Neural Networks 23 (2010) 295–305

Contents lists available at ScienceDirect

Neural Networks

journal homepage: www.elsevier.com/locate/neunet

Evolutionary swarm neural network game engine for Capture Go

Xindi Cai [a,*], Ganesh K. Venayagamoorthy [b], Donald C. Wunsch II [c]

[a] APC-MGE by Schneider Electric, O'Fallon, MO 63368, USA

[b] Real-Time Power and Intelligence System Laboratory, Department of Electrical and Computer Engineering, Missouri University of Science and Technology, Rolla, MO 65409, USA

[c] Applied Computational Intelligence Laboratory (ACIL), Department of Electrical and Computer Engineering, Missouri University of Science and Technology, Rolla, MO 65409, USA

ARTICLE INFO

Article history:
Received 18 August 2007
Received in revised form 1 November 2009
Accepted 6 November 2009

Keywords:
Evolutionary algorithm
Particle swarm optimization
Neural networks
Evolutionary computation
Go
Capture Go
Game engine

ABSTRACT

Evaluation of the current board position is critical in computer game engines. In sufficiently complex games, such a task is too difficult for a traditional brute force search to accomplish, even when combined with expert knowledge bases. This motivates the investigation of alternatives. This paper investigates the combination of neural networks, particle swarm optimization (PSO), and evolutionary algorithms (EAs) to train a board evaluator from zero knowledge. By enhancing the survivors of an EA with PSO, the hybrid algorithm successfully trains the high-dimensional neural networks to provide an evaluation of the game board through self-play. Experimental results, on the benchmark game of Capture Go, demonstrate that the hybrid algorithm can be more powerful than its individual parts, with the system playing against EA and PSO trained game engines. Also, the winning results of tournaments against a Hill-Climbing trained game engine confirm that the improvement comes from the hybrid algorithm itself. The hybrid game engine is also demonstrated against a hand-coded defensive player and a web player.

1. Introduction

Intelligence is revealed through appropriate decisions and adaptive behaviors, to satisfy specific goals under certain environments (Chellapilla & Fogel, 1999). Games have served as one of the best benchmarks for studying intelligence. Governed by a set of rules, the players apply strategies to allocate available resources in achieving maximum payoffs, either short-term or long-term. The nature of goal-driven decision making and interaction in a range of environments makes game playing a flexible and general concept that is utilized in many aspects of real life (Axelrod, 1984; Maynard Smith, 1982; von Neumann & Morgenstern, 1944).

The foundation of game theory was first laid down by von Neumann and Morgenstern (1944), and Morgenstern (1949). Claude Shannon (1950) 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 could do in any circumstance (Morris, 1994). Guided by pioneering works, computer game engines have beaten their designers in a variety of games, from Tic-Tac-Toe to Chess. The algorithm proposed by Shannon, combined with an expert database, achieved noteworthy success, given the computational power of current machines, when Chinook won the world Checkers title in 1994 (Schaeffer, 1996) and IBM's deep blue beat the World Chess Champion Garry Kasparov in 1997 (Hsu, 2002).

Unfortunately, traditional game solutions to the game of Go are inefficient, 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. Unlike most other strategy games, Go has remained elusive for computers. It is increasingly recognized as a "grand challenge" of artificial intelligence, and attracts researchers from a diversity of domains, such as game theory (Berlekamp & Wolfe, 1994), pattern recognition, reinforcement learning (Zaman, Prokhorov, & Wunsch, 1997), and even cognitive psychology (Chen et al., 2003).

Even though the best endgame analysis system can beat professional masters (Berlekamp & Wolfe, 1994), computer Go still plays an amateur level for the rest of the game (especially the middle game), despite continued, albeit slow, progress in performance (Fotland, 1999; Muller, 1999). In particular, neural networks and evolutionary computation provide useful means for evaluating a board position (Enzenberger, 1996; Richards, Moriarty, Mc Questen, & Miikkulainen, 1998). Unlike a rigid look-up table or hand-coded feature extraction in a knowledge database, the neural evaluator adapts to the nonlinear stimulus-response mappings of the game. Evolutionary algorithms have shown to be a promising approach to solve complex constrained optimization problems. For example, Chellapilla and Fogel succeeded in evolving an expert-level neural board position evaluator for Checkers

* Corresponding author.
E-mail addresses: xindi.cai@apc.com (X. Cai), gkumar@ieee.org (G.K. Venayagamoorthy), dwunsch@mst.edu (D.C. Wunsch II).

0893-6080/$ – see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.neunet.2009.11.001

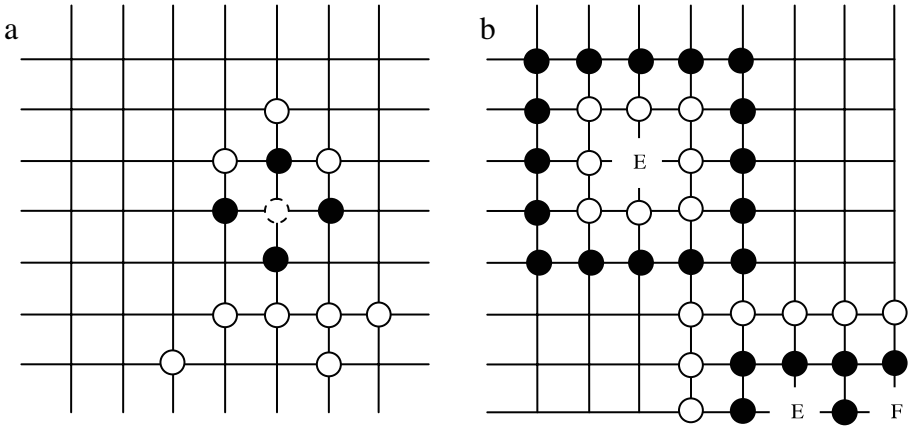

Fig. 1. Go board, terms, and rules. Only a portion of the 19 × 19 Go board is shown. In (a), the lower 5 T-like white stones form a string, and hence a group with the white stone to their left. In the middle, black captures a white stone, the dashed one, and removes it from the board, by placing the 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 board position. That is the rule of "Ko". (It applies to the game of Go, but not to capture go.) 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 since white cannot seize both of the eyes simultaneously.

(Chellapilla & Fogel, 1999, 2001; Fogel, 2002) and further for Chess (Fogel, Hays, Hahn, & Quon, 2004, 2005) without any domain expertise. 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 Chellapilla and Fogel (1999) 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 as Go, but has a different goal — whoever captures first, wins. The game engine for Capture Go, with minor modifications, should be a useful subsystem for an overall Go player (specifically, a life-and-death module). Previous work (Konidaris, Shell, & Oren, 2002) 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 (Gomez & Miikkulainen, 1997), open ended evolution and scalability. Growing from zero knowledge, this game engine extends our work (Cai & Wunsch, 2004). A neural network with more than 6000 parameters, part of the large-scale game engine, is trained by a PSO-enhanced evolutionary algorithm through self-playing. The innovative hybrid training algorithm inheriting the advantages of PSO and EA, i.e., fast convergence of PSO and good diversity of EA, is more powerful than its individual parts, and this has been shown in other applications (Cai, Zhang, Venayagamoorthy, & Wunsch, 2006). The hybrid algorithm 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.

Section 2 presents background information on both Go and Capture Go. The architecture of the neural network board position evaluator is presented in Section 3. Section 4 describes PSO, EA, and the hybrid algorithm. Tournament design and parameter setup are provided in Section 5. Simulation results and their discussion are given in Section 6. Finally, conclusions are given in Section 7.

2. Go and Capture Go

2.1. The game of Go

Go is a deterministic, perfect information, zero-sum game of strategy between two players. More detail on the game of Go can be found in Smith (1956). Players take turns to place black and white pieces (called *stones*) on the intersections of the lines in a 19 × 19 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) by one's own stones than is surrounded by the opponent's stones.

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 that surround the intersection to be captured. A string with two eyes cannot be captured because filling one of the eyes would be a suicidal move and is therefore illegal. Having two eyes is the line between "alive" strings and "dead" ones. The potential of making two eyes for a string, e.g., whether surrounding enough large eye space, capturing enemy stones in neighborhood or connecting to "alive" friendly strings, will greatly affect the final fate of the string. With some minor exception for ties (or Seki), the strings that are incapable of forming two eyes will be considered as captured and hence removed when calculating territories at the end of the game. Evaluating whether stones can be formed into strings, and furthermore into groups, and whether strings are capable of forming two eyes is the fundamental skill in Go because it represents the game strategies of "attack", making oneself strong and being aggressive, and "defense", avoiding vulnerability and surviving.

To prevent loops, it is illegal to make moves that recreate prior board positions (rule of *Ko*). The rule for Go is simple: one can place a stone on any empty intersection unless it is a suicidal or Ko move. A player can pass any time. The game ends when both players pass in consecutive turns (please see Fig. 1 for graphic explanations about the terms and rules mentioned above).

Go starts with an empty board and fills with stones as the game progresses. The branching factor, the legal moves on the board at any time, 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 in size

from 10^{360} to 10^{690} nodes. In general, a game tree of approximately 10^{575} nodes (Allis, van den Herik, & Herschberg, 1991) is accepted by most researchers. A 14-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 vague and frequently changing rank information of strings/groups make board evaluation extremely difficult. A single stone has no rank at all. The importance of a stone, a string, and a group changes, affected by numbers of factors such as connectivity, liberty, position on the board, correlation with neighboring friend and/or foe, and even the player aesthetics, as the game carries on. Additionally, tactical skills are not as important as in Chess. Winning a tactical struggle over a group may not necessarily lead to winning the game. Therefore, creating an evaluation function by employing pattern recognition, tactical search and rule-based reasoning, mainly based on matching the expert knowledge, is not promising. It is true that such techniques are predominant in the top computer Go engines, and it is important not to forget that these are still the strongest engines existing now. But, all of these engines have remained weak. When dealing with complex, subtle, and unsettled board situations, such as the middle-game, learning is crucial (Fürnkranz, 2001; Mechner, 1998).

In order to reduce the tremendous size of the game tree and approximate the dynamic nonlinear evaluation function of Go, researchers have explored the use of neural networks with machine learning methods such as supervised learning (training on moves from professional games) (Enderton, 1991) and reinforcement learning (Richards et al., 1998; Schraudolph, Dayan, & Sejnowski, 1994). In this paper, a hybrid of particle swarm optimization (PSO) and evolutionary computation (EC) is applied for those tasks on the simplified version of Go described below.

2.2. Capture Go

Capture Go (also known as Atari Go) inherits all rules from Go, with a modified winning criterion, i.e., whoever captures first wins the game. In case of no capture, the player who has occupied more territory wins. Moreover, there is no Ko move in Capture Go because the first capture ends the game and repeated board positions never occur. In this paper, Capture Go games are played on a 9×9 board.

Capture Go, though simple, maintains most characteristics of Go. This simplified version emphasizes the most important strategies, i.e., how to attack and defend. It also addresses the strategic balance between capture and territory expansion, in the sense that both players must still attend to fortifying their own territories. In addition, compared to the high branching factor and long-range spatiotemporal interaction in Go, Capture Go has a typically shorter game length and clear game ending criteria, i.e., first capture wins. Also, the candidate moves often fall into a small neighborhood of the last move, which greatly reduces the game tree branches. A well-developed Capture Go evaluator can be plugged directly into an overall computer Go engine for string's life-death analysis. All these make Capture Go a perfect starting point in searching for effective multi-objective mechanisms for Go.

3. Architecture of swarm-neuro-engine

The idea from Chellapilla and Fogel (1999) is used to evolve a feedforward neural network to approximate the board evaluation function, assigning credits for leaves in the game search tree of Capture Go. The best candidate move is then chosen from the game tree according to alpha–beta minimax search. 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 multi-layer-perceptron (MLP) neural evaluator consists of three hidden layers and one output node. The second, third, and output layer are fully connected. The first hidden layer is designed specially, following Chellapilla and Fogel (1999), to process spatial information from the board. Each neuron has a bipolar sigmoid activation function, i.e., $\tanh(\lambda x) = \frac{e^{\lambda x} - e^{-\lambda x}}{e^{\lambda x} + e^{-\lambda x}}$ with a variable bias term. Fig. 2 shows the general structure of the swarm-neuro-engine.

The one-dimensional input vector destroys the spatial characteristics such as neighborhood and distance of the board. The first hidden layer is implemented to remove the handicap. Each neuron in the layer covers an $n \times n$, $n = 3, \ldots, 9$, square overlapping a subsection of the board. Fig. 3 illustrates examples of the overlapping subsections. All 49 possible 3×3 square subsections are assigned to the first 49 neurons in the first hidden layer. The following thirty-six 4×4 squares are allocated to the next 36 neurons until the entire board is allocated to the last neuron in the layer (Chellapilla & Fogel, 1999). Therefore, 140 neurons are set in the first hidden layer to enable the capture of possible features that are processed in subsequent hidden layers (of 40 and 10 hidden nodes (Chellapilla & Fogel, 1999)). The connecting weights between input layer and first hidden layer are designed specially to reflect the symmetric property of the board (see Fig. 4). Each intersection on the 9×9 board is assigned a weight. Only 15 of them, in the dashed triangle area in Fig. 4, are allowed to be different. The rest of the weights are duplicated by those 15, flipping along horizontal, vertical, and diagonal axes, the dashed lines in Fig. 4. The board information of each square subsection, i.e., black, white, and empty, is weighted, summed up, and passed to the first hidden layer for further processing. This weight assignment guarantees that patterns shown on symmetric square subsections will contribute equally to the network. Also, those weights, when trained, will teach the neuro-engine to play at favorable subareas, e.g., corners, of the board.

4. PSO, EA and the hybrid PSO–EA algorithm

4.1. Particle swarm optimization

Particle swarm optimization is a form of stochastic optimization technique developed by Kennedy and Eberhart (1995), and Kennedy, Eberhart, and Shi (2001). 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 criteria 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 $\mathbf{x}_i$. 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 previous best position, which is recorded in a vector $\mathbf{p}_i$, where $f(\mathbf{p}_i)$ is the best fitness it has achieved so far. Furthermore, the global 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 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

$$\mathbf{v}_i(t+1) = w \times \mathbf{v}_i(t) + c_1\mathbf{r}_1(t)(\mathbf{p}_i(t) - \mathbf{x}_i(t)) + c_2\mathbf{r}_2(t)(\mathbf{p}_g(t) - \mathbf{x}_i(t)) \quad (1)$$

where c_1 and c_2 are positive acceleration constants, $\mathbf{r}_1$ and $\mathbf{r}_2$ are uniformly distributed random vectors (the same dimension as $\mathbf{x}_i$ and each element in the range [0, 1]) and w is the inertia weight,

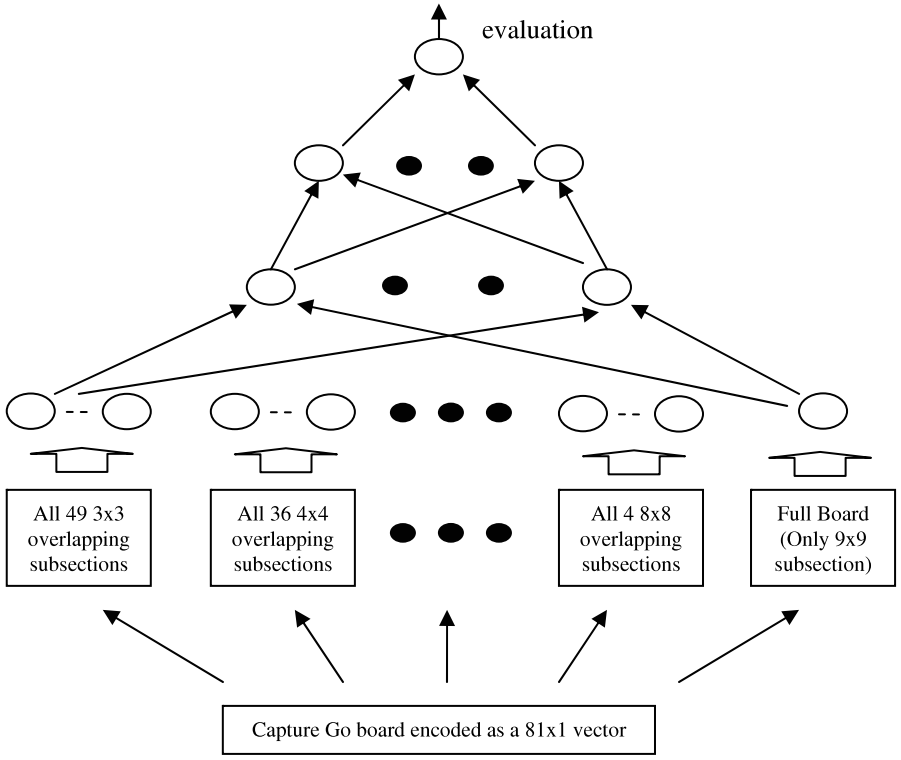

Fig. 2. Architecture of the swarm-neuro-engine. This feedforward neural network evaluates the given board. Different sets of parameters of the swarm-neruo-engine lead to different credit assignments for the given board, and hence represent different strategies. The board pattern is interpreted by 140 subsquares. Each of these subsquares is assigned to a node in the first layer of the network for spatial preprocessing purpose. 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 the black.

with a typical value between 0.4 and 0.9. The term $\mathbf{v}_i$ is limited to the range $\pm\mathbf{v}_{\max}$ (the value of $\mathbf{v}_{\max}$ and whether it is required are problem dependent (Clerc & Kennedy, 2002; van den Bergh, 2002)). If the velocity violates this limit, it is set at $\mathbf{v}_{\max}/-\mathbf{v}_{\max}$. Changing the velocity this way prevents it from becoming unbounded. Based on the updated velocities, each particle updates its position as:

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1). \tag{2}$$

Based on (1) and (2), the population of particles tends to cluster together with each particle initially moving in a random direction (Fig. 5). Fig. 6 illustrates the procedure of the PSO algorithm. The computation of PSO is easy, and adds only a slight computational load when it is incorporated into an EA.

4.2. Optimal PSO

The velocity update of a PSO particle, as given in Eq. (1), comprises three parts. The first is the momentum part, which prevents abrupt velocity change. The second is the "cognitive" part, which represents learning achieved from particle's own search experience. The third is the "social" part, which represents the cooperation among particles — learning from the group best's search experience. The inertia weight w controls the balance of global and local search ability. A large w facilitates the global search while a small w enhances local search (Shi, 2004).

There always exist cases, perhaps pathological ones, where heuristics, such as presented here, will fail (Wolpert & Macready, 2001). With that caveat, our observations are that the static parameter settings of PSO, if well selected, can do a good job (Doctor, Venayagamoorthy, & Gudise, 2004; van den Bergh, 2002), but much better performance can be achieved if a dynamically changing scheme for the parameters is well designed. Examples

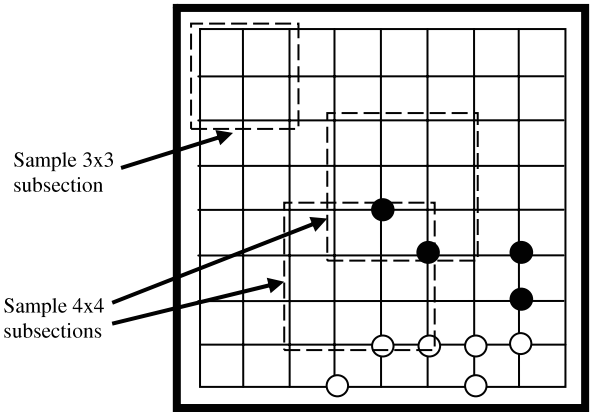

Fig. 3. Game board preprocessing in the first hidden layer of swarm-neuro-engine. The number of $n \times n$ subsections is $(9 - n + 1) * (9 - n + 1)$ for $n = 3, \ldots, 9$. The neuro-player was able to learn the string/group patterns of Capture Go based on these subsections.

include a linearly decreasing w (Shi & Eberhart, 1998), a linearly increasing w (Zheng, Ma, Zhang, & Qian, 2003), a nonlinearly fuzzy changing w (Shi & Eberhart, 2001), or involving a random w (Shi & Eberhart, 2001), and dynamically changing c_1, c_2 and $v_{\max}$ (Clerc & Kennedy, 2002; Doctor & Venayagamoorthy, 2005). All intuitively assume that the PSO should favor global search ability at the beginning and local search around the minima.

Finding a proper set of $\{w, c_1, c_2\}$ for a swarm of particles (Clerc & Kennedy, 2002; van den Bergh & Engelbrecht, 2004) is critical. The cost function for such an optimization is the game performance of the entire swarm, where the gradient-based algorithms, such as Rosenbrock and Fletcher–Powell methods, fail due to the lack of gradient information with respect to the parameters. Therefore, a 2-level hierarchy of the swarm algorithm, similar to the direct

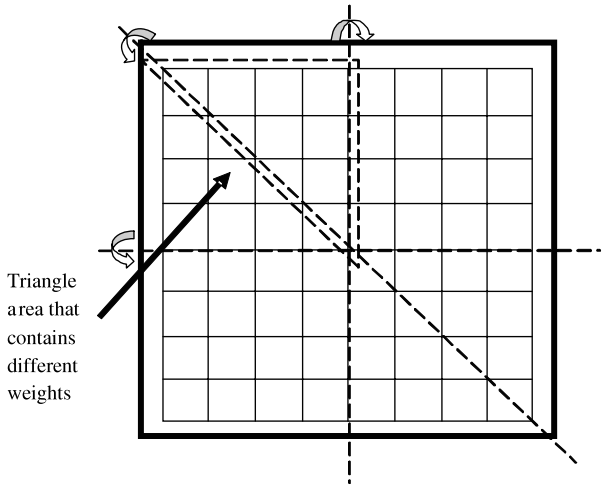

Fig. 4. Shared-weight designed to represent the symmetric nature of the game board. Weights in the dashed triangle area are duplicated along three axes to reflect the board symmetry. This shared-weight scheme guarantees that patterns shown on symmetric square subsections will contribute equally to the network.

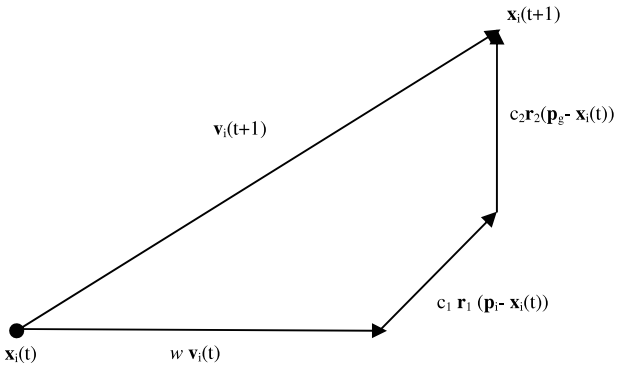

Fig. 5. Concept of a swarm particle's position. $\mathbf{x}_i(t)$ and $\mathbf{v}_i(t)$ denote the particle's position and the associated velocity vector in the search space at generation t, respectively. Vector $c_1\mathbf{r}_1(\mathbf{p}_i - \mathbf{x}_i(t))$ and $c_2\mathbf{r}_2(\mathbf{p}_g - \mathbf{x}_i(t))$ describe the particle's "cognitive" and "social" activities, respectively. The new velocity $\mathbf{v}_i(t+1)$ is determined by the momentum part, "cognitive" part, and "social" part, Eq. (1). The particle's position at generation $t+1$ is updated with $\mathbf{x}_i(t)$ and $\mathbf{v}_i(t+1)$, Eq. (2).

search method such as the Hooke–Jeeve method, is used (Doctor et al., 2004). In the inner level, PSO is applied. The structure of the inner swarm is the actual implementation of the problem, i.e., a population of particles evolved for the best neural evaluator for capture Go. In the outer swarm, a particle represents a set of parameters, i.e., $\{w_{\text{in}}, c_{1\text{ in}}, c_{2\text{ in}}\}$, used for its inner swarm to optimize the parameters. The global best of the inner swarm is marked as the local best of the corresponding particle of the outer swarm. The outer swarm performs the standard PSO procedure to update its particles. The new parameter settings (positions of the particles of the outer swarm) are then sent to the inner swarms for the next round of optimization.

The hierarchy is explained in Fig. 7. For the outer swarm, the particles' positions and velocities are initialized in a three dimensional space corresponding to $w_{\text{in}}, c_{1\text{ in}}$ and $c_{2\text{ in}}$. The parameters of each particle of the outer swarm are used by the inner swarm, and a fitness value is returned. The fitness function assigns to each outer swarm particle, the best performance of all particles in the corresponding inner swarm, when played against a fixed opponent (The fixed player has a hybrid trained engine for 15 generations. The outer swarm measures how good the improvement, from such a fixed stage, is with a certain set of parameters. The fixed player serves as a base point for all sets of tested parameters, for comparison). The PSO process is applied,

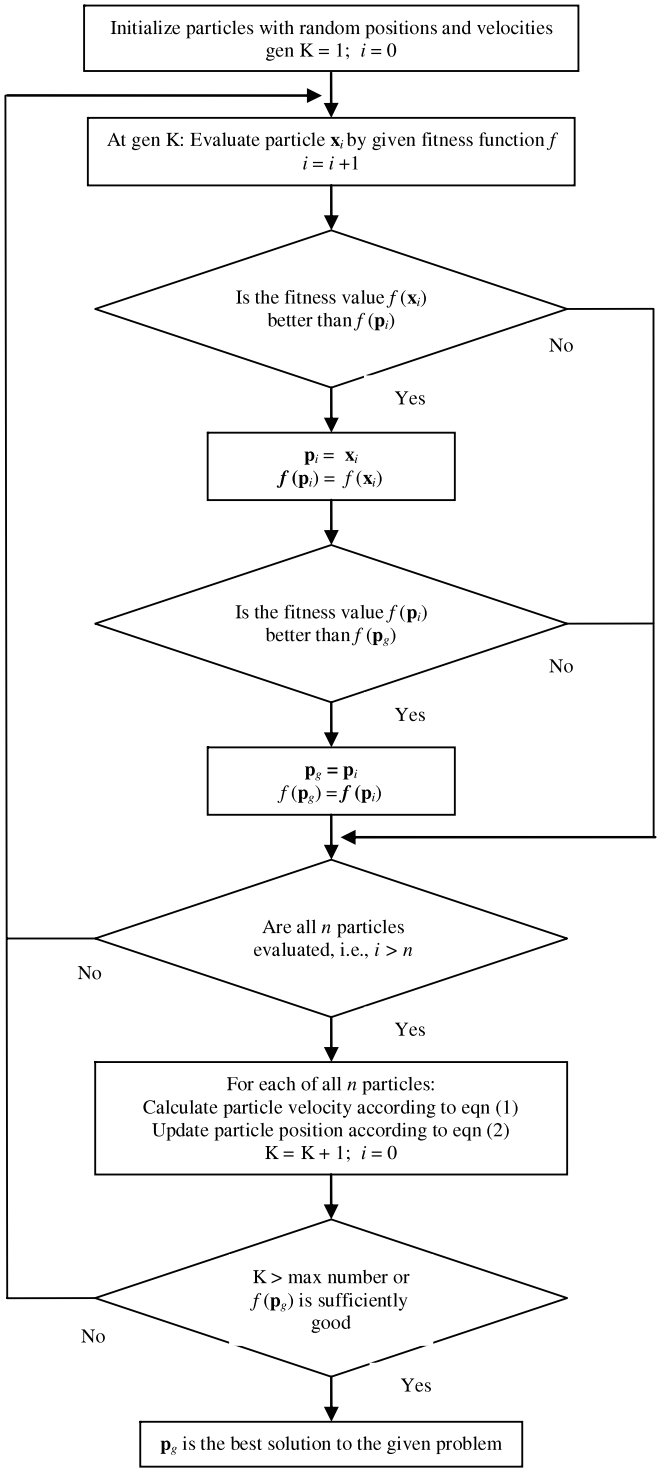

Fig. 6. Flow chart of PSO procedure.

based on the fitness returned for each particle, until the outer swarm termination criterion is met. Thus, the solution vector, that is the position vector of the gbest of the outer swarm, gives the optimal set of values for w, c_1, and c_2. The values of w_{out}, $c_{1\text{ out}}$, and $c_{2\text{ out}}$ of the outer swarm are 0.8, 2, and 2, respectively.

4.3. Evolutionary algorithm

The evolutionary algorithm (Engelbrecht, 2002; Schwefel, 1995) begins with a uniformly random population of n neural

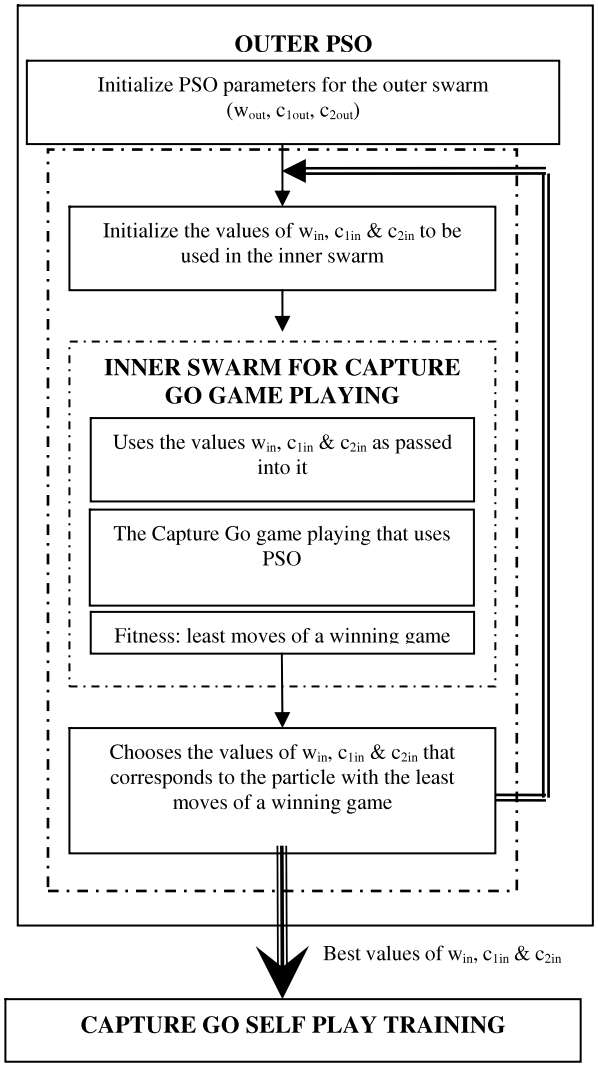

Fig. 7. Flowchart for determining optimal PSO parameters ($w_{\text{in}}, c_{1\text{ in}}, c_{2\text{ in}}$) using PSO (outer PSO).

networks, K_i, $i = 1, \ldots, n$ (see Section 5 for detail). 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

$$\sigma_i'(j) = \sigma_i(j) \exp(\tau N_j(0, 1)), \quad j = 1, \ldots, N_w \tag{3}$$

$$w_i'(j) = w_i(j) + \sigma_i' N_j(0, 1), \quad j = 1, \ldots, N_w \tag{4}$$

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 (Chellapilla & Fogel, 1999).

For the Capture Go game training, a population of 40 individuals competed for the best game board evaluator. Each individual represents a MLP described in Section 3. The number of weights and biases, i.e., N_w, in such a MLP (size $= 140 \times 40 \times 10 \times 1$) was 6216, and hence τ was 0.0796. The best half of the population was used as parents to create offspring for the next generation.

4.4. Hybrid of PSO and EA

The PSO algorithm's emphasis is on the co-operation among the particles to obtain the best fitness value. Each particle tracks the best performance in its own history, and shares such information within a certain neighborhood throughout the entire search process. Such a mechanism guides particles to pursue better fitness values quickly (Trelea, 2003; van den Bergh & Engelbrecht, 2004). However, particles of standard PSO are not eliminated even if they are ranked as having 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 (Esquivel & Coello Coello, 2006; Higashi & Iba, 2003; Miranda & Fonseca, 2002; Ting, Wong, & Chung, 2005; Xia, Wu, Zhang, & Yang, 2004). Clearly, the advantage of one algorithm can complement for the other's shortcoming. Thus, the motivation is to develop a hybrid 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 PSO and EA. In other words, the hybrid algorithm maintains the properties of competition and diversity in EA, and applies PSO to improve the surviving individuals as well. 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 (Various improved versions of PSO exist (Clerc & Kennedy, 2002; Miranda & Fonseca, 2002; van den Bergh & Engelbrecht, 2004), so we point out that a user may choose to use one of these, instead of basic PSO, in a hybridization, if desired). 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. 8 illustrates this hybrid PSO+EA algorithm.

5. Simulation design

Section 5.1 of this section describes how a PSO algorithm is used to obtain the optimal parameters of a second PSO algorithm that trains the hybrid engine. Section 5.2. explains the experimental procedure for self-play of PSO, EA, and the hybrid. Section 5.3. covers how the hybrid engine is also trained to play against a defensive player.

5.1. PSO optimization

A smaller population of ten particles is used in the outer particle swarm. It is reported in Gudise and Venayagamoorthy (2003) that 8 particles can obtain a similar performance as a swarm of 25 particles but much faster. Each particle has three dimensions representing $w_{\text{in}}, c_{1\text{ in}}$ and $c_{2\text{ in}}$, that is used in the inner particle swarm evolution consisting of 20 individuals. In the inner swarm, each particle plays against a fixed opponent for one game. The particle is scored not only on the game result, i.e., win, draw, or lose, but also on how well it plays the game, i.e., how many moves for a win, draw, or loss. For example, a win is assigned a score of "Sup(M)−M", where Sup(M) denotes the maximum number of moves possible for a game, 81 moves in the 9×9 case, and M is the number of moves actually used; a loss is scored as "−(Sup(M)−M)", and a draw is scored as "0". Therefore, a particle that wins quickly will gain a big positive score. Even if it loses, it can get a small negative score when it plays well (defined by taking a long time to lose). This evaluation distinguishes the quality of players better than that of "+1, 0, −1" when only one game is played. In the outer swarm, the fitness value of each particle is the best score of the corresponding

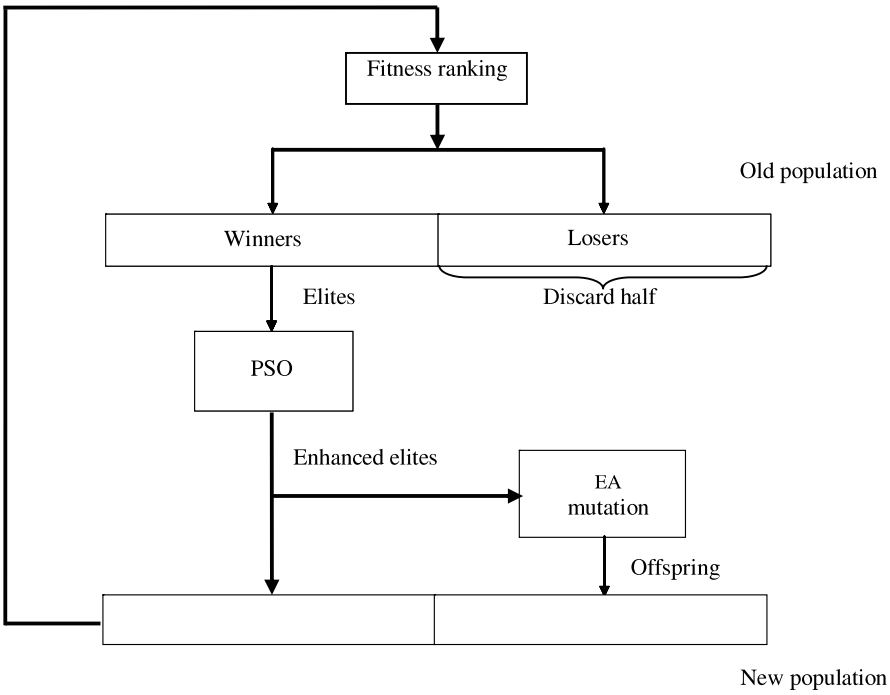

Fig. 8. Flow chart of the hybrid PSO–EA method. The winners, which contain half of the population, are enhanced by PSO and kept in the population for the next generation. Those enhanced winners also work as the parents in EA to produce offspring. The offspring replace the discarded losers to keep a constant 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.

inner swarm. If there is a draw, the number of winning games in the inner swarm works as a tie breaker.

5.2. Self-play training

In the self-play stage, a population of 40 individuals, each representing a game strategy, 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.

In the population evolved by PSO, particles compete with their recorded best to determine the new local best. The new local best also compete for a new global best candidate, and it challenges the stored global best (Considering that particles are improving during the evolution, scores earned in different generations may not be fair to compare. Therefore, a game is played between the currently found best and the recorded best, and the winner is stored as the best). The positions of particles are then updated according to Eqs. (1) and (2). In the population evolved by the EA, 20 strategies that receive the greatest total scores are retained as parents and generate offspring based on Eqs. (3) and (4) to replace the losing 20 strategies. In the population evolved by the hybrid algorithm, the 20 particles with the highest total scores are enhanced by standard PSO, i.e., Eqs. (1) and (2). Using an EA, the enhanced particles are also used as parents to produce offspring, based on Eqs. (3) and (4), to replace the 20 discarded particles as shown in Fig. 8.

Each game is played using an alpha–beta minimax search of the associated game tree for each board position, looking a pre-defined number of moves into the future. The depth of the search is set to six, ensuring the board is evaluated after the opponent has an opportunity to respond to the player's move. In addition, a forced move, such as immediate capture attempt/save, will not be counted as one of the six depths in the alpha–beta search procedure.

Following the same self-play methodology (same configuration in each generation and same number of evolving generations), 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 (Pollack & Blair, 1998). Recent research (Runarsson & M Lucas, 2005) also shows that such a parent/child weighted averaging is important in order to get best performance from co-evolutionary algorithms. The tournament results show that PSO, EA, and the hybrid players outperform the HC player (with training parameter beta $= 0.05$ (Pollack & Blair, 1998)), which indicates that the improving of game engines comes mainly from the learning algorithms.

5.3. Against hand-coded defensive player

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 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. 9). The game result indicates that it is more likely to defeat this defensive player by occupying more territories, rather than capturing its stones. Competing with this player teaches our hybrid engine to manage the balance between seizing its own territories and capturing enemy stones.

6. Simulation results

6.1. PSO optimization

In the outer swarm, each dimension of a particle, w_{in}, $c_{1\,\text{in}}$ and $c_{2\,\text{in}}$, is initialized from a uniform distribution over [0.2, 1.2],

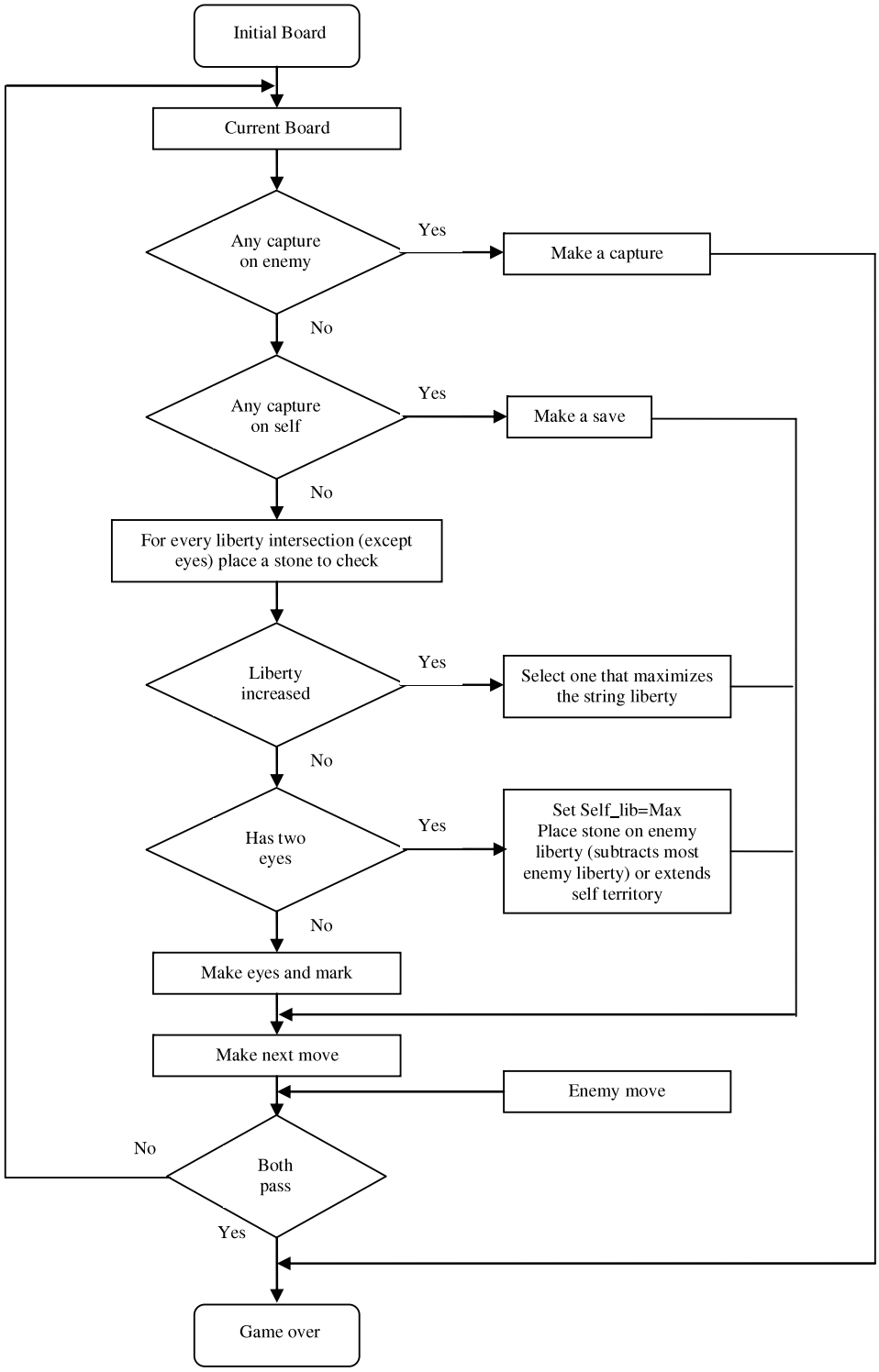

Fig. 9. Flowchart of the defensive player.

[0.25, 2.5], and [0.25, 2.5], respectively. The value of $v_{\max}$ is set to 1.0 and the position of $(w_{\text{in}}, c_{1\,\text{in}}, c_{2\,\text{in}})$ is restricted in the three intervals above. In the inner swarm, each dimension of a particle is initialized from a uniform distribution over $[-0.2, 0.2]$. The value of $v_{\max}$ is set to 2.0.

The best performance among 200 particles (10 sets of parameters with 20 particles assigned to each set) at generation 1 is losing a game with 65 moves, and at generation 30, is winning with 8 moves. After 30 generations, the best parameter setting found is $(w_{\text{in}} = 0.4667, c_{1\,\text{in}} = 1.7971, c_{2\,\text{in}} = 2.4878)$. (This set of parameters is applied to all PSO-related algorithms below if not specified otherwise). This set of parameters results in a particle that has won with 8 moves for the last 7 consecutive generations.

6.2. Self-play training

The initial 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

Table 1
Tournament results among hybrid PSO–EA, PSO, EA, HC and random players in 1200 total games. Black players are listed in row and white players in column, and 100 games were played for each entry of the table. For example, result in row 2 column 4 means that hybrid PSO–EA player in black wins 90–10 against EA player in white. Note that the advantage of playing black is significant, so that the algorithms must be evaluated in comparison with each other.

	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

Table 2
Statistical tournament results among hybrid PSO–EA, PSO, EA, HC players, in 120,000 total games. There are 10 players in each category. 100 games are played between any two players. The average wins and standard deviation are calculated for each pair of categories, in a total of 10 000 games for each entry of the table. Black players are listed in row and white players in column. For example, result in row 2 column 4 means that hybrid PSO–EA players in black win 83 games in average with a standard deviation of 6 against EA players in white. Note that the advantage of playing black is significant, so that the algorithms must be evaluated in comparison with each other.

	Hybrid PSO–EA	PSO	EA	HC
Hybrid PSO–EA	/	77 ± 9	83 ± 6	80 ± 8
PSO	61 ± 7	/	70 ± 5	80 ± 8
EA	60 ± 8	72 ± 8	/	76 ± 9
HC	49 ± 12	51 ± 13	45 ± 13	/

iterated for 100 generations. Because the best engines of PSO, EA and the hybrid are deterministic, randomness is introduced to avoid duplication during the 100 games. For each side, the best move suggested by the engine is picked 90% of the time. Even though that each game is different (due to 10% random move each side), each engine faces roughly the same amount of noisy situations. Since the random move does not favor one particular engine, how well, shown by the game results, the engine handles these unforeseen moves still illustrates the abilities of developing strategy from unexpected board positions, recovering from its own errors, and taking advantage of opponent errors. At last, the best swarm-neuro-engine (at the generation 100) of each category, i.e., PSO, EA, and the hybrid, is then used to play against each other and a random player. Table 1 summarizes their performance in 100 games in the tournament. All players illustrate success in learning of strategies in Capture Go game because they overwhelm the random player with only 28 moves per game on average (which indicates the S-measure $= 1$ (Messerschmidt & Engelbrecht, 2004)). The hybrid PSO–EA player in black dominates both PSO and EA players. The PSO and EA players are at the same level.

In addition, each algorithm trains 10 groups of players with different initial weights for 100 generations. After training, each best player of the 10 groups plays, in both black and white, against every best player trained by other methods for 100 games. The tournament among those players contains $10 \times 10 \times 100 \times 2$ games for each pair of methods. The statistical performance of each method, measured by the average winning games and the standard deviation, is reported in Table 2 for total 120,000 games played. Finally, a web Capture Go player (admittedly weak, as they all are) (schachverein, 0000) is brought for illustration, and the 10 best hybrid players wins by 23 on average (standard deviation $= 1$) in 25 games.

6.3. Against hand-coded defensive player

The 10 best hybrid players wins by 56 in average (standard deviation $= 3$) over 100 games against the defensive player. Tables 3 and 4 contain the complete sequence of moves of two selected games between the best-evolved hybrid PSO–EA and the defensive player. The notation for each move is given in the form of $n \cdot A : (a, b)$, where n is the nth move, A denotes the black/white player and (a, b) is the board position, row a and column b, of the move. Table 3 illustrates the process of the first game where the hybrid player manages to expand its own territory and restrict its opponent's. The hybrid engine wins the game by seizing 19 more intersections than the defensive player (see Fig. 10). Table 4 illustrates the process of the second game where the hybrid demonstrates its capability in making eyes and hence maintaining safety for multiple groups. The hybrid player wins the game by seizing 11 more intersections than the defensive player (see Fig. 11). The defensive player also beats the web player, with a 21/4 in 25 games.

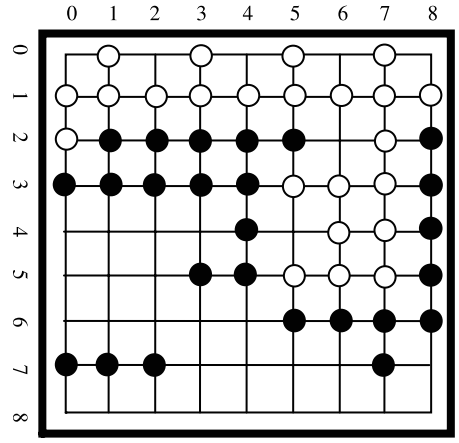

Fig. 10. Final board of game one between the hybrid PSO–EA and the defensive player.

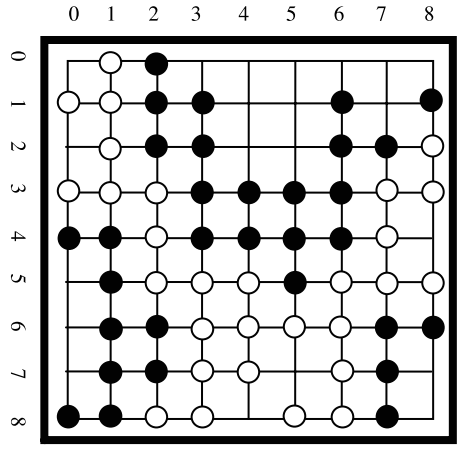

Fig. 11. Final board of game two between the hybrid PSO–EA and the defensive player.

7. Conclusions

A novel hybrid PSO–EA algorithm is employed to train a computer engine to play the benchmark game Capture Go. The algorithm is used to train a swarm-neuro-engine using a feedforward neural network as the evaluation function of the leaf nodes in a game tree, with zero expertise involved. To determine the optimal parameters of PSO, a second PSO is applied to determine an inertia weight w and acceleration constants (c_1, c_2), based on games played. Such training results in a set of PSO parameters that quickly teaches randomly initialized players to beat the fixed opponent in a few generations. With the optimal PSO parameters, the hybrid algorithm utilizes the PSO to improve

Table 3
Game one between the hybrid PSO–EA and the defensive player. The hybrid player plays black, the defensive player plays white; Comments on moves are offered in brackets.

Hybrid PSO–EA engine	Defensive expert	Comments
1. B: (4, 4)	1. W: (5, 6)	
2. B: (6, 8)	2. W: (5, 5)	
3. B: (7, 7)	3. W: (5, 7)	[Good move of hybrid engine, trying to make an eye at (7, 8) or (8, 8)]
4. B: (6, 7)	4. W: (4, 7)	
5. B: (6, 6)	5. W: (3, 7)	
6. B: (5, 8)	6. W: (3, 6)	
7. B: (5, 4)	7. W: (3, 5)	
8. B: (5, 3)	8. W: (2, 7)	
9. B: (4, 8)	9. W: (1, 7)	[Defensive expert tries to extends its territory and liberties in all directions in the past 8 moves]
10. B: (3, 8)	10. W: (1, 6)	[Hybrid engine seized opponent's liberties and potential eye position]
11. B: (3, 4)	11. W: (1, 5)	
12. B: (3, 3)	12. W: (1, 4)	
13. B: (3, 2)	13. W: (1, 3)	
14. B: (3, 1)	14. W: (1, 2)	
15. B: (3, 0)	15. W: (4, 6)	[Both sides set up the territory boundaries in the past consecutive moves]
16. B: (6, 5)	16. W: (1, 1)	[Hybrid engine blocks the entrance of possible invasion to its territory]
17. B: (2, 3)	17. W: (1, 0)	
18. B: (2, 2)	18. W: (1, 8)	
19. B: (2, 8)	19. W: (0, 7)	
20. B: PASS	20. W: (0, 1)	
21. B: (2, 1)	21. W: (0, 5)	[The hybrid seizes the rival's liberties in the past 5 moves]
22. B: PASS	22. W: (0, 3)	[The defensive is making eyes in the past 4 moves]
23. B: (2, 5)	23. W: (2, 0)	[The hybrid make a vulnerable move]
24. B: (2, 4)	24. W: PASS	[The hybrid quickly saves the vulnerable move]
25. B: (7, 2)	25. W: PASS	
26. B: (7, 1)	26. W: PASS	
27. B: (7, 0)	27. W: PASS	[The hybrid occupies possible invasion positions in its territory in the past 3 moves]
28. B: PASS	28. W: PASS	[Both sides pass and game ends. The hybrid wins by seizing 19 more intersections]

Table 4
Game two between the hybrid PSO–EA and the defensive player. The hybrid player plays black, the defensive player plays white; Comments on moves are offered in brackets.

Hybrid PSO–EA engine	Defensive expert	Comments
1. B: (4, 4)	1. W: (6, 6)	
2. B: (2, 7)	2. W: (6, 5)	
3. B: (6, 7)	3. W: (6, 4)	
4. B: (7, 2)	4. W: (6, 3)	
5. B: (6, 1)	5. W: (5, 6)	
6. B: (4, 5)	6. W: (5, 3)	
7. B: (3, 3)	7. W: (5, 2)	
8. B: (5, 1)	8. W: (7, 6)	
9. B: (5, 5)	9. W: (7, 4)	[The defensive player tries to make an eye]
10. B: (6, 2)	10. W: (4, 2)	[The hybrid engine enhances its territory at the bottom-left corner]
11. B: (4, 0)	11. W: (3, 2)	
12. B: (3, 4)	12. W: (3, 1)	
13. B: (2, 3)	13. W: (2, 1)	
14. B: (4, 3)	14. W: (1, 1)	
15. B: (1, 2)	15. W: (1, 0)	
16. B: (0, 2)	16. W: (5, 7)	[The hybrid draws the top-left boundary of its top territory in the past 5 moves]
17. B: (7, 7)	17. W: (4, 7)	[The hybrid protects its eye space at the bottom-right corner]
18. B: (2, 6)	18. W: (5, 8)	[The hybrid tries to close the top-right boundary of its top territory]
19. B: (3, 6)	19. W: (8, 6)	
20. B: (6, 8)	20. W: (8, 5)	[The hybrid is making an eye at the bottom-right corner to save its group there]
21. B: (8, 7)	21. W: (5, 4)	[The hybrid makes an eye at the bottom-right corner and saves its group there]
22. B: (3, 5)	22. W: (0, 1)	[The defensive makes an eyes at top-left corner]
23. B: (1, 3)	23. W: (3, 0)	
24. B: (4, 1)	24. W: (3, 7)	
25. B: (4, 6)	25. W: (3, 8)	
26. B: (1, 8)	26. W: (7, 3)	[Both sides try to settle down their territory boundary at the top-right corner]
27. B: (7, 1)	27. W: (8, 3)	
28. B: (8, 1)	28. W: (8, 2)	
29. B: (8, 0)	29. W: (2, 8)	[Both sides finalize their territory boundary at the bottom-left corner]
30. B: (1, 6)	30. W: PASS	[Both sides finalize their territory boundary at the top-right corner]
31. B: (2, 2)	31. W: PASS	
32. B: PASS		[Both sides pass and game ends. The hybrid wins by seizing 11 more intersections]

the winning individuals of EA. The enhanced elites are then used as parents to produce offspring, by EA mutation, to replace losing individuals. After generations of self-play training, results of tournaments among hybrid PSO–EA, PSO, and EA show that the hybrid PSO–EA approach performs better than both PSO and EA. Defeating the HC trained game engine also verifies that the hybrid algorithm is responsible for successfully learning the strategy.

A hybrid engine is also trained and tested to play with a defensive knowledge-based Capture Go player. The hybrid engine manages to expand and fortify its own territory, instead of capturing the opponent's stone, to countervail the defensive strategies of the hand-coded player. Two winning games by the swarm-neuro-engine demonstrate that it balances the goals of occupying territory and capturing stones, as well.

The success in learning one of the key strategies of Go, i.e., capture and defense, through Capture Go will demonstrate that other strategies, such as piece sacrifice, 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.

Acknowledgements

The authors gratefully acknowledge the support from the National Science Foundation and the M.K. Finley Missouri endowment.

References

Allis, L. V., van den Herik, H. J., & Herschberg, I. S. (1991). Which games will survive? In D. N. L. Levy, & D. F. Beal (Eds.), *Heuristic programming in artificial intelligence 2—The second computer olympiad* (pp. 232–243). Ellis Horwood.

Axelrod, R. (1984). *The evolution of cooperation*. New York: Basic Books.

Berlekamp, E., & Wolfe, D. (1994). *Mathematical Go: Chilling gets the last point*. MA: A.K. Peter Ltd..

Cai, X., & Wunsch, D. II (2004). Evolutionary computation in playing Capture Go game. In *Proc. of international conference on cognitive and neural systems*.

Cai, X., Zhang, N., Venayagamoorthy, G. K., & Wunsch, D., II (2006). Time series prediction with recurrent neural networks using hybrid PSO–EA algorithm. *Neurocomputing*, *70*(13–15), 2342–2353.

Chellapilla, K., & Fogel, D. (1999). Evolution, neural networks, games, and intelligence. *Proceedings of the IEEE*, *87*(9), 1471–1496.

Chellapilla, K., & Fogel, D. (1999). Evolving neural networks to play Checkers without relying on expert knowledge. *IEEE Transactions on Neural Networks*, *10*(6), 1382–1391.

Chellapilla, K., & Fogel, D. (2001). Evolving an expert Checkers playing programs without using human expertise. *IEEE Transactions on Evolutionary Computation*, *5*(4), 422–428.

Chen, X., Zhang, D., Zhang, X., Li, Z., Meng, X., He, S., et al. (2003). A functional MRI study of high-level cognition II. The game of Go. *Cognitive Brain Research*, *16*(1), 32–37.

Clerc, M., & Kennedy, J. (2002). The particle swarm: Explosion, stability, and convergence in a multi-dimension complex space. *IEEE Transactions on Evolutionary Computation*, *6*(1), 58–73.

Doctor, S., Venayagamoorthy, G. K., & Gudise, V. G. (2004). Optimal PSO for collective robotic search applications. In *Proc. of the IEEE congress on evolutionary computation* (pp. 1390–1395).

Doctor, S., & Venayagamoorthy, G. K. (2005). Improving the performance of particle swarm optimization using adaptive critics designs. *IEEE swarm intelligence symposium*.

Enderton, H. D. (1991). The golem Go program. *Tech. rep. CMU-CS-92-101*. Carnegie Mellon University.

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

Engelbrecht, A. P. (2002). *Computational intelligence—An introduction*. Wiley, ISBN: 0-470-84870-7.

Esquivel, S. C., & Coello Coello, C. A. (2006). Hybrid particle swarm optimizer for a class of dynamic fitness landscape. *Engineering Optimization*, *38*(8), 873–888.

Fogel, D. (2002). *Blondie24: Playing at the edge of AI*. SF, CA: Morgan Kaufmann.

Fogel, D., Hays, T. J., Hahn, S. L., & Quon, J. (2004). A self-learning evolutionary Chess program. *Proceedings of the IEEE*, *92*(12), 1947–1954.

Fogel, D., Hays, T. J., Hahn, S. L., & Quon, J. (2005). Further evolution of a self-learning Chess program. *Proc. of IEEE symposium on computational intelligence in games* (pp. 73–77).

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

Fürnkranz, J. (2001). Machines that learn to play games. In J. Fürnkranz, & M. Kubat (Eds.), *Machine learning in games: A survey* (pp. 11–59). Huntington, NY: Nova Scientific Publishers, [Chapter 2].

Gomez, F., & Miikkulainen, R. (1997). Incremental evolution of complex general behavior. *Adaptive Behavior*, *5*, 317–342.

Gudise, V. G., & Venayagamoorthy, G. K. (2003). Comparison of particle swarm optimization & backpropagation as training algorithms for neural networks. *IEEE swarm intelligence symposium* (pp. 110–117).

Higashi, N., & Iba, H. (2003). Particle swarm optimization with Gaussian mutation. In *Proc. of IEEE swarm intelligence symposium* (pp. 72–79).

Hsu, F. (2002). *Behind deep blue*. Princeton, NJ: Princeton Univ. Press.

Kennedy, J., & Eberhart, R. (1995). Particle swarm optimization. In *IEEE international conference on neural networks* (pp. 1942–1948). *Vol. 4*.

Kennedy, J., Eberhart, R., & Shi, Y. (2001). *Swarm intelligence*. San Meteo, CA: Morgan Kaufmann.

Konidaris, G., Shell, D., & Oren, N. (2002). Evolving neural networks for the capture game. In *Proc. of the 2002 annual research conference of the south African institute of computer scientists & information technologists postgraduate symposium*. Available from: http://www-robotics.usc.edu/~dshell/res/evneurocapt.pdf.

Maynard Smith, J. (1982). *Evolution & the theory of games*. Cambridge, UK: Cambridge University Press.

Mechner, D. (1998). All systems Go. *The Sciences*, *38*(1).

Messerschmidt, L., & Engelbrecht, A. P. (2004). Learning to play games using a PSO-based competitive learning approach. *IEEE Transactions on Evolutionary Computation*, *8*(3), 280–288.

Miranda, V., & Fonseca, N. (2002). EPSO-best-of-two-worlds meta-heuristic applied to power system problems. In *Proc. of IEEE congress on evolutionary computation* (pp. 1080–1085).

Morgenstern, O. (1949). Economics & the theory of games. *Kyklos*, *3*(4), 294–308.

Morris, P. (1994). *Introduction to game theory*. NY: Springer-Verlag.

Muller, M. (1999). Decomposition search: A combinatorial games approach to game tree search, with applications to solving Go endgames. In *Proc. of international joint conference on artificial intelligence* (pp. 578–583).

Pollack, J. B., & Blair, A. D. (1998). Co-evolution in the successful learning of Backgammon strategy. *Machine Learning*, *32*, 226–240.

Richards, N., Moriarty, D., Mc Questen, P., & Miikkulainen, R. (1998). Evolving neural networks to play Go. *Applied Intelligence*, *8*, 85–96.

Runarsson, T. P., & M Lucas, S. (2005). Co-evolution versus self-play temporal difference learning for acquiring position evaluation in small-board Go. *IEEE Transactions on Evolutionary Computation*, *9*(6), 628–640.

http://www.schachverein-goerlitz.de/Foren/Fun/Go/go.htm.

Schaeffer, J. (1996). *One jump ahead: Challenging human supremacy in Checkers*. New York: Springer.

Schraudolph, N., Dayan, P., & Sejnowski, T. (1994). Temporal difference learning of position evaluation in the game of Go. *Advances in Neural Information Processing*, *6*, 817–824.

Schwefel, H. P. (1995). *Evolution & optimum seeking*. NY: Wiley.

Shannon, C. E. (1950). Automatic chess player. *Scientific American*, *182*(2), 48–51.

Shi, Y., & Eberhart, R. C. (1998). Parameter selection in particle swarm optimization. In *Proceedings of the 1998 annual conference on evolutionary computation*.

Shi, Y., & Eberhart, R. C. (2001). Fuzzy adaptive particle swarm optimization. In *Proceedings of the 2001 Congress on evolutionary computation*.

Shi, Y., & Eberhart, R. C. (2001). Particle swarm optimization with fuzzy adaptive inertia weight. In *Proceedings of the workshop on particle swarm optimization*.

Shi, Y. (2004). Particle swarm optimization. In *IEEE connectors, the Newsletter of the IEEE Neural Networks Society*: *Vol. 2(1)* (pp. 8–13).

Smith, A. (1956). *The game of Go*. Tokyo, Japan: Charles Tuttle Co..

Ting, T. O., Wong, K. P., & Chung, C. Y. (2005). Investigation of hybrid genetic algorithm/particle swarm optimization approach for the power flow problem. In *Proc. of international conference on machine learning & cybernetics 2005* (pp. 436–440).

Trelea, L. C. (2003). The particle swarm optimization algorithm: Convergence analysis & parameter selection. *Information Processing Letters*, *85*(6), 317–325.

van den Bergh, F. (2002). An analysis of particle swarm optimizers. *Ph.D. dissertation*. Dept. of Computer Science. Univ. Pretoria, Pretoria, South Africa.

van den Bergh, F., & Engelbrecht, A. P. (2004). A cooperative approach to particle swarm optimization. *IEEE Transactions on Evolutionary Computation*, *8*(3), 225–239.

von Neumann, J., & Morgenstern, O. (1944). *Theory of games & economic behavior*. Princeton, NJ: Princeton Univ. Press.

Wolpert, D. H., & Macready, W. G. (2001). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, *1*(1), 67–82.

Xia, W., Wu, Z., Zhang, W., & Yang, G. (2004). A new hybrid optimization algorithm for the job-shop scheduling problem. In *Proc. of American control conference 2004* (pp. 5552–5557).

Zaman, R., Prokhorov, D. V., & Wunsch, D. C. II (1997). Adaptive critic design in learning to play the game of Go. In *Proc. of the international joint conference on neural networks* (pp. 1–4).

Zheng, Y., Ma, L., Zhang, L., & Qian, J. (2003). Empirical study of particle swarm optimizer with an increasing inertia weight. In *Proc. of IEEE congress on evolutionary computation* (pp. 221–226).