A Multi-Agent Approach to the Game of Go Using Genetic Algorithms

Todd Blackman and Arvin Agah

Department of Electrical Engineering and Computer Science
The University of Kansas, Lawrence, Kansas, USA

ABSTRACT

Many researchers have written or attempted to write programs that play the ancient Chinese board game called Go. Although some programs play the game quite well compared with beginners, few play extremely well, and none of the best programs rely on soft computing artificial intelligence techniques like genetic algorithms or neural networks. This paper explores the advantages and possibilities of using genetic algorithms to evolve a multiagent Go player. We show that although individual agents may play poorly, collectively the agents working together play the game significantly better.

KEYWORDS

Game of Go, Genetic Algorithms, Multiagent Systems

1. INTRODUCTION

Games have often been used to test new concepts in artificial intelligence because of their relative simplicity compared with other more complex possibilities like simulations and real-world testing. Go has the potential to excel as a testbed for artificial intelligence concepts because of the complexity of the tactics and strategies used to play the game well. These complexities resemble real-world problems better than most other games do. Brute-force search cannot be used exclusively to play this

Reprint requests to: Arvin Agah, Professor, Department of Electrical Engineering & Computer Science, The University of Kansas, Lawrence, KS 66045-7621 USA; e-mail: agah@ku.edu; tdblackman@gmail.com URL: http://people.ku.edu/~agah/

game, as in other games, because of Go's huge branching factor, which starts out at 361 at the beginning of a game and approximately decreases by one after each move. Some have suggested that Go may have to replace chess as the game for AI practitioners.

With pure search ruled out as a viable method for playing Go, one must turn to more intelligent methods such as pattern recognition or rule-based deduction. Complexity often plagues Go programmers because of the intricacies of how a player must think about the game–often remote locations on the board influence a local situation. Current Go programs play at only the level of a skilled novice, and these limitations exist perhaps because of the programs' architectures and their insistence on using only methods such as pattern-matching, hard-coded rules in computer code, and minimax with alpha-beta pruning.

This paper is based on the concept that programs should play Go using relatively simple agents that combine to play the game well. Traditional methods have their place in Go programs, but to play an abstract and multi-faceted game one must use an abstract and multi-faceted approach. Genetic algorithms (GA) have been employed to play complex games, but they often use evolved values that are too low-level to allow the program to attain the skills required. These values allow for the evolution of useful information such as patterns or algorithmic code, but to play the game on a professional level one would need too many of these individual pieces of information. Analogously, it would be similar to creating a neural network with $3x19^2$ inputs, representing the, 19^2 board locations and the three possible states for each location (white, black, or empty) and, 19^2 outputs. Training an artificial neural network of this size will remain inconceivable for quite some time. Likewise, trying to evolve a set of rules using a GA would fail in much the same way. Numerous rules exist, and evolving them would take too long.

The approach presented in this paper differs from most current Go programs. Other programs are extremely complex, representing huge amounts of Go knowledge. Such programs eventually become unwieldy, difficult to maintain, hard to follow, and tricky to improve upon. The motivation for this work is thus to study whether a set of relatively simple agents can each look at the problem from their own perspective, after which GA-evolved weights will allow the agents' solutions to be summed together to produce a final solution. This approach exchanges the ability to fine-tune the program with the ability to incorporate more agents, and thus more knowledge, in a consistent and scalable way. This paper will illustrate a novel

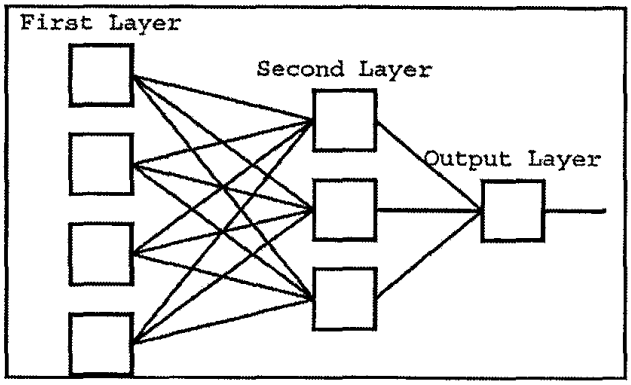


Fig. 1: Summation network architecture

multiagent approach to playing Go that uses a multilayer network to choose moves based on those suggested by each individual agent.

The approach involves developing a set of agents that generate a value for each location on the board, with higher values representing a more highly recommended move. These values are entered into a matrix representing the board locations. These matrices are then normalized and combined non-linearly using GA-evolved weights. The summation network architecture, shown in Figure 1, resembles a neural network in that the resulting matrix of values is generated from a weighted sum of a set of weighted sums. The resulting move to play will then be either the highest value in the matrix or chosen probabilistically with higher values receiving a greater probability of being chosen.

Thus, the Goal is to investigate the novel multiagent approach to playing Go, where as more agents are added and evolved, the program plays better and simple agents can be used to achieve collective performance. The agents have specific and well-defined expertise, i.e., a clear focus.

This paper is organized into five sections. Section 2 begins with information about the game of Go, including the rules of the game, a few direct implications of the rules, basic concepts, how to score a game, and how players are ranked. This is followed by a description of some techniques relevant to this paper. Following these, the applications of AI to games, and in particular, to the game of Go are discussed. Section 3 explores the design of the AI program written for this paper, providing a top-level view of the program's architecture. In Section 4 the experiments performed along with their results are presented. Section 5 concludes this paper.

2. BACKGROUND AND RELATED WORK

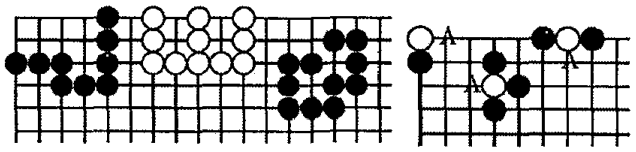
2.1 The Game of Go

The game of Go (also called Goe, iGo, baduk, wei-chi, weichi, weiqi, wei-qi, etc.) is a board game played by two opponents on a, 19x19 grid. Go is a game of perfect information, i.e., players can see the entire state of the game at all times. No guessing or probability is involved as in such games as backgammon or bridge, which have uncertainty and hidden state, respectively. The players take turns placing pieces called stones on intersections on the board. The game continues until both players pass their turns in succession. No stone can be moved unless it is captured (as explained later), and all stones are completely equal in power.

2.1.1 *Surrounding Territory.* The Goal of the game is to surround more territory than one's opponent with a secondary Goal of capturing the opponent's stones. Each surrounded intersection or captured stone is worth one point. An exception exists when a stalemate condition arises, as explained later. In general, surrounding territory is considered much more important by anyone versed in Go. Figure 2a shows three examples of surrounded territory: The black group on the left surrounds nine points, the white group surrounds four points, and the right-most black group surrounds two points.

2.1.2 *Capturing.* The secondary way to gain points is by capturing the stones of the opponent. To capture a single stone, one must play on all adjacent intersection points that are at right angles to the stone(s) to be captured. Figure 2b shows three examples of a white stone about to be captured by a black stone if black were to play on the locations marked A. To capture groups one must play on all the liberties of the group, where a liberty is an empty location adjacent to a group. A group is defined as a set of stones that connect adjacently to each other through the straight lines on the board, and diagonals do not count. Figure 3a shows an example board where white could play at the location marked A to capture ten black stones. On the other hand, black could play at A to capture four white stones. If white were to play at location B in Figure 3b, then white would only capture seven stones.

2.1.3 *Eyes.* Any group is unconditionally safe if it can partition itself into at least two sections (also called eyes). The two rightmost groups in Figure 2a have two eyes each. If the opponent plays in an eye with a single intersection inside that eye, then it commits suicide, as all of its liberties are taken. If the eyes are too big, one's opponent could create a living group inside an eye and then the eye could become useless.



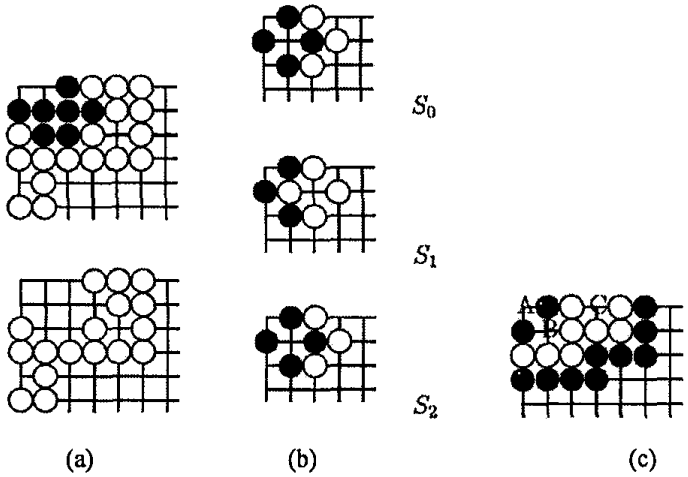
(a) (b)

Fig. 2: (a) Surrounded territory; (b) White stones to be captured next move.

(a) (b)

Fig. 3: (a) Capturing; (b) Capturing II

2.1.4 *Live and dead stones.* Surrounding territory is crucial while playing Go, but there is an important twist that can make what seems like one's territory actually become the opponent's. At the end of the game, after both sides pass in succession, if a group of stones could not survive an attack (it does not have two eyes or the ability to make them if pressed), then that group is removed from the board and given to the opponent. With experience, one learns how to identify dead groups of stones, and if a disagreement arises about whether a group is alive, then the game continues. Figure 4a shows a group that is dead and the resulting board fragment after it is removed.



(a) (b) (c)

Fig. 4: (a) Dead group; (b) Without the Ko rule; (c) Seki.

2.1.5 *Rule of ko.* An important concept in Go is that of the rule of ko stating that no board state may be repeated, i.e., livelock is not allowed to occur. The sequence of plays in Figure 4b illustrates an example of what could happen without this rule. The first move by white ($S_0 \rightarrow S_1$) captures a black stone, while the second move ($S_1 \rightarrow S_2$) captures a white stone. This second move ($S_1 \rightarrow S_2$) is illegal, and black must play somewhere else. Without the ko rule, a livelock situation would arise and both sides would continually gain the same number of stones.

2.1.6 *Seki* (*Stalemate*). Seki can be viewed as a localized stalemate condition. In Go, there are situations when neither side can count a territory because both sides would have dead groups if they played first. Figure 4c illustrates this condition. If white's turn, it could not play at A (suicide), while playing at C would fill in its own eye. The only option is for white to play at B, but that would allow black to play at C on the next turn, capturing the white group. Likewise, if it were black's turn to play, C would be suicide and A would be filling in its own eye. Black's play at B would allow the small group to be killed with a white play at A. Thus, locations A, B, and C are not counted as territory.

2.1.7 *Scoring.* Many variations exist for scoring finished Go games, but for simplicity a player's score is calculated by first removing dead groups (which become prisoners), then counting the number of captured prisoners, and finally counting the number of intersections completely surrounded by one's own color. In many games the person to play second may get additional points called a komi to compensate for going second, which can vary from 0.5 to 5.5 points.

2.1.8 *Other board sizes.* Go can be played on boards of any size, but historically, games have been played on boards of size, 19, 17, 13, and 9 (361, 289, 169, and 81 intersections, respectively). Size 9 is used to teach beginners and is often used in computer Go games, because of its smaller search space.

2.1.9 *Go player ranking.* Go has a well standardized hierarchy of skill levels that allows players to compete on a fairly equal basis with standardized handicaps. A complete beginner that has played a game and knows the rules starts off at 30 kyu. The scale progresses to one kyu which is the best kyu ranking one can attain. After this, the scale continues at one dan up to nine dan, the highest amateur ranking possible. For the kyu ranks, lower values imply a better player, while for the dan ranks, higher values imply a better player. Professionals rank themselves on the dan scale as well from one to nine, but their rankings are usually considered stronger. Different countries and groups may not completely coincide with each other on

strength. For example, a Korean eight kyu might not be equal to a British eight kyu. Computer programs are often given honorary diplomas of a certain level, but this can be misleading as programs play well but make significant mistakes at times.

2.2 Relevant AI and Computational Techniques

Numerous AI and computational techniques have been applied to Go programs, including search techniques, neural networks, thread pools, multiagent systems, and GAs (Nichols et al. 1998; Stuart et al. 1995; Wooldridge, 1999).

2.2.1 *Search techniques.* Many search techniques are variations on breadth-first or depth-first search, including uniform-cost, iterative deepening, bidirectional, and depth-limited searches. Canonical breadth-first search expands a new tree layer at each iteration, whereas depth-first search expands one element from the next layer at every iteration. Uniform-cost search expands the next cheapest node at each iteration; and iterative deepening search is breadth-first with the number of layers increased at each iteration. Bidirectional search attempts to search from both the Goal and the starting state simultaneously. Finally, depth-limited search includes a provision for the maximum depth that will be searched before backing up.

These search methods can be improved upon by incorporating knowledge about the problem space to help make the search more efficient. The simplest informed method is a greedy search that always expands the node that appears closest to the solution. Another method is A^* and its close relative IDA* (Iterative Deepening A^*) (Stuart et al. 1995). In A^* the next node to be expanded is the node that has the lowest value for a variable defined as the cost from the initial node to the current node plus the estimated cost of the best path to the goal.

2.2.2 *Neural networks.* A neural network can be viewed as a random search method that yields a mapping function that may not be found by more traditional methods. Neural networks are often composed of multiple layers of neurons, and each neuron in each layer can be connected to the neurons in the immediately adjacent layer. Each connection has a weight (strength) associated with it, and each input neuron receives some part of an input signal which is passed through a non-linear activation function that determines if the neuron fires. A neuron that fires has its output signal multiplied by the weights which is then directed into all the connected neurons in the next layer. Each neuron in this layer receives a signal from each input neuron, and these signals are then summed and once-again passed through

an activation function to determine its output. This process continues for all neurons in all layers until an output is received at the output layer.

Many paradigms exist for training neural networks, and the most common is the backpropagation technique that compares the output of the network with a correct training example. The error between the expected and actual output is propagated backward through the network to modify the weights between the neurons. Other methods exist to modify the weights (Valluru Rao & Hayagriva, 1995). The applicability of neural networks is influenced by such factors as the number of neurons, the number of layers, connections within a layer, connections back to previous layers, the training data, the learning rate, and the training method. Much trial-and-error is involved in neural network design.

2.2.3 *Multiagent systems.* Although there is much disagreement on the exact definition of an agent, most agree that agents are indeed autonomous. An agent has been defined as "*...a computer system that is situated in some environment, and that is capable of autonomous action in the environment in order to meet its design objectives*" (Wooldridge, 1999). Intelligent agents have the characteristics of agents but also have intelligent traits such as reactivity, pro-activeness, and social ability. A robot capable of interacting in its environment might be considered an intelligent agent, whereas a home thermostat would not.

A number of issues must be addressed to build a multiagent system, such as what kind of information the agent will have about its environment. A computer program running a single thread for each agent is quite different from a robot traversing polar ice-sheets. Is the environment real or virtual? Will the agent receive all inputs through a socket, via shared memory, or by way of an external bump sensor? All of these questions are important to designing a multiagent system. Closely related to this concept of an agent's environment is the idea of an ontology which is, "*...a specification of the objects, concepts, and relationships in an area of interest*" (Wooldridge, 1999). All agents must exist within some framework. Distributed agents may communicate over a wireless network or with physical means such as actual speech. Languages and protocols exist for agent communication (Huhns & Stephens, 1999).

Agents interact to be effective. Agents can be cooperative or self-interested which would lead to different types of agent interactions. Cooperative agents may actually negotiate an agreed upon goal, whereas self-interested agents might achieve their own goals, leading to a net benefit to at least themselves.

2.2.4 *Genetic algorithms.* Genetic algorithms are search methods based on simulated evolution in an attempt to find a solution or goal for a particular problem (Mitchell, 1999). Essentially, a GA begins with a set of random chromosomes of alleles. These "arrays of bits" are translated into parameters or data that can then be tested to see how well they approximate the solution that is being sought. The function that determines how well an individual chromosome performs is called a fitness function. The fitness of each chromosome is calculated and pairs of chromosomes are selected, with those with higher fitness values more likely to be picked. These pairs undergo crossover and/or mutation. The resulting chromosomes become new members of the population. This process is repeated until a chromosome with some minimum fitness is found, or reaching a maximum number of generations.

The efficacy of using GAs relies heavily on the choice of fitness function, the size of each generation, the maximum number of generations, the crossover rate, the mutation rate, and sometimes a scaling factor (Goldberg, 1989). Many variations have been proposed that modify the basic GA paradigm, such as a co-evolution method that evolves two separate populations that compete with each other after each generation (Rosin & Belew, 1995). This is analogous to different species competing in the real world.

2.2.5 *Thread Pools.* Although not an AI technique, the concept of thread pools is related to this work. Thread pools begin with a finite number of threads, each capable of doing some work. When it becomes available, work is given to a thread in the thread pool. If there is more work to do than threads available, then the work must wait to be done. This method saves some overhead because each time work must be done, the operating system does not need to create and destroy a new thread which can take much time. A problem with this method is that an optimal number of threads to start within the pool must be determined. The number of processors, the complexity of the problem, and the amount of work to be done all affect the usefulness and size of thread pools.

2.3 AI and Games

Artificial intelligence techniques have been applied to the development of game playing programs, particularly two-person games of perfect information. One such technique, namely Minimax search, involves enumerating all possible moves for one player, followed by enumerating every possible response to each of these moves.

This process is continued until all the leaves of the tree represent final game states. A tree such as this allows the program to play a game perfectly, but for all but the most trivial games this approach requires too much time and memory. As a compromise, programmers can allow the tree to expand to a certain level and then assign an estimate of the quality of the game state using an evaluation function. At any point, the minimax tree allows the picking of the best move assuming the evaluation function is accurate. The issue is that the evaluation function only approximates the quality of a position and could be difficult to evaluate.

The alpha-beta pruning techniques address this inefficiency by keeping track of the highest and lowest evaluation values. Assuming that players always play the best move possible, one can prune tree branches that are worse than some other possible move that the algorithm has seen before. For example, while performing a depth-first search one finds that one player can achieve an evaluation of nine. Then, deeper in the search, a possible sequence of moves leads to a value of five. The subtree with the value five will be pruned and no further moves from that path will be considered. This method always returns the same value as pure minimax search with depth-limitation, so it is often used.

Games of perfect information such as Reversi, Pente, checkers, chess, and go-moku can derive benefits from AI techniques. The main difference between these games and Go lays in their branching factors and the manner in which each piece affects other pieces. Reversi, checkers, and chess all have relatively small branching factors, making them much more conducive to alpha-beta search with move-ordering and other advanced pruning techniques. For example, minimax search with alpha-beta pruning can be improved by ordering the moves at each node in the search tree in an attempt to allow pruning to remove more nodes (Norvig, 1992). This ordering can be accomplished if certain locations on the board are more advantageous than others, i.e., better moves are placed first. Pente and go-moku have branching factors similar to Go, but have much simpler interactions between the pieces.

Another useful technique is generating abstract heuristic values that are relevant to the game, such as mobility in the game of Othello or pawn structure in the game of chess (Norvig, 1992). Go, for example, has potential candidates for approaches of this nature, such as thickness and good shape that describe abstract concepts that relate to good moves. In Go, one should build thickness and make good shape. Another method is forward pruning, which requires a function that removes obviously poor moves from the search but is difficult to do and very subjective. Although out of favor as a rigorous

method, this approach is a necessity for games with large branching factors. Programs that think while the opponent is playing can gain some advantage, and the use of board hashing and opening book databases can help the programs' strengths as well. Also, exhaustive searching near the end of a game can be an option for some games such as Othello, but may not be feasible in Go.

2.4 AI and Go

2.4.1 *Search space.* The number of possible states on a Go board of size, 19 is $3^{19^2} \approx 1.74 \times 10^{172}$ as there are 19^2 intersections on the board, and each location has three possible states of black, white, or empty. Although many states are very unlikely to occur, and accounting for symmetries (color-inversion, rotational, reflection) and that the numbers of stones of each color are usually roughly the same, the size of the search space is extremely large. One of the greatest difficulties in programming Go is the immense branching factor in the game. The first move in a game of Go can be any one of $19^2 = 361$ moves, whereas chess has only 20 initial moves and Reversi has only four initial moves. Although the number of possible moves fluctuates as play progresses, these games cannot be compared with the order of magnitude difference in the branching factor of Go. Chess has $20 \times 20 = 400$ game states after the first two moves, while Go has $19^2 \times (19^2 - 1) = 129{,}960$ game states. Including a third move brings chess up to approximately 10,000 states, whereas Go has 46,655,640 states. This illustrates why Go cannot be played well by brute force.

2.4.2 *Neural network techniques for Go.* Many researchers have used neural networks to play Go, including neural networks with GA-evolved weights. One approach created an architecture for a program called Neurogo that evaluated the board using a neural network with backpropagation and temporal difference learning (Enzenberger, 1996). The network received its input from a feature expert while a relation expert controlled the connections between the network layers. An external expert was included that could override the network's output for a small class of problems. The novel idea was using experts to extract features from the gobans–another term for go boards.

Neural networks have also been used evolved using GAs (Donnelly et al. 1998). A 9x9 board was used along with a three layer non-recurrent network. It was postulated that recurrent networks with more than a single hidden layer might be better suited for the non-linearities of Go. The experiments consisted of generating a population of 32 networks that played against each other. The network winning the

most games overwrote the network that lost the most at the end of the cycle. The networks were used to evaluate the quality of a given position which was accomplished via a single output neuron and a set of input neurons that derived their inputs directly from the goban. Each location on the board corresponded to three individual input neurons (white, black, and empty). The resulting input layer had $9 \times 9 \times 3 = 243$ neurons. The networks were found to slowly get better but still played poorly compared with modern Go programs.

2.4.3 *Traditional techniques in Go programs.* Certain Go programs do not use any soft computing technique, i.e., the programs do not rely on learning, GAs, neural networks, cellular automatons, or other similar approaches. In one approach to Go, moves were generated by first enumerating possible moves based on small, local views of the goban (Müller, 1999). These moves are filtered, ordered, checked, and refiltered, and the best move was executed. If a ko ensued, then a special ko module was called. If no move survived this process, then the program passed. At the core of this approach was a pattern-matching database that used Patricia trees—a method normally used to search large text databases like dictionaries. This program contained about 3000 patterns. This reflects a prominent trend across many Go programs: they often rely heavily on vast databases of patterns that have been built manually. These pattern databases make these programs better; however, the implementation of these databases is not a trivial task. The possibility does remain of learning patterns as the program plays.

Another prominent program, The Many Faces of Go, had an opening move database containing around 45,000 moves and a pattern database of about 1,000 patterns (Fotland, 1993). This program contained a rule-based expert system with around 200 rules used to suggest moves to look into further. Additionally, dynamic knowledge was stored about the state of the board.

These traditional techniques represent some prominent themes in most strong Go programs: they construct meta-data based on the state of the board and use this meta-data along with large databases of patterns to decide on moves. Rarely does learning or extensive minimax-style search play a role in the skill of these games.

2.4.4 *GA techniques for Go.* Many attempts have been made to build a program that plays Go by using GAs. In one approach, GAs were used to evolve a Go evaluation function for 7x7 boards (da Silva, 1996). The evaluation function worked by attempting to translate a given board into a new board that represented the final configuration of the game. The evaluation function then looked at who won to calculate the fitness.

Essentially, the GA attempted to evolve an evaluation function that could be used in minimax searches with alpha-beta pruning. The evolved parameters were a set of low-level functions that performed simple calculations based on the board state. These functions, organized as the chromosome dictates, produced what was called an S-expression–a significant component of calculating a board evaluation and consequently the fitness. This approach yielded a player that on average never defeated an opponent called Wally, a freely available public domain Go program.

In another approach, a program was written using GAs to play Go (Greenberg, 2001). Knowledge in this program was represented by triples similar to Prolog predicates; and these statements could be nested. Each variable comprised a board location, the color (white, black, or empty), and the action to take (move, pass, or resign). Although this possibly layered traversal of the statements, moves were chosen. The program, "*...was very poor at breeding individuals that could match. And when it did, the individual would often resign after but a few moves*".

Researchers have used GA and the game of Go to create algorithms that incorporate qualities of true human experts (Kojima et al. 1997). One inclusion was to incorporate useful but infrequently used rules, and another was to model ecological systems. The ecologic models dictated that many species coexist. The idea was that species live together in an environment, yet they can be radically different. Rules, in their system, increased in number and ate virtual food. Rules whose activations decreased to zero, died, whereas rules whose activations became too high split into the original rule and a more specific rule. A training datum was considered food, which was eaten by a rule that matched it; the activation value of the rule then increased. The GA was used entirely to evolve rules based on patterns found on the board. The playing skill of the program was not reported, but the rules that the program generated were shown to Go experts. The experts determined that about 42% of the rules were good, 21% were average, and 37% were bad.

2.4.5 *Other techniques and hybrids*. One approach is based on a system that evolves neural networks to play Go by using GAs (Richards et al. 1998). The program, called SANE, started with no prior Go knowledge. The process evolved individual neurons using crossover mutations and random point mutations. Each neuron was defined as a set of bits that described connections and the connection weights. Each neuron had a fixed number of connections, but each connection could be attached to either the output or the input layer. Network blueprints (sets of neurons that work well together) were also evolved along with the individual neurons. Entire networks were

evolved based on the final state of the game rather than assigning credit to individual moves. One could argue, however, that perhaps game records between two masters represent on average the best move at each point in the game.

The evolution of neural networks on a variant of the SANE architecture that evolves individual neurons, but evaluates the fitness of entire networks is presented in (Daniel & Risto Miikkulainen.2000). In addition, blueprints evolved. These neurons are only for the single hidden layer of the network. SANE has been shown to work well in continuous domains and gamesthat have hidden state information.

3. METHODOLOGY

The approach proposed in this paper consists of a three layer summation network with each layer fully connected to its adjacent layers. Each connection is characterized by an integer weight, and each node's sums arrays. These arrays each contain an element that corresponds to a location on the board (a one-to-one mapping). The cornerstone of this design is to evolve these weights using GA, thus each chromosome specifies a set of integer weights for the summation network. The initial inputs to the network are the outputs from the individual agents.

3.1 Design Overview

The system provides the end user with the ability to run regressions, evolve a GA player using stored game training sets, and play a human player with extensibility in mind to allow future integration with IGS (Internet Go Server) and gomodem (a protocol for serial communication between two computers, each playing Go). The software was designed according to the object-oriented paradigm and consists of a moderator that allows two move generation classes (called interfaces) to play against each other. Through this abstract interface class, an ASCII text player was developed that interfaces with a human user, a simple Perl/Tk interface that also interfaces with a human user, a GA player, and a GA trainer that is designed to play against the GA player in order to calculate the fitness of the GA player.

3.2 Stone, Board, and Game Classes

The stone class represents a single location on the goban, which was implemented with speed as the primary concern. It uses bit operations to test various traits of a location such as if the location has a black stone or a white stone. It also has functions

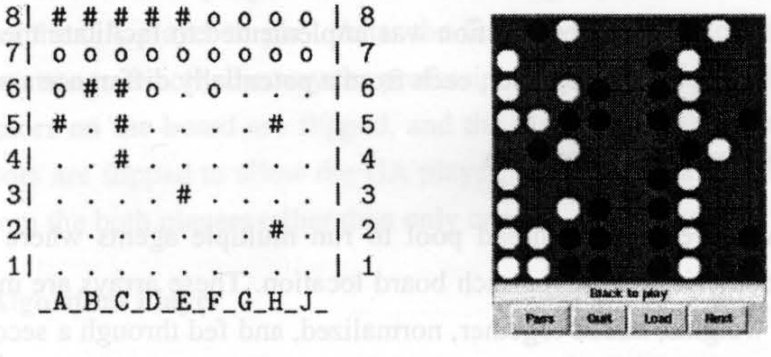
that test if the location is on an edge. The next layer of abstraction encapsulates the concept of a board, which is a one-dimensional array of stones. A one-dimensional array was chosen to speed up board manipulations by reducing the need for pointer arithmetic that is required in multi-array offset calculations. On top of the board abstraction there is a game class that stores a linked list of boards and keeps track of which side's turn it is. The game class enforces certain optional rules like whether to allow suicide and also provides such functions as play a move and legal.

3.3 User Interfaces

The software contains two distinct user interfaces, namely, a text interface and a graphical user interface (GUI). The text interface displays the goban using ASCII characters with a # representing black and an o representing white. Figure 5a shows an ASCII board for a 9x9 game. This interface is useful when visual appeal is not an issue, such as testing code. The other interface is a GUI interface that uses an external Perl/Tk program to display the goban. Figure 5b shows a screen shot. This interface is important for playing games against the program, as a graphical board is easier to interact with. This interface allowed for a quicker way to play with the program in an attempt to test and correct the defects.

3.4 Genetic Algorithm

The code for performing GAs was originally based on Goldberg (1989), converted to C++, and syntactically modified to better suit an object-oriented approach. The GA



(a) (b)

Fig. 5: (a) An ASCII board; (b) Graphical user interface screen shot.

code allows for the choice of fitness functions. The program keeps statistics on the performance of the GA and tracts the minimum, maximum, sum, average, variance, and standard deviation of the fitness values from each generation. The F-test and T-test values are computed for each generation and compared with the initial generation. The F-test calculates whether two distributions have significantly different variances. The (Student's) T-test measures whether two distributions have significantly different means. Two versions of the T-test were used for distributions with statistically different variances and statistically identical variances. The best chromosome is extracted following the calculation of the fitness values from the last generation and used to configure the GA player, which then plays against an opponent that serves as a trainer. The trainer plays its moves based entirely on recorded games. The GA player is further tested by a similar trainer opponent, but this time with a different set of recorded games.

3.5 Moderator

The moderator essentially loads two move generators, which can be from a user interface, a random move generator, or a GA player, and also a GA trainer. The moderator is multi-threaded (multiple concurrent processes), allowing a thread for each move generator. This design allows both sides to have processing time throughout the entire game–not just during one side's turn. Message passing is used to allow communication between the moderator and the two movegenerators.

The probability board stores an array of values which correspond with the locations on the goban. The semantics are such that the values at each element represent how highly that location is valued as a possible next move. Each agent constructs one of these, and a function was implemented to choose a move probabilistically, and another function was implemented to facilitate the addition of two or more of the boards together, each from a potentially difference source.

3.6 Agent Network Architecture

The GA player uses a thread pool to run multiple agents where each agent generates a numerical value for each board location. These arrays are multiplied by GA-evolved weights, added together, normalized, and fed through a second layer of summation nodes. The resulting array is then normalized. The highest value in the resulting array then becomes the move to be played. This process can be described as follows:

- Each of N agents computes a value for each location on the goban. This probability board is a vector and is denoted as where is the agent number.
- Each of the K second level nodes γ_k sum all β_n values multiplied by a scalar value $w_{k,n}$. Thus, $\gamma_k = \sum_{n=0}^{N} w_{k,n}\beta_n$
- These γ_k are vectors that are then normalized so that the values add up to 1 in each vector, unless all values in a vector are zero, in which case they are left that way.
- These normalized γ vectors are then multiplied by a second set of weights and added together: $\varepsilon = \sum_{k=0}^{K} w_k \gamma_k$
- This final vector, ε, is normalized and represents a distribution of which move to play. The first highest value is then chosen.

This approach theoretically allows for a large number of agents, limited primarily by the size of the thread pool and the number of processors available to the program. A major goal of this project was to create a design that was scalable and could benefit from a highly parallel machine. Although scalability was not tested, the possibility of adding more agents could easily be realized. Figuring out what each agent would do could become a significant bottleneck, though.

3.7 Genetic Algorithm Trainer

The move generator was designed to play against the GA player. The move generator reads a sequence of moves from a data file that is derived from recorded games of professionals in the public domain. The move generator sets up the board and then allows the GA player to play. After the GA player has played, the trainer resets the game state to exactly the way the professional actually played in the game record. The colors on the board are flipped, and the GA player is allowed to play again. The colors are flipped to allow the GA player, which plays a single color, to gain benefit from the both players rather than only one

3.8 Genetic Algorithm Player

The GA player loads the parameters for the weights in the summation network and computes the move to play by running the agents, filtering their values through the summation network, and then picking either the first highest value or normalizing

and then choosing the move probabilistically. The fitness is calculated by setting up a goban, as dictated by stored games from the Internet. The fitness of the GA player is the percentage of moves correctly played. Many other GA Go programs calculate fitness by using some variation of attempting to guess how the current board configuration relates to the final division of points at the end of the game. This approach sidesteps this difficulty which relates closely with the difficulty of scoring a finished game.

3.9 Agents

Six different agents are designed and implemented, where they choose moves in significantly different ways. These includethe following:

a. a random agent that plays random legal moves,
b. a follower agent that tries to play close to the opponent,
c. an opener agent that plays in the locations usually played in at the beginning of a game,
d. a capture agent that attempts to kill groups by reducing other groups' liberties,
e. an agent that attempts to create a strong configuration known as a tiger's mouth, and
f. an extension agent that favors moves close to the friendly stones.

The random agent plays random legal moves and is developed to allow the testing of code that directly uses the agents and to allow the testing of the code that lets the agents interact. Additionally, the random agent is used as a baseline for comparison with other agents. The standard by which the success of the GA is judged is the set of five random agents.

The follower agent prefers playing on locations adjacent to opponent stones. As is often found in games of Go, many good moves are often near opponent stones, i.e., attacking them. Playing close to opponent stones not only attacks them, but also attempts to push the opponent group in the opposite direction.

The opener agent suggests moves around the perimeter of the board near the third or fourth row. The values decrease the further the game progresses. The reasoning behind this type of agent is that at the very beginning of most games, stones are played near the edges and sides because this is where it is easiest to make territory. In a corner, one only has to worry about attacks from two directions. On a

side, attacks are only possible from three directions, whereas in the middle, attacks can be made from all directions. These considerations justify having an opener agent.

The capturer agent attempts to capture opponent stones by filling in their last liberties. This agent has no knowledge of living or dead groups, thus it plays by calculating which groups have one or two liberties left and then plays in those liberties. This agent does not take into account moves that would reduce a friendly group's liberty count down to one. This agent would play a move that reduces an opponent's group to a single liberty while that move may allow the opponent to capture a friendly group on the next turn.

The tiger's mouth agent attempts to create a powerful configuration called a tiger's mouth. This formation retains the same name regardless of all symmetries. This configuration is considered strong because it allows three stones to not be connected while retaining the ability to become connected by playing in the center location. Another strength is that if an opponent stone tries to keep these stones from connecting, then that opponent stone can be captured on the next turn if it is not part of another group. The versatility of this formation provides justification for the inclusion of this agent.

The extension agent plays many of the common extensions. Each type of extension has a different weight (or value) that is derived from the GA chromosome. The extension agent is also the only agent that uses alleles from the chromosomes to set these internal values. The alleles specify the relative value of each extension type. These extensions are shown in Figure 6.

4. EXPERIMENTS AND RESULTS

The experiments and their results show the successful evolution of summation network weights for a multiagent approach to playing Go. The key point is that

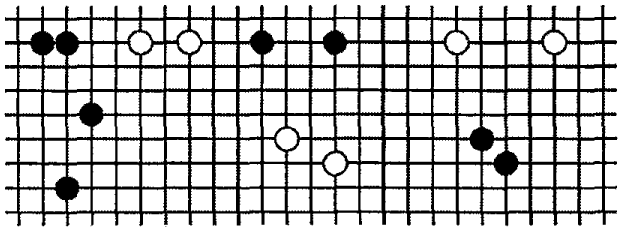


Fig. 6: Extensions

although each individual agent may play poorly, the agents playing together actually play better. Random search using a GA finds weights for the summation network that improve over multiple generations. The fitness function for the experiments uses recorded games from 9x9 games between professionals from the public domain that occurred between, 1995 and 2000 on an international Go server (Anonymous 2001).

4.1 Individual Agent Experiments

The GA was run for a number generations with the program configured to use only a single agent. These agents were the random, follower, opener, capturer, tiger's mouth, and the extension agents. In each case, the populations contained 10 individuals. Such a small population and small number of generations were used because no major changes were observed when these parameters were set to higher values, and because of the large amount of time it took to run the GAs with this configuration. These runs took around three days on a dual-processor, 1.2GHz computer. The crossover value was 40% with a mutation probability of 0.0333. The F-multiplier was 2. The random agent was used as a baseline. After evolution, the best individual in the final population was used to play against a testing data-set which consisted of game records that were different from those used for training.

4.1.1 *Opener Agent.* Figure 7 shows the results of the GA run using only the opener agent. Because the evolved weights are not used if a single agent is used, the fitness values did not change. The best chromosome of the last generation chose 1.14% of the training moves correctly and 1.55% of the testing moves correctly. Considering that this agent was designed to play opening moves, that it fared poorly is not a surprise.

4.1.2 *Single Random Agent.* The single randomly playing agent did not fare well. Since the random agent always picks legal moves, the number of possible moves near the end of any game becomes smaller, which increases the likelihood that a random guess would be correct. The random agent is not actually randomly choosing locations to play, but instead assigns the same value for every legal position. The final resulting probability board contains an array of values, which in this case would all be the same. The program can be configured to either pick the first highest or to pick one probabilistically. For these experiment–and all of the others–the former method was used. The result is that the first legal move is always chosen, which ends up being correct a static number of times. The testing data yielded 0.62% correct moves because choosing the same legal location is correct a certain number of times.

4.1.3 *Extension Agent.* The extension agent has internal parameters that derive their values from the evolved chromosomes. Figure 8 shows the results of the GA run using only the extension agent. As the generations progressed, the mean fitness and the maximum fitness increased. Additionally, the minimum fitness had a net decrease of 0.0301. The F-test predicted that the first and the final generations had insignificantly different variances. The T-test was used to predict with a probability

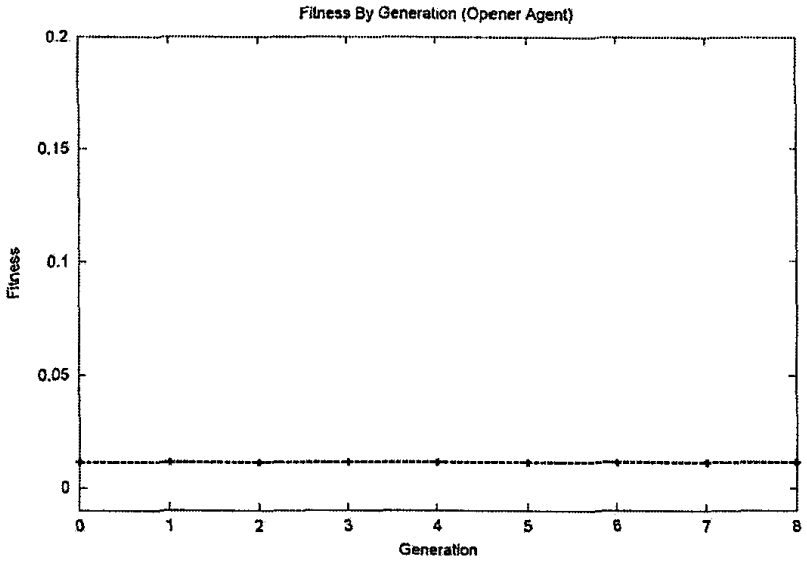


Fig. 7: GA with the opener agent.

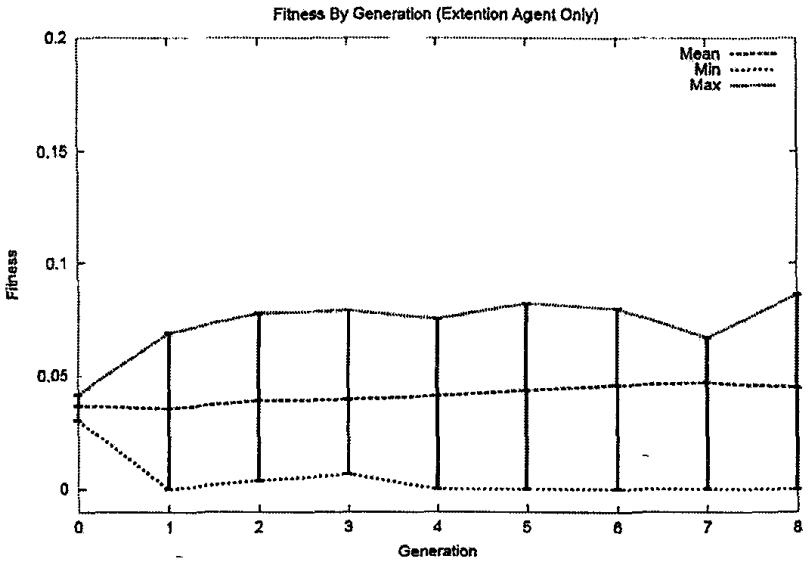


Fig. 8: GA with the extension agent

of 99.9642% that the improvement was real, i.e., not a result of chance. The best chromosome of the last generation chose 5.2% of the training moves correctly and 3.8% of the testing moves correctly.

4.1.4 *Other Agents.* The results for the capturer agent did not improve because this agent lacks internal parameters that could benefit from evolution. The best chromosome from the last generation correctly determined 7.8% of the training and 4.0% of the testing moves. For the follower agent, the best chromosome of the last generation correctly determined 4.2% of the training and 3.3% of the testing moves. In case of the tiger's mouth agent, the best chromosome from correctly determined 2.4% of the training and 2.2% of the testing moves.

4.2 Multiagent Experiments

The multiagent experiments were similar to the individual agent experiments with the exception that in these cases the program was run with all of the agents at once, excluding the random agent. A separate run that included five random agents was used as a baseline for comparisons. Not surprisingly, the GA configured with five identical random legal move generating agents performed rather poorly. The results, shown in Figure 9, were nearly identical to those resulting from the single random agent.

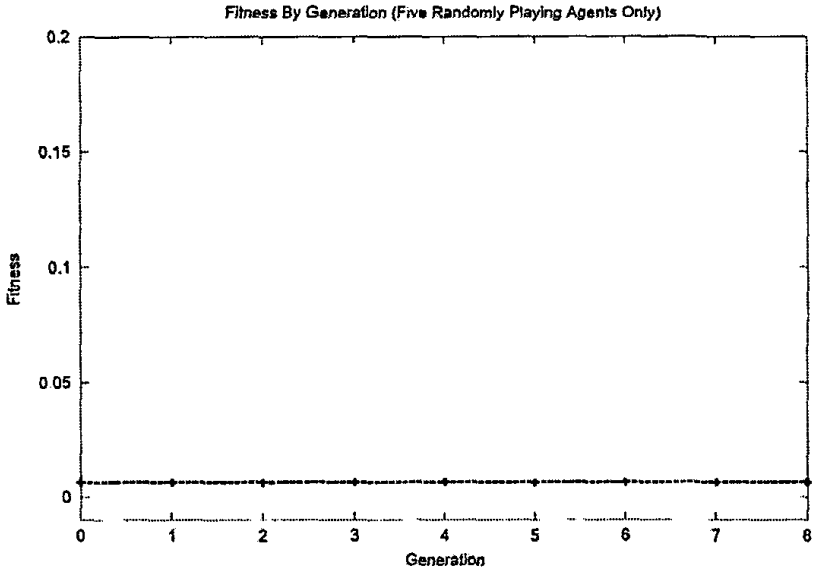


Fig. 9: GA with five random agents

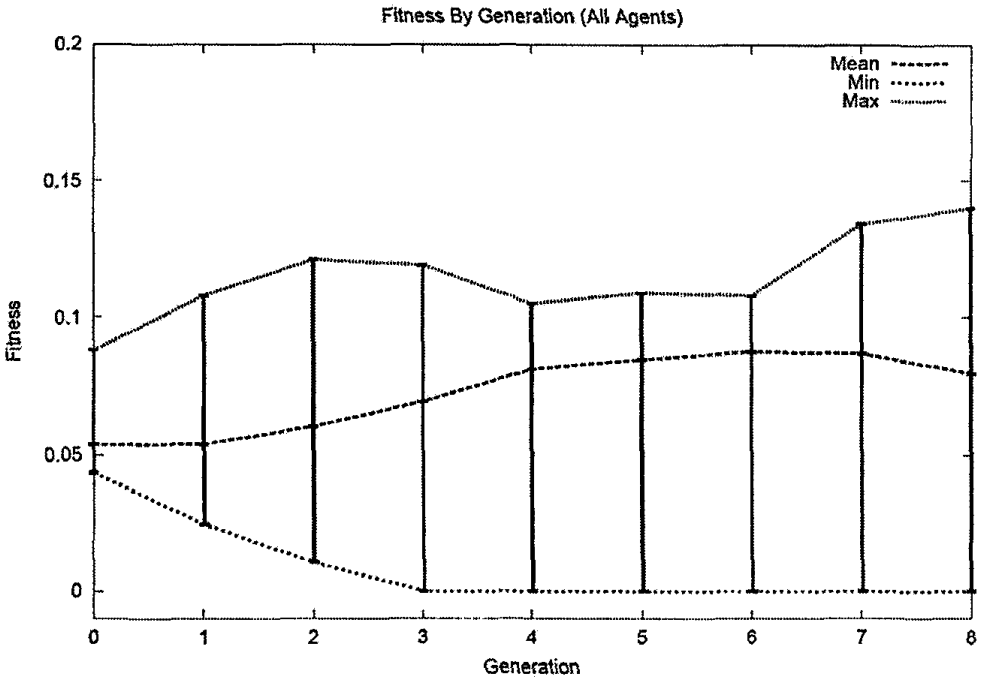


Fig. 10: GA with all agents.

Figure 10 shows the results of the algorithm using five agents, loaded in the following order: opener, tiger's mouth, capturer, follower, and extension agents. Three hidden-layer nodes were used, and each generation had 10 individuals. Initially, the maximum fitness was 0.0881 and the mean fitness was 0.0537. By the final generation, the maximum fitness had risen to 0.14 and the mean fitness had risen to 0.0798. The question then becomes one of deciding if this difference should be attributed to chance or to actual improvement. Using the F-test, the difference in the variances was not significant. The T-test value of the final generation was –4.23, which implied a probability of 0.000504 that these results were from chance and not from a different population as the initial population, i.e., the confidence interval was 99.95% that the difference in the means was significant. The best chromosome from the final generation determined 10.2% of the moves correct while it got 5.6% of the testing set correct. Additional experiments were performed with multiagent configuration with a population size of 100. All parameters were the same as for the smaller multiagent configuration except for the population size. The results were similar to those obtained from the smaller multiagent experiments.

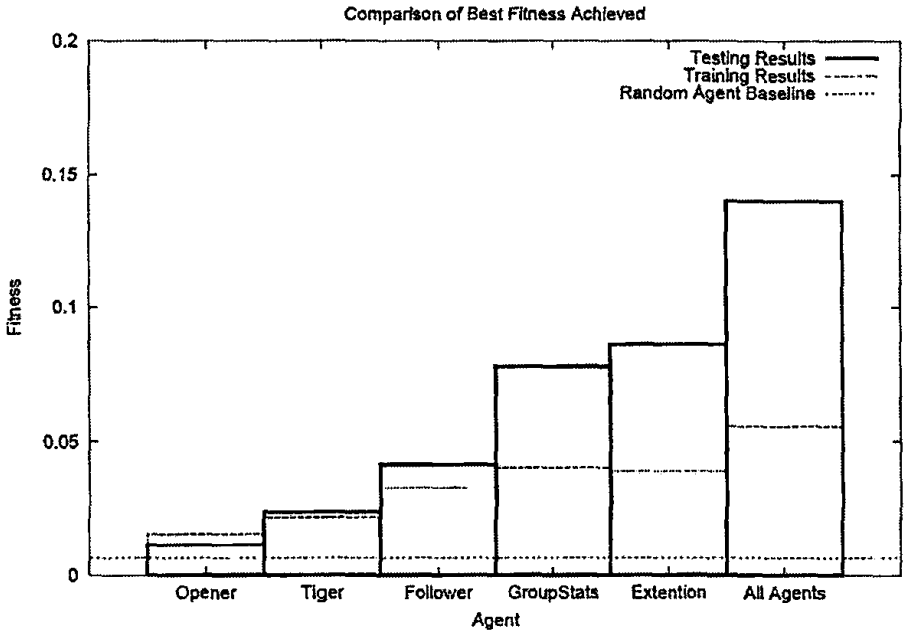


Fig. 11: Agent comparison.

Figure 11 shows a comparison of the best fitness values achieved by all agent configurations. The randomly playing agent played the poorest, and the two configurations that could benefit from the GA (extension agent and the multiagent systems) actually did. The testing data showed some variability, and in some cases an agent that performed better on the training data did worse on the testing data. Overall, there was a benefit in using the GA to evolve Go player multiagent systems.

CONCLUSION

We found that a multiagent approach using a summation network does indeed yield a viable Go player. Furthermore, improvements were gained over the course of multiple generations. In addition to these results, a unique approach to playing Go was demonstrated. We showed that it is feasible to build a system that plays Go using probabilistic methods incorporating multiple agents whose interactions (the summation network) have been evolved or learned in some way. This approach shows that it may be possible to break down certain large intractable problems and use genetic algorithms to combine multiple sources of information without knowing exactly how the information interacts to form a solution. The presented architecture

exemplifies the possibility of trading the ability to fine-tune the behavior of a system with the ability to scale the system.

This approach to playing Go has a number of potential limitations. Foremost, a multi-agent approach relies heavily on the ability of the designer to create agents that contribute to the skill of the system. Generating good agents is a challenge. Another limitation is that genetic algorithms take long periods of time to run. Larger training sets, larger testing sets, larger populations, more intricate summation networks, and more generations could all help improve the program, but unfortunately all of these would contribute to a significantly slower program.

Although scalability was an important goal, the realization of a massively parallel multiagent Go program must be quelled by the prohibitive cost and the scarcity of machines with dozens of processors. The future may not hold a limitation such as this, but currently it is a real limitation to increasing the number of agents extensively. Yet another restrictive aspect of this work was the use of 9x9 boards, as experimenting with, 19x19 boards would have taken too long. Future work includes testing larger networks utilizing a larger number of agents, and using larger boards.

In conclusion, we have shown that a multiagent approach using a summation network does indeed yield a viable Go player. Furthermore, improvement was gained over the course of multiple generations. Perhaps in the decades to come someone will create a Go program that can play at the level of the masters. This is a goal that many await patiently.

REFERENCES

da Silva, S.F. 1996. *Go and genetic programming: Playing Go with filter functions.* Master's thesis, Universiteit Leiden; Computer Science Department.

Donnelly, P., Corr, P. and Crookes, D. 1998. Evolving Go playing strategy in neural networks, *Applied Intelligence*, **8(1)**, 85-96.

Enzenberger, M. 1996. *The integration of a priori knowledge into a Go playing neural network.* http://www.cgl.ucsf.edu/go/programs/neurogo-html/neurogo.html, September, 1996.

Fotland, D. 1993. *Knowledge representation in the many faces of Go.* Available at: http://www. smart-games.com/knowpap.txt

Goldberg, D.E. 1989. *Genetic algorithms in search, optimization, and machine learning*, Addison-Wesley.

Greenberg, J. 2001. *Breeding software to play the game of Go.* Available at: http://www. inventivity.com/OpenGo/Papers/jeffg/breed.html.

Huhns, M.N. and Stephens, L.M. 1989. *Multiagent systems: a modern approach to distributed artificial intelligence,* first edition, Cambridge, Massachusetts, The MIT Press, 79-118.

Kojima, T., Ueda, K. and Nagano, S. 1997. Evolutionary algorithm extended by ecological analogy and its application to the game of Go. *Proceedings of the fifteenth international joint conference on artificial intelligence (IJCAI-97)*, The University of Tokyo, College of Arts and Sciences, 684-9.

Müller, M. 1995. *Computer Go as a sum of local games: an application of combinatorial game theory.* PhD dissertation, ETH Zürich.

Mitchell, M. 1999. *An introduction to genetic algorithms,* first edition, Cambridge, Massachusetts, The MIT Press.

Nichols, B, Buttlar, D. and Farrell, J.P. 1998. *Pthreads programming; A POSIX standard for better multiprocessing,* Cambridge, Massachusetts, O'Reilly & Associates, Inc., 1-60.

Norvig, P. 1992. *Paradigms of artificial intelligence programming: case studies in common lisp,* San Francisco, California, Morgan Kaufmann Publishers, 596-653.

Polani, D. and Miikkulainen, R. 2000. Eugenic neuro-evolution for reinforcement learning. *Proceedings of the genetic and evolutionary computation conference (GECCO-2000)*, edited by Whitley, D., Goldberg, D., Cantu-Paz, E. Spector, L., Parmee, I. and Beyer, H.-G., Las Vegas, Nevada, USA, San Francisco, California, Morgan Kaufmann, 1041-6.

Press, W.H., Teukolsky, S.A., Vetterling, W.T. and Flannery, B.P. 1997. *Numerical recipes in C: The art of scientific computing*, second edition, The Press Syndicate of the University of Cambridge, 227, 616-9.

Rao, V. and Rao, H. 1995. *C++ neural networks and fuzzy logic*, second edition, Cambridge Massachusetts, The MIT Press, a subsidiary of Henry Holt and Company, Inc., 1995.

Richards, N., Moriarty, D. and Miikkulainen, R., 1998. Evolving neural networks to play Go. *Applied Intelligence*, **8**, 85-96.

Rosin C.D. and Belew, R.K. 1995. Methods for competitive coevolution: Finding opponents worth beating. *Proceedings of the sixth international conference on genetic algorithms,* La Jolla: CA. Cognitive Computer Science Research Group, Department of Computer Science and Engineering, University of California, ICGA.

Russell, S.J. and Norvig, P. 1995. *Artificial intelligence: a modern approach*, first edition, Prentice Hall, 122-45.

Wooldridge, M. 1999. *Multiagent systems: a modern approach to distributed artificial intelligence*, first edition, The MIT Press, 28-31.