

Learning to Play Complex Games

David M. Kaiser

Florida International University
Computer Science Department
SW 8th Street
Miami Florida
dkaiser01@fiu.edu

Abstract. We describe an algorithm, inspired by the biological behavior of ants, to efficiently determine intelligent moves and the genetic algorithm used to learn better strategies in a broad class of games. We present a class of games called Simple War Games that includes both deterministic and non-deterministic, perfect information, two-player, zero sum games. Simple War Games have a branching factor and game tree that is magnitudes larger than that of either chess or go and thus standard search techniques are inappropriate.

1 Introduction

Computer game playing is one of the oldest areas of endeavor in Artificial Intelligence. However, most research in Computer Game-Playing has been focused on a small number of specific games (most notably chess, checkers, backgammon, Othello and go). The result has been the development of programs capable of playing a single game very well, yet can play no other. Not enough research has been done in the way of general game playing. There is a need for more research on a general theory of game playing, in particular theory that is suitable for implementation. This paper is a step in that direction.

This paper defines an original class of games, called Simple War Games, which includes both deterministic and non-deterministic games. This class contains games that are comparable in complexity to checkers and chess. This paper also describes several example games.

We developed the program WAR, which is capable of playing any Simple War Game. The program uses a unique algorithm, which we created to evaluate moves in Simple War Games. The program also has a learning component that uses a genetic algorithm. This paper describes the evaluation algorithm and the results of the learning process.

This paper is organized in the following manner. Section 2 examines the area of computer game playing. Section 3 introduces the class of games called Simple War Games. Section 4 describes WAR, a program to play Simple War Games. Section 5 presents the results of several runs of WAR on different games and gives an analysis of its performance. Section 6 relates our program to other programs capable of playing more than a single game. Section 7 presents direction for future research.

2 Computer Game Playing

Many different games have been studied in Computer Game Playing. However, most research revolves around just a few very similar games. This chapter presents these games, examines their similarities, compares their complexity, and explores strategies that researchers have used to create successful game playing programs. The intent is to give the reader a brief overview of the current status of Computer Game Playing, show that research in this area has been too restricted, provide a reference for comparing the class of games presented in the next chapter, and glean insight into which methods might be used in solving other complex games.

2.1 Two-Person, Perfect Information, Deterministic, Zero-Sum Games

We define a *game* as a decision problem with two or more decision makers – players – where the outcome for each player may depend on the decisions made by all players. In a *two-person* game there are two players that make moves alternately. Chess, checkers and tic-tac-toe are examples of two-person games.

A game in which all the players have complete information of the current situation in the game is called a *perfect information* game. Othello is a perfect information game. Most card games are *imperfect information* games because neither player knows what order the cards are in.

A *deterministic* game is one in which the outcome of each move, or transition from one state in the game to another, is known before hand. Chess is a deterministic game. Both players know exactly what will happen when a player moves a piece from one location to another. Backgammon is an example of a *non-deterministic* game. Neither player knows what the roll of the dice will be and therefore do not know what moves will be possible on future turns.

A *zero-sum* game is a game in which one player's winnings equal the other player's losses. If we add up the wins and losses in a game, treating losses as negatives, and we find that the sum is zero for each set of strategies chosen, then the game is a zero-sum game. Most familiar games like bridge, chess and checkers are zero-sum games.

2.2 Game Complexity

One measure of complexity is the game's *average branching factor*. This is the average number of moves that a player can make during their turn at any given point in the game.

Another way of measuring the complexity of a game is to determine the size of the *game tree*. In deterministic games, the nodes in such a tree correspond to game states, and the arcs correspond to moves. The initial state of the game is the root node; leaves of the tree correspond to terminal states. In non-deterministic games, chance nodes must also be used to represent the variable elements of the game, such as dice

rolls in backgammon. Each level in the game tree represents a single *ply*, or one player's turn.

The size game tree can be calculated by taking the average branching factor and raising it to the power of the number of ply the game usually lasts before ending. In chess for example, each player has, on average, about 36 possible moves. Thus, the average branching factor for Chess is 36. Since the average game of chess last around 40 moves per player (or 80 ply) the game tree is 36^{80} or approximately 10^{124} .

By examining the entire game tree, all the way down to the terminal states, it would be possible to find the best strategy for playing the game. However, generating the entire game tree for complicated games, like chess, is completely infeasible.

2.3 Popular Games

In Fürnkranz's bibliography on Machine Learning in Games [1] more than 37% of the papers deal with chess. The four most popular games chess, Othello, go, and checkers are employed in over 64% of the papers. A full 80% of the papers use two-player, perfect information, deterministic, zero-sum as test beds for Machine Learning. Only 6% of the papers cover non-deterministic games such as backgammon, while another 6% contain games of imperfect information (Scrabble, bridge, etc.).

Obviously, these are not all the games played in the world, nor even all the games studied in the field of Computer Game Playing. There are many variations on these popular games, not to mention completely different games not mentioned. However, these games are fairly representative of games used in the area of Computer Game Playing and are the ones that have garnered the lion's share of attention in this area.

2.5 Constructing a Game Playing Program

Since searching the entire game tree exhaustively is usually not feasible, other techniques that rely on searching only a part of the game tree have been developed. Most computer game playing programs use some variant of the Alpha-Beta algorithm. The algorithm has proven to be a valuable tool for the design of two-player, zero-sum, deterministic games with perfect information. Since its creation in the 1960's, the basic structure has changed little, although there have been numerous algorithmic enhancements to improve the search efficiency [2].

One method that developers have used to improve the performance of game playing programs is called an *end game database*. Information, on which positions near the end of the game are won, lost or drawn, is stored in a database. The effect of end-game databases is to effectively extend the programs search ability. This method has proved successful in games like Othello and checkers.

Since it is usually not possible to search the entire game tree, it is usually necessary to evaluate different positions in order to choose the next move. This is done with an *evaluation function*. The evaluation function returns an estimate of the expected outcome of the game from a given position. The performance of a game-playing program is dependent on the quality of the evaluation function. If it does not accurately reflect the actual chances of winning, then the program will choose moves that lead to losing positions. The actual numeric values of the evaluation function are not important so long as a better position has a higher value than a worse position [2].

2.6 Complex Programs

We will now briefly present methods that have proven useful in creating superior game playing programs, and list the relative complexity for a handful of the popular games. Figure 2.3 shows the average branching factor and estimated game tree size for several different games.

Chess: Deep Blue is arguably the best computer Champion. It uses a relatively simple evaluation function with a 10-ply search. Custom built for chess by a team of IBM scientists, Deep Blue weighs 1.4 tons, and has 32 microprocessors that give it the ability to look at 200 million chess positions each second [3].

Backgammon: With 30 pieces, 26 locations and all possible combinations of the dice roll, backgammon has branching factor of several hundred per ply. Tesauro's TD-Gammon has attained a Champion level of performance using a neural network, trained by self-play. The program is better than any other computer program playing backgammon and plays at a level nearly equal to the world's best players. TD-Gammon uses only a 2-ply search [4].

Go: Go has nine rules, two kinds of pieces and a 19x19 board. Despite its simplicity, it has an average branching factor of 250 and the size of the game tree is about 10^{360} . Computer programmers have found it challenging to create programs that can compete against average players. Go4++ and Handtalk are among the strongest of all go programs. Go4++ uses a process of matching 15 high level patterns with the current game state to generate about 50 candidate moves that are then analyzed to find the best move. Handtalk also uses pattern matching to evaluate a very small number of candidate moves [5].

3 A New Class of Games

In order to create computer programs capable of playing many different games, we developed a new class of games that has the following properties: (a) it is large enough to include interesting games; (b) it is simple to describe so that it is manageable to work with; (c) it includes both deterministic and non-deterministic games; (d) the games are two-person, perfect information, zero-sum games.

We call this original class of games Simple War Games. The basic idea is that each player can move some, none, or all of their pieces each turn. This creates a branching factor that is staggering. If this were possible in chess, for example, the average branching factor would be over 10,000 as compared to 35. The solutions that have proven successful in many other games might prove less adequate in this domain.

3.1 Simple War Games

Simple War Games are concerned only with moving pieces around the board and resolving combat between them. The three main elements for any game in this class are: the board, the pieces, and combat resolution. While definition of the rules for Simple War Games is original, the intent was to include components that are common to a large number of existing games.

Order of Play Game play alternates between the two players. During their turn each player is allowed to move as few or as many of their pieces as they like. The game ends when any of the following conditions are met: (a) One player has no pieces; (b)

One player has scored the winning number of points or more; (c) The maximum number of turns for the game has been reached.

The Pieces Pieces are what the player uses to change the state of the game. Pieces have a specific location on the board and have various other attributes to describe their state. There may be no more than one piece in a given location.

Every piece starts out with a number of points representing how much damage it can sustain before being removed from the board. In chess, the points for each piece would be one. Each piece also has a score, which is used to determine the winner of the game.

The Board A board is defined as a set of locations where pieces can be placed. The locations are arranged in a regular matrix in either of the two formations: *square* which is a grid similar to that used in games like chess, checkers, or Othello and *offset*. The offset arrangement is similar to the hexagon board used by most military war games and allows a more realistic representation of movement.

Movement Each piece has a movement allowance, which represents the distance it can move in one turn. Each location on the board has a movement cost. A piece can move from one location to another so long as it has enough movement points. Each turn, the player moves any or all of their pieces. Pieces are allowed to another location so long as the movement cost of the path is less than the movement allowance of the piece. Movement points cannot be saved from one turn to the next.

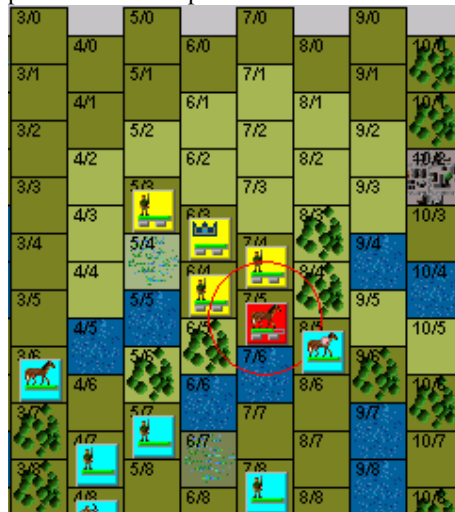


Figure 3.5 – Legal Moves for Piece at 7/5

Combat In a game like chess, combat is pretty simple; when one piece attacks, the opponent's is eliminated. But if you were trying to be a bit more realistic, perhaps the

enemy soldier was only wounded, or three men in the squad were killed but the remaining five survived.

Combat can be either deterministic (like chess) or non-deterministic (roll the dice and see what happens). If the points of damage resulting from combat are greater than the defending piece's remaining hit points, the defending piece is removed from the board, and the attacking player's score is increased by the score of the eliminated piece.

3.2 Sample Game Definitions

In this section we will describe the two sample games used for developing and training the WAR program. Definitions that are more detailed can be found in [6]

SIMPLE The first sample game is called SIMPLE. SIMPLE is meant to be a very easy introduction to the class of Simple War Games. The board is rather small: 5 by 5 offset squares. Each side has four pieces: 2 infantry, 1 horse, and 1 king. The infantry and king can move 1 square; the horse can move 2 squares. The game is deterministic: a horse or king can be eliminated by four hits from any enemy piece; two hits will eliminate an infantry piece. The game is over when a player eliminates the opponent's king (i.e. scores 900 points) or after each player completes fourteen turns. The setup for SIMPLE is shown in Figure 3.8.

In order to compare the complexity of four sample Simple War Games with the popular games, we examine the branching factor and average ply. The branching factor for SIMPLE ranges from 30, at the start of the game, to over 1100 in the middle game. Each player has 4 pieces that can move between 4 and 18 different locations each turn and then have a choice of attacking between 0 and 4 different enemy pieces. The average branching factor of SIMPLE is approximately 300. The game tree size for SIMPLE (10^{34}) is comparable with that of checkers (10^{31}).

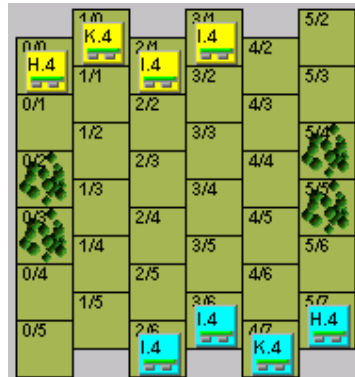


Figure 3.8 – The board for SIMPLE

TANK TANK is a combat between two companies of armored vehicles. TANK is meant to be complex enough to show the breadth of the class, yet not strain the user too much waiting for the program to make a move. There are 7 different piece types

and each player has 14 pieces. The game ends only after 14 turns or when one player has no pieces. The major difference here is that combat is resolved non-deterministically. An attack might result in eliminating the enemy piece or no damage at all. The setup for TANK is shown in Figure 3.11.

Despite the fact that some pieces in the sample game TANK have no movement capability, the combinations of moves available to each player is huge. The average number branching factor of TANK is around 10^{10} . While the average ply is only around 16 this still creates a game tree size (10^{166}) which is comparable with chess (10^{124}).

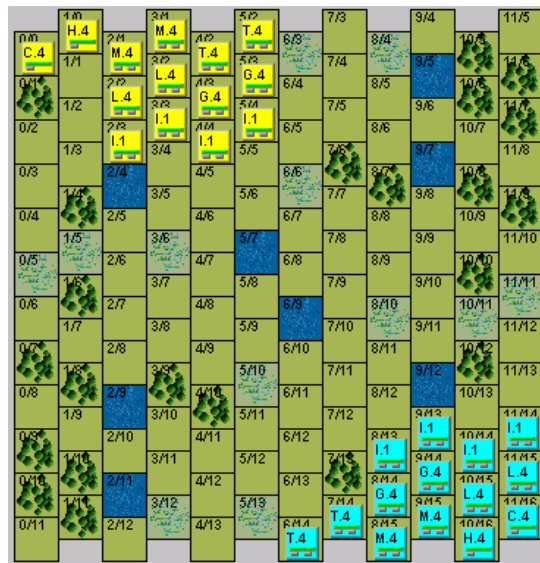


Figure 3.11 – The board for TANK

3.5 Comparing Game Classes

The class of Simple War Games is an easily described class of games, which encompasses both deterministic and non-deterministic games. The class contains games that are as complex as checkers and possibly more difficult than go. It provides a set of games that is more realistic than chess, yet they are simple enough to offer a good analysis. Despite their descriptive simplicity, they form a large class, rich enough to present implementation challenges. By defining a class of games instead of one single game, it will be possible for researchers to test theories on very simple games, and scale up to larger and more complex games without having to change to a completely new game. Code that implements the mechanics of the game need not be changed, as would be the case when moving from tic-tac-toe to checkers. This will allow more time to be spent focusing on methods to improve game playing performance.

4 The WAR Program

We created a program, called WAR, to play any game in the class of Simple War Games. The program was implemented in Java 1.0 on a 90mhz Pentium. Of its approximately 9100 lines of code, 72% was devoted to the playing environment, 4% to learning, and 24% to the game playing strategy described here.

Many methods used to create successful programs for popular games such as chess or Othello are not applicable to Simple War Games. Generating databases of opening moves and end games for specific Simple War Games, even the sample games we described in Chapter 3, is counter to our goal of generalization.

4.1 Strategy for WAR

The strategy we have developed is inspired by the way ant colonies function. Our approach is to give each piece a little bit of brainpower with the idea that each local good decision will contribute towards a globally good strategy. In other words, rather than formulate a strategy for the entire army, we have one set of rules that is the same for each soldier. Other search methods have also been inspired by ant behavior [7].

Every piece has the same goals:

- 1) Attack. Eliminate opponent's pieces.
- 2) Safety. Protect yourself.

To determine where to move and which pieces to attack, WAR uses an evaluation function. This evaluation function is comprised of two primary advisors, Attack and Safety. These two advisors, described in the following sections, are themselves made up of several advisors. All the advisors use the game definition and the current game state to determine the best move for each piece.

When it is the programs turn to move, each piece is selected in turn and the best move is made for that piece. The "best move" is determined by rating each location to which the currently selected piece can move. The locations are rated by the evaluation function using the advisors. The process continues until the program is finished moving and attacking with all the pieces it controls.

Attack The purpose of the attack advisor is to rate a location in terms of how beneficial the position would be for this piece with respect to attacking the enemy. The idea is to move the piece into position to do the greatest amount of damage to the enemy as possible, without taking the safety of the piece into consideration. The Attack Advisor is made up of three advisors: the King Advisor, the Inverse Range Advisor and the Highest Priority Advisor.

Piece Priority The Piece Priority Advisor helps determine which piece should be attacked. If there are multiple targets available, which one should the program choose to attack? All other things being equal, it would be more beneficial to eliminate an opponent's a queen rather than a pawn. The queen would have a higher priority. But if capturing the enemy pawn wins the game, the pawn would be the better choice. This advisor rates a piece by assigning a number that ranks the piece relative to the other pieces.

The values calculated by the Piece Priority advisor are dynamic. Pieces will be rated differently as the game progresses. The value is determined by the current condition of the piece (i.e. whether or not it is wounded), the location of the piece and the

other pieces left in the game. The value returned by the Piece Priority advisor is a weighted sum of the values returned by the following advisors:

Safety The Safety Advisor generates an integer that measures the relative danger at any given location. This Safety advisor gives an indication of whether the piece should enter the given location. The safety value must take into consideration the location of all the pieces on the board, friend and foe alike. The safety value for a piece/location returned by the Safety advisor encapsulates information such as: Terrain defense modifier; Minimum range to a friendly piece; Minimum range to an opponent piece; Range to the center of the friendly formation; Range to the center of the opponent formation; Number of opponent pieces that can attack; Number of friendly pieces that can attack.

5 Experiments

The program WAR uses the advisors presented in the previous section to determine how to play the game. A separate text file, with different weights for each advisor, was created for each game. Although the same program, WAR, is used to play Simple War Games, each separate text file, of advisor weights, represents a different strategy for playing. A number of these strategies were created for testing the WAR program. These are *agents*.

An agent that moves randomly in this class of games is simply useless. In tic-tac-toe a player might have 10% chance or better of making a good move, simply because there are less than ten choices available on any given turn. However, such a random player in a Simple War Game would always lose and be of little value. Therefore, we hand crafted a baseline agent, and named it AGENT1.

To improve WAR's performance, it was necessary to adjust the weights of each advisor for each specific game. We determine the best weights for each sample game thru self-play using a genetic algorithm. We chose AGENT1 as a seed to the Genetic Algorithm, rather than purely random numbers, with the notion that the future generations would benefit from starting at a good location.

Four sample games, two of which were presented in section 4, SIMPLE, KING, CHESSLIKE, and TANK, were used to test and train the agents for the WAR program. One best agent, called a champion, was created for each of these sample games.

5.1 The WAR Genetic Algorithm

Genetic algorithms are adaptive methods that have been used successfully to solve search and optimization problems. Genetic algorithms are based on the principles of natural selection and "survival of the fittest" [8].

The *population size* was set to 16 because it made implementation somewhat easier and the genetic algorithm ran significantly faster than with a population size of 32. The initial population was seeded with two individuals, seven individuals were generated randomly and the remaining seven were offspring of the first two.

The *fitness* of each individual was determined by competition with the other members of the population. There are four single elimination rounds in each generation. Individuals that lose during the first round are removed from the population. The in-

dividual that wins the most games during the current generation is guaranteed to breed. All individuals that won during the first round will be included in the next generation. Individuals are also chosen randomly from the first round winners to breed the next generation. This tournament scheme is similar to that presented in [9].

Genetic algorithms use a method called *crossover* to combine individuals. There are many other crossover operators that can be used, however there is no proven best method [10] so we chose the 1-point crossover.

Mutation randomly alters each characteristic with a small probability. It is applied to each child after crossover. Mutation is typically viewed as a background operator with a very small chance of occurrence. The probability is usually set to less than 1%. However, it has been suggested that mutation is more important than originally thought [10]. We set the mutation rate so that each weight has a 20% chance of being changed by a random integer amount between -20 and +20.

For each sample game, the genetic algorithm was run 4 times for 100 generations, to come up with four separate playing strategies (agents). The four agents were pitted against each other and the winner was designated the champion of that particular game, resulting in four champion agents. These champion agents are called CHAMP1, CHAMP2, CHAMP3 and CHAMP4.

5.2 Results

To insure that the resulting four champion agents were indeed improvements within their specific games, they were pitted against each other and the baseline agent, AGENT1, in a “finals” round. Each Champion agent played two games against every other player, alternating sides between games. For example, CHAMP1 played two games of SIMPLE against AGENT1, two games against CHAMP2, two against CHAMP3 and two against CHAMP4. The results are shown in Figure 5.2.

Game	AGENT1	CHAMP1	CHAMP2	CHAMP3	CHAMP4
SIMPLE	0	8	0	0	0
KING	0	0	8	0	0
CHESSLIKE	0	0	0	8	0
TANK	0	0	0	0	8

Figure 5.2 – Total wins by each agent in the finals.

The results suggest that the genetic algorithm performed as expected, generating agents that played their respective game better.

Since the class of Military War Games was designed to emulate war in general, it is not surprising that concepts of warfare are applicable to the strategy in Simple War Games. For example, the baseline player AGENT1 simply attacks with great fervor much like the barbarian armies of old. This strategy works best when you have overwhelming forces. It also works best when your units are all the same type. Since each piece determines its own best move, without regard to the other pieces, AGENT1 cavalry tend to zoom out to the front, do some damage, and get eliminated before the infantry can catch up.

6 Related Work

General game playing has not been explored in great depth. But there are a fewThere have been few is chapter discusses other work in the area of general game playing and compares it to ours.

SAL Michael Gherrity created a program called SAL [11] that has the ability to learn any two-player, deterministic, perfect information, zero-sum game. It does not learn from the rules, but by trial and error from actually playing (and being given valid moves at each turn).

HOYLE Susan Epstein created a program called HOYLE [12] that can learn to play two-person, perfect information, finite board games. It uses a mixture of generic and specific advisors and weighted for each particular game to improve its performance. HOYLE has 23 game independent advisors, but requires a certain amount of hand crafting (i.e. programmer intervention) for each game.

METAGAMER Barney Pell created a game definition grammar, a game generator, a grammar interpreter, and the game-playing program METAGAMER [13]. METAGAMER plays a class of games called symmetric chess-like games (a subset of two person, perfect information, deterministic, zero-sum games). The class includes the games of chess, tic-tac-toe, checkers, Chinese-chess and many others.

MORPH II Robert Levinson developed MORPH II [14], a domain independent machine learning system and problem solver. MORPH II has a low reliance on search; just 2-ply is average. Games are presented to the system using a graph-theoretic representation scheme. MORPH II abstracts its own features and patterns for each game.

7 Summary

We defined a class of games, called Simple War Games. This class contains both deterministic and non-deterministic games. The definition of Simple War Games is one of our original contributions. We also defined many games in this domain.

Further we developed an algorithm for determining moves for Simple War Games. This algorithm is based on the biological behavior of ants.

Finally we developed the program WAR that is able to play Simple War Games. The program had a learning component that used a genetic algorithm. While genetic algorithms have been used in learning game playing, we tailored ours to Simple War Games.

7.1 Future Research

It would prove useful to extend the Simple War Game syntax to include more games, such as Metagames [13]. All the pieces in Simple War Games move in exactly the same way. The distance they can move is adjustable, but the way they move is not. A method of describing how the pieces (i.e. in a straight line or hopping), different methods of capture (e.g. hopping over or landing on) and piece promotion would be necessary. Having a way to describe popular games would enable a direct comparison of the WAR program with other game playing programs.

There might be unexpected side benefits to explicitly describing games. Such a tool may ease the burden of identifying features for the evaluation functions. Determining the important features of a game is of primary importance when creating an evaluation function. Feature selection is also critical information in Machine Learning; it is necessary to establish what information should be learned and what is noise.

General game classification is area that would benefit from further investigation. Research on one game would be more readily applied to other games if done in a common context. Classifying games so that useful methods developed for one game can be easily applied to another would be most helpful.

References

1. Johannes Fürnkranz. Bibliography on Machine Learning in Strategic Game Playing. [web page] March 2000; <URL:<http://www.ai.univie.ac.at/~juffi/lig/lig-bib.html>>. [Accessed November 1, 2004].
2. Stuart Russell and Peter Norvig. *Artificial Intelligence, A Modern Approach*. Prentice Hall Inc., New Jersey, 1995.
3. IBM Coporation. Kasparov vs Deep Blue Press Material 1997 [URL:http://www.research.ibm.com/deepblue/press/html/g.6.1.html](http://www.research.ibm.com/deepblue/press/html/g.6.1.html) [Accessed November 1, 2004]
4. Gerald Tesauro. [Temporal Difference Learning and TD-Gammon](#). In *Communications of the ACM, March 1995 / Vol. 38, No. 3*. 1995.
5. Burmeister, J. and Wiles, J. [An Introduction to the Computer Go Field and Associated Internet Resources](#). Technical Report 339, Department of Computer Science, University of Queensland, 1995.
6. David Kaiser. A Generic Game Playing Machine. Master's thesis, Florida International University, Miami, FL, 2000.
7. Marco Dorigo, Vittorio Maniezzo and Alberto Coloni. The Ant System: Optimization by a colony of cooperating agents. In *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, Vol.26, No.1, 1996, pp.1-13.
8. John Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge, MA, 1993
9. Gabriel J. Ferrer. [Using genetic programming to evolve board evaluation functions](#). Master's thesis, Department of Computer Science, School of Engineering and Applied Science, University of Virginia, Charlottesville, VA, 1996.
10. David Beasley, David R. Bull and Ralph R. Martin. An Overview of Genetic Algorithms: Part 2, Research Topics. In *University Computing*, 15(4) 170-181. 1993.
11. Michael Gherrity. [A Game-Learning Machine](#). PhD thesis, University of California, San Diego, CA, 1993.
12. Susan L. Epstein, Jack Gelfand, and Joanna Lesniak. [Pattern-based learning and spatially-oriented concept formation in a multi-agent, decision-making expert](#). *Computational Intelligence*, 12(1):199-221, 1996.
13. Barney Pell. [A strategic metagame player for general chess-like games](#). In *Computational Intelligence, Volume 11, Number 4*, 1995.
14. Robert Levinson. [MORPH II: A universal agent: Progress report and proposal](#). Technical Report Number UCSC-CRL-94-22, Department of Computer Science, University of California, Santa Cruz, Jack Baskin School of Engineering. June 1994.