56

# A Tsume-Go Life & Death Problem Solver

by

Adrian B. Danieli

Submitted to the
Department of Electrical Engineering and Computer Science

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 19, 1998
[June 1998]

Author_______________________________________
Department of Electrical Engineering and Computer Science
May 19, 1998

Certified by_______________________________________
Robert Berwick
Thesis Supervisor

Accepted by_______________________________________
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

# Table of Contents

# List of Figures and Tables

# A Tsume Go
# Life & Death Problem Solver

by
Adrian B. Danieli

Submitted to the
Department of Electrical Engineering and Computer Science

May 19, 1998

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

## ABSTRACT

This thesis describes the techniques and problems one needs to consider when attempting to create a Tsume-Go life and death problem solver; a program (or portion thereof) that finds solutions to a sub-problem of the oriental strategy game Go. A Tsume-Go puzzle is typically a configuration of stones on a small section of the full 19x19 Go board in which the next player to move must make the correct, usually non-obvious, move in order to "live". If the player fails to make the crucial move, even perfect follow-up play cannot save the player's stones (assuming his opponent makes no mistakes as well). The Tsume-Go problem solver developed here can solve very simple problems.

Thesis Supervisor: Robert Berwick
Title: Professor, MIT Artificial Intelligence Laboratory

# 1 Introduction

## 1.1 Computer Games

People have tried to create programs to play conventional games of skill ever since computers were first built. The idea of writing a program that can play a game better than you, better than your friends, better than the average person, perhaps better than a grandmaster, has fascinated both researchers and common programmers alike. Maybe it is the naïve idea that if you can create a machine that plays at a level usually only achieved by extremely intelligent people, you yourself are an extremely intelligent person. Regardless of the motivation for these programs, game playing is one of the classic problems in artificial intelligence.

Researchers have invested much effort into improving the quality of play in many game programs. Simple (though non-trivial) games like Connect-Four have already been solved; in Othello, computer programs are considered superior to the best humans; in Backgammon, Checkers, and Chess, computers are among the strongest players – the best capable of defeating human world champions [Müller 95]. However, these glorious successes have not been achieved by computers in other games like Bridge, Chinese Chess, and Go, where they are far outplayed by their human adversaries [Plaat 96].

If we briefly consider some of the differences between Chess and Go, it is easy to see why Chess has been a more attractive research domain in the past. Go has a larger board size, more moves in a complete game, the branching factor is larger, and evaluating the current state of a game is more difficult since there is a very poor correlation between the game state and the number of stones on the game board or the amount of territory enclosed by a player's stones [Burmeister/Wiles 95]. Simply put, Chess is *easy* compared to Go.

## 1.2 Tsume-Go

Somewhere between these two extremes lies Tsume-Go. Tsume-Go is a more tractable sub-problem of Go. Instead of dealing with the vast 19x19 grid of a normal-sized Go board, Tsume-Go problems are localize Go games that occupy a much smaller region of the board. A player is faced with a certain arrangement of stones and he/she must discover and make the correct move (there may be more than one). If the correct move is chosen, that player is guaranteed a successful outcome assuming perfect subsequent follow up play. On the other hand, if the wrong move is chosen, the player cannot "win" this localized game if his/her opponent plays correctly. A successful outcome may be defined as "black to move and kill white unconditionally" or "white to move and live".

Tsume-Go puzzles share common features with both Go and Chess; move generation, stone placement, and the goal of capturing territory is the same as Go. However, the thinking skills required to do this are more Chess-like: a more limited number of moves and countermoves, searched until all lines of play are exhausted (or repeated) – and less like Go: global board analysis and pattern matching complemented with intuition about the game's "hot spots".

## 1.3 Searching

Game-playing programs all rely on a search algorithm that mimics the behavior of a human player. When a human tries to find a move, he/she typically looks ahead at a few moves, predicts the responses of his opponent (and subsequent replies) and chooses the most promising move. Programs do the same – searching each line of play until a specified depth or until no more moves are possible, at which point the current board state is evaluated. The first player to move attempts to maximize this score (to select the best move). In response, the second player attempts to do the opposite – minimize the first player's score (that is, maximize his/her own score). This process of backing up the value of the best move at each level is called minimaxing. This thesis discusses the implementation (and feasibility) of minimax search, as well as improvements on it like alpha-beta pruning, alpha-beta with transposition tables, and null-window alpha-beta searches like MTD(f) discussed later [Plaat 97].

## 1.4 Overview

The following is a short overview of the rest of this thesis. The intended reader of this thesis is a computer science student with a basic grasp of artificial intelligence search techniques. Section 2 briefly introduces the game of Go and Tsume-Go, no prior experience with or knowledge of this game is necessary.

Section 3 discusses the current state of computer Go, the differences between Chess and Go (from an artificial intelligence standpoint), and where Tsume-Go fits in.

Section 4 contains information about the Tsume-Go problem solver developed in this thesis, the development environment, the basic class/module hierarchy , and the user interface.

Section 5 details the node representation as implemented in this program. Specific algorithms for node creation, analysis, Ko detection, move generation, and board evaluation are explained.

After establishing a node structure and move generation, Section 6 introduces minimax and alpha-beta search. Our alpha-beta search is then augmented with a hash table. Finally, the search algorithm MTD(f) – memory enhanced test driver – is introduced. Section 7 compares these algorithms on some Tsume-Go problems.

Section 8 concludes this thesis with a discussion of the results from Section 7. Specific problem areas in this implementation are detailed, and possible directions for future work are presented.

# 2 The Basics of Go

Go is a two-player board game which originated between 2,500 and 4,000 years ago in China. Go is one of the oldest board games still actively played. It is an intellectual battle solely based on skill. In China, Japan, Korea, and Taiwan, Go shares a similar status as Chess does in western cultures [Burmeister/Wiles 95]. Part of Go's appeal is that it only has a few simple rules that can be learned quickly. Go's complexity arises from the huge number of possible board configurations, and the fact that nearly identical stone configurations can have vastly different meanings.

## 2.1 The Game Board

A Go board consists of a grid formed from the intersection of horizontal and vertical lines. The size of the board is generally 19x19, although quicker games can be played on 13x13 or 9x9 grids. The two players, black and white, alternately place a stone of their color on the intersection points of the board (including the corners and edges). A stone can only be placed on an empty intersections and it cannot move to another location. A stone is removed from the board only when it is captured. A player can pass if making no move is most advantageous. The game ends with two successive passes. The ultimate goal of Go is to capture more territory and prisoners than your opponent.

## 2.2 Liberties, Capturing Stones, and Suicide

The empty intersections that neighbor a stone are called the stone's liberties. A single stone may have up to four liberties (see stone D2 in **figure 2.1**). Stones placed on edges or corners only have 3 or 2 adjacent locations, respectively. If a stone has no liberties, it is captured and is removed from the game board (in **figure 2.2**, white surrounds the black stone). Capturing is the only time a stone is moved once it has been placed on the game board.

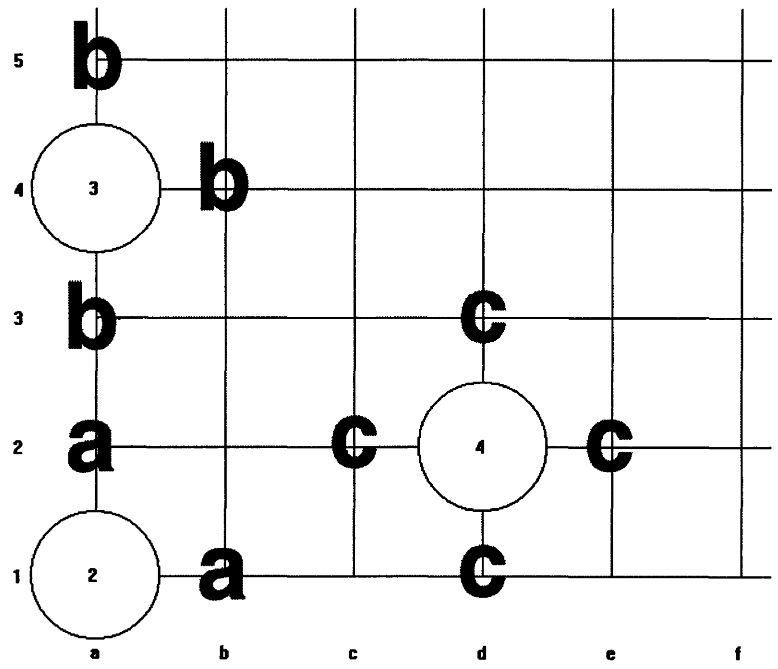


**Figure 2.1: The white stones at A1, A4, and D2 have 2, 3, and 4 liberties, respectively. The number of liberties a stone has equals the number of empty intersections near the stone.**

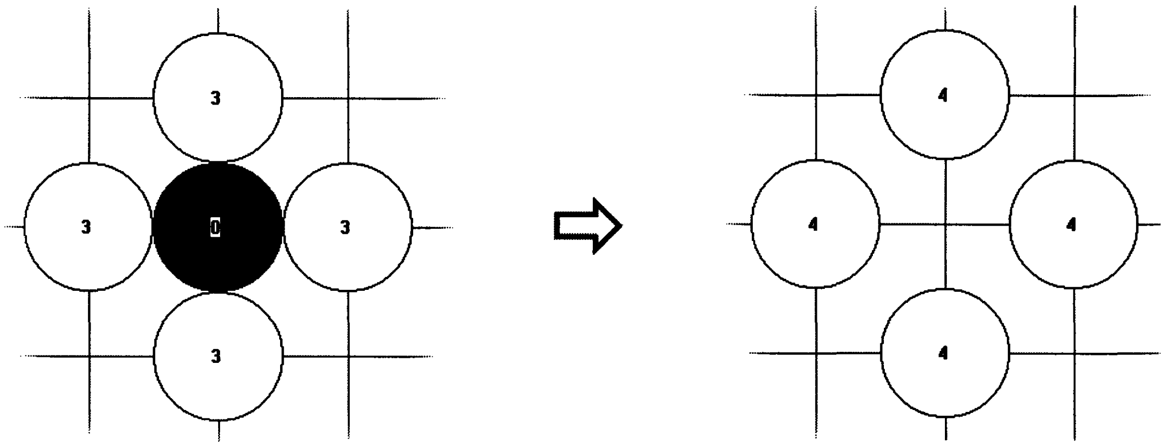


Figure 2.2: White reduces the black stone's liberties to zero. The black stone is removed from the board when this happens.

Suicide is not allowed in Go. A player cannot place a stone on an intersection such that it (or its group) has no free liberties. For example, white cannot play at intersection C3 in **figure 2.3**.

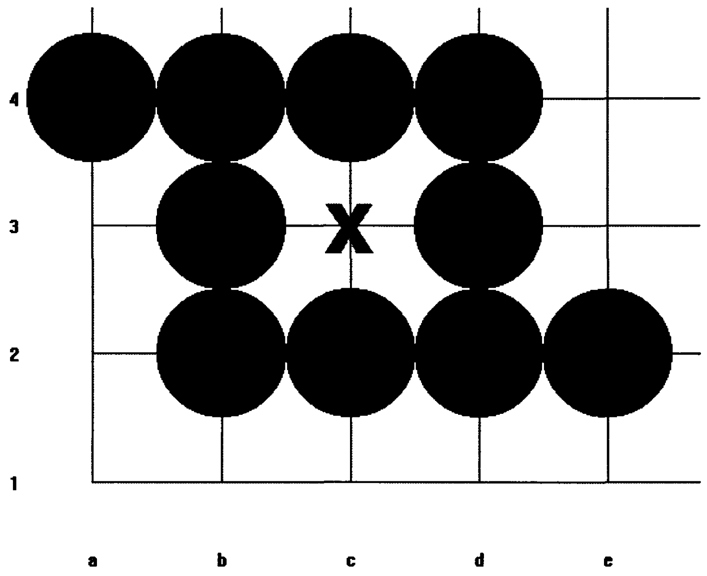


Figure 2.3: White cannot play a stone at C3 because it would have no liberties and would be immediately removed.

## 2.3 Groups (Strings)

Same-colored stones form groups (or strings) when connected to each other horizontally of vertically, but not diagonally. When a group of stones is formed, each stone in the group no longer has individual liberties. Instead, a liberty count is calculated for the group as a whole – the sum of the number of unique, empty, intersection points for each stone in the group. **Figure 2.4** shows two white groups with 7 liberties each, and one black group with 5 liberties. A group (or stone) with only one liberty is said to be in *atari*. An entire group can be captured by completely reducing its liberty count to zero as demonstrated in **figure 2.5** on the following page.

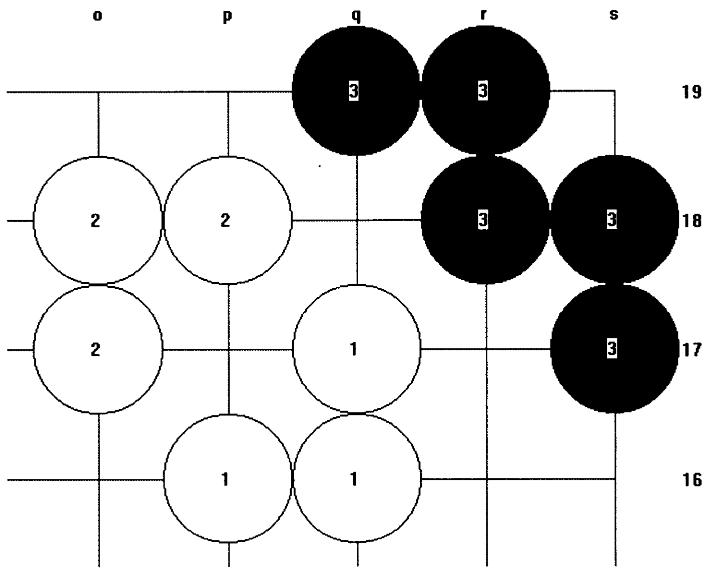


**Figure 2.4: There are two white groups, each with 7 liberties. The black group has 5 liberties.**

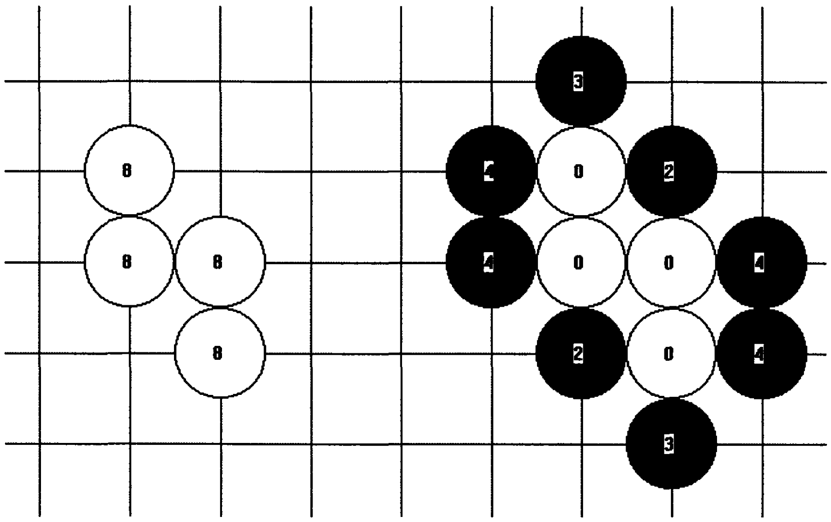


Figure 2.5: To capture a group, a player must reduce his/her opponent's group liberties to zero.

## 2.4 Eyes, Life, and Death

A group is said to have an eye when it completely surrounds an empty intersection point (see **figure 2.6**). A group with a single eye can be captured by first surrounding it, then filling the eye with a stone. The group needs to be completely surrounded before attempting to fill the eye, otherwise the suicide rule forbids the move. The reason the suicide rule is not applicable in the above case is because the opponent's captured stones are first removed, then liberties are computed for the stones on the board.

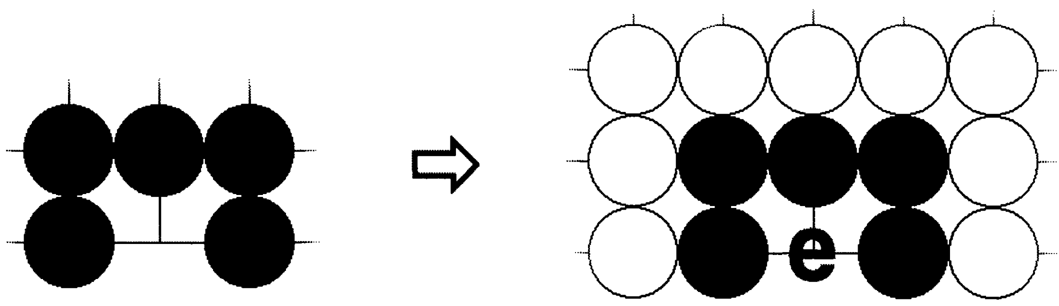


Figure 2.6: This black group has a single eye at location "e". It can still be captured by white if white first surrounds this group, then plays a stone in this eye space.

A group which has two separate eyes (not one two-intersection eye), as in **figure 2.7**, is considered absolutely alive since it cannot be captured no matter how many consecutive moves are given to the opponent. Similarly, a group is considered dead if there is no way of stopping it from being captured no matter who moves first.

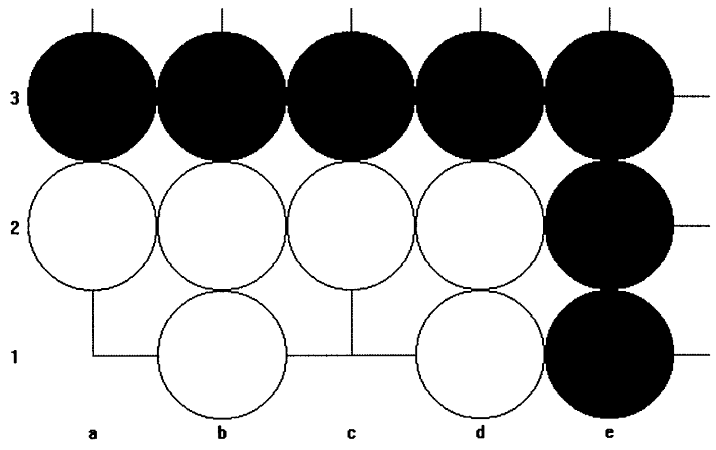


**Figure 2.7: This white group is considered unconditionally alive since black cannot capture it no matter what black does.**

## 2.5 Ko – Keeping Games Finite

A player cannot play a stone that causes a previous board position to be repeated. This is done to prevent infinite games. The basic Ko rule prohibits the simplest and most common Ko shape in **figure 2.8** – if a single stone captures a single stone, then no single stone may recapture it immediately [Jasiek 97]. In the figure, white captures the black stone. Black cannot not capture the white stone since it leads back to a previously seen board configuration.

A Ko fight occurs when a player loses a stone in a Ko situation similar to **figure 2.8**. Black cannot capture the white stone, but he/she may be able to threaten white

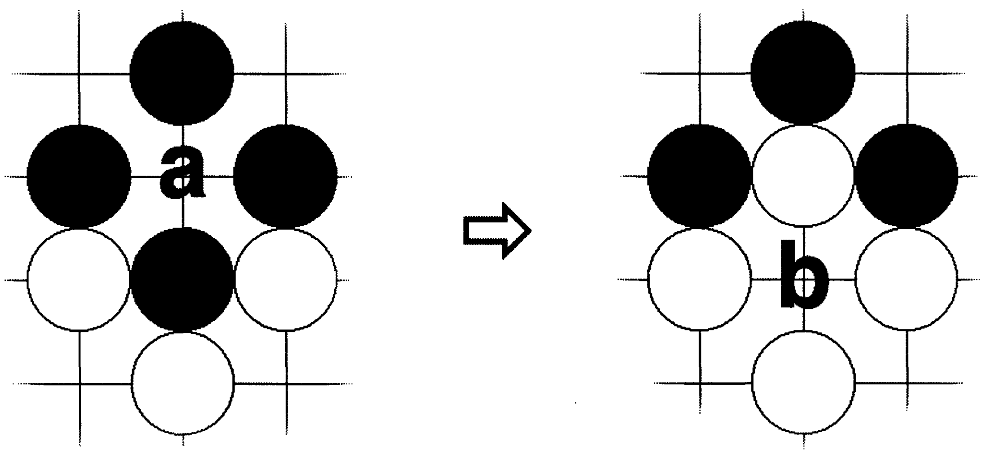


**Figure 2.8: This is the most common Ko shape. White captures black's stone by playing at "a", but black cannot recapture white's stone at "b" without playing somewhere else on the board first.**

somewhere else on the game board (not shown) forcing white's response. Now, black can capture the white stone in the figure since it is a new board configuration.

## 2.6 Tsume-Go

Tsume-Go puzzles are very localized Go problems (or battles) between white and black, usually of the form "Black to play and kill unconditionally." This means it is black's move and there is at least one legal move that will guarantee that white cannot save its stones, *even with perfect play*. For example, in **figure 2.9**, there are 7 legal moves for black: O1, P1, S1, Q2, R2, and "pass". Move S1 is the correct choice, although the counterproof of all the alternatives is beyond the scope of this thesis.

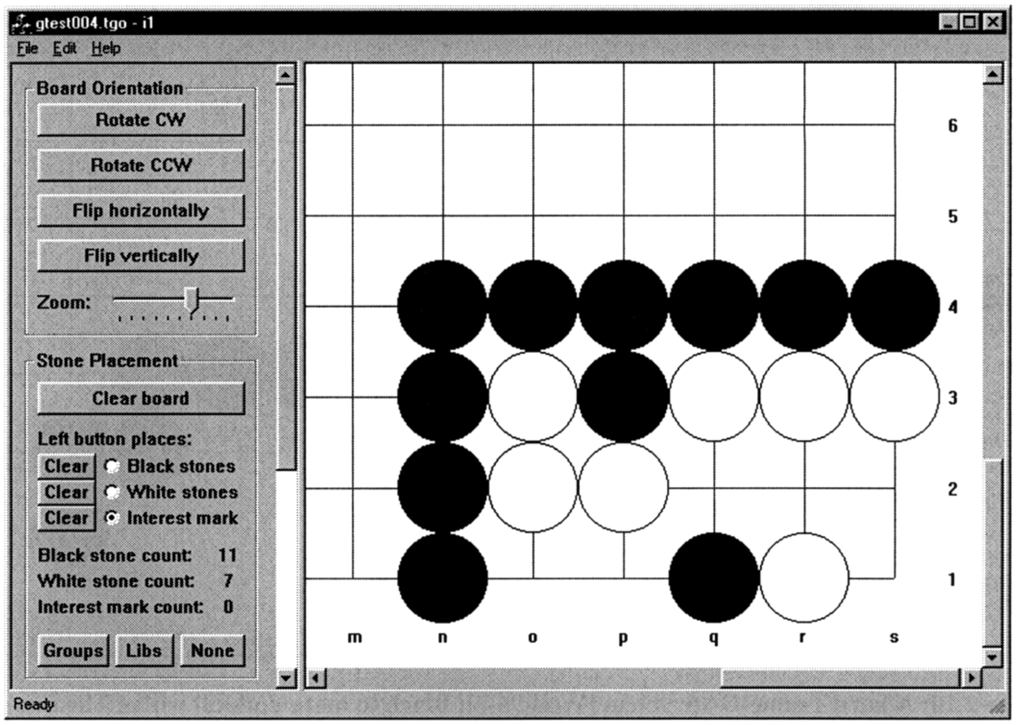
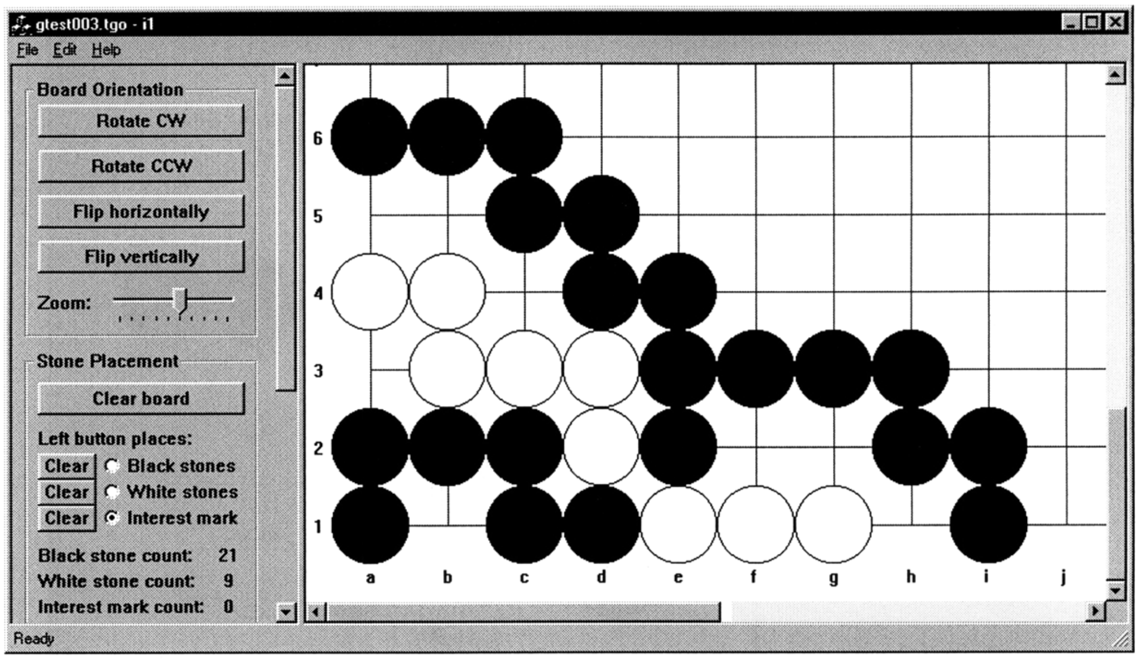


**Figure 2.9: An example Tsume-Go problem [Dyer 96]: Black to move and kill white. Here, the correct move for black is S1.**

Tsume-Go-like board positions do arise in normal Go games. If a player is especially good at identifying and solving these problems, he will have a great advantage over those who do not because he will not need to waste time worrying about saving or attacking dead groups.

**Figure 2.10** shows an example of a difficult Tsume-Go problem that requires black to make a very non-obvious move – filling its own group's eye space at B1 [Wolf 96-7].



Black stone count: 21
White stone count: 9
Interest mark count: 0

**Figure 2.10: A hard Tsume-Go problem [Wolf 96-7]: Black to move and kill white. The non-obvious solution is for black to fill its own eye space at B2 – a move most efficient Go programs will overlook.**

# 3 The Challenge of Computer Go

## 3.1 The Current State of Go and Tsume-Go

Computers that play Go have not had nearly as much success as their counterparts in Checkers, *Awari*, Nine Men's Morris, or Chess. Despite extremely simple rules, Go resists the bruce force search techniques that have worked successfully for these other games [Müller 95].

The best programs have garnered only slightly more than a beginner ranking on the human ranking scale. **Figure 3.1** shows the human ranking scale for Go players. Someone just learning Go usually has a rank of 30 to 20 Kyu, which decreases as the player's skill improves. After 1 Kyu, the amateur ranks progress from 1 Dan to 6 Dan. The professional Go ranks are on a separate scale, progressing from 1 Dan to 9 Dan [Mariano 97].

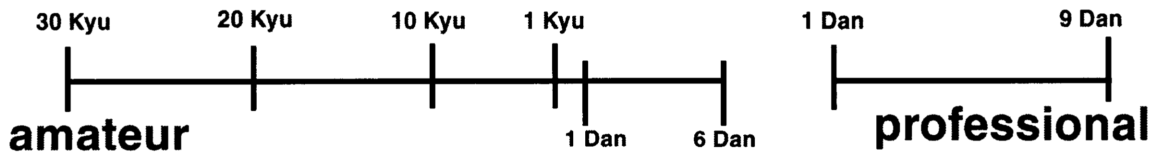


**Figure 3.1: The amateur scale ranges from 30 Kyu to 1 Kyu, then 1 Dan to 6 Dan. Professional ranks begin at 1 Dan and continue up to 9 Dan (on a separate scale). The best computer Go programs rank between 10-15 Kyu, while specialized Tsume-Go programs achieve near Dan-level performance.**

General computer Go programs rank somewhere between 10-15 Kyu, but are often easily defeated by human opponents of the same skill after the human has learned the computer's style of play. Specialized Tsume-Go programs achieve higher ratings nearing Dan-level performance. This thesis explores the design and construction of a Tsume-Go problem solver that aspires to amateur level performance.

## 3.2 Differences between Chess, Go, and Tsume-Go

**Table 3.2** contains a comparison of the features between Chess, Go, and Tsume-Go. These are the main reasons why the brute-force search techniques used in Chess do not work at a similar level in Go. A regulation chess board has 64 squares, a typical game lasts 80 moves, and a player has 35 possible moves, on average, for a given board position. In Go, there are 361 intersections where stones can be played, a typical game has more than 300 moves, and the number of moves a player could consider for a given board position exceeds 200, on average [Burmeister/Wiles 95]. Chess programs are often able to reduce the apparent branching factor further by evaluating moves by certain pieces first, thus allowing more tree-pruning opportunities. In Go, there is only one kind of piece – a stone – that is placed once and only removed if it is captured.

The board changes gradually from move to move in Go unlike Chess. There is no good correlation between the number and quality of pieces and the final outcome of a Go game. No simple evaluation function seems to exist. In Chess, the end of a game is easy to determine: checkmate or resignation. It is not always clear (to beginners) when a Go game should be called. Naturally, one would stop when making new moves cannot improve one's score, but this is a difficult determination for Go programs to make accurately.

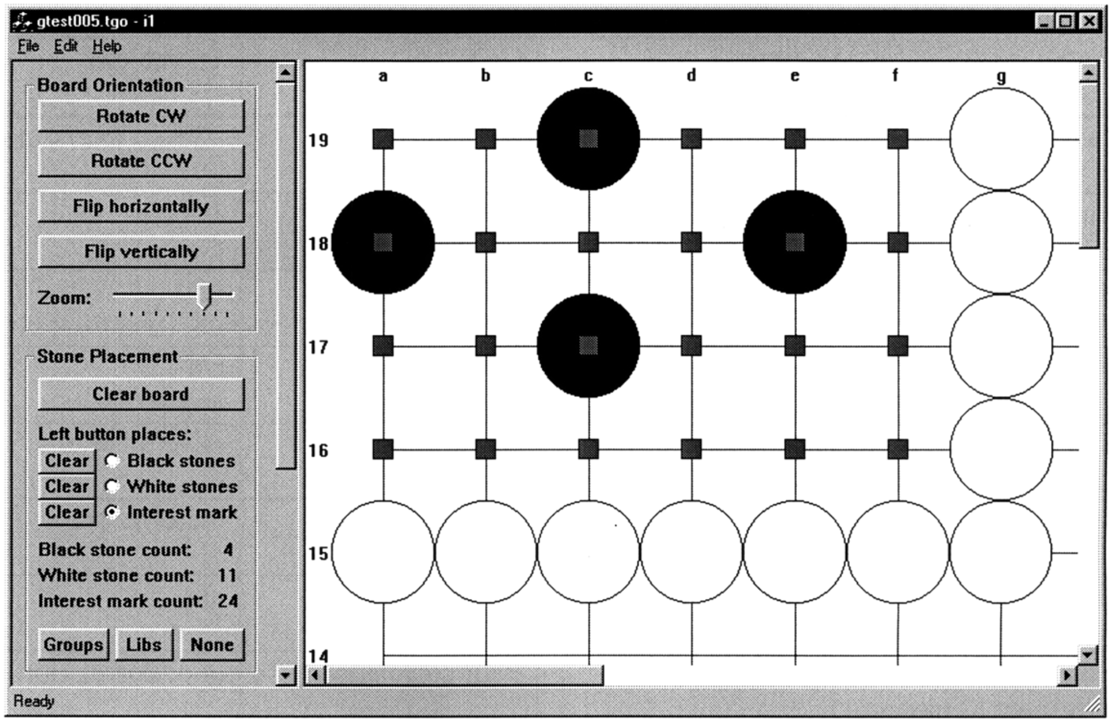
In essence, Go programs lack both a good evaluation function and a human's intuition about which stone groups can be captured or not, as well as which player has the advantage. Go's search space is simply too large to perform brute-force searching without a good evaluation function to measure progress.

| | Chess | Go | Tsume-Go |
|---|---|---|---|
| Region of play | 64 squares | 361 intersections | ~20-80 intersections |
| Number of pieces | 6 | 1 | 1 |
| Branching factor | ~35 | ~200 | ~30 |
| Moves per game | ~80 | ~300 | n/a |
| End of game determination | checkmate | counting territory | alive or dead |
| State of board | changes rapidly | incremental | incremental |
| Horizon effect | grandmaster level | beginner level | n/a |
| Board evaluation w/respect to pieces | good correlation | poor correlation | poor correlation |
| Programming approaches | tree search, good evaluation criteria | not amenable to tree search | specialized tree search |

**Table 3.2: A comparison of features between Chess, Go, and Tsume-Go. [based on a table by Burmeister/Wiles 95]**

## 3.3 The Middle Ground: Tsume-Go

Tsume-Go problem solvers sit somewhere between Chess programs and Go programs. Tsume-Go inherits Go's simple rules but it is amenable to chess-like tree searches becasue it is only concerned with a small, localized area of the whole game board. Still, Tsume-Go life and death problem solver can face extraordinarily complex and deep problems (from a computational standpoint) even on relatively simple problems. Sparse Tsume-Go problems like the one posed in **Figure 3.3** can take many hours to solve, even with advanced search techniques and heuristics. If a player's group escapes the local battle the entire Go board is brought into the picture making further search futile.



Left button places:
Black stones
White stones
[✓ Interest mark]

Black stone count: 4
White stone count: 11
Interest mark count: 24

Ready

**Figure 3.3: A difficult Tsume-Go problem for computers because of the high branching factor. The small squares are "interest points" – the region of moves the computer will search through.**

## 3.4 Existing Work

GoTools is perhaps the strongest Tsume-Go problem solver in the world today. It was written by Thomas Wolf, and has received the 1997 Award of the Japanese Computer Go Forum. Some have credited GoTools with Dan-level performance. More information about this program can be found at http://alpha.qmw.ac.uk/~ugah006/gotools/ [Wolf 97].

Dave Dyer has also written a life & death problem solver, currently unreleased to the public. His website contains valuable resources for all Go programmers: http://www.andromeda.com/people/ddyer/go-program.html [Dyer 96].

# 4 A Tsume-Go Problem Solver

This thesis has two main purposes: (1) to explore the applicability of tree-search algorithms in Tsume-Go that have proven successful in other games like Chess, Othello, and Checkers; and (2) to give the reader explicit algorithms and code examples developed from the exploration of these techniques. It is my hope that this thesis will lower the initial learning curve required to implement these algorithms by taking away some of their mystique.

This thesis presents my basic progression from the initial idea of creating a Tsume-Go life & death problem solver, to a real implementation that can actually solve some simple problems.

## 4.1 Development Environment

In the hopes of creating a real program that can run almost anywhere today and that is arguably user-friendly, this Tsume-Go program was developed for the Microsoft Windows 95/NT operating system on the Intel x86 platform. An additional advantage of this choice was the ability to use a decent compiler – Microsoft Visual C/C++ – with modern debugging tools. The Microsoft Foundation Class libraries (MFC 4.2) were used as the C++ wrapper for the Win32 system calls.

The development system was a Pentium II/266mhz MMX machine with 64MB of RAM, although the program was tested on many Pentium-class machines with less memory. Even these machines proved inadequate for solving certain Tsume-Go problems which are considered relatively easy by human standards.

## 4.2 Classes / Modules

The program was developed using MFC's Document/View Architecture as an SDI (single-document interface) application. **Table 4.1** lists the eight major classes used in this program.

| Class / Module | Description |
|---|---|
| **`CI1App`** | the application object class (Windows-specific) |
| **`CMainFrame`** | the main frame window class (Windows-specific) |
| **`CViewBoard`** | the game board "view", user-interface code responsible for drawing and updating the game board (Windows-specific) |
| **`CViewCtrl`** | the control panel "view" (Windows-specific) |
| **`CGoDoc`** | the application's document, contains code to perform minimax search, alpha-beta, alpha-beta w/memory, and MTD(f). |
| **`goboard`** | the Go board representation used in the user interface |
| **`node`** | a Go board node, optimized for use in the various searches (includes move generation & evaluation functions) |
| **`hashtable`** | a Go node-specific transposition table |

**Table 4.1: Major modules in this Tsume-Go program.**

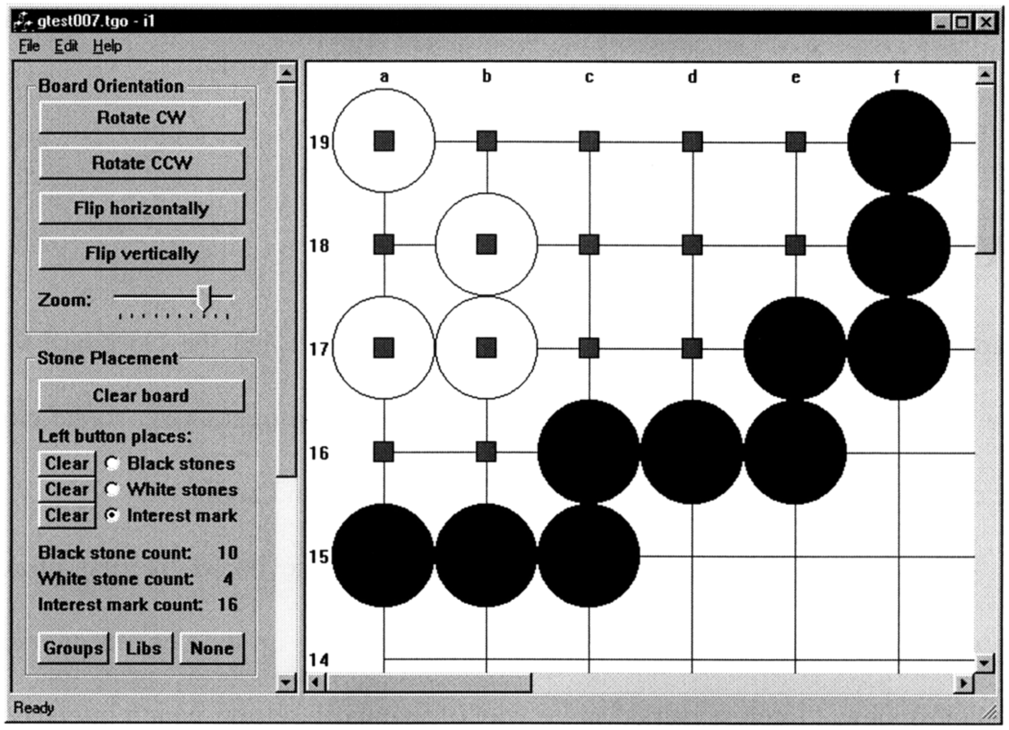
The classes **`CI1App`**, **`CMainFrame`**, **`CViewBoard`**, and **`CViewCtrl`** are Windows specific classes used in single-document interface design. **`CGoDoc`** is the application's document, and contains the code that performs minimax searching, alpha-beta pruning, alpha-beta with memory, and MTD(f) – a memory-enhanced test driver algorithm (see Section 6).

## 4.3 User Interface

The application is divided into two views: a control panel and the Go board view (see **figure 4.2**). The control panel contains the following board operations: rotate clockwise, rotate counter-clockwise, flip horizontally, flip vertically, and zoom.

A user may place stones or interest marks on the board by first clicking the appropriate radio button in the control panel, then clicking on the desired intersection(s) in the board view. "Interest marks" are a way of manually setting which intersections are of interest for a particular Tsume-Go problem (clearly, any intersections beyond the black stones in **figure 4.2** are not relevant).

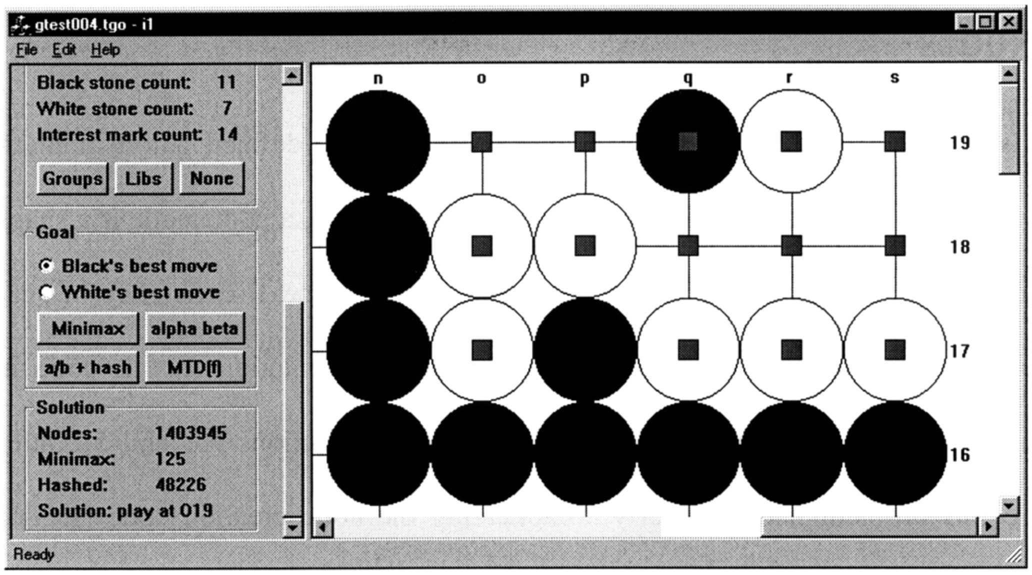
The control panel maintains counts of the number of black stones, white stones, and interest marks on the game board. Clicking on the "Groups" button will label each group in the go board view with a unique number. Clicking on the "Libs" button will label each group with its liberty count.



Left button places: [✓ Interest mark]
Black stone count: 10
White stone count: 4
Interest mark count: 16

**Figure 4.2: Screen shot from the Tsume-Go program. The control panel view is on the left, the Go board view on the right. The small squares in the board view are interest marks.**

At the bottom of the control view, one selects which player is to make the next move, and then presses the appropriate search-method button. The program attempts to solve for that player's best move. In **figure 4.3**, after searching to a depth of 16, the program believes O19 will guarantee black a score of at least 125 (the evaluation function here is 127 minus the number of white stones). 1,403,945 nodes were searched in this process. In this example, the search was performed by the alpha-beta algorithm augmented by a hash-table. 48,226 unique nodes were stored in the table. This proposed move is incorrect since a search depth of 16 is not sufficient to analyze all possible paths to their conclusion.


Black stone count: 11
White stone count: 7
Interest mark count: 14

Goal
(•) Black's best move
( ) White's best move

Solution
Nodes: 1403945
Minimax: 125
Hashed: 48226
Solution: play at O19

**Figure 4.3: After searching for Black's best move to a depth of 16 using alpha-beta with a hash table, the program believes that playing at O19 will yield a score no worse than 125. (The evaluation function is 127 minus the number of white stones remaining) [Dyer 96].**

# 5 Nodes

This section covers the first essential object that must be defined before implementing any game-playing machine using tree-search. In Tsume-Go, a game node is a snapshot of the entire Go board, along with which player is moving next. Efficiency of space and time must be a priority when creating the node representation as this code may be executed millions of times during a deep tree-search. Sections 5.1 and 5.2 attempt to tackle this problem without overly complicating the node creation code (section 5.3), move generation (5.4), and board evaluation (section 5.5) procedures.

## 5.1 Knowledge Representation

A node contains a snapshot of a Go game. **Table 5.1** summarizes the member variables in class *node*, and a description of what each holds. The game board is thought of as a 20x20 array of intersections, the 0th-row and 0th-column are unused. A "move" is the *x* and *y* indexes into this array, with the exception of two special cases: a pass (0,0) and no-move (0,1). The move leading to the root node is a "no-move".

Since there are only two players in a Go game, the color of the player who moves next is stored in the boolean variable **`mColor`**. Black is defined as 0 (false), white as 1 (true). This variable is used to generate appropriate moves for that player given the current state of the game board. **`mPossibleMoveX[]`** and **`mPossibleMoveY[]`** are arrays which store the possible moves for the current player (see section 5.4 for details). In the current implementation, these are fixed-sized, 64-byte arrays. Thus, a maximum branching factor of 64 is allowed, much more than any Tsume-Go problem requires. The member variable **mPossibleMoveCount** holds the actual number of possible moves (after move generation).

| Variable | Type | Description |
|---|---|---|
| **mColor** | boolean | color of player making next move |
| **mParent** | node pointer | this node's parent |
| **mDepth** | integer | this node's depth in the search tree |
| **mMoveX** | byte | x-coordinate of move leading to this node |
| **mMoveY** | byte | y-coordinate of move leading to this node |
| **mPossibleMoveCount** | integer | number of possible moves |
| **mPossibleMoveX[]** | 64-byte array | x-coordinates of possible moves |
| **mPossibleMoveY[]** | 64-byte array | y-coordinates of possible moves |
| **mKOthreats** | integer | number of KO fights this player can win |
| **mCompressedBoard[]** | 25-dword array | compressed version of 19x19 game board |

**Table 5.1: Member variables of class *node*.**

A pointer to this node's parent is maintained in **mParent**. In addition, the node's depth in the search tree is stored in **mDepth**, and the opponent's move which led to this board position is stored in **mMoveX** and **mMoveY**, both bytes. Some Tsume-Go problems can only be "won" by winning one or more Ko fights. If the variable **mKOthreats** is greater than zero, then up to *mKOthreats* duplicate board positions are allowed to occur for this player – that many Ko fights to be won.

A compressed version of the game board is stored in a 32-bit word array called **mCompressedBoard[]** of length 25. Each intersection can be empty, black, or white; so only 2-bits are needed per location. A grid of 20x20 intersections can be stored in 800 bits, or 100 bytes, or 25 double words. The reason for compressing the game board is to speed up node comparisons (necessary for Ko detection, see section 5.3). The data maintained in a single *node* entry is less than 256 bytes total.

## 5.2 Static Workspace

When the root node is created, certain static variables in the *node* class are initialized. The color of the root node determines the evaluation function. This information is stored in the boolean variable **`sEvalAsWhite`**. Since the root node color is constant for the duration of any given tree-search, there is no need to replicate this information in every node. Similarly, the maximum recursion depth, **`sMaxDepth`**, is constant throughout a search.

The root node of the search tree is created based on the current Go board state in the user interface modules. The "interest marks" are stored in a static 20x20 boolean array called **`sInterest[][]`**, which remains constant. This information is used by the move generator to create only relevant moves. **Table 5.2** summarizes the static variables in the *node* class.

| Variable | Type | Description |
|---|---|---|
| **`sEvalAsWhite`** | boolean | evaluate leaf nodes from white's perspective (is the root node white?) |
| **`sMaxDepth`** | integer | maximum recursion depth |
| **`sInterest[][]`** | 20x20 boolean array | interest mark settings for each interesection on the Go board |
| **`sWorkBoard[][]`** | 20x20 dword array | uncompressed workboard |

**Table 5.2: Static member variables of class *node*.**

In order to faciliate move generation further, the compressed game board at every node is uncompressed into the 20x20 double-word array **`sWorkBoard[][]`** that maintains extended information about the Go board for the move generation routines. Each entry in this table maintains the following information (**Table 5.3**, **Figure 5.4**): if the intersection is occupied by a stone, what color the stone is if it exists, the group this stone belongs to, the number of liberties this group has, and a few bits so the move generation and analysis routines do not repeat work.

| Bit positions | Number of bits | Field description |
|---|---|---|
| 0-8 | 9 | group number (range 0-511) |
| 9-17 | 9 | group liberties (range 0-511) |
| 18 | 1 | occupied (set if occupied) |
| 19 | 1 | color (set if white, not set if black) |
| 20 | 1 | clean/dirty bit (set if clean, for board analysis) |
| 21 | 1 | checked/unchecked bit (set if checked, board analysis) |
| 22-31 | 10 | unused |

**Table 5.3: Information maintained in each entry of the 20x20 work board structure. Since there are only 361 intersections on a Go board, 9 bits each for the group number & liberties will suffice.**

| | | | | | | |
|---|---|---|---|---|---|---|
| unused | x | d | c | o | group liberties | group number |
| 10 bits | 1 | 1 | 1 | 1 | 9 bits | 9 bits |

**Figure 5.4: A workboard single table entry.**
**"x" = check/unchecked bit, "d" = clean/dirty bit,"c" = color bit, and "o" = occupied bit.**

## 5.3 Node Creation, Analysis, and Ko Detection

The root node is created by passing a reference to the user-interface's Go board, the player who is making the next move, the maximum tree depth, and the number of Ko fights this player is allowed to win. This information is expanded into the **`sWorkBoard[][]`** and **`sInterest[][]`** arrays. A board analysis is performed to remove dead stone groups (if any), then the move generation algorithm is called (see Section 5.4).

```
CONSTRUCTOR node(board:UIGoBoard, maxdepth:INT, color:BOOL,
                 KOthreats:INT)
      1. initialize node member variables
            a. parent is null
            b. save node color
            c. node depth is zero
            d. save KOthreats
            e. last move does not exist
      2. initialize static information (constant for the tree-search)
            a. save root node color
            b. save maximum recursion depth
      3. generate sWorkBoard[][] from UIGoBoard
      4. generate sInterest[][] from UIGoBoard
      5. analyze go board
      6. generate moves
      7. compress game board
```

The board analysis routine contains three major parts. First, it scans the entire go board looking for groups. When it finds a stone that has not been explored yet, it performs a recursive search starting at the current location that progresses to adjacent stones in order to find all the members of this group. As this search progresses, we keep track of the number of unique, empty intersections adjacent to this group – this final value is the total number of liberties of the entire group.

After we've discovered every group on the board and its liberty count, we need to remove any of *the current player's* stone groups with zero liberties. Remember, the *opponent* made the move leading up to this board position – which may well have captured one of the this player's strings. Finally, if any groups were removed, the entire board is rescanned to reflect changes in liberties that need to be propagated due to this removal.

```
BOOL analyze(void)
      1. scan entire Go board for stone groups
            a. found a new group, analyze-newgroup()
            b. update all members so that they reflect the group's
               liberties
      2. scan board for this player's groups with zero-liberties
            a. remove group from board
      3. if groups were removed, rescan entire Go board as in step 1.
      4. return TRUE if enemy stone groups exist on the board
```

```
INT analyze-newgroup(xpos:INT, ypos:INT, group:INT, color:BOOL)
      1. check the four neighboring intersections of (xpos,ypos)
            a. if position is off the game board, or it has already
               been examined, skip this intersection
            b. mark location as previously examined
            c. if this position is empty, increment the liberty count
            d. if this intersection is occupied by our stone color,
               then recurse on this new location
      2. sum the liberty counts returned by each neighboring
         intersection and return this value
```

To explore children of a node, the search algorithm calls the **getchild()** member function of a node *n*. This looks to see if node *n* has any unexplored, possible children. If it does, a new node is created by passing a reference to the parent and the attempted move to the node constructor. A child node is created in a similar fashion to the root node – a stone of the appropriate player is placed on the board, and the board is analyzed. If this child node is a leaf, a static evaluation is performed. Otherwise, move generation proceeds. Now **getchild()** has a possible child, but we must check for Ko situations. If all is well, the child node is returned. If this fails and node *n* has no more children, a **NULL** value is returned by **getchild()**.

```
NODE* getchild(void)
      1. If there are no possible moves from this board position
         remaining, return NULL.
      2. Create a new node from the next move in the possible moves
         array. Return if NULL.
            a. Find this node's grandparent (two board positions ago)
            b. Compare these nodes. If they are the same, check if we
               can win any more KO fights.
                  i.  Yes? Return this node.
                  ii. No? This move isn't possible, repeat step 2.
            c. Nodes are different; find grandparent's grandparent.
            d. Recurse to step b until we run out of grandparents.
      3. No KO situation found, return this node.
```

## 5.4 Move Generation

If one disregards Ko situations for a moment, a list of possible moves can be generated by scanning the Go board for empty intersections. A particular empty intersection is a valid move for player A if any one of the neighboring intersections meets one of these conditions:

- the intersection is empty (at least one liberty will exist for this stone)
- the intersection is occupied by a stone of Player B that has only one liberty (an enemy group will be captured, leaving at least one liberty for this stone)
- the intersection is occupied by a stone of Player A that has more than one liberty (this stone will become part of a larger group, with at least one liberty)

Also, passing (move 0,0) is always a valid move. However, two successive passes marks the end of the game. Adding passes increases the branching factor $b$ by one, but $(b+1)^d$ is still $O(b^d)$, where $d$ is the average search depth of a problem. Correctness does not make finding the solution exponentially worse.

After move generation, the arrays **`mPossibleMoveX[]`** and **`mPossibleMoveY[]`** contain possible moves from the current board position. Some of these moves may be refuted due to Ko rules, as described in the previous section.

## 5.5 Board Evaluation

When solving a Tsume-Go problem, a program can have two slightly different goals depending on what kind of problem is being solved. One goal is absolute life or death for a certain player – only one color remains on this localized region of the board, or rather no further plays by the enemy can result in an alive group. The other goal is finding the best move for a player, which could result in *seki*, a stalemate situation where no territory is awarded.

In Tsume-Go problems where you know the solution (e.g., black to move and kill white unconditionally), the first goal is applicable – all reasonable lines of play must be searched completely until a solution is found which guarantees a win. In a general computer Go player, the other alternative is more appropriate. The program doesn't know if absolute victory is possible. Instead, it wants to find the move that maximizes its score, whatever that may be. In this case, the Tsume-Go routine acts more like a general purpose Go problem solver.

This thesis attempts to develop a program that will solve Tsume-Go puzzles with known solutions. We can use this simple evaluation function:

```
board value = 127 - enemy stone count
```

We know Tsume-Go problems of the form "white to move and kill" will have a final minimax value of 127. Armed with this knowledge, we can substantially improve on the alpha-beta pruning algorithms (see Section 6.4, 6.5).

## 5.6 Improvements

One of the hardest things to do in Tsume-Go is leaf detection. When should the computer stop searching? Benson's proof sets forth the criteria for unconditional safety of a group, but problem solvers can stop searching sooner – as soon as a group is safe assuming perfect play [Müller 95].

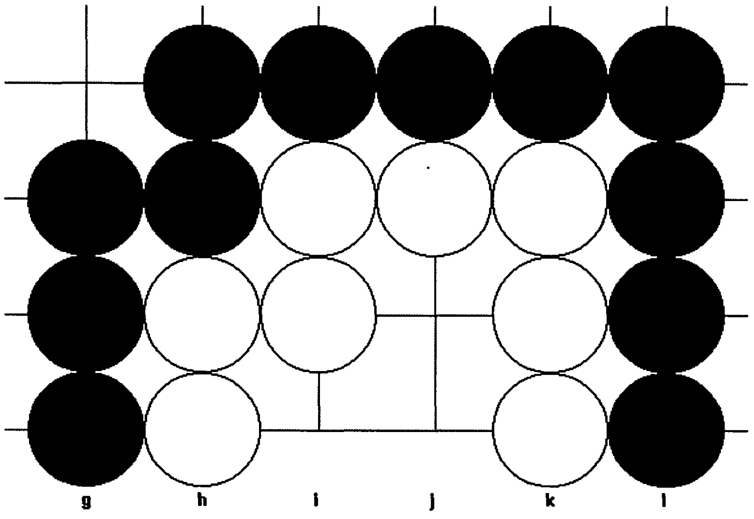


**Figure 5.5: An example of an eye-space of size 3.**

Existing Tsume-Go programs attempt to classify the state of an eye (see Section 2.4) surrounded by a single chain either algorithmically (GoTools [Wolf 97]) or through the use of a large "eye shape" lookup table (Dave Dyer [Dyer 96]). The purpose of this algorithm or database is to determine, given an eye-shape such as the one in **figure 5.5**, what the outcome is and the move that leads to it, *without explicit searching*. This leads to large performance gains by significantly decreasing the number of nodes searched.

# 6 Adversarial Search

This chapter explains the search algorithms that can be used to solve some Tsume-Go problems. Section 6.1 covers minimax search, the basic method for deciding which move to make in a given situation. Alpha-beta pruning (Section 6.2) is an idea that significantly reduces the number of nodes that need to be explored to arrive at the minimax value. This is accomplished by aborting losing lines of play.

While the basic Alpha-beta algorithm is described in many A.I. textbooks, it is seldom used in real world programs. Section 6.3 presents a memory-enhanced version of the alpha-beta algorithm. By storing previously explored nodes in a hash table, alternate lines of play that lead to identical positions can be quickly retrieved without re-exploration. Sections 6.4 and 6.5 explore a new minimax search algorithm, MTD(f), that performs zero-window alpha-beta searches and its applicability to Tsume-Go.

## 6.1 Minimax Search

Minimax search simulates one way humans decide on which move to make next in a game. Player A examines a particular move, then guesses Player B's response, considers his/her reply to this move, and so on. After analyzing every move in this fashion, the move which appears to lead to the most promising board position is selected [Winston 92].

Programs perform the minimax algorithm by doing a depth-first search through the game-tree to a specified, maximum depth. Unlike breadth-first or best-first searches, depth-first search requires very little memory space, and its path through the search space is easy to follow. When a program reaches the end of a line of play, either the maximum search depth has been reached or no legal moves can be made, the current node is assigned a value from a static evaluator (see Section 5.5).

Player A, the maximizer, wants to choose a move which leads to a board position of maximum value. Player B, the minimizer, wants to choose a move which leads to a board position of minimum value. The maximizer decides the best move to make at all even levels in the tree, the minimizer decides *its* best move at odd levels in the search tree. The root node has a depth of zero. Scores are backed up from leaves to the root node. **Figure 6.1** shows the minimax algorithm in progress.

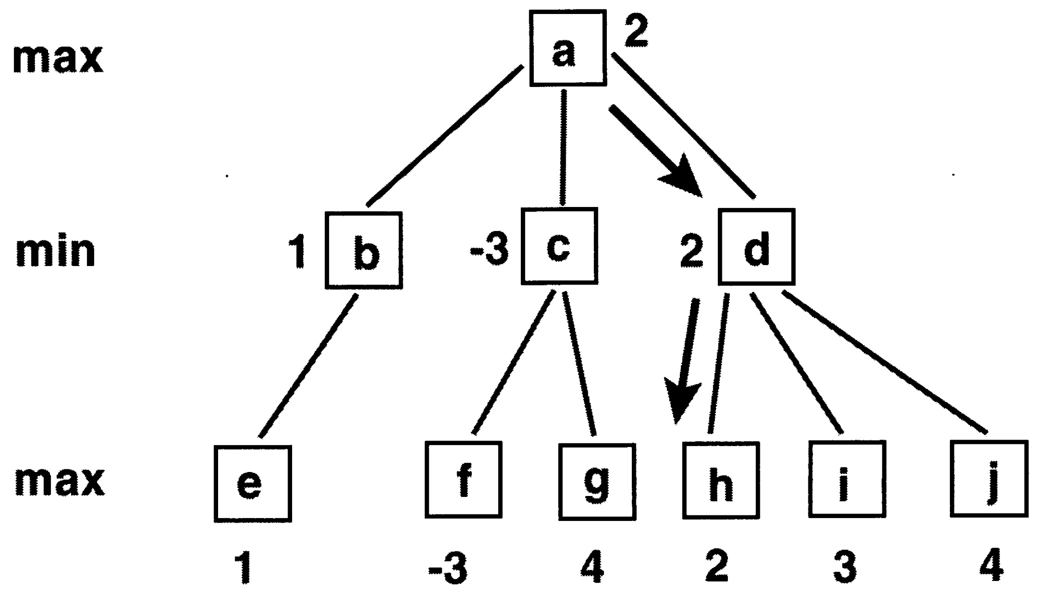


**Figure 6.1: Backing up scores in minimax. If Player A makes move "d", he/she is assured a score of at least 2 based on this minimax tree.**

Here is some pseudo-code for a minimax procedure. One can start a minimax search by creating a root node *r* and specifying the maximum search depth *d* during its creation, then calling **minimax(*r*, 0)**. The value returned is the minimax value of this node, searched to depth *d*, as well as the best move itself.

```
minimax(n:NODE *,depth:INT)
  returns bestvalue:INT, bestmoveX:BYTE, bestmoveY:BYTE
      1. Check if n is a leaf node. If it is, return its value.
      2. If depth is odd, this is a minimizing level
            a. let the minimum score seen thus far be MAXSCORE
            b. get a child c of node n. if c is NULL, break to step 2f
            c. recursively call minimax(c,depth+1)
            d. if returned value is less than min
               i.  let min=value
               ii. save the move that led to c as the best seen so far
            e. delete node c, then loop to step 2b
            f. done, return min and best move found
      3. If depth is even, this is a maximizing level
            a. let the maximum score seen thus far be MINSCORE
            b. get a child c of node n. if c is NULL, break to step 3f
            c. recursively call minimax(c,depth+1)
            d. if returned value is more than max
               i.  let max=value
               ii. save the move that led to c as the best seen so far
            e. delete node c, then loop to step 3b
            f. done, return max and best move found
```

## 6.2 Alpha-Beta Pruning

The alpha-beta pruning algorithm augments minimax search by only a few lines of a code. It cuts off certain paths in the search tree that have no effect on the final minimax value. This is illustrated in **figure 6.2**. Here, the minimizer at board position X has evaluated one of its children whose value is -3. But, the maximize one level above position X is already guaranteed a value of at least 1, so there is no need to explore the other children of X since the minimizer would never return anything better than -3.

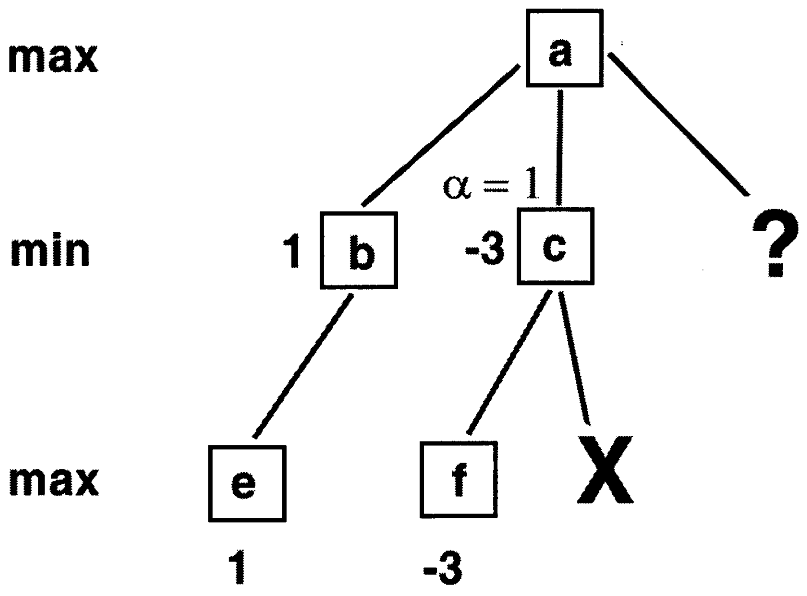


**Figure 6.2: Alpha-beta cutoff example.**

In alpha-beta pruning, *alpha* is a lower-bound on the minimax value as the search progresses, *beta* is the upper-bound. The maximizer trys to increase the lower-bound, *alpha*, while the minimizer trys to lower the upper-bound, *beta*. If the lower-bound and upper-bound collide, searching can be terminated. The minimax value (*alpha* returned by the top maximizing level) has been found.

Here is some pseudo-code for an alpha-beta procedure. It is essentially like minimax, except you feed in default values for *alpha* and *beta*, usually the minimum and maximum leaf value.

```
alphabeta(n:NODE *,depth:INT,alpha:INT,beta:INT)
  returns bestvalue:INT, bestmoveX:BYTE, bestmoveY:BYTE
      1. Check if n is a leaf node. If it is, return its value.
      2. If depth is odd, this is a minimizing level
            a. while alpha < beta, do steps 2b thru 2e
            b. get a child c of node n. if c is NULL, break to step 2f
            c. recursively call alphabeta(c,depth+1,alpha,beta)
            d. if returned value is less than beta
               i.  let beta=value
               ii. save the move that led to c as the best seen so far
            e. delete node c, then loop to step 2a
            f. done, return beta and best move found
      3. If depth is even, this is a maximizing level
            a. while alpha < beta, do steps 3b thru 3e
            b. get a child c of node n. if c is NULL, break to step 3f
            c. recursively call alphabeta(c,depth+1,alpha,beta)
            d. if returned value is more than alpha
               i.  let alpha=value
               ii. save the move that led to c as the best seen so far
            e. delete node c, then loop to step 3a
            f. done, return alpha and best move found
```

Move ordering plays a large role in how big a savings is seen with alpha-beta pruning. In the worst case, alpha-beta is no better than minimax. Good move-ordering heuristics are essential in most game playing applications.

A simple improvement for Tsume-Go is based on this point: *my opponent's best move is often my best move*. We can implement this easily by inserting code after lines 2C and 3C, respectively, that reorders the remaining children of node *n* such that the move just returned by my opponent (node *c*'s best move) is searched next, if it is a child of *n* and has not been examined yet. This simple heuristic always seems to improve the alpha-beta search in Tsume-Go [Dyer 96].

## 6.3 Alpha-Beta with Memory

Often, identical board positions arise while performing an alpha-beta search, usually as the result of some transposed moves. Consider starting the Go board in **figure 6.3**, with black to move next. In the move sequences **b-c-d-a** and **b-a-d-c** white's moves are reversed, but the same board position B arises after each move sequence. Many programs maintain a large hash table (or transposition table) so that repeat board positions are not fully searched each time. In Go especially, a high percentage of identical positions occur because the Go board tends to change so gradually as play progresses.

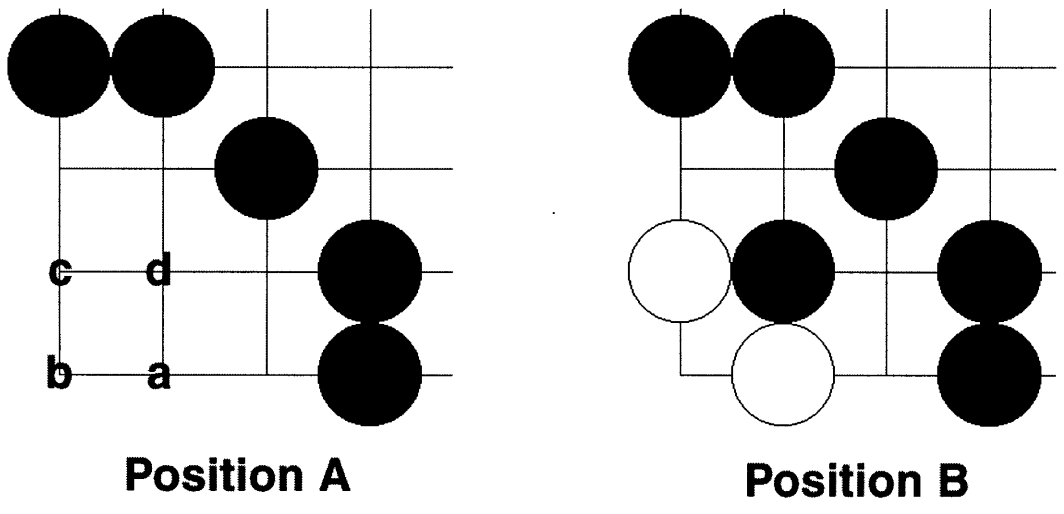


**Figure 6.3: Board position B can arise from move sequence b-c-d-a and b-a-d-c.**

A hash table entry for a particular node would contain the following information [Levy 91]:

- the score of that node (determined from a previous alpha-beta visit)
- a bit indicator whether this is an exact value, or an upper/lower bound on the minimax score

- the depth of searching done beyond this node, from which this score was derived
- the best move to make given this position
- the position (node / Go board) itself

In order to be effective, a hash table should be large (ranging from 4,000 to 4,000,000 entries), and is typically a power of 2 [Levy 91]. Of course, with a node-size of 256 bytes, storing the entire Go board inside each hash table entry is unreasonable. Instead, a hash function calculates a hash code of length *m* bits from a node. *m* is typically 36 to 64 bits in length, far smaller than what is necessary to uniquely specify a node. Hash functions are designed to minimize the possibility that two different positions are assigned the same hash code. The bottom *k* bits of the hash code ($k \leq m$) are used to find the position in the hash table (assume a table size of $2^k$).

The hash table used in the Tsume-Go program assigns each board a 38-bit hash code. The bottom 18 bits are used as an index into the hash table itself ($2^{18}$ entries x 64 bits each = 2MB hash table). **Table 6.4** and **figure 6.5** detail this implementation's hash table entry structure.

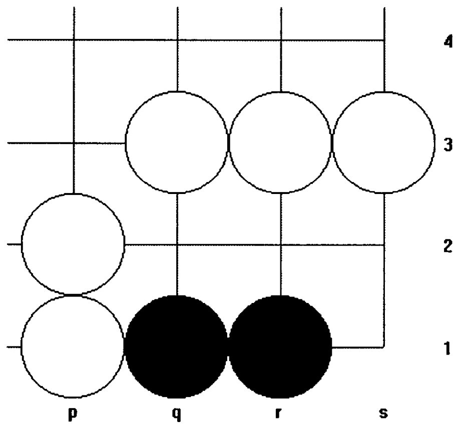
| Bit positions | Number of bits | Field description |
|---|---|---|
| 0-31 (LOB) | 32 | hash code (low 32-bits) |
| 0-5 (HOB) | 6 | hash code (high 6-bits) |
| 6-12 | 7 | node score (range 0-127) |
| 13-21 | 9 | depth of search beyond this node (range 0-511) |
| 22-30 | 9 | best move (`x = val%20, y = val/20`) |
| 31 | 1 | indicates bound score (not exact) |

**Table 6.4: Information maintained in each entry of the hash table.**

| 32 low-order bits of hash code | | | | |
|---|---|---|---|---|
| b | best move | depth | node score | hash code (high) |
| 1 | 9 bits | 9 bits | 7 bits | 6 bits |

**Figure 6.5: A single hash table entry is comprised of two 32-bit words. The lower word is the 32 low-order bits of the hash code. The upper word contains the remaining 6 bits of the hash code, 7 bits for score, 9 bits for depth, 9 bits for the best move, and 1 bit to indicate a bound score.**

Most programs determine a hash code for a board position using a hash function technique first described by Al Zobrist in 1970. This hash function uses a piece-square table of dimensions 2 (black/white pieces) by 361 (intersections) filled with random numbers the size of our hash code. A node is assigned a hash code by exclusive-or'ing the random numbers assigned to the appropriate piece squares [Levy 91]. Identical boards do not mean identical nodes however, so we need additional entries for which player has the next move, and how many Ko threats are available to this player. In **Figure 6.6**, with black to move and zero Ko threats, this node gets a hypothetical 16-bit hash code of 0000001101110011.



| | |
|---|---|
| P1-white | 0001 1010 1011 1111 |
| Q1-black | 1010 1000 0010 0100 |
| R1-black | 0100 1100 1111 1110 |
| P2-white | 1100 0000 0010 0010 |
| Q3-white | 0101 1110 1011 1100 |
| R3-white | 0101 0101 0101 0100 |
| S3-white | 1110 0001 0001 0110 |
| B-move | 0100 1010 0001 0001 |
| KO = 0 | 1001 1101 1101 1011 |
| all XOR'd | 0000 0011 0111 0011 |

**Figure 6.6: Given the above board, black to move, and zero Ko threats, this node hashes to the value 0000001101110011.**

We are almost ready to present a modified version of the alpha-beta pruning algorithm enhanced with a hash table. If a node *n* is found in the hash table, when can the alpha-beta algorithm use this information to terminate a line of play? An example where we cannot use it is when the depth of the stored node is too shallow. If we store a deep position in the search tree in the hash table whose minimax value and best move were calculated only to a depth of *x*, and then encounter this same position higher up in the search tree with more than *x* levels below it, we cannot terminate because we have not searched all lines of play to the specified maximum depth. We need expand this node and update the hash table entry appropriately.

A node *n* found in the hash table can only be considered terminal if the following two conditions are satisfied:

- the depth stored in *n* is equal to or greater than the *maximum depth* - *current depth*
- the score is exact, or at least a sufficient bound to cause an alpha-beta cutoff.

The following pseudo-code is identical to the normal alpha-beta pruning algorithm except for the lookup and store calls in the hash table.

```
alphabeta_hash(n:NODE *,depth:INT,alpha:INT,beta:INT)
  returns bestvalue:INT, bestmoveX:BYTE, bestmoveY:BYTE
      1. Check if n is a leaf node. If it is, return its value.

      2. If depth is odd, minimizing level
            a. initialize minchild to MAXSCORE
            b. while alpha < beta, do steps 2b thru 2g
            c. get a child c of node n. if c is NULL, break to step 2h
            d. lookup node in hash table
                  i. if current depth > max depth - hash node's depth
                  ii. if exact score or (score<=alpha)
                  iii. then terminal node. Use stored value.
            e. node not in hash, or can't be used
                  i. recurse on alphabeta_hash(c,depth+1,alpha,beta)
            f. if returned value is less than beta
                  i. let beta=value
                  ii. save the move that led to c as the best seen
            g. delete node c, then loop to step 2b
            h. done, store minchild in hash
                  i. if n has children remaining, store as a bound
                  ii. otherwise, store value & bestmove as exact

      3. If depth is even, maximizing level
            a. initialize maxchild to MINSCORE
            b. while alpha < beta, do steps 3b thru 3g
            c. get a child c of node n. if c is NULL, break to step 3h
            d. lookup node in hash table
                  i. if current depth > max depth - hash node's depth
                  ii. if exact score or (score>=beta)
                  iii. then terminal node. Use stored value.
            e. node not in hash, or can't be used
                  i. recurse on alphabeta_hash(c,depth+1,alpha,beta)
            f. if returned value is greater than alpha
                  i. let alpha=value
                  ii. save the move that led to c as the best seen
            g. delete node c, then loop to step 3b
            h. done, store maxchild in hash
                  i. if n has children remaining, store as a bound
                  ii. otherwise, store value & bestmove as exact
```

## 6.4 MTD(f) – Memory-enhanced Test Driver

MTD(f) is a minimax search algorithm that is simple to implement and efficient (provided you already have an existing alpha-beta-hash-table algorithm) [Plaat et al. 94]. MTD(f), an acronym for Memory-enhanced Test Driver, gets its efficiency from performing only zero-window (or null-window) alpha-beta searches, and using a good bound variable to perform these searches.

As we have seen, normal alpha-beta algorithms are called with a wide search window – typically alpha is the smallest value a leaf can have and beta the largest. This ensures that the returned value lies between these two values and hence is the true minimax value of the search tree. A narrower alpha-beta window gets more cutoffs, which improves search efficiency. Zero-window alpha-beta causes the most cutoffs but at the expense of information – only a bound on the minimax value is found. If alpha-beta fails high, we have a *lowerbound* on the minimax value. If alpha-beta fails low, we have an upperbound on the minimax value [Plaat 97].

The MTD(f) algorithm repeatedly calls alpha-beta with a zero window and a good bound. For example, suppose we call **`alphabeta_hash(root,d,beta-1,beta)`**, where *root* is the root node, *beta* is the good bound, and *d* is the search depth. This will return some value *v*. If *v* is less than *beta*, alpha-beta has failed low hence our new upperbound is *v*. Otherwise, alpha-beta has failed high and our new lowerbound is *v*. When the lower and upper bounds collide, we have found the minimax value. The algorithm zooms in on the minimax value by repeatedly calling alpha-beta. It is essential the alpha-beta is implemented with a hash table. If it was not, each pass of MTD(f) would re-explore most of the nodes.

```
mtdf(root:NODE *,depth:INT,initialguess:INT)
  returns bestvalue, bestmoveX:BYTE, bestmoveY:BYTE
      1. let x = initialguess
      2. initialize upperbound to MAXSCORE
      3. initialize lowerbound to MINSCORE
      4. while lowerbound < upperbound
           a. if x = lowerbound, beta = x + 1
           otherwise, beta = x
           b. x = alphabeta_hash(root,d,beta-1,beta)
           c. if x < beta then upperbound = x
              otherwise, lowerbound = x
      5. return x, bestmove
```

In the above pseudo-code at line 4A, if *x* is the same as *lowerbound*, we are trying to improve the lowerbound. Otherwise, we are trying to improve the *upperbound.* In line 4C, we update the appropriate bound depending on whether **alphabeta_hash** failed high or low. If our initial guess is the minimax value, **alphabeta_hash** is called exactly twice. Once for the upperbound, and once for the lowerbound.

## 6.5 Fast Tsume-Go Tree Searches

The maximum value for a leaf node as defined by our evaluation function in Section 5.5 is 127. If we are solving a Tsume-Go problem in which one player (either black or white) dies unconditionally with no stones remaining, we know the minimax value of this search tree will be 127. We do not know which move (or moves) will lead to this outcome.

The following algorithm will find a move with the desired minimax value of *goal* with only one zero-window alpha-beta call. It is important that all lines of play are searched completely – *alpha-beta must not abort at some fixed maximum depth.* In the Tsume-Go problem solver developed for this thesis, one would create a root node with no maximum depth cutoff and then call **tsumego_td(root,127)**.

```
tsumego_td(root:NODE *, goal:INT)
  returns bestmoveX:BYTE, bestmoveY:BYTE
      1. x = alphabeta_hash(root,0,goal-1,goal)
      2. if x equals goal, return the bestmove
      3. Otherwise, no solution was found.
```

This algorithm is a special case of MTD(f). We know the minimax value ahead of time, and thus need only one pass to verify that the lowerbound really is our *goal.*

# 7 Experiments

The following experiments were created to test the correctness of the Tsume-Go problem solver created for this thesis. The first couple examples show that the node class and minimax code work properly. Then, the performance of the four search algorithms, minimax, alpha-beta, alpha-beta with a hash table, and MTD(f) using alpha-beta w/hash, are compared against each other on increasingly difficult problems.

These experiments were performed on an Intel Pentium II/266mhz MMX processor with 64MB of RAM.

## 7.1 Obvious Moves

To verify its correct functionality, the Tsume-Go program solved many extremely simple problems. One of the most basic tests is shown in **figure 7.1**. It is white's move. The interest marks are arranged so that only moves A17, B17, and C17 are allowed for white. The problem solver correctly identifies B17 as the optimal move, with a minimax score of 127. In this example, a board position without any black stones is defined as a leaf, so minimax stops. This is not a sufficient condition for search termination, in general. Of the three moves generated at the root node, B17 is neither the first nor the last tried.

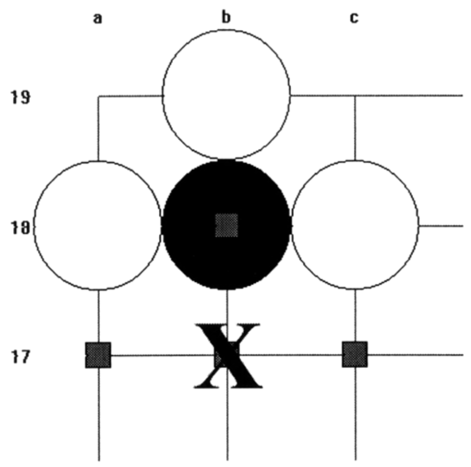


**Figure 7.1: Problem filename `OTEST1.TGO`.**
**White to move. Play at B17 to capture the black stone.**

The problem in **figure 7.2** tests the programs ability to recognize Ko situations. Black plays at "x". If it had made the other move, white would have captured black's stone. White would not then be able to capture white's stone because of the Ko rule, hence the board would contain 3 black stones and 4 white stones. By playing at "x", the board is split evenly 4 black stones versus 4 white stones.

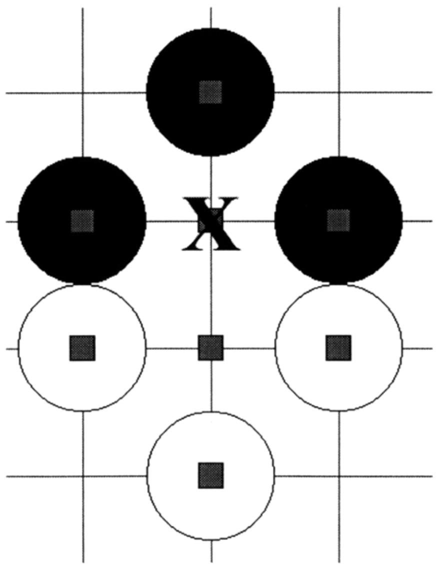
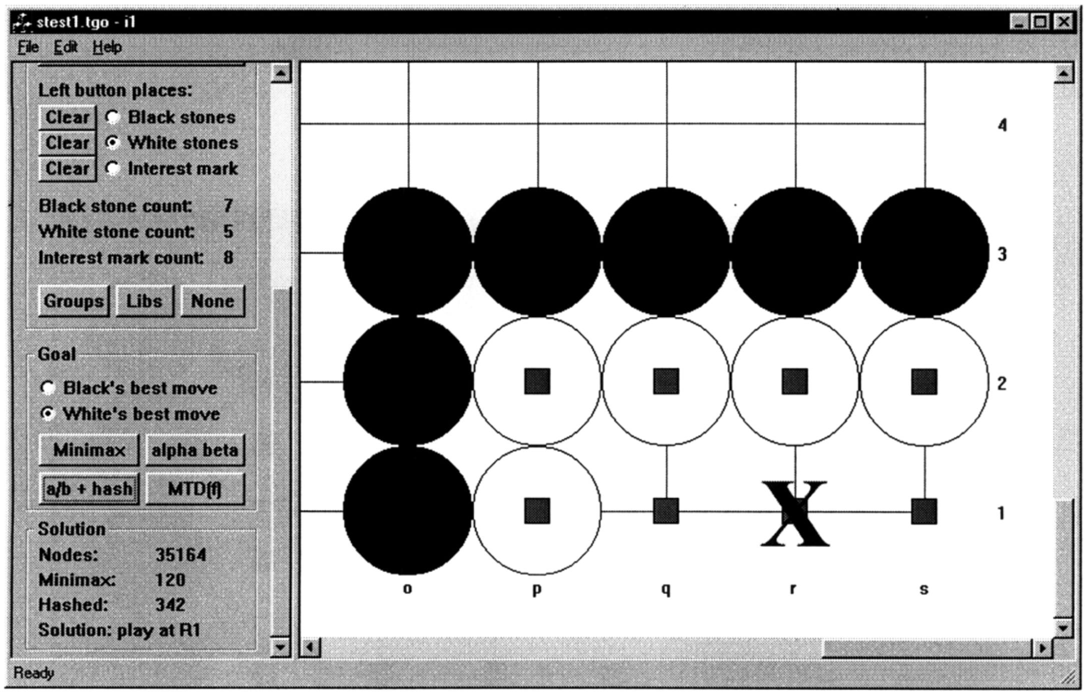


**Figure 7.2: Problem filename `OTEST2.TGO`.**
**Black to move. Play at X, avoiding white capture and Ko situation.**

## 7.2 Simple Problems

A more interesting problem is a situation in which white is forced to make a move to avoid capture. In **figure 7.3**, white only has 3 move choices at the root node. However, searching all lines of play in their entirety involves many useless lines of play. For example, if white played at Q1, black would respond either at R1 or S1, forcing atari. White has no legal moves, so black would play and capture the white group. Now, five intersections are free and white continues to hopelessly battle. **Table 7.4** compares the four different search algorithms on this problem.


stest1.tgo - i1

Left button places:
Clear ( ) Black stones
Clear [✓ White stones]
Clear ( ) Interest mark

Black stone count: 7
White stone count: 5
Interest mark count: 8

Goal
( ) Black's best move
[✓ White's best move]

Solution
Nodes: 35164
Minimax: 120
Hashed: 342
Solution: play at R1

Ready

**Figure 7.3: Problem filename `STEST1.TGO`.**
**White to move. Play at R1 forming two distinct eyes. The white group is now unconditionally alive.**

| max depth | minimax | alpha-beta | alpha-beta-hash | MTD(f) using alpha-beta-hash |
|---|---|---|---|---|
| 4 | 88 | 36 | 32 | 180 |
| 6 | 1216 | 176 | 142 | 311 |
| 8 | 32838 | 933 | 721 | 1653 |
| 10 | 516005 | 3899 | 2666 | 6479 |
| 12 |  | 11130 | 6451 | 15483 |
| 14 |  | 26383 | 13385 | 31729 |
| 16 |  | 51601 | 24584 | 51020 |
| 18 |  | 87359 | 28876 | 73516 |
| none |  |  | 39115 | 60257 |

**Table 7.4: Number of nodes explored at specified depth by each algorithm. A shaded box means operation took too long to complete.**

The problem in **Figure 7.5** showed particularly good results for the MTD(f) algorithm. **Table 7.6** compares all the searches.

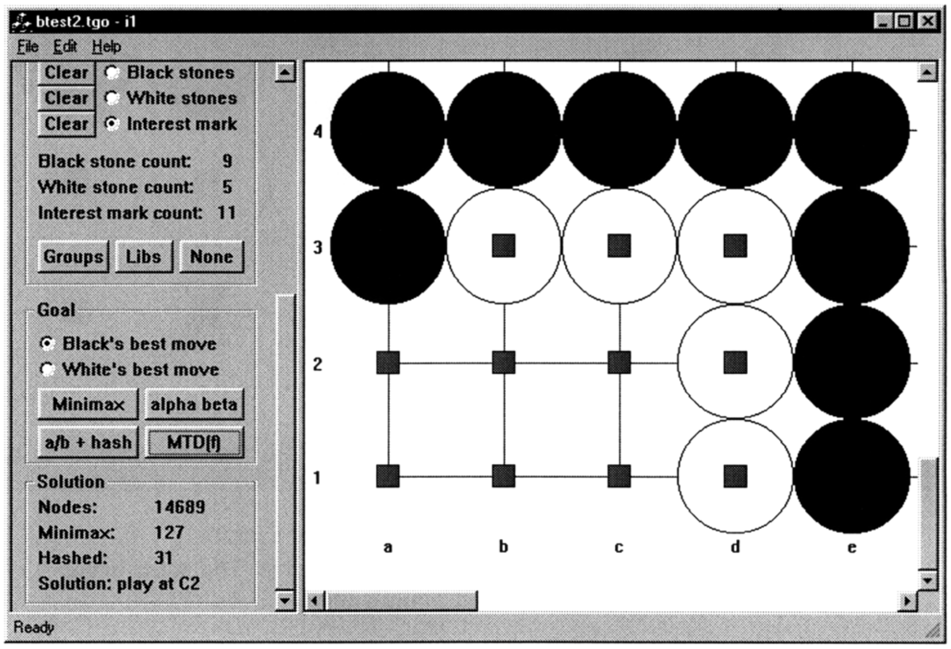

```
btest2.tgo - i1
File  Edit  Help
Clear   Black stones
Clear   White stones
Clear   [✓ Interest mark]
Black stone count:   9
White stone count:   5
Interest mark count:  11
Groups  Libs  None
Goal
[✓ Black's best move]
White's best move
Minimax  alpha beta
a/b + hash  MTD(f)
Solution
Nodes:      14689
Minimax:    127
Hashed:     31
Solution: play at C2
Ready
```

**Figure 7.5: Problem filename BTEST2.TGO.**
**Black to move. After searching the entire tree with MTD(f), the program has determined that move C2 will kill the white group.**

| max depth | minimax | alpha-beta | alpha-beta-hash | MTD(f) using alpha-beta-hash |
|---|---|---|---|---|
| 4 | 1202 | 140 | 142 | 712 |
| 6 | 13632 | 1203 | 1026 | 1455 |
| 8 | 114924 | 4196 | 3412 | 3797 |
| 10 | ▨ | 5508 | 3944 | 950 |
| 12 | ▨ | 15969 | 9300 | 2998 |
| 14 | ▨ | 31553 | 14170 | 7487 |
| 16 | ▨ | 41579 | 18212 | 10020 |
| 18 | ▨ | 61913 | 23931 | 13611 |
| none | ▨ | ▨ | 31497 | 14689 |

**Table 7.6: Number of nodes explored at specified depth by each algorithm. A shaded box means operation took too long to complete.**

## 7.3 Harder Problems

The problem in **Figure 7.7** is a real Tsume-Go problem. If black plays at S1, he/she can kill all of white's groups. **Table 7.8** contains the analysis of the search algorithms on this problem. Unfortunately, the search tree proved too deep, so an incorrect answer was returned because of the truncated lines of play.

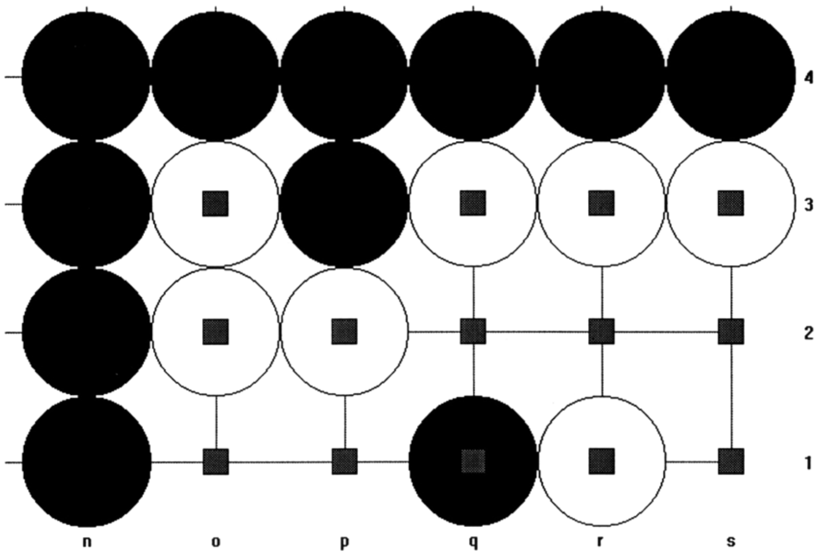


**Figure 7.7: Problem filename `HTEST1.TGO [Dyer 96]`.**
**Black to move and kill white unconditionally. Correct play is at S1.**

| max depth | minimax | alpha-beta | alpha-beta-hash | MTD(f) using alpha-beta-hash |
|---|---|---|---|---|
| 4 | 1194 | 154 | 147 | 703 |
| 6 | 17038 | 1292 | 1006 | 3870 |
| 8 | 237688 | 7498 | 3392 | 9354 |
| 10 | ▨ | 54630 | 25037 | 36184 |
| 12 | ▨ | 169475 | 73002 | 41415 |
| 14 | ▨ | ▨ | 365443 | 160579 |
| 16 | ▨ | ▨ | ▨ | 514519 |
| 18 | ▨ | ▨ | ▨ | ▨ |
| none | ▨ | ▨ | ▨ | ▨ |

**Table 7.8: Number of nodes explored at specified depth by each algorithm. A shaded box means operation took too long to complete.**

The problem in **Figure 7.9** is a hard Tsume-Go problem taken from a paper on GoTools by Thomas Wolf, credited to Denis Feldmann. Black moves first and can kill white by playing at B1. The branching factor has increased significantly, hence the search algorithms do not perform well. **Table 7.10** shows the results of this experiment.

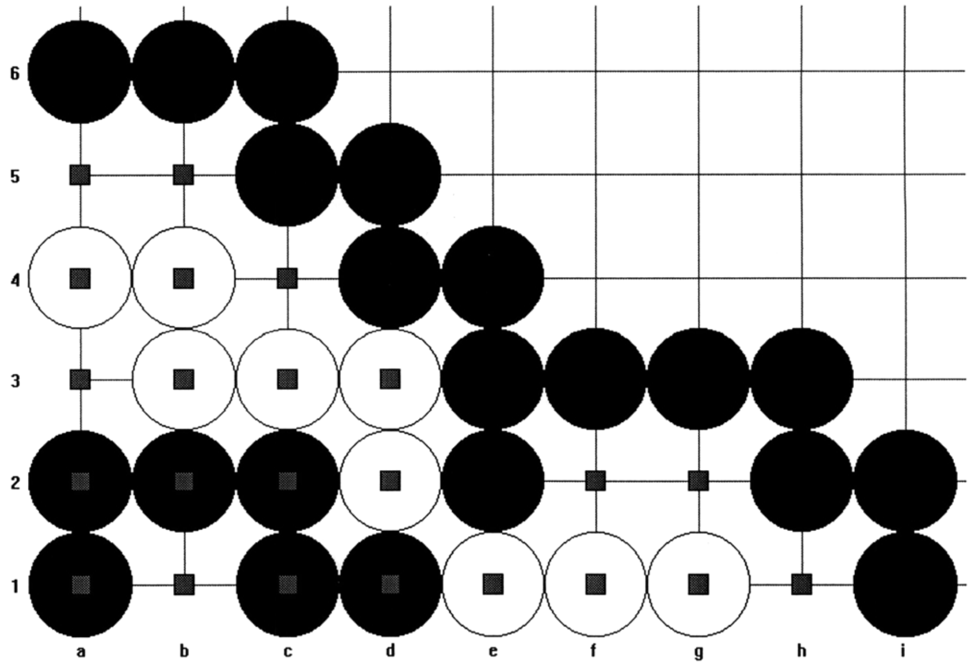


**Figure 7.9: Problem filename `HTEST2.TGO [Wolf 96-7]`.
This is a very hard Tsume-Go problem. Black to move and kill.
The correct move is for black to fill its own group's eye-space at B1.**

| max depth | minimax | alpha-beta | alpha-beta-hash | MTD(f) using alpha-beta-hash |
|---|---|---|---|---|
| 4 | 3312 | 187 | 187 | 1641 |
| 6 | 145536 | 7446 | 4980 | 16566 |
| 8 | ▨ | 49742 | 31907 | 135128 |
| 10 | ▨ | 1079949 | 559844 | 1850622 |
| 12 | ▨ | ▨ | ▨ | ▨ |
| 14 | ▨ | ▨ | ▨ | ▨ |
| 16 | ▨ | ▨ | ▨ | ▨ |
| 18 | ▨ | ▨ | ▨ | ▨ |
| none | ▨ | ▨ | ▨ | ▨ |

**Table 7.10: Number of nodes explored at specified depth by each algorithm. A shaded box means operation took too long to complete.**

Part of the reason why MTD(f) performs so miserably in this experiment is that MTD(f) requires a good initial guess. Here, we guess the solution is 127, the static evaluation of the game board if all white pieces were removed. Because the branching factor and depth of this search tree is so great, cutting off the search at a depth of 10 yields a very low minimax value. MTD(f) performs many iterations trying to find this. In a real Go program, one would use iterative deepening to get a better initial guess when calling MTD(f).

# 8 Conclusion

## 8.1 Results

From the raw node counts in Section 7, one instantly realizes that plain minimax searching has no place in a Tsume-Go program. It was used as a reference point to compare the other, more promising algorithms.

Alpha-beta pruning expanded an order of magnitude less nodes than minimax. A Go board changes slowly as play progresses, yet still provided alpha-beta with a great deal of cutoffs. Although not shown in Section 7, each of these example problems were analyzed from many different orientations. Altering the orientation of the problems changes the order in which moves are searched, which has a big effect on how efficient alpha-beta is. In the worst case, alpha-beta will do little better than minimax.

When we added a transposition table, the number of nodes searched decreased in all the problems – usually less than half the nodes expanded in the plain alpha-beta search. When augmented with a hash table, alpha-beta does not exhibit the wide fluctuations in node counts as the board is scanned in different orientations. Some good heuristics will also improve the alpha-beta algorithms, but even the simple heuristic including in these algorithms – *my opponent's best move may be my best move* – significantly increased the cutoff rate.

The MTD(f) algorithm beat out minimax and plain alpha-beta, but it was not quite as dominate over the alpha-beta variant with a hash table. MTD(f) relies on a good initial estimate of what the minimax value will be. In this program, we feed it the value of a Go board with no enemy stones, 127. MTD(f) performed the worse when the actually minimax value for the depth-limited tree was low. In the last example in Section 7, MTD(f) performed a null-window alpha-beta search on 127, then 126, 125, 124, ..., until it reached 113. Even with a memory enhanced alpha-beta algorithm, there was a great deal of researching. In a real program, one would probably use iterative deepening by first searching to depth 1, then feeding the result of that as the initial guess for a search of depth 2, then depth 3, and so on.

## 8.2 Problem Areas

The largest problem in Tsume-Go is determining when to stop a search. This program does not identify leaf nodes well, and consequently must search an absurd number of nonsense lines of play in order to find an answer. Like many beginning Go players, this program does not know when to give up on a line of play.

Another problem area is move generation. In our representation, the groups and liberties of each are recalculated for each node. This is extremely wasteful, especially since a Go board changes a relatively small amount between any two successive plays. Some careful consideration about how stone groups grow in Go, and how adding a single stone affects only a local region of the board, may well lead to much more efficient analysis and move generation algorithms.

Finally, the node representation itself is somewhat cumbersome. Since we are performing only depth-first search, the amount of space required grows linearly with the search depth. Instead of constantly compressing and decompressing the game board stored in each node, it would probably be faster to store an uncompressed Go board in each node. This would only mean a minimal increase in space, especially when compared to the size of a hash table. To simplify and speed up Ko detection, one could also use the same hash function that is used in the hash table and store this small hash code in the node structure.

## 8.3 Future Research

The memory-enhanced test driver algorithm seems promising in Tsume-Go. Zero-window alpha-beta searches allow for the most cutoffs. Since we know what the minimax value will be, we only need to call the alpha-beta-hash algorithm once to determine the winning move. This approach assumes both a winning move does exist and that no maximum depth cutoffs will occur. With an algorithm to detect alive/dead eye-spaces or a database table of eye shapes, a variant on the MTD(f) algorithm may prove superior to the conventional memory-enhanced alpha-beta routines currently used.

# References

| | |
|---|---|
| [Allis et al. 91] | Which games survive? ALLIS, L.V., VAN DEN HERIK, H.J., HERSCHBERG, I.S. In: Heuristic Programming in Artificial Intelligence 2. LEVY, D.N.L., and BEAL, D.F.(eds), pages 232-243, Ellis Horwood, 1991. |
| [Burmeister/Wiles 95] | An Introduction to the Computer Go Field and Associated Internet Resources. BURMESITER, J., WILES, J., World-Wide-Web page http://www.psy.uq.edu.au/~jay/, 1995. |
| [Dyer 96] | Searches, Tree Pruning, and Tree Ordering in Go. DYER, D., World-Wide-Web page http://www.andromeda.com/people/ddyer/go/search.html, 1997. |
| [Jasiek 97] | Ko Rules. JASIEK, R., World-Wide-Web page, http://www.inx.de/~jasiek/korules.html, 1997. |
| [Levy 91] | How Computers Play Chess. LEVY, D., and NEWBORN, M. W.H. Freeman and Company, pages 153-224, New York, NY, 1991. |
| [Mariano 97] | Go Frequently Asked Questions. MARIANO, A. FAQ posted on internet newsgroup rec.games.go, 1997. |
| [Müller 95] | Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory. MULLER, M. Thesis, Swiss Federal · Institute of Technology, Zurich, 1997. |
| [Plaat et al. 94] | A New Paradigm for Minimax Search. PLAAT, A., SCHAEFFER, J., PIJLS, W., and Bruin, A, University of Alberta, Edmonton, 1994. |
| [Plaat 96] | Research Re:Search & Re-search. PLAAT, A., PhD thesis, Erasmus University, Rotterdam, Netherlands, 1996. |
| [Plaat 97] | MTD(f) A Minimax Algorithm Faster than NegaScout. PLAAT, A., World-Wide-Web page, http://theory.lcs.mit.edu/~plaat/mtdf.html, 1997. |
| [Richards et al. 96] | Evolving Neural Networks to Play Go. RICHARDS, N., MORIARTY, D., MIIKKULAINEN, R., University of Texas as Austin, Austin, TX, 1996. |
| [Silva 96] | Go and Genetic Programming. Playing Go with Filter Functions. SILVA, S.F., 1996. |
| [Winston 92] | Artificial Intelligence, 3rd Edition. WINSTON, P., Addison-Wesley Publishing Company, pages 101-118, Reading, MA, 1992. |
| [Wolf 96-7] | About Problems in Generalizing a TsumeGo Program to Open Positions. WOLF, T., School of Mathematical Sciences, Queen Mary and Westfield College, University of London, London, 1996. |

| | |
|---|---|
| [Wolf 96-11] | The Program GoTools and its Computer-Generated Tsume Go Database. WOLF, T., School of Mathematical Sciences, Queen Mary & Westfield College, London, 1996. |
| [Wolf 97] | GoTools – The Tsume-Go program. WOLF, T., World-Wide-Web page http://alpha.qmw.ac.uk/~ugah006/gotools/, 1997. |

5519-56