Computer Go: A Grand Challenge to AI

Xindi Cai and Donald C. Wunsch II

University of Missouri – Rolla

Summary. The oriental game of Go is among the most tantalizing unconquered challenges in artificial intelligence after IBM's DEEP BLUE beat the world Chess champion in 1997. Its high branching factor prevents the conventional tree search approach, and long-range spatiotemporal interactions make position evaluation extremely difficult. Thus, Go attracts researchers from diverse fields who are attempting to understand how computers can represent human playing and win the game against humans. Numerous publications already exist on this topic with different motivations and a variety of application contexts. This chapter surveys methods and some related works used in computer Go published from 1970 until now, and offers a basic overview for future study. We also present our attempts and simulation results in building a non-knowledge game engine, using a novel hybrid evolutionary computation algorithm, for the Capture Go game.

1 Introduction

Games have served as one of the best test benches in artificial intelligence fields since shortly after computers were invented. Human-designed computer game engines have beaten their designers in varieties of games, from those as simple as Tic-Tac-Toe to as complex as Chess. The brute-force search algorithm, combined with an expert database, achieved noteworthy success, given the computational power of current machines, when IBM DEEP BLUE beat the World Chess Champion Garry Kasparov in 1997 [21]. Unfortunately, this game's tree search approach is hampered by the traditional Chinese game Go.

Unlike most other games of strategy, Go has remained an elusive skill for computers to acquire, and it is increasingly recognized as a "grand challenge" of artificial intelligence, which attracts researchers from a diversity of domains, such as game theory [2], pattern recognition, reinforcement learning [47], and even cognitive psychology [10].

Even though computers still play Go at an amateur level, numerous publications demonstrate some quite meaningful exploration that helps us to understand the nature of this computer game better. In our review of computer Go literature on what has been achieved, where the barriers lie, and how to make a breakthrough, we emphasize that developing an efficient self-learning mechanism in Go may best reveal the nature of goal-driven decision making and interacting in a range of environments, where strategies are acquired to allocate available resources in achieving maximum payoffs, either short-term or long-term. Even though other AI approaches are promising in computer Go, we particularly favor applying reinforcement learning and neural network techniques to build a zero-knowledge board evaluation function. Such neural network evaluators have been successfully trained by evolutionary algorithms in Checkers and Chess [6, 7, 16]. We utilize a hybrid of an evolutionary algorithm and particle swarm optimization to train our neural network evaluator and obtain encouraging results on Capture Go, a simplified version of Go. The success in learning one of the key strategies of Go, i.e., capture and defense, through Capture Go will demonstrate that other strategies of similar complexity are solvable with the same technique. With more strategy boxes accumulated, this divide-conquer approach may result in an overall game engine incrementally built in a hierarchy of high levels, employing those boxes as the building blocks and eventually becoming competitive against human beings.

The chapter is organized as follows: Section 2 provides background information about Go and some comparison between computer Go and a Chess program. In Section 3, we track the development of computer Go and focus on several successful programs. The theme for Section 4 is the current promising approaches in self-learning with little built-in knowledge where conventional architecture and algorithms have failed in computer Go. In Section 5, we present our simulation results in training a zero knowledge game engine to play Capture Go using an innovative hybrid population computation. We round off the article with an overview for future directions of study and conclusions in Section 6.

2 Background

This section furnishes some background knowledge for both the traditional game Go and the computer version so that we can discuss the methods presented in this chapter. First, we introduce the game Go with its terms and rules. Then, we briefly discuss the techniques used for computer games. Finally, we compare computer Go and computer Chess to show why the conventional methods failed.

2.1 The Game Go

Go is a deterministic, perfect information, zero-sum game of strategy between two players. Players take turns placing black and white pieces (called stones) on the intersections of the lines in a 19x19 grid called the Go board. Once played, a stone cannot be removed unless captured by the other player. To win the game, each player seeks to surround more territory (empty grids) with one's own stones than the opponent.

Adjacent stones of the same color form strings, and hence groups; an empty intersection adjacent to a stone, a string, etc. is called its liberty. A group is captured when its last liberty is occupied by the opponent's stone. A player cannot make a suicidal move by placing a stone on an intersection with no liberty. An eye is a formation of stones of special significance in Go. When an empty intersection is completely surrounded by stones of the same color, it is known as an eye. An opponent cannot place a stone on that intersection unless it is a capturing move, i.e. unless placing the stone causes one or more of the stones surrounding the intersection to be captured. A string with two eyes cannot be captured because filling one of the eyes would be a suicidal, and therefore illegal, move. Having formed two eyes, or having the potential to do so, is the line between "alive" strings and "dead" ones. Those strings that are incapable of forming two eyes will be considered as captured and hence are removed when calculating the territories at the end of the game. Evaluating whether stones can be formed into strings, furthermore into groups, and whether strings are capable of forming two eyes is the fundamental skill in Go as it represents the game strategies of "attack," making ourselves strong and being aggressive, and "defense," avoiding vulnerabilities and surviving.

To prevent loop, it is illegal to make moves that recreate prior board positions (rule of Ko). The rule for Go is simple: one can place his/her stone on any empty intersection unless it is a suicidal or Ko move. A player can pass his/her turn at any time. The game ends when both players pass in consecutive turns (see Fig. 1). There are excellent books available on the game of Go [43].

2.2 Techniques Used in Computer Games

Claude Shannon [42] proposed that a mechanical algorithm could play a game if that algorithm contained two ingredients: an evaluation function—a mathematical formula that assigns credits according to different board positions—and a rationale, which he called "minimax," that seeks to minimize the maximum damage that the opponent can do in any circumstance.

The evaluation function quantifies how good or bad each legal move on the board was, while the minimax procedure provides a way to evaluate the possible alternative positions, given the credits from the evaluation function, by favoring the position that had the least advantage for the rival. This mechanism is the bar code of almost every computer game product.

Numerous algorithms, such as minimax, Alpha-Beta search, MFD, and different parallel versions, are proposed in the minimax procedure for the purpose of achieving a deeper and wider search with the same computational power. On the other hand, the evaluation function is evolved from static

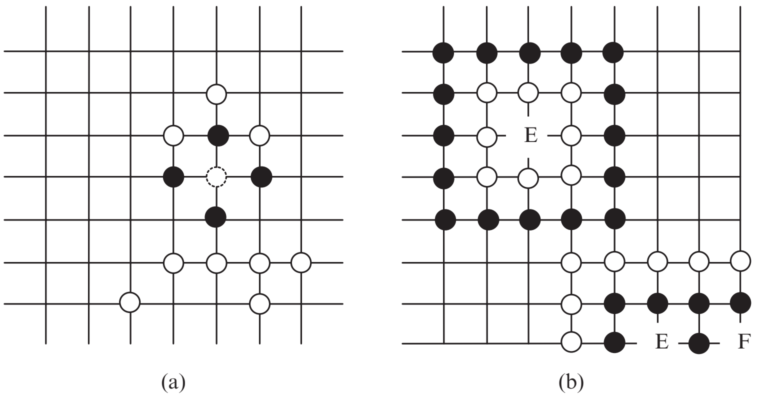

Fig. 1. Go board, terms, and rules. Only a portion of the 19x19 Go board is shown. In (a), the lower 5 T-like white stones form a string, and hence a group with the white stone to their left. In the middle, black captures a white stone, the dashed one, and removes it from the board by placing the top black stone on the board. However, white cannot immediately put a stone onto the dashed-stone intersection to capture the top black stone because such a move would repeat the previous board position, thus violating the rule of "Ko." In the middle of (b), white has an eye at "E." White cannot put his stone at "E," which results in no liberty of the whole string. That is a suicide move and forbidden by the rules. Black can put his stone at "E" even though there is no liberty of that stone either, but it is allowed because it captures the white string. So the white string with only one eye is dead and will be removed from the board at the end of the game. The black string at the corner has two eyes, i.e., "E" and "F," and hence is alive because white cannot seize both of the eyes simultaneously

knowledge-based pattern recognition to dynamic neural networks trained by Heuristic Dynamic Programming (HDP) or Evolutionary Algorithm (EA).

2.3 Computer Go and Computer Chess

Given the same programming techniques that were so successful in Chess, computer Go has so far significantly bypassed what Chess programs have achieved in a high level of game playing. We separate these two games into parts, based on Shannon's plot, to illustrate the gap we need to overcome.

In the game tree aspect, good Chess programs look seven plys, a move by each player, ahead or deeper, and there are about 35–40 moves available, on average, to a player. Exploration on such a search tree results in an evaluation of about 60 billion scenarios with good prune skills. The IBM Deep Blue is capable of evaluating 200 million positions a second, which allows Chess programs to execute a massive search and evaluation with the help of expert knowledge.

Unlike Chess, Go starts with an empty board and fills with stones as the game progresses. Theoretically speaking, there are 361! leaves at the bottom of the game tree with no capture occurring during the game. The branching factor of the search tree, or the number of legal moves on the board, is around 200 on average, more at the beginning and less at the end of the game. The game length varies from 150 to 300 moves. All these factors result in a game tree varying from 10^{360} to 10^{690}. In general, a game tree of approximately 10^{575} leaves [1] is accepted by most researchers. A 7-ply look-ahead search needs to handle ten thousand trillion positions, which makes a brute force search approach infeasible, even with the most powerful computer.

The evaluation function is much easier in Chess because the pieces have their own rank values. Assessing the strength of either side can be simply reduced to comparing the piece value, plus some measure of strength position and threat to the King, since normally a piece advantage will lead to a victory. The idea of rank in Go is very vague. A single stone has no rank at all. Moreover, the importance of a stone, a string, and a group changes based on a number of factors, such as connectivity, liberty, position on the board, correlation with neighboring friend and/or foe, and even the player's preferences, as the game progresses. Sometimes, the same stones can be either protected or sacrificed with the same aim to gain the maximum territory. Additionally, tactical skills play a more important role in Chess. A tactical evaluation based on piece quality of the board correctly yields the likely winner. In Go, winning a tactical struggle over a group may not clearly lead to winning the game. In other words, it is more important to know how to apply those tactical skills properly at different board areas, under different game conditions, or even in different sequences than how to play each tactic correctly. Unfortunately, the former is very game/situation dependent, and even human masters do not agree with each other on which tactical method to pick (sometimes totally opposite methods are selected just because of masters' personal tastes). Therefore, installation of an expert knowledge database, which usually contains the procedures of tactical skills, as the evaluation function, as used in Chess, is implausible for Go because defining and formulating such knowledge into Go moves is difficult.

Besides the above two facets, computer Go also lacks the capability to perform deep, narrow look-ahead. "Ladder situation" is a typical example. Another problem in Go is to determine the finish of the game. In Chess, the game ends when one of the players resigns, when a checkmate is achieved, or when stalemate/draw positions by rule (e.g. king + knight vs. king, position repetition, 50-move rule) are reached, which all are immediate, definite, and easily recognizable states. In Go, the game ends when both players choose to pass consecutively. They agree to pass when they feel that it would not improve their territory by placing more stones on the board. Beginners frequently play beyond the optimal point at which an expert would stop. Current Go programs display similar behavior, especially in self-playing.

3 Development of Computer Go

The history of computer Go is rather short. The first paper discussing computer Go was published in 1970 by Albert Zobrist [49], who wrote the first computer program to play a complete Go game. Since then, many programs have been developed, including, but not limited to: *Go Nemesis* [46], *Wally* [29], *Many Faces of Go* [17], *Handtalk* [11] and *Go*4++ [33], some of which will be discussed in full later in this section. Also, quite a few international computer Go tournaments have been held since the 1st international computer Go congress took place in 1986.

3.1 Early Attempts

Computer Go pioneers inherited most of their techniques from computer Chess. A small game tree was searched due to limited computational power, and certain board patterns, based on fixed, pre-built expert knowledge working as feature evaluation, were checked in order to generate candidate moves.

Zobrist introduced the idea of using influence functions to quantify the impact of black and white stones, and hence segment the board into black and white territories. Each stone radiates its influence, measured by a numeric value with positive for black stones and negative for white ones on the board. The influence attenuates as the hamming distance increases. Also, a black stone itself is given a value of $+50$ and a white stone, -50. For every intersection on the board, the influence function accumulates the influence credits spread by all black and white stones. The black and white territories can be recognized as the areas of contiguous positive and negative values. On the other hand, Zobrist quantified the board in various 19x19 arrays that contain information such as the occupation of an intersection (black, white, or empty); the number of white and black neighbors; the number of stones and liberties for each string; size (including empty points) and number of stones in each segment. This internal representation was constructed for future feature evaluation and pattern recognition.

The move generator in Zobrist's program was built by integrated feature evaluations. With the help of internal representation, the program performed pattern recognition over the entire board, hoping to find some patterns matching those stored in the expert knowledge database. In this database, each pattern was associated with a candidate move and a numeric value to reflect the priority of the move. At the end of the process, the pattern with the highest value was picked, and its associated move became the program's next move. In addition, a limited look-ahead, a depth of three moves, or heuristic search was executed on a local area in handling forming eyes, ladders, saving/capturing strings, and connecting/cutting strings.

Zobrist's program performed somewhat weakly. It beat some beginners but was fairly vulnerable when playing against experienced players.

Ryder's program [37] extended Zobrist's work with refined influence functions and larger summed feature evaluations. The contribution Ryder made was that he combined both strategic, long-term objectives and tactical, short-term objectives because he saw that Go requires a balance between fortifying actual and/or potential territories and avoiding loss of key stones to the opponent. In maintaining this balance, Ryder formulated three domains of interest: how well each side controls his/her regions on the board; where the best moves for both sides are located; and what the life-and-death statuses of strings for both sides are.

Move generation in Ryder's program consisted of two phases. At first, the summed feature evaluation, working as a move filter, reduced the candidate moves from all legal ones to about the 15 best moves according to the tactical status of all strings. In the second phase, the best 15 moves were further analyzed with respect to both tactical and naïve strategy theories. The move with the highest combined score after the first and second round was then recommended as the next move.

Besides refining the credits each stone contributed through the influence function, Ryder also used the influence function to describe the strength of connectedness and relation between stones, and hence to form individual stones and neighboring empty points into some local properties, such as walls, strings, groups, and armies, based on some predefined influence thresholds.

Reitman and Wilcox [34, 35, 36] built their Go program, from INTERIM.2 to Nemesis, by replicating human perceptual and cognitive abilities in Go. The three essential ingredients they incorporated into their Go program were perception, knowledge, and coordination. By perception, they meant the representation of different board positions as a skilled Go player would do. Types of knowledge stored in the program were: tactical – including how to save/kill a group and how to make territory; strategy – evaluating the board position as the game proceeded; coordination – controlling the flow of both perceptions and knowledge for the purpose of generating the next move.

The INTERIM.2, and later the Nemesis [46], had tremendous data structures cascading from simple to complex: stringboard, linkboard, gameboard, and gamemap. They were designed to contain the representations mentioned above. The program also employed a tactician, PROBE, to answer specific questions and propose reasonable initial moves. Those moves, instead of all legal candidates, worked as the branches of the game tree for certain depths of plays until a board position could be judged as either a success or a failure for the specific question. Typically, around 60–80 moves were searched for a particular problem based on a hierarchy of experts in PROBE. Therefore, such a look-ahead drive was rather a narrow, goal-driven but faster one than a general, full-board yet time-consuming one with the price of being greedy at local patterns.

Restricted by the computational powers they had at the time, researchers of computer Go in the 70s and 80s implemented quite an amount of Go knowledge, borrowed directly from human experts, in their programs. The

Go knowledge encoded in the form of patterns greatly pruned the game tree, which by no means brute force can handle, by concentrating on the most promising candidate moves. On the other hand, the room for improvement for these knowledge-intensive Go engines was limited due to their hard learning mode. No ability in Go beyond their fixed database had been demonstrated in their performance. Besides rigid playing, these early programs lacked overall game strategy because the search involved was local rather than full-width. The preferred patterns, and hence corresponding moves, on the sub-board may not be the right play for a long-term goal. These were the reasons why early Go programs ranked at low Kyu level.

3.2 Knowledge Representation and Rule-Based Go Engines

Computer Go engines saw some breakthrough in the 1990s. *Many Faces of Go*, *Handtalk*, and *Go*4++ were among the strongest Go programs commercially available during that time. All these programs employed complicated data structures to describe board position, sophisticated tactical strategies to generate candidate moves, and expert patterns to modify move values.

Many Faces of Go is a typical representative of this category. It contains dynamic data, such as intersections, eye, string, connection, and group, to illustrate board positions. The dynamic data structure is modified incrementally as stones are added to or removed from the board. The data are also recalculated, ether locally or globally, when regions of the board have been affected by a move or move sequence.

The evaluation function, which maintains the data structure, is a tactical analyzer. It reads strings to evaluate liberties, eye, connection, group strength, and territories. Therefore, it assigns a score to board positions depending on how strongly they are controlled by white or black.

The game engine of *Many Faces of Go* is driven by a strategy function, which scans the dynamic data to pick up important areas on the board to act. It also switches the engine among open game, middle game, and end game, hence utilizing different knowledge/rule databases for move generation. Other functionalities include game judgment ("ahead," "even," "behind," etc.), sente evaluation, and urgent defense/capture.

Knowledge and rules are implemented intensively in *Many Faces of Go* for candidate move generation. A rule-based expert system with more than 200 rules is responsible for suggesting plausible moves for full board level. A Joseki database consisting of over 36,000 moves is designed for corner fights. In the pattern database, each of the 1200 patterns of size 8x8 is associated with a move tree. Patterns match leads to move suggestion, eye/connection clarification, in different game phases (middle or end game), and at different areas of the board (middle, edge, or corner).

Combining a strategy function, a move suggestion expert system, and a pattern database, the *Many Faces of Go* evaluates only a small number of moves and plays the one with the highest score.

Handtalk and *Go*4++ process similar procedures; *Handtalk* implements patterns in assemble code to enhance the matching speed, and *Go*4++ emphasizes the connectivity in its data structure and evaluation function.

The common and distinguishing feature of computer Go programs in this category is that candidate moves are heuristically generated by an expert knowledge database and/or pattern matching. The success of the program depends heavily on the sound design of the expert knowledge database and sufficient patterns to cover every case, which is usually very difficult, especially in middle game.

3.3 Combinatorial Game Theory Approach in Computer Go

Combinatorial game theory [12] treats a game as a sum of local subgames and provides a mathematical basis to analyze the game in a divide-conquer manner.

Combinatorial game theory has been applied to many aspects of computer Go. This approach achieved a major breakthrough in computer Go by beating a human professional master at the end game [3]. Another attempt of combinatory game theory is to employ it in position evaluation by focusing on different zones of the board and then computing a full board evaluation from these local evaluations [27]. It is also utilized to decompose Go game tree search [26].

However, the theory has one drawback for its emphasis on being exact. At the end game, the territories held by each side are very clear; thus, subgames can be precisely divided. In open and middle games, however, such assumption is usually not true. Current research involves using heuristic rules/conditions to replace [9] or relax [28] the theory.

4 Learning in Computer Go

Unlike the knowledge-based approaches, which retrieve the built-in expert solutions via pattern recognition, the learning approaches really focus on teaching the computer to analyze the environment and then play the game from its own experience. Neural networks, reinforcement learning, and evolutionary computation are heavily involved in these approaches. Neural networks usually act as a game engine, such as a board evaluator and/or move filter for game tree search, utilizing reinforcement learning techniques to describe the game environment/goals in sequence and employing evolutionary computation methods to tune the weights in order to minimize the cost function.

Temporal difference [44] methods are incremental learning procedures specialized for prediction problems where the sensory inputs are applied in sequence. They have been successfully employed for the prediction evaluation function at different board positions in backgammon [45]. Temporal difference algorithms minimize the following criterion function:

$$J(w) = \sum_{p=1}^{P} \sum_{k=1}^{N_p} \lambda^{N_p - k} (zN_p - G(x_p(k))^2 \tag{4.1}$$

by adjusting the weights in a neural network as:

$$\Delta w_t = \alpha(P_{t+1} - P_t) \sum_{k=1}^{t} \lambda^{t-k} \Delta_w P_k \tag{4.2}$$

In both equations, P is the number of examples, i.e., the number of games; $\mathrm{N_p}$ is the number of steps in the $\mathrm{p^{th}}$ example, which is not known until the outcome is determined; $\mathrm{zN_p}$ is the actual outcome of the $\mathrm{p^{th}}$ example; $\mathrm{G(x_p(k))}$ is the output of the network when presented with $\mathrm{x_p(k)}$; and λ[0,1] is a parameter used to place more emphasis on predictions temporally close to the outcome.

Using the TD approach, a computer Go program can pick up moves that result in better board positions once appropriate weights are learned. In addition, as the weight evolves, the program strategy will also evolve, leading to different performance and triggering, and, in return, another round of weight adjustment. Such a process guides the computer Go engine to play a game without supervision, i.e., explicitly labeled good/bad moves. In fact, any legal move can be used for training, and the game engine gradually learns the strategies itself by playing both sides according to its current evaluation function, with little expert knowledge involved during the whole process.

Schraudolph et al. [39, 40] implemented a Go position evaluator on a neural network system, trained by the TD(0) algorithm. The architecture of the neural system was designed to reflect the spatial board information and eliminate symmetric effects on the board (color and position reflection/rotation) with hidden units and weight sharing. The network predicts the fate of every point on the board rather than just the overall score and then evaluates whole positions accordingly. After extensive self-play training (moves acquired stochastically by Gibbs sampling to avoid duplication and local minima), the system managed to edge past *Wally*, a weak computer Go program, and even beat *Many Faces of Go* at some low level playing.

Zaman et al. [47, 48] used an Heuristic Dynamic Programming (HDP) type adaptive critic design [32] for evaluating a Go board. The main difference between HDP and the above-mentioned TD approaches is that HDP uses an additional utility function (per step cost/reward) in the training signal.

An HDP-type critic estimates the function J (cost-to-go) in the Bellman equation of dynamic programming expressed as:

$$J(t) = \sum_{k=0}^{\infty} \gamma^k U(t+k) \tag{4.3}$$

where γ is a discount factor for finite horizon problems ($0 < \gamma < 1$), and U(.) is a non-negative utility function or local cost/reward. The critic is trained forward in time and tries to minimize the following error measure over time:

$$\|E\| = \sum_{t} E^2(t) \tag{4.4}$$

where,

$$E(t) = J(t) - [\gamma J(t+1) + U(t)] \tag{4.5}$$

the terms inside the square bracket make the desired signal at time t, if t is not the terminal state. At the end of the game, the desired signal is simply U(t). J(t) is a function of R(t), i.e., the observable states. In terms of Go, R(t) can be the board representations at step t. The function U(t) denotes an incremental area measure from board R(t − 1) to R(t). When the area associated with R(t − 1) is larger than that of R(t) (loss of area between two steps, t − 1 and t), U(t) is set to zero because U(.) is strictly non-negative by the principle of dynamic programming. In [47], the U(t) function is given by:

$$U(t) = \begin{cases} util(t) - util(t-1); & if\ util(t) > util(t-1) \\ 0; & otherwise \end{cases} \tag{4.6}$$

where,

$$util(t) = \eta \frac{N_W}{N_W + N_B} + \nu \frac{A_W}{A_W + A_B} + \rho \frac{P_B}{P_W + P_B} \tag{4.7}$$

N_W and N_B are the number of WHITE and BLACK stones on the board, respectively; A_W and A_B are the areas occupied by WHITE and BLACK stones, respectively; P_B and P_W are BLACK and WHITE prisoners held by the opponent, respectively.

The architecture of Zaman's computer Go engine includes five distinct modules: Critic, Action, Wally, Go, and Utility. The Critic module is a multi-layer perceptron trained by the TD(0) method. It estimates the sum of discounted reward/cost for the network's future actions starting from the current state. The Action network plays as a move filter. Firstly, it lists all legal moves for the current state; secondly, it generates a new board position for each move; after that, it evaluates the critic's evaluation of the resulting board; and finally, it selects the move based on the critic's board evaluation. The utility module is used to measure network cost/reward at the current state of the board. The Go module incorporates rules of the game, and Wally serves as the network's opponent. The game engine surpassed the strength of Wally.

Besides the reinforcement technique discussed above, evolutionary computation, especially genetic algorithm, is also very popular in computer Go engine implementation. Like natural evolution, the genetic algorithm evolves the neural networks to tackle evaluation function problems in Go.

Genetic operators, such as selection, crossover, and mutation, are applied to neural networks for effective architecture and/or weights to solve the credit assignment problem. Instead of punishing or rewarding individual moves, the evolutionary approach evaluates and selects networks, i.e., game strategies, based on their overall performance in the game. SANE [38] evolved both architecture and weights of a three-layer feedforward network simultaneously to evaluate move value for each board position. After a few hundred generations of evolution, SANE defeated Wally 75% of the time and exhibited several aspects of general Go playing, indicating a good scale-up.

Genetic algorithms are also used to direct minimax searches away from poor information [25] and optimize search heuristics for life-and-death problems [31].

5 An Example of Training a Non-Knowledge Game Engine for Capture Go

Evolutionary algorithms have shown to be a promising approach to solving complex constrained optimization problems. Chellapilla and Fogel succeeded in evolving an expert-level neural board position evaluator for Checkers without any domain expertise [6, 7, 15]. Their work concludes that computer game engines can learn, without any expert knowledge, to play a game at an expert level, using a co-evolutionary approach.

The trend of [6, 7] is followed, and PSO is applied in combination with an EA to develop a neural evaluator for the game of Capture Go. As a simplified version of Go, Capture Go has the same rules but a different goal – whoever captures first, wins. The system for this game, with minor modifications, should be a useful subsystem for an overall Go player. Previous work [24] on Capture Go showed that the simplified game is a suitable test bench to analyze typical problems involved in evolution-based algorithms, such as lifetime learning, incremental evolution [20], open ended evolution, and scalability. Growing from zero knowledge, this game engine extends our work [4]. The large-scale game engine, a neural network with more than 6000 parameters, is trained by a particle swarm optimization (PSO)-enhanced evolutionary algorithm through self-playing. The innovative hybrid training algorithm inheriting the advantages, i.e., fast convergence and good diversity, of both PSO and EA is more powerful than its individual parts, and this has been shown in other applications [5]. It is compared with a Hill-Climbing (HC) algorithm as a first-order benchmark. The hybrid, method-based game engine is also played against a hand-coded defensive and a web player to show its competence. The Capture Go games are played on a 9x9 board.

5.1 Particle Swarm Optimization

Particle swarm optimization is a form of evolutionary computation developed by Kennedy and Eberhart [22, 23]. Similar to EAs, PSO is a population-based optimization tool, where the population is initialized with random potential solutions and the algorithm searches for optima, satisfying some performance index over iterations. It is unlike an EA, however, in that each potential solution (called a particle) is also assigned a randomized “velocity” and is then “flown” through an m-dimensional problem space.

Each particle i has a position represented by a position vector $\mathbf{x}_i$ (the possible solution for the given problem). A swarm of particles moves through the problem space, with the velocity of each particle represented by a vector $\mathbf{v}_i$. At each time step, a function f representing a quality measure is calculated by using $\mathbf{x}_i$ as input. Each particle keeps track of its own best position, which is recorded in a vector $\mathbf{p}_i$, and $f(\mathbf{p}_i)$, the best fitness it has achieved so far. Furthermore, the best position among all the particles obtained so far in the population is recorded as $\mathbf{p}_g$, and its corresponding fitness as $f(\mathbf{p}_g)$.

At each time step t, by using the individual's best position, $\mathbf{p}_i(t)$, and the global best position, $\mathbf{p}_g(t)$, a new velocity for particle i is calculated using (5.1) below

$$\begin{aligned} v_i(t+1) = w \times v_i(t) + c_1 r_1(p_i(t) - x_i(t)) \\ + c_2 r_2(p_g(t) - x_i(t)) \end{aligned} \tag{5.1}$$

where c_1 and c_2 are positive acceleration constants, r_1 and r_2 are uniformly distributed random numbers in the range [0, 1], and w is the inertia weight, with a typical value between 0.4 and 0.9. The term $\mathbf{v}_i$ is limited to the range $\pm v_{\max}$. If the velocity violates this limit, it is set at its proper limit. Changing velocity this way enables the particle i to search around the individual's best position $\mathbf{p}_i$ and global best position $\mathbf{p}_g$. Based on the updated velocities, each particle updates its position according to the following:

$$x_i(t+1) = x_i(t) + v_i(t+1) \tag{5.2}$$

Based on the above equations, the population of particles tends to cluster together with each particle initially moving in a random direction. Fig. 2 illustrates the procedure of the PSO algorithm. Computing PSO is easy and adds only a slight computational load when incorporated into an EA.

5.2 Evolutionary Algorithm

The evolutionary algorithm (also called evolution strategy in [41]) begins with a uniformly random population of n neural networks, K_i, $i = 1, \ldots, n$. Each neural network has an associated self-adaptive parameter vector σ_i, $i = 1, \ldots, n$, where each component controls the step size of mutation applied to its corresponding weights or bias.

Each parent generates an offspring strategy by varying all associated weights and biases. Specifically, for each parent K_i, $i = 1, \ldots, n$, an offspring $K_i, i = 1, \ldots, n$, was created by

$$\sigma_i'(j) = \sigma_i(j) \exp(\tau N_j(0,1)), \quad j = 1, \ldots, N_w \tag{5.3}$$

$$w_i'(j) = w_i(j) + \sigma_i' N_j(0,1), \quad j = 1, \ldots, N_w \tag{5.4}$$

where N_w is the number of weights and biases in the feedforward neural network, $\tau = 1/\sqrt{2\sqrt{N_w}}$, and $N_j(0,1)$ is a standard Gaussian random variable resampled for every j[6, 7].

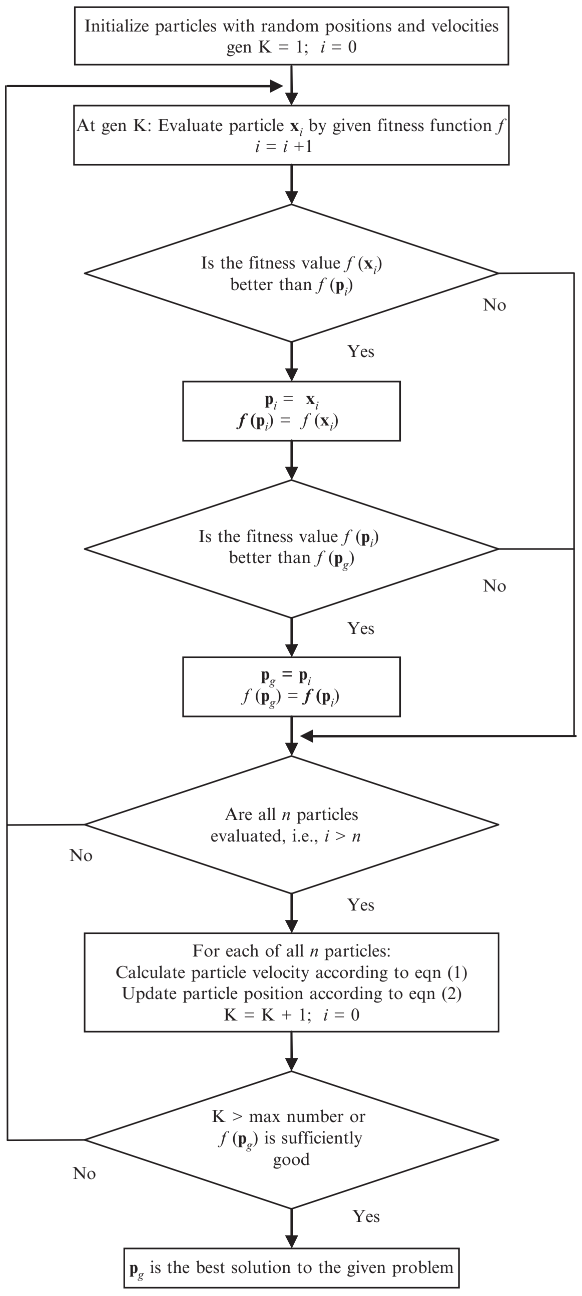
Initialize particles with random positions and velocities
gen K = 1; $i = 0$
At gen K: Evaluate particle $\mathbf{x}_i$ by given fitness function f
$i = i + 1$
Is the fitness value $f(\mathbf{x}_i)$ better than $f(\mathbf{p}_i)$
No
Yes
$\mathbf{p}_i = \mathbf{x}_i$
$f(\mathbf{p}_i) = f(\mathbf{x}_i)$
Is the fitness value $f(\mathbf{p}_i)$ better than $f(\mathbf{p}_g)$
No
Yes
$\mathbf{p}_g = \mathbf{p}_i$
$f(\mathbf{p}_g) = f(\mathbf{p}_i)$
Are all n particles evaluated, i.e., $i > n$
No
Yes
For each of all n particles:
Calculate particle velocity according to eqn (1)
Update particle position according to eqn (2)
K = K + 1; $i = 0$
K > max number or $f(\mathbf{p}_g)$ is sufficiently good
No
Yes
$\mathbf{p}_g$ is the best solution to the given problem

Fig. 2. Flow chart of PSO procedure

5.3 Hybrid of PSO and EA

PSO focuses more on the cooperation among the particles. With memory, each particle tracks the best performance in its own history and its neighborhood throughout the entire evolution when sharing the memory. Such a mechanism guides particles to pursue high fitness values more quickly than mere selection operation in EA. However, particles of PSO are not eliminated even if they are ranked to have the worst fitness in the population, which may waste the limited computational resources. On the other hand, individuals in EA compete for survival. Also, their diversity, maintained by mutation, prevents the population from the premature convergence often found in PSO. Clearly, the advantage of one algorithm can complement the other's shortcoming. Thus, the motivation is to develop a hybrid-based learning algorithm.

Based on the complementary properties of PSO and EA, a hybrid algorithm is used to combine the cooperative and competitive characteristics of both. In other words, PSO is applied to improve the surviving individuals and maintain the properties of competition and diversity in EA. In each generation, the hybrid algorithm selects half of the population as the winners according to their fitness and discards the rest as losers. These elites are enhanced, sharing the information in the community and benefiting from their learning history, by the standard PSO procedure. The enhanced elites then serve as parents for an EA mutation procedure. The offspring also inherit the social and cognitive information from the corresponding parents, in case they become winners in the next generation. Fig. 3 illustrates this hybrid PSO + EA algorithm.

5.4 Simulation Results

A feedforward neural network (multi-layer perceptron MLP) is designed to carry out the board evaluation function, assigning credits for leaves in the game search tree of Capture Go. The best candidate move is then chosen according to the alpha-beta minimax search from the game tree. The board information is represented by a vector of length 81, with each element corresponding to an intersection on the board. Elements in the vector are from $\{-1, 0, +1\}$, where "-1" denotes that the position on the board is occupied by a black stone, "0" denotes an empty position, and "1" denotes a white stone. The feedforward neural evaluator consists of three hidden layers and one output node. The second, third, and output layers are fully connected. Each neuron has a bipolar sigmoid activation function:

$$\tanh(\lambda x) = \frac{e^{\lambda x} - e^{-\lambda x}}{e^{\lambda x} + e^{-\lambda x}} \tag{5.5}$$

with a variable bias term.

The first hidden layer is designed specially, following [6, 7], to process spatial information from the board. In order to grasp the spatial characteristics such as neighborhood and distance of the board, each neuron in the first

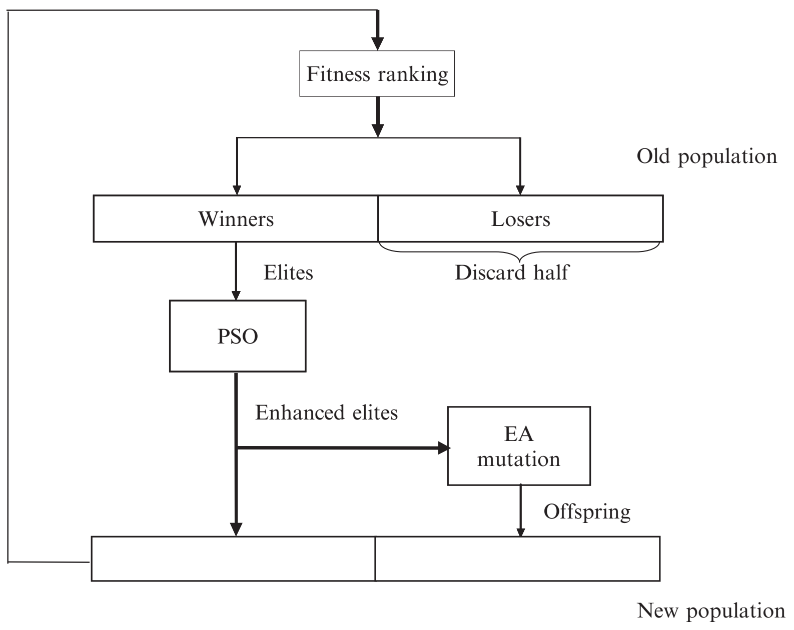

Fig. 3. Flow chart of the hybrid PSO-EA method. The winners, which contain half of the population, are enhanced by PSO and kept in the population for the next generation. Those enhanced winners also work as the parents in EA to produce offspring. The offspring replace the discarded losers to keep a constant number of individuals in the population for the next generation. If the PSO block is removed, the hybrid algorithm is reduced to the conventional EA

hidden layer covers an $n \times n$, $n = 3, \ldots, 9$, square overlapping a subsection of the board. In addition, the connecting weights between the input layer and the first hidden layer are designed specially to reflect the symmetric property of the board. Fig. 4 shows the general structure of the game engine.

In the self-play training, a population of 40 individuals, each representing a game engine, is evolved by playing games of Capture Go. Each individual earns credits based on its game results. Each player, always black, plays one game against each of eight randomly selected opponents, always white, from the population. The game is scored for each player as -2, 0, or $+1$ points depending on the results of loss, draw, or win. In total, there are 320 games per generation, with each engine participating in an average of 16 games. After all games are complete, each individual accumulates the scores it earned as its fitness value and updates according to the algorithms employed, i.e., PSO, EA, or the hybrid.

The weights of each swarm neuro-engine are generated randomly from a uniform distribution over $[-0.2, 0.2]$. The self-adaptive parameters for the EA are initially set to 0.05. The value of v_{max} for PSO is set to 2.0. The whole evolutionary process is iterated for 100 generations. At last, the best

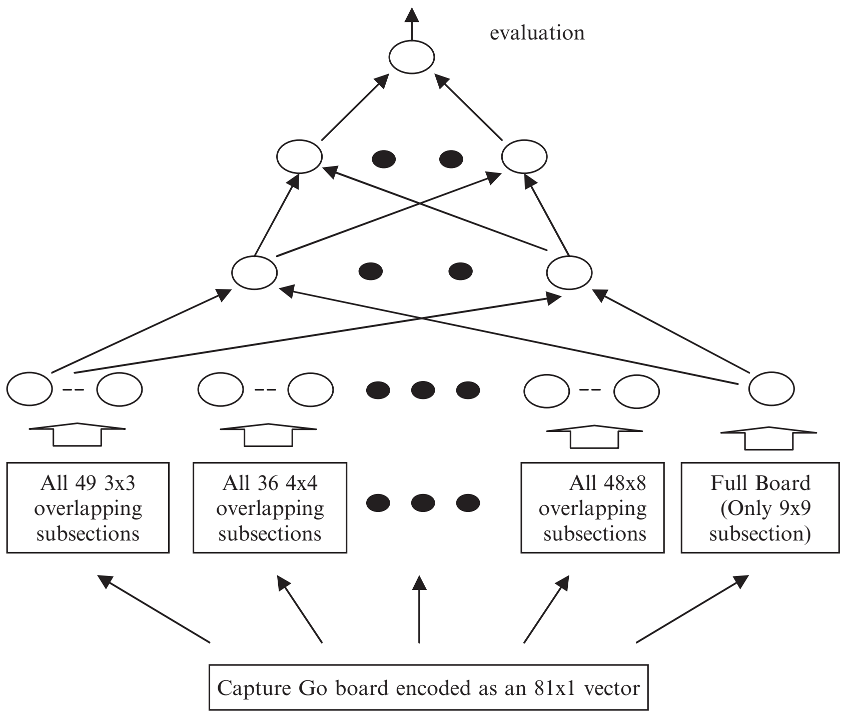
evaluation
All 49 3x3 overlapping subsections
All 36 4x4 overlapping subsections
All 48x8 overlapping subsections
Full Board (Only 9x9 subsection)
Capture Go board encoded as an 81x1 vector

Fig. 4. Architecture of the game engine. This feedforward neural network evaluates the given board. Different sets of parameters of the neuro-game engine lead to different credit assignments for the given board, and hence represent different strategies. The board pattern is interpreted by 140 subsquares. Each of these subsquares is assigned to a node in the first layer of the network for spatial preprocessing purposes. The outputs are then passed through two hidden layers of 40 and 10 nodes, respectively. The output node of the entire network is scaled between [−1, 1] with "−1" in favor for the white and "1" for black

Table 1. Tournament results among hybrid PSO-EA, PSO, EA, HC and random players in 100 games. Black players are listed in rows and white players in columns. For example, the result in row 2, column 4 means that the hybrid PSO-EA player in black wins 90 to 10 against the EA player in white

	Hybrid PSO-EA	PSO	EA	HC	Random
Hybrid PSO-EA	/	79/21	90/10	76/24	100/0
PSO	62/38	/	76/24	70/30	100/0
EA	53/47	68/32	/	70/30	100/0

neuro-engine (at generation 100) of each category, i.e., PSO, EA, and the hybrid, is then used to play against each other and a random player (in each game, roughly 20% of the moves are randomly generated for both sides to avoid duplication). Table 1 summarizes their performance in 100 games in the tournament. All players illustrate success in learning strategies in Capture Go because they overwhelm the random player with only 28 moves per game, on average. The hybrid PSO-EA player in black dominates both PSO and EA players. Considering the advantage that black plays first, the hybrid in white is roughly equivalent to the EA and slightly weaker than the PSO. The PSO and EA players are at the same level. Following the same self-play methodology, another game engine is trained by a simple learning method, hill-climbing (HC), for Capture Go, to verify if the dynamics of the game and the co-evolutionary setup of the training are the key issues of the game engine learning [30]. The tournament results show that PSO, EA, and hybrid players outperform the HC player (with training parameter beta = 0.05 [30]), which indicates that the improvement of game engines comes mainly from the learning algorithms. Finally, a web Capture Go player [19] is brought for illustration, and the best hybrid player wins 23 of 25 games.

In addition to self-play, a defensive player of Capture Go is hand-coded. This player takes defensive strategies with the following priorities: 1) connect all its stones into one string; 2) choose a move that maximizes its liberties (the liberty count saturates when it makes two eyes); 3) surround more empty intersections with a wall; and 4) attack the weak stone(s) of its opponent. The player is hard to capture because it is difficult to seize all its liberties before it makes two eyes or captures an opponent's stone(s) instead (see Fig. 5). The game's result indicates that an opponent is more likely to defeat this defensive player by occupying more territories rather than by capturing its stones. Competing with this player teaches our hybrid engine to manage the balance between seizing its own territories and capturing enemy stones (see Figs. 6 & 7).

6 Conclusion

The game Go remains unconquered for traditional artificial intelligence techniques due to the tremendous size of its game tree, vague rank information of its stones/strings/groups, and dynamic changes of the crucial building blocks.

Early computer Go programs built the game board evaluation function by employing pattern recognition, tactical search, and rule-based reasoning, mainly based on matching the expert knowledge. Even though such techniques are predominant in the top computer Go engines existing now, the rigid or hand-coded look-up table in a knowledge database prevents the game engines from correctly handling the complex and subtle environment beyond the provided expert knowledge.

The emergence of neural networks, reinforcement learning, and evolutionary computation techniques guide the computer Go engines to learn, rather than embed, the game strategies through playing Go games. The game engines without pre-defined knowledge gradually adapt to the nonlinear stimulus-response mappings of the game from their own experience. Multiple at-

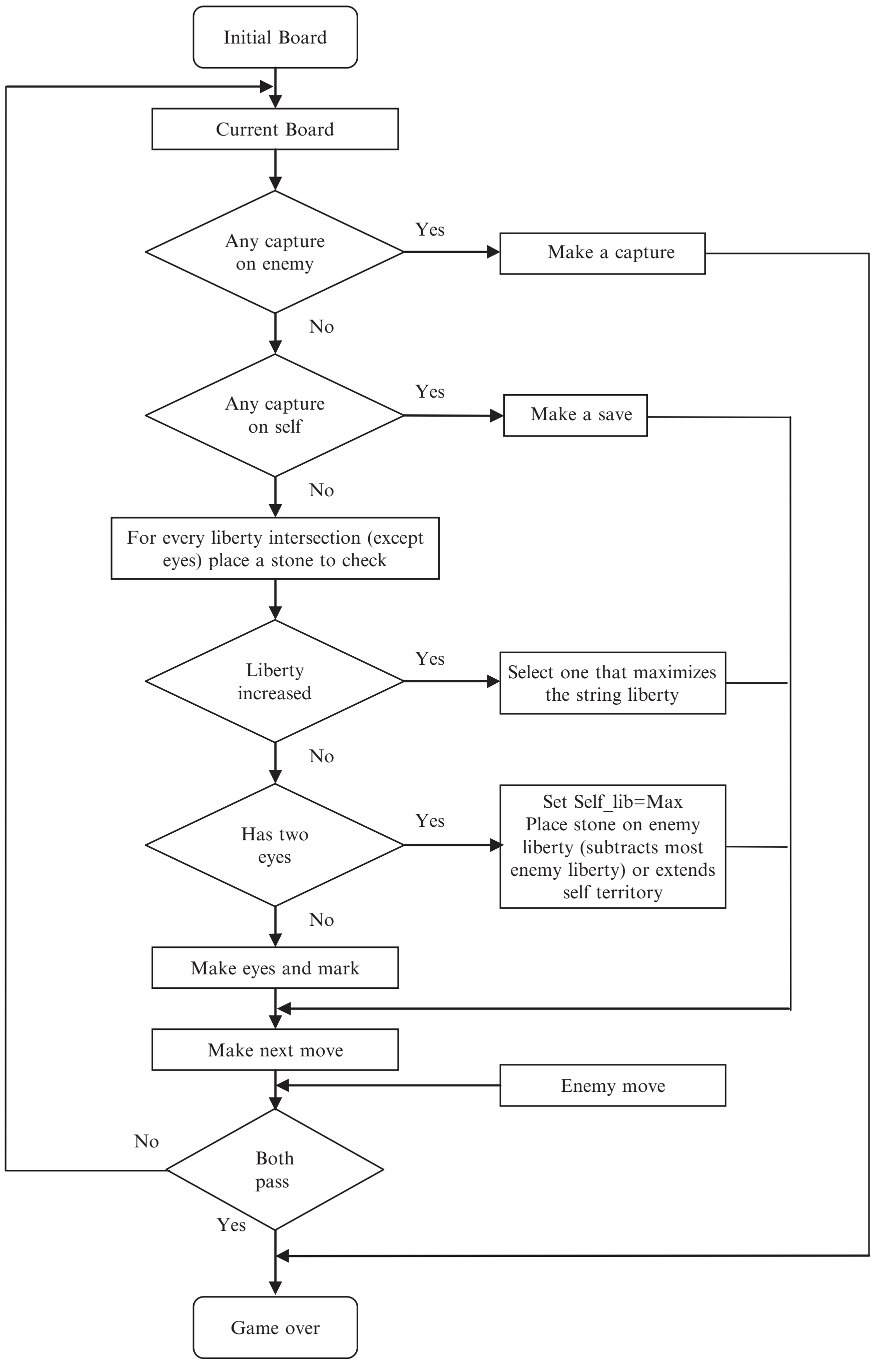
Initial Board
Current Board
Any capture on enemy
Yes
Make a capture
No
Any capture on self
Yes
Make a save
No
For every liberty intersection (except eyes) place a stone to check
Liberty increased
Yes
Select one that maximizes the string liberty
No
Has two eyes
Yes
Set Self_lib=Max Place stone on enemy liberty (subtracts most enemy liberty) or extends self territory
No
Make eyes and mark
Make next move
Enemy move
No
Both pass
Yes
Game over

Fig. 5. Flowchart of the defensive player

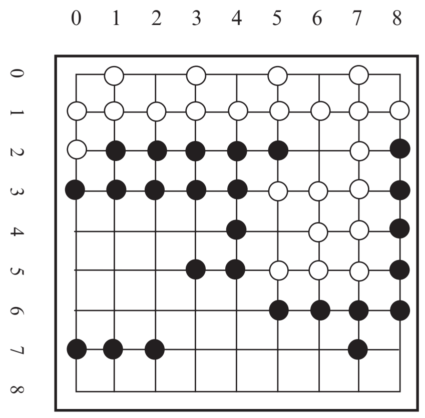

Fig. 6. Final board of game one between the hybrid PSO-EA and the defensive player

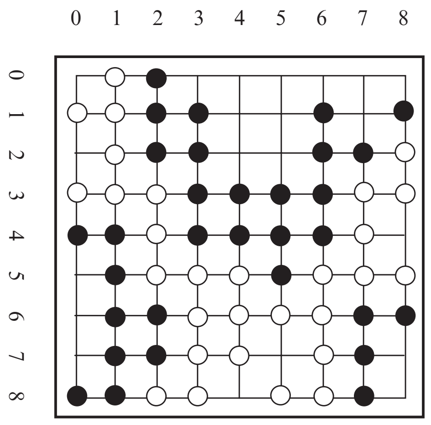

Fig. 7. Final board of game two between the hybrid PSO-EA and the defensive player

tempts in this category have achieved primary success in beating some of the knowledge-based counterparts and have demonstrated the ability to scale up and grasp the general Go strategy.

The acquisition, integration, and use of knowledge are critical to the progress of computer Go programs. Allowing the network to access mature features instead of looking at the raw board may allow it to learn faster and deal with more complex situations. The possibility of merging approaches of network evolution and pattern matching is worthy of further exploration. Evolving a hierarchy of networks where the lower levels, mainly the well-studied features, would provide the inputs for the networks at a high level may be the next breakthrough towards the ultimate goal – defeating human masters in Go.

References

[1] Allis LV, Van den Herik HJ, Herschberg IS (1991) "Which games will survive?" In D. N. L. Levy and D. F. Beal, editors, *Heuristic Programming in Artificial Intelligence 2-The Second Computer Olympiad*, pp. 232–243, Ellis Horwood.

[2] Berlekamp E, Conway J, Guy R (1982) *Winning Ways*, Academic Press, New York.

[3] Berlekamp E, Wolfe D (1994) *Mathematical Go: Chilling Gets the Last Point.* A. K. Peters., MA, USA

[4] Cai X, Wunsch II DC (2004) "Evolutionary computation in playing CaptureGo game," *Proc. of ICCNS'04*, Boston.

[5] Cai X, Zhang N, Venayagamoorthy GK, Wunsch II DC (2005) "Time series prediction with recurrent neural networks using hybrid PSO-EA algorithm," *Neurocomputing, (Accepted).*

[6] Chellapilla K, Fogel D (1999) "Evolution, neural networks, games, and intelligence," *Proceedings of the IEEE*, vol. 87, no. 9, pp. 1471–1496, September.

[7] Chellapilla K, Fogel D (1999) "Evolving neural networks to play Checkers without relying on expert knowledge," *IEEE Trans. on Neural Networks*, vol. 10, no. 6, pp. 1382–1391, November.

[8] Chellapilla K, Fogel D (2001) "Evolving an expert Checkers playing programs without using human expertise," *IEEE Trans. on Evolutionary Computation*, vol. 5, no. 4, pp. 422–428, August.

[9] Chen K, Chen Z (1999) "Static analysis of life and death in the game of Go," *Information Science*, Vol. 121, pp. 113–134.

[10] Chen X, Zhang D, Zhang X, Li Z, Meng X, He S, Hu X (2003) "A functional MRI study of high-level cognition II. The game of GO," *Cognitive Brain Research*, 16(1): 32–37.

[11] Chen Z (1995) "Programming technics in Handtalk," http://www.wulu.com/ht-techn.htm.

[12] Convey J (1976) *On Numbers and Games*, Academic Press, London/New York.

[13] Enderton HD (1991) "The Golem Go program," Tech. Rep. CMU-CS-92-101, Carnegie Mellon University.

[14] Enzenberger M (1996) "The integration of a priori knowledge into a Go playing neural network".
Available: http://www.markus-enzenberger.de/neurogo.ps.gz

[15] Fogel D (2002) *Blondie24: Playing at the Edge of AI.* SF, CA: Morgan Kaufmann.

[16] Fogel D, Hays TJ, Hahn SL and Quon J (2004) "A self-learning evolutionary Chess program," *Proc. of the IEEE*, vol. 92, no. 12, pp. 1947–1954, December.

[17] Fotland D (1999) The 1999 FOST (Fusion of Science and Technology) cup world open computer championship, Tokyo. Available: http://www.britgo.org/results/computer/fost99htm

[18] Fürnkranz J (2001) Machine learning in games: A survey. J. Fürnkranz & M. Kubat (eds.): Machines that Learn to Play Games, Nova Scientific Publishers, Chapter 2, pp. 11–59, Huntington, NY.

[19] Goerlitz S http://www.schachverein-goerlitz.de/Foren/Fun/Go/go.htm

[20] Gomez F, Miikkulainen R (1997) "Incremental evolution of complex general behavior," *Adaptive Behavior*, vol. 5, pp. 317–342.

[21] Hsu F (2002), *Behind Deep Blue.* Princeton, NJ: Princeton Univ. Press.

[22] Kennedy J,. Eberhart R (1995) "Particle Swarm Optimization," *IEEE International Conference on Neural Networks*, vol. 4, pp. 1942–1948, Nov. 27-Dec. 1, Perth, Australia.

[23] Kennedy J,. Eberhart R, Shi Y (2001) *Swarm Intelligence.* San Meteo, CA: Morgan Kaufmann.

[24] Konidaris G, Shell D, Oren N (2002) "Evolving neural networks for the capture game," *Proc. of the SAICSIT postgraduate symposium*, Port Elizabeth, South Africa. Available from: http://www-robotics.usc.edu/∼dshell/res/evneurocapt.pdf

[25] Moriarty DE, Miikkulainen R (1994) "Evolving neural networks to focus minimax search," *Proc. of National Conference on Artificial Intelligence (AAAI-94)*, pp. 1371–1377.

[26] Muller M (1999) "Decomposition search: A combinatorial games approach to game tree search, with applications to solving Go endgames," *Proc. of IJCAI*, vol. 1, pp. 578–583.

[27] Muller M (2002) "Position evaluation in computer Go," *ICGA Journal*, Vol. 25, No. 4, pp. 219–228.

[28] Muller M (2003) "Conditional combinatorial games and their application to analyzing capturing race in Go," *Information Science*, Vol. 154, pp. 189–202.

[29] Newman B (1988) "Wally, a simple minded Go-program," ftp://imageek.york.cuny.edu/nngs/Go/comp/.

[30] Pollack JB, Blair AD (1998) "Co-evolution in the successful learning of Backgammon strategy," *Machine Learning*, Vol. 32, pp. 226–240.

[31] Pratola M, Wolfe T (2003) "Optimizing GoTools' search heuristics using genetic algorithms," *ICGA Journal*, vol. 26, no. 1, pp. 28–48.

[32] Prokhorov D and Wunsch II DC (1997) "Adaptive critic designs," *IEEE Trans. on Neural Networks*, vol. 8, no. 5, pp. 997–1007, September.

[33] Reiss M (1995) e-mail sent in January 1995 to the computer Go mailing list, http://www.cs.uoregon.edu/∼richard/computer-go/.

[34] Reitman W, Kerwin J, Nado R, Reitman J, Wilcox B (1974) "Goals and plans in a program for playing Go," *Proc. of the 29th National Conference of the ACM*, pp. 123–127.

[35] Reitman W, Wilcox B (1975) “Perception and representation of spatial relations in a program for playing Go,” *Proc. of the 30th National Conference of the ACM*, pp. 37–41.

[36] Reitman W, Wilcox B (1978) “Pattern recognition and pattern-directed inference in a program for playing Go,” In D. Waterman and F. Hayes-Roth, editors, *Pattern Directed Inference Systems*, pp. 503–523, Academic Press, New York.

[37] Ryder J (1971) “Heuristic analysis of large tree as generated in the game of Go,” PhD thesis, Department of Computer Science, Stanford University.

[38] Richards N, Moriarty D, McQuesten P, Miikkulainen R (1998) “Evolving neural networks to play Go,” *Applied Intelligence*, vol. 8, pp. 85–96.

[39] Schraudolph N, Dayan P, Sejnowski T (1994) “Temporal difference learning of position evaluation in the game of Go,” *Advances in Neural Information Processing*, vol. 6, pp. 817–824.

[40] Schraudolph N, Dayan P, Sejnowski T (2000) “Learning to evaluate Go position via temporal difference methods,” In L. Jain and N. Baba Eds, *Soft Computing Techniques in Game Playing*, Springer Verlag, Berlin.

[41] Schwefel (1995) *Evolution and Optimum Seeking*. Wiley, NY.

[42] Shannon CE (1950) “Automatic Chess player,” *Scientific American* 182, No. 48.

[43] Smith A (1956) *The Game of Go*, Charles Tuttle Co., Tokyo, Japan.

[44] Sutton R (1988) “Learning to predict by the method of temporal differences,” *Machine Learning*, No. 3, pp. 9–44.

[45] Tesauro G (1992) “Practical issue in temporal difference learning,”*Machine Learning*, No. 8, pp. 257–278.

[46] Wilcox B (1985) “Reflections on building two Go programs,” *ACM SIGART Newsletter*, pp. 29–43.

[47] Zaman R, Prokhorov DV, Wunsch II DC (1997) “Adaptive critic design in learning to play the game of Go,” *Proc. of the International Joint Conference on Neural Networks*, vol. 1, pp. 1–4, Houston.

[48] Zaman R, Wunsch II DC (1999) “TD methods applied to mixture of experts for learning 9x9 Go evaluation function,” *Proc. of the International Joint Conference on Neural Networks*, vol. 6, pp. 3734–3739

[49] Zobrist A (1970) “Feature extractions and representation for pattern recognition and the game of Go,” PhD thesis, Graduate School of the University of Wisconsin.

This research was supported in part by the National Science Foundation and the M. K. Finley Missouri endowment.

Authors’ addresses: Applied Computational Intelligence Laboratory, Dept. of Electrical & Computer Engineering, University of Missouri – Rolla, 1870 Miner Circle, Rolla, MO 65409-0040; email: cai@umr.edu, dwunsch@ece.umr.edu