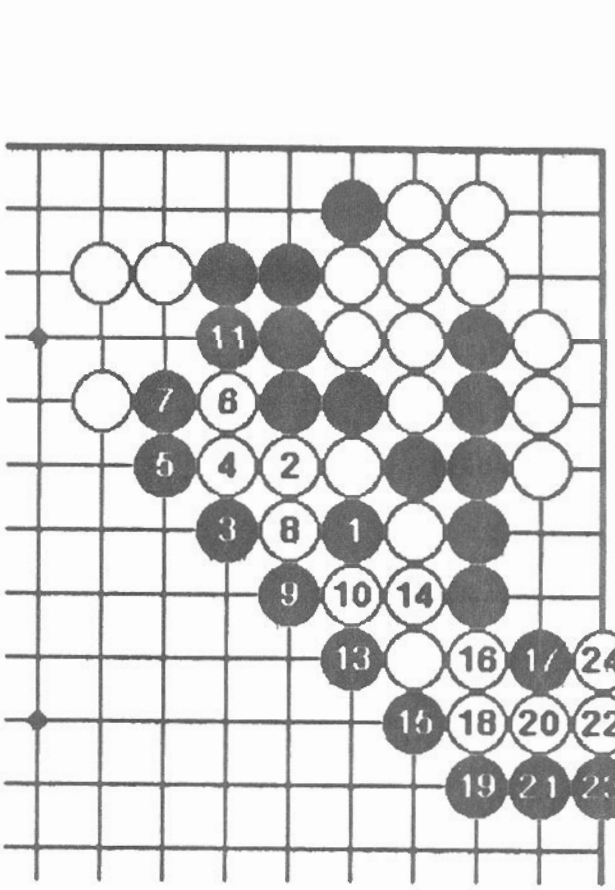
[Tesuji] Davies, James.
Tesuji.
Ishi Press, Tokyo, 1975.

[Treasure] Nakayama, Noriyuki.
The Treasure Chest Enigma.
?

[Wirth 82] Wirth, Niklaus.
Programming in Modula-2.
Springer Verlag, 1982.

spiral8:





Appendix: Loose Ladder Examples

nodes : number of nodes searched
time : seconds used
sol. : solution depth (in how many moves it can be catched)
max. : maximal depth searched
source, page, dia.: the source of the problem

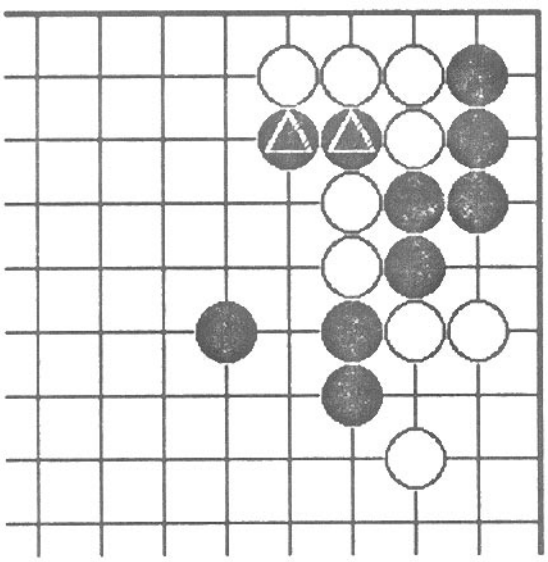
	nodes	time	sol.	max.	source	page	dia.	
loose1	81	3	11	11	[Tesuji]	17	1	
loose2	992	39	21	24	[Tesuji]	17	4	
loose3	370	19	15	18	[Tesuji]	17	5	
spiral1	41	2	11	11	[Fundamentals]	211	1	(a)
spiral2	55	3	13	13	[Fundamentals]	213	7	
spiral3	332	13	17	17	[Fundamentals]	212	3	
spiral4	188	10	21	22	[Fundamentals]	212	4	
spiral5	226	9	27	27	[Fundamentals]	213	5	
spiral6	182	8	25	25	[Fundamentals]	213	6	
spiral8	176	8	25	25	[Fundamentals]	219	26	(b)
slapp1	928	40	11	15 (c)	[Tesuji]	18	1	
slapp2	170	10	9	18	[Tesuji]	19	7	
nose3	3553	153	17	20	[Tesuji]	23	5	
japan	684	48	31	33	[Treasure]	?	?	

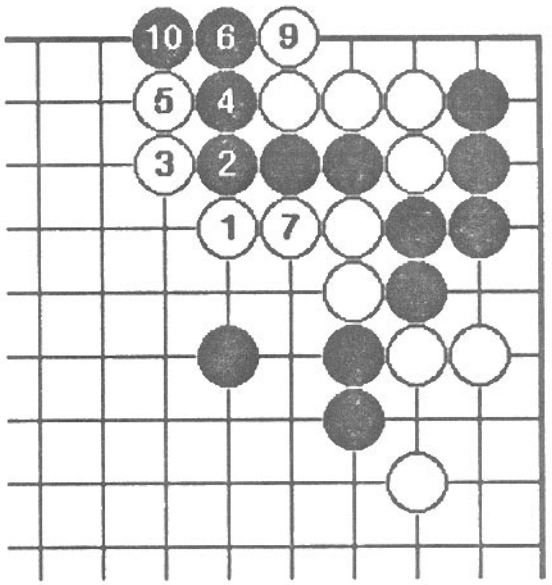
(a) Another solution is found if the atari on the prey has not yet been played.

(b) The essence of spiral8 is to threaten one block in order to become strong and be able to catch another block. The computer solves only the second part of the problem, which is a loose ladder.

(c) Without restricting the search, slapp1 explodes. The problem is solved correctly when the maximal search depth is set to 15. With a maximal depth of 50, 30000 nodes are searched to find the solution.

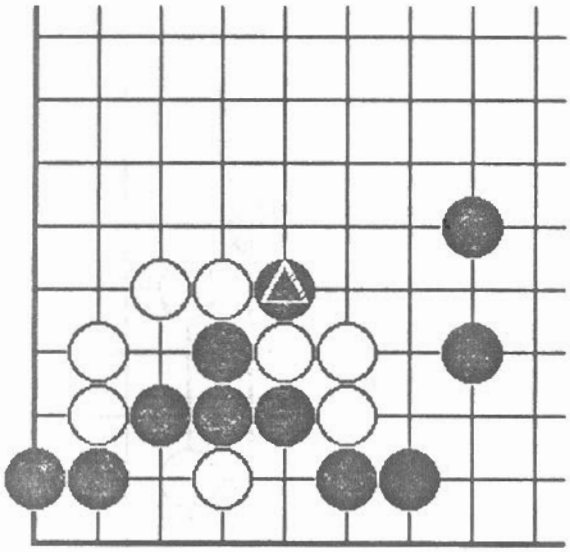
loose1:

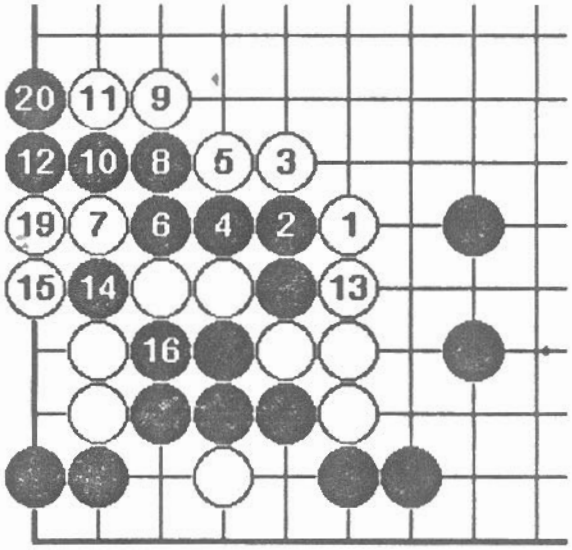




8: pass

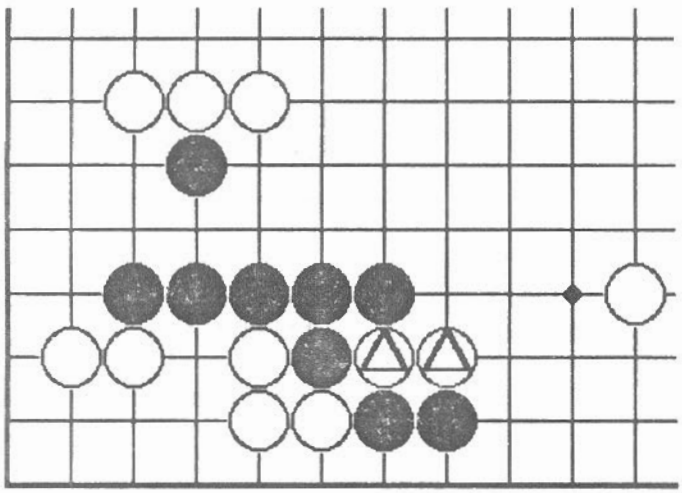
loose2:

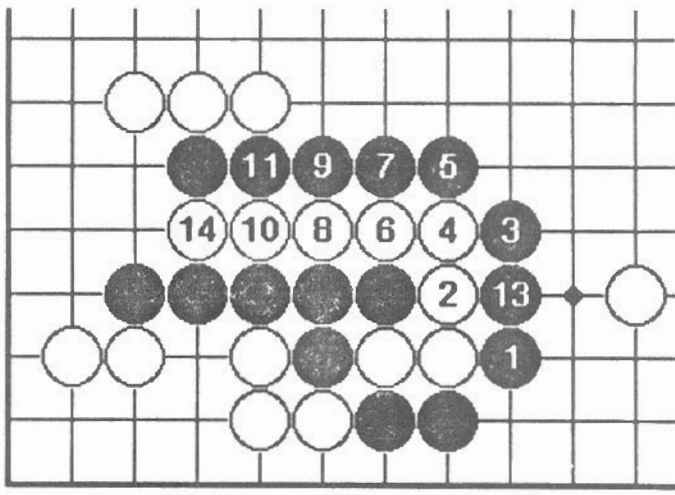




17 at 14, 18: pass

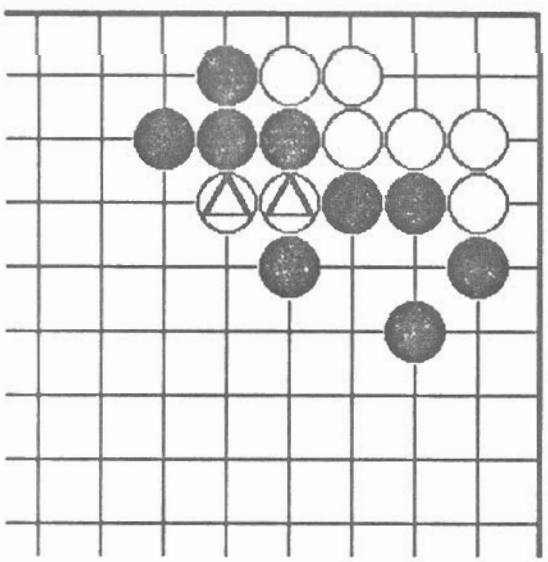
loose3:

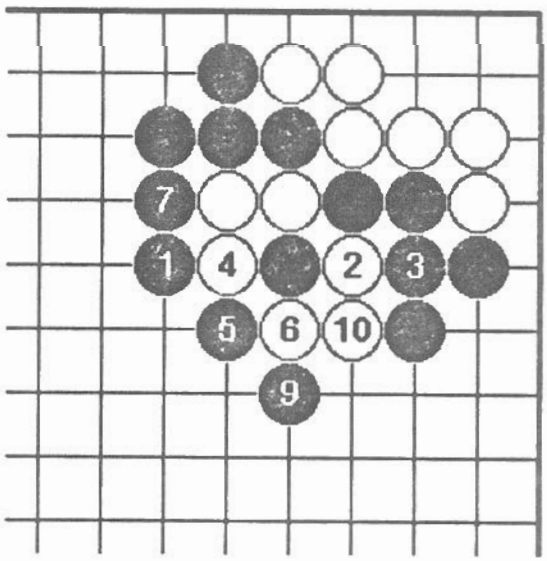




12: pass

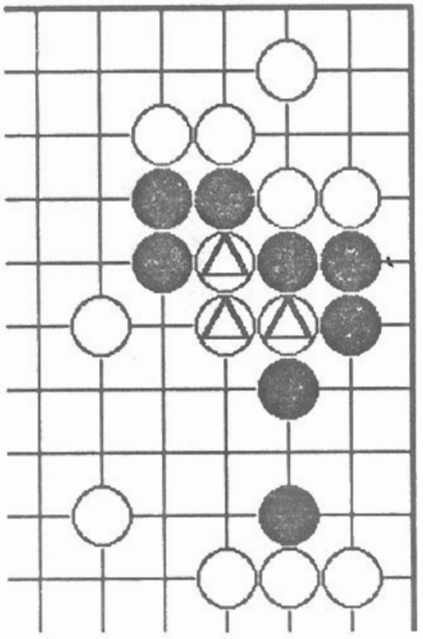
spiral1:

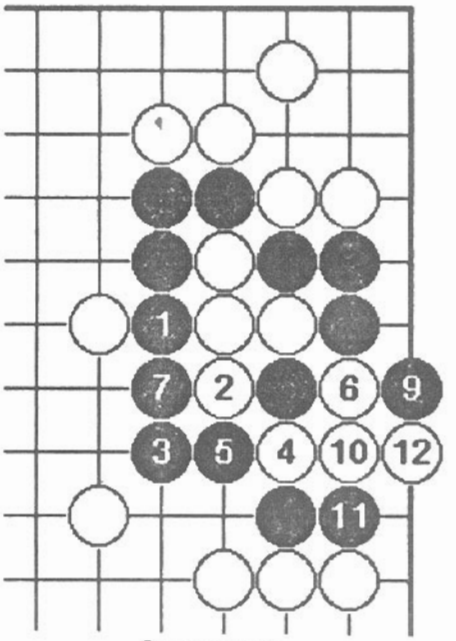




8 connects

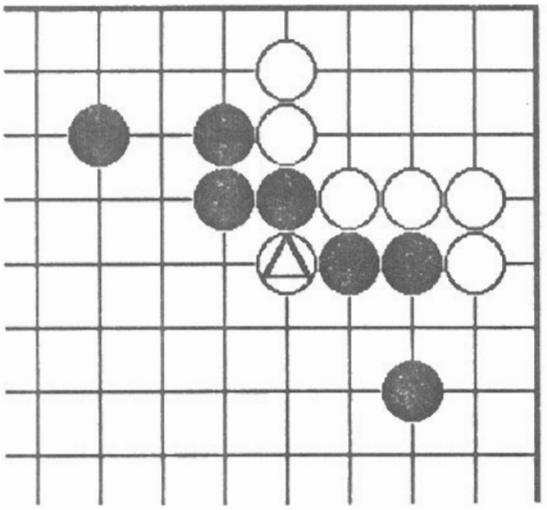
spiral2:

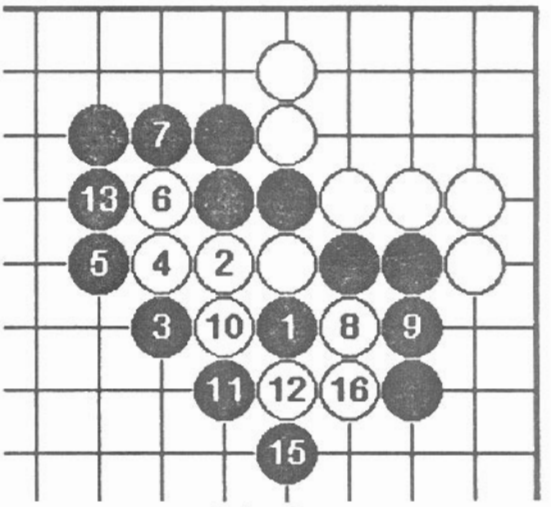




8 connects

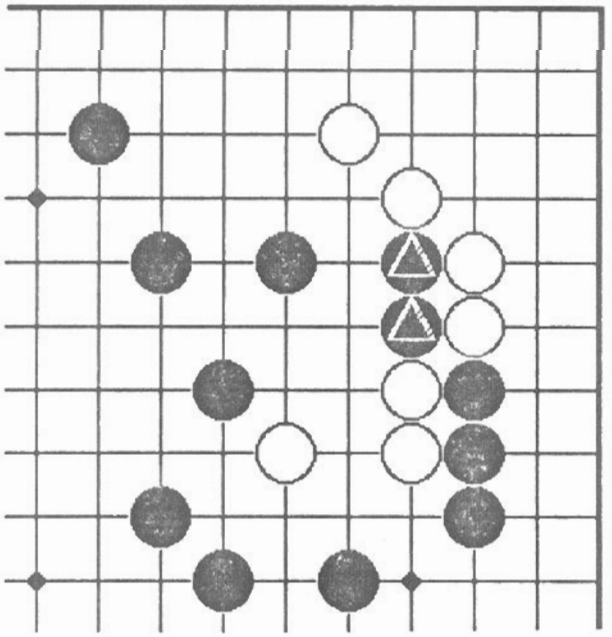
spiral3:

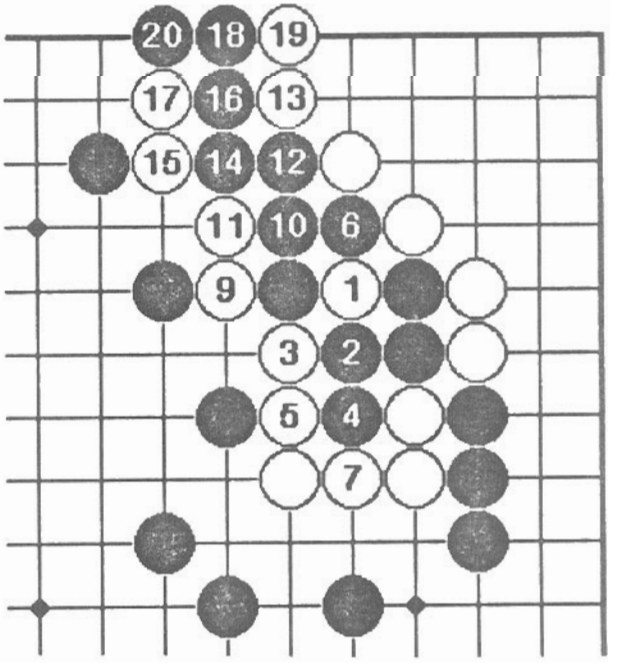




14 at 1

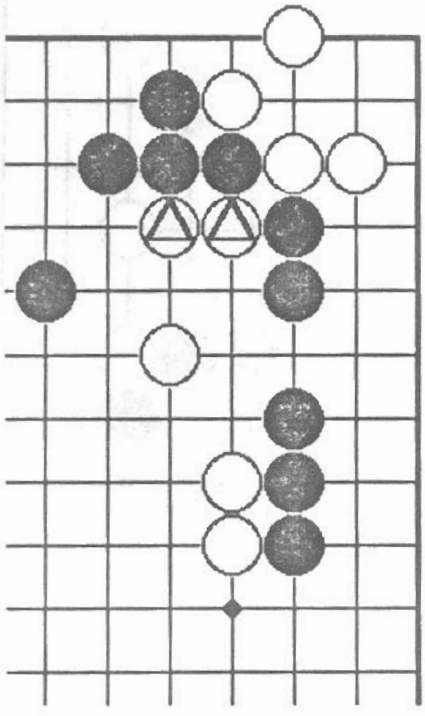
spiral4:

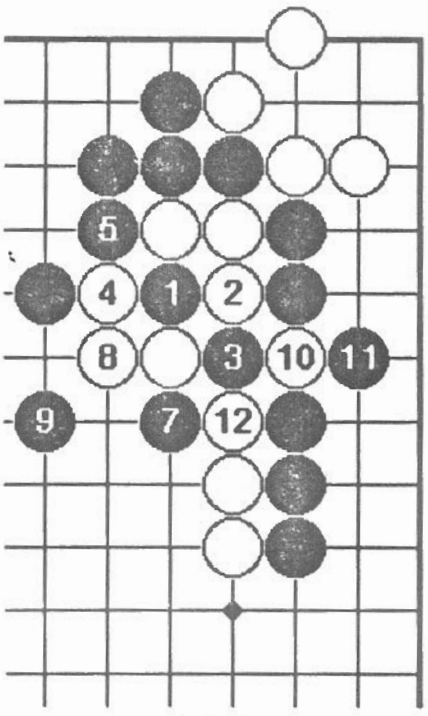




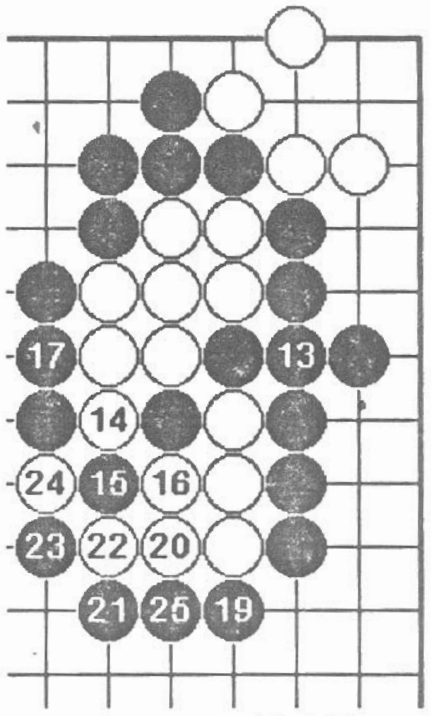
8 at 1

spiral5:



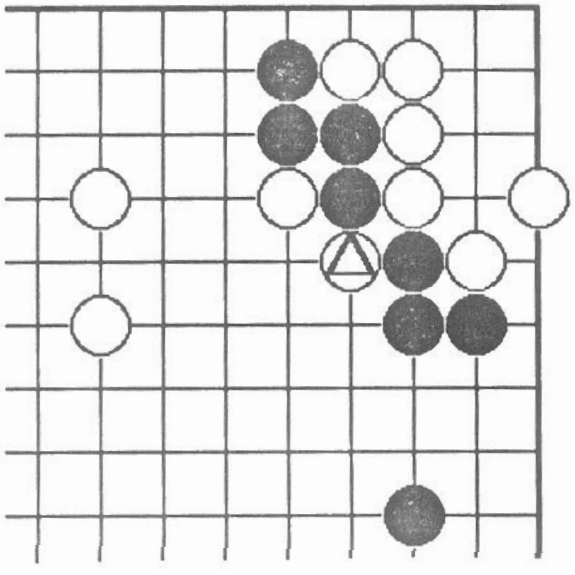


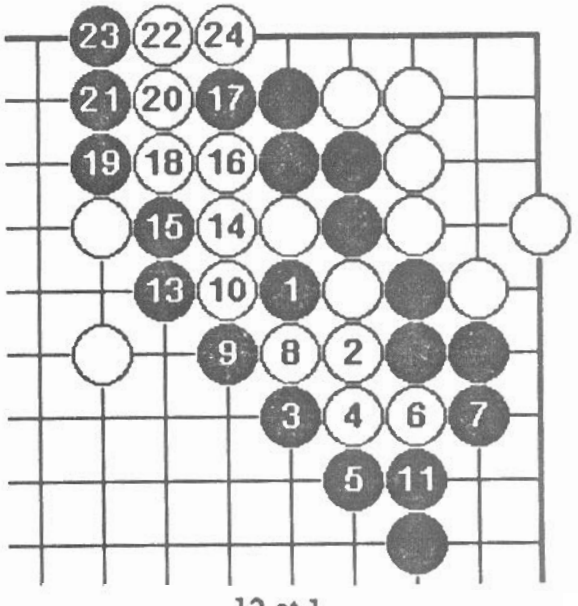
6 at 1



18 connects, 26 at 15

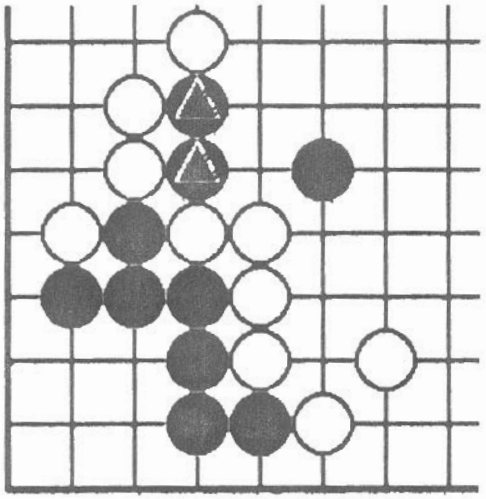
spiral6:

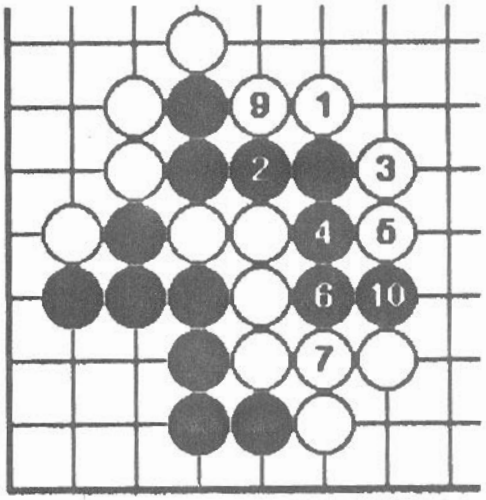




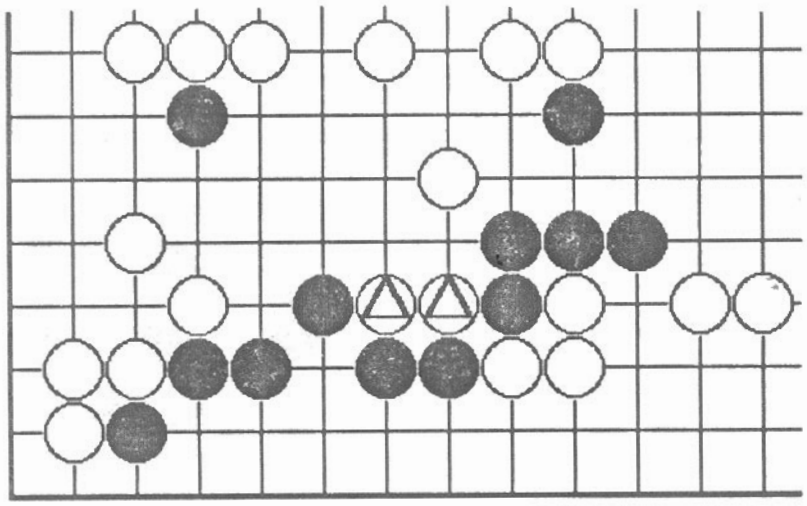
12 at 1

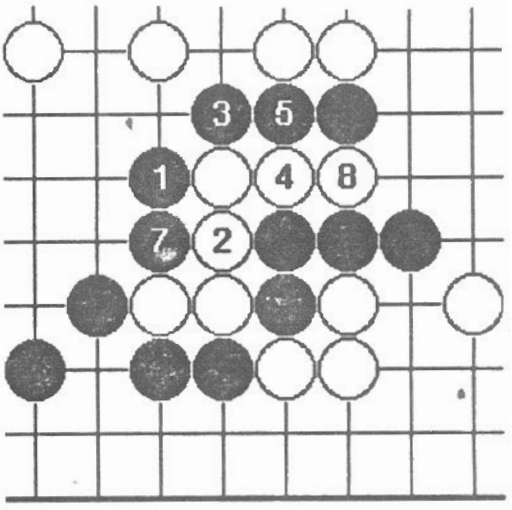
slapp1:





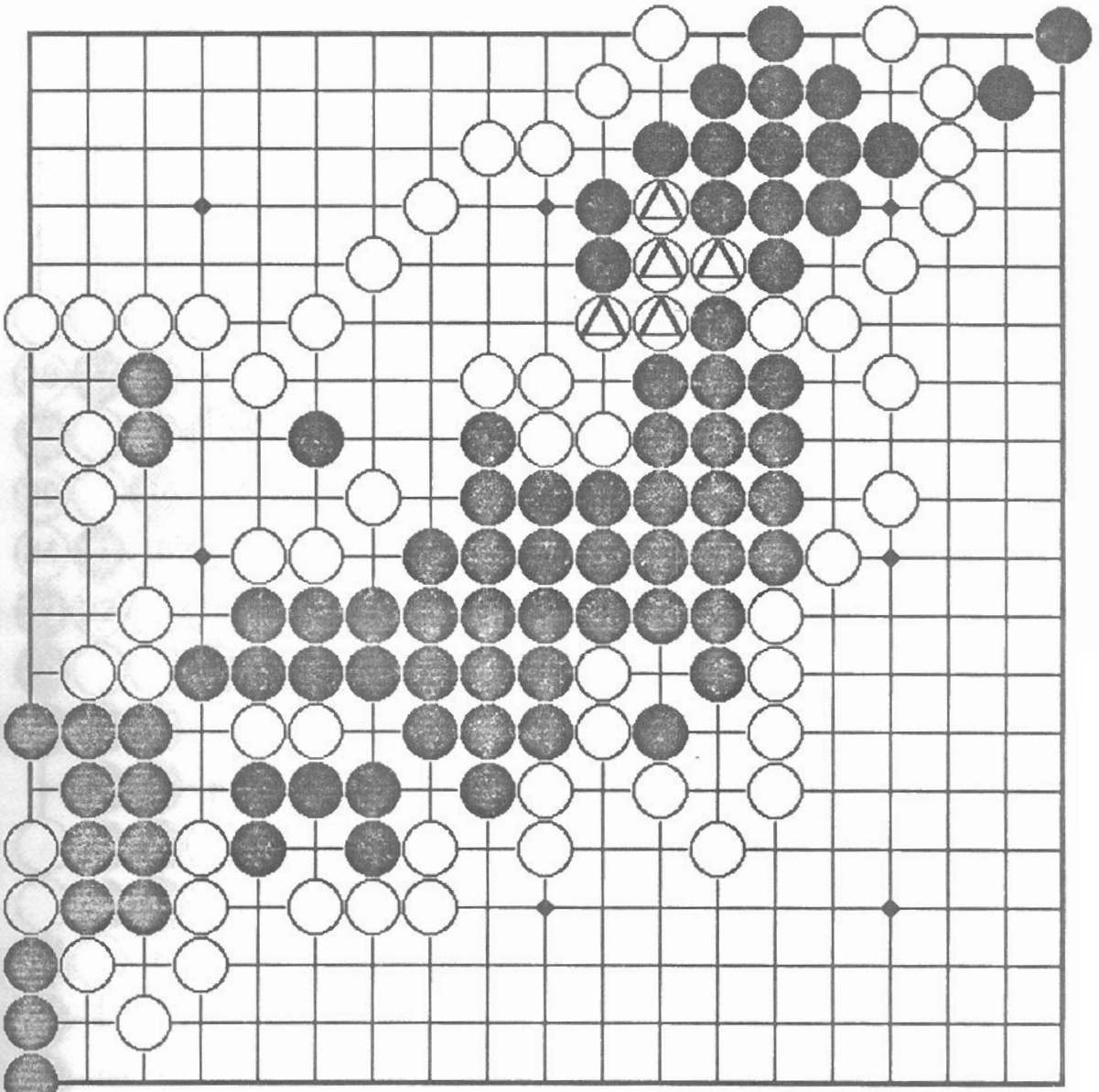
slapp2:



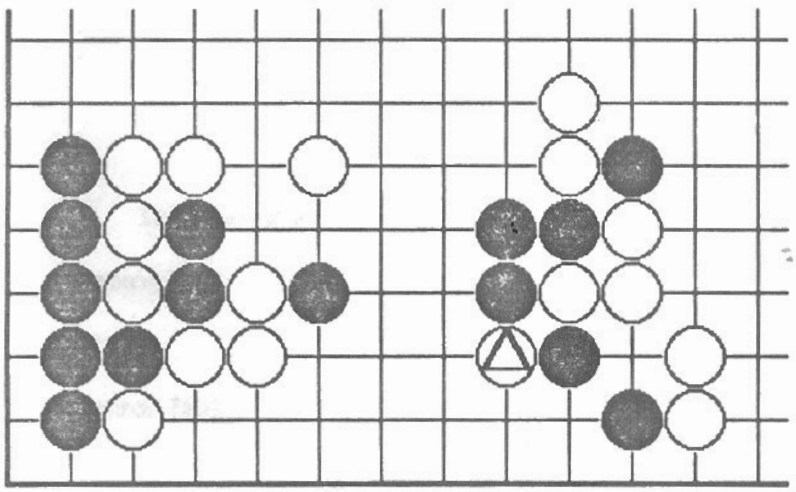


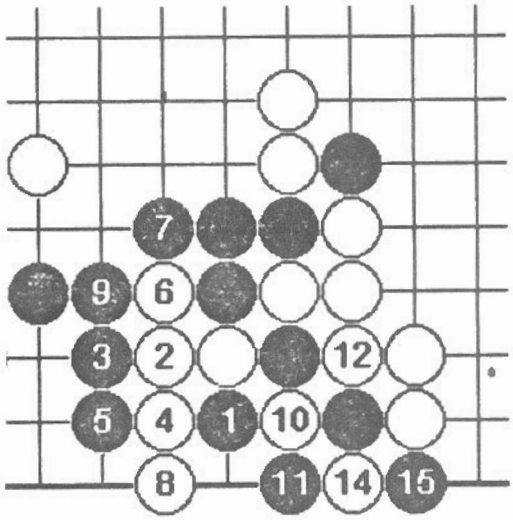
6: pass

japan:

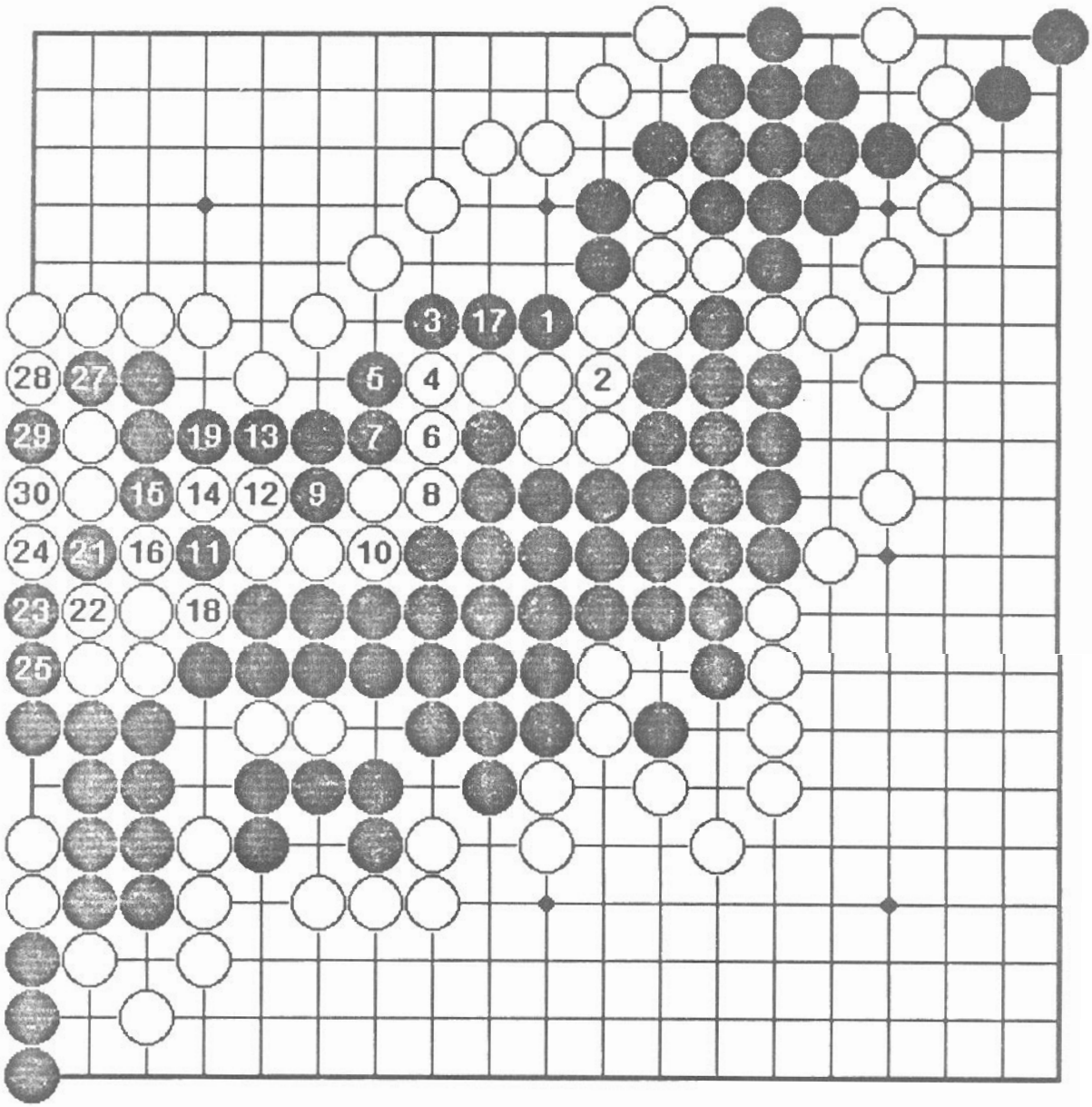


nose3:





13 at 10, 16: pass



20 at 11, 26 at 21