Jaigo Technical Report

Sean Foy Noel Weichbrodt

December 19, 2007

Abstract

Despite its elegantly simple rules, Go's complexity as a game of deterministic strategy has made it a recurring subject in studies of artificial intelligence and game theory [**?**].

Contents

List of Tables

Chapter 1

Introduction

Go is a two-player board game which originated in Ancient China and remains one of the most richly complex games of strategy today. The objective is to control more area or territory than one's opponent on the game board. The game is played on a square line grid; the standard size of this grid is 19 x 19, though 13 x 13 and 9 x 9 grids are used occasionally. Players place stones – either black or white – at the intersecting points of the grid to signify control of that position. Once placed on the board a stone has a number of liberties – unoccupied grid positions which are orthogonally adjacent to that stone. The goal of the game is to claim as much territory as possible by forming chains (contiguous extensions of stones of the same color) which cannot be captured by one's opponent. A stone or chain is captured and removed from the board when it has no liberties, the result of being surrounded by stones of the opposing color.

Players are not allowed to commit "suicide" with their stones, that is, they may not play into a spot where a stone would have no liberties. This prohibition actually enables players to create certain arrangements of stones which cannot be captured. Players are also forbidden from repeating one of the previous two states of the game; players cannot repeat their previous move. This is known as the Ko rule. At each turn, a player can either place a stone or pass. The game ends when both players pass consecutively on a turn.

Go players are classified by their skill into equivalence classes called kyu and dan. A beginning player might have a rank of 30 kyu, while a more experienced player might be 15kyu. Avid amateurs and professionals who outrank 1 kyu fill the range from 1 dan (low) to 8 dan (high) and on to 1 professional dan (low) to 10p (high).

Game theorists would describe Go as a zero-sum, perfect information, partisan, deterministic strategy game. In zero-sum games, the utility of any action with respect to player a is the additive inverse of the utility of that action with respect to player b. Perfect information games are fully observable in AI parlance. Go has a turn-taking structure, so it is partisan (i.e., some moves are available to only one player). There is no element of chance in Go. These

characteristics invite comparison with games such as Chess.

While chess eventually yielded to clever hardware and algorithms, the sheer state-space of Go has rendered specialize search approaches weak. In 1965, noted statistician I.J. Good noted presciently:

> Go on a computer? In order to programme [sic] a computer to play a reasonable game of Go rather than merely a legal game it is necessary to formalise the principles of good strategy, or to design a learning programme. The principles are more qualitative and mysterious than in chess, and depend more on judgment. So I think it will be even more difficult to programme a computer to play a reasonable game of Go than of chess. [**?**]

Unlike many similar problems that were also announced in this manner, Go remains both a challenging and tantalizing problem.

Two major factors make computer Go more difficult to program than computer Chess:

1. Go is PSPACE-hard with a branching factor of 361, and so is not amenable to brute force searching.

2. The traditional concepts and representations of the state of the board do not translate easily to traditional areas of AI research.

Chapter 2

Related theory and practice

2.1 Board representation

With 3361 board states, and perhaps 10768 games, representing the board may be the most difficult part of writing a go engine.

The exploration of alternative future board states often yields duplicate (potential) states; it is helpful to memoize board evaluation such that a board state may be evaluated only once, regardless of the paths by which it is reached [**?**]. The Zobrist hash function is useful in this capacity: (reduce xor (map f board-positions)), f: (x, y, color) -¿ Z.

2.2 Known techniques

Search techniques have been successful in games including Chess, but they have been much less effective in Go. The large branching factor of Go prohibits exhaustive search. Wolfe has shown by reduction from 3-QBF that Go endgames are PSPACE-hard [**?**] and Go itself is EXPTIME-complete [**?**], so it is unlikely that any algorithm can guarantee optimal and efficient play. Computer Go is thus focused on approximation algorithms, and suboptimal or incomplete but empirically successful heuristic searches. Minimax with α-β pruning and the anytime beam stack search methods are of limited use given the very short horizon and absence of good heuristics for Go.

The difficulty of formalizing the heuristics used by humans motivates us to consider applying machine learning algorithms to pattern-matching in Go. Researchers including [**?**] have enjoyed limited success with neural networks, but both training and playing are slow, and feature vector characterization is difficult. Strategies for overcoming the need for a priori knowledge in feature vector formulation and connection weights include constructive and genetic algorithms. Unfortunately, experiments with these techniques have yielded "erratic and unorganized" play despite "prohibitively large" time requirements for playing on full-size boards [**?**].

Brügmann's approach plays games by choosing moves randomly according

to probability distributions found by simulated annealing; the moves are evaluated according to their occurrence in successful games. A key insight from this research is that moves can be iteratively reprioritized by observed value rather than played in some fixed order; this is crucial for the simulated annealing application [**?**].

Chapter 3

Current Issues

* Finding a good evaluation function is hard. The human heuristics are vague, and when/how they get applied is even more vague. Zobrist divided the game into phases for the purpose of selecting evaluation functions [**?**]. Ryder distinguished between tactical and strategic utility [**?**]. * fight between optimizing the main loop (evaluating the board) vs optimizing the small loops (given a board evaluation, local tricks to improve evaluation function). GNUGO v Go++ * GA & NN seemed promising at first, but no commercial package uses them due to lack of speed. * If all old techniques are inadequate and new techniques don't pan out, whither computer go?

Our project seeks to create a JavaScript-based Go-engine. JavaScript is a dynamically typed, imperative language that supports the functional and object-oriented paradigms [**?**]. It is ubiquitous due to its inclusion in nearly every Web browser, and recently has enjoyed a surge of popularity among programmers due to the availability of cross-platform libraries and the success of applications such as GMail. As professional programmers, we viewed this project as an opportunity to explore not only AI but also renewed interest in the JavaScript language. Also, JavaScript is the only supported means of developing iPhone applications. iPhone is an intrinsically interesting platform; its communication capabilities and computational limitations only heighten the interest.

Chapter 4

Interaction/UI

We decided that our software should be interoperable with other Go software, allowing us to focus on the AI and making our software more useful. The Go Text Protocol [**?**] is a defacto standard for interoperability, so we have focused our effort on implementing the role of the engine in the GTP. Besides providing an interface to competitive engines and automated testing frameworks, it allows us to avoid writing our own UI by reusing any of several existing UI implementations. During early development, we used a simple set of HTML input controls to interact with the engine via GTP.

Chapter 5

Scoring

The board is scored using Benson's Algorithm [**?**]. "c" stands for black or white, "-c" for the opposite color. Each point of the board is, of course, either black, white or empty.

A c chain is a non-empty maximum connected set of c points.

A c region is a non-empty maximum connected set of -c and empty points.

A c chain is unconditionally alive if and only if it belongs to a set of c chains S such that every chain B in S is adjacent to at least two c regions R that satisfy:

1. (P1) all empty points in R are adjacent to B, and
2. (P2) all c chains adjacent to R belong to S.

Chapter 6

Engine choice

There are a number of Go engines, so we sought to extend an existing engine rather than write something completely new. We could not find an engine in JavaScript, so the first order of business was to choose an engine to port. Our criteria for choosing an engine was simple: What is the engine with the smallest (estimated) expression in JavaScript and the highest kyu ranking?

SimpleGo v1

50kyu greedy search (random move from list of moves with worst opponent score)

SimpleGo v3

30kyu α-β pruned search

Crawler

?? (30kyu) maintain line strength/shape

Chapter 7

Technical details

Code Listing - Implementation of Benson's algorithm for determining unconditional life: Analyze_Color_Unconditional_Status - Greedy Search algorithm: select_scored_move - Crawler algorithm: select_crawler_move

Chapter 8

Discussions, observations, and comparisons

We implemented two move choice engines based on two different algorithms, scoring for color and blocks based on Benson's algorithm, and representations of the board, the game, eyes, blocks, and numerous language-specific utilities. Our goals were compactness and speed due to the target platform limitations.

Due to issues with the jaigo board representation, our measurements were limited to the first 50 moves

Regarding compactness, the total size of all jaigo files, including images, scripts, and html, was 102221 bytes for our initial release. The full implementation of the jaigo code is 66071 bytes [**?**].

Regarding speed, we do not anticipate that for the first 50 moves on a 19x19 board with human (black) versus crawler (white), $(noname)10967525.68flood_mark161814.17(noname)19467811.75st$

for the first 50 moves of a game on a 19x19 board, with crawler (black) versus crawler (white),

(no name) $40449432.8flood_mark383815.94iterate4044958.75(noname)7351908.72string_as_move2427655.86(nonan$

Median move response time for 10 games into the first 50 moves was was 1 second, with the high being 3 seconds and the low being unmeasurable.

On an iPhone over 54mbps 802.11g wifi with a full signal connected to a 3mbps cable modem, the median jaigo load time was 5 seconds, with the initial load time being the longest measurement at 15 seconds. On an iPhone over ATT 2.5G EDGE with a full signal, the median jaigo load time was 21 seconds, with a low of 17 seconds and a high of 30 seconds.