

REINFORCEMENT LEARNING: SELF LEARNING IN GAMES USING TEMPORAL DIFFERENCE LEARNING AND CO-EVOLUTIONARY LEARNING

*A Major Thesis & Project Report submitted in the partial fulfillment of the
requirements for award of the degree of*

BACHELOR OF ENGINEERING

In

COMPUTER ENGINEERING

Submitted By:

AMIT GUPTA 213/CO/08

AYUSH JAIN 237/CO/08

Under the guidance of

PROF. SATISH CHAND



**DEPARTMENT OF COMPUTER ENGINEERING
NETAJI SUBHAS INSTITUTE OF TECHNOLOGY**

UNIVERSITY OF DELHI

DELHI- 110078

Academic Session: 2008-2012

Certificate

This is to certify that B.E. Project Thesis titled “Reinforcement Learning: Self Learning in Games using Temporal Difference Learning and Co-evolutionary Learning” has been carried out under my supervision by Ayush Jain (237/CO/08) and Amit Gupta (213/CO/08). As per my knowledge this project has not been submitted elsewhere for award of a degree.

Prof. Satish Chand,

Professor, COE Department,

Netaji Subhas Institute of Technology

Acknowledgement

We owe a debt of deepest gratitude to our thesis supervisor, Prof. Satish Chand, Department of Computer Engineering, for his guidance, support, motivation and encouragement throughout the period this work was carried out. His readiness for consultation at all times, insights and concern have been invaluable. We also thank the other faculty and staff members of our department for their invaluable help and guidance

We would like to use this section to thank all those who helped us track down and obtain the many technical papers which have made this project possible.

Ayush Jain

237/CO/08

Amit Gupta

213/CO/08

Abstract

Reinforcement learning is the problem faced by an agent that learns behaviour through trial-and-error interactions with a dynamic environment. In this project, we apply reinforcement learning to games to develop AI players without human supervision as it trains itself by playing against itself and learning from the outcomes of the games. The main approaches used to develop self learning in games use Temporal Difference Learning or Co-evolutionary Learning. Although the temporal difference and co-evolutionary algorithm show good results in board evaluation optimization, the hybrid of both approaches is rarely addressed in the literature. This project aims to develop such self learning programs using Co-evolutionary Temporal Difference Learning (CTDL), a novel way of hybridizing Co-evolutionary Learning with Temporal Difference Learning that works by interlacing a population competitive Co-evolution with Temporal Difference Learning. The Co-evolutionary part of the algorithm provides for exploration of the solution space, while the temporal difference learning performs its exploitation by local search. We apply CTDL to the game of Tic Tac Toe and Othello, using weighted piece counter for representing players' strategies. The results of an extensive computational experiment demonstrate CTDL's superiority when compared to Co-evolution and reinforcement learning alone.

Temporal Difference Learning (TDL) is a variant of reinforcement learning, where the playing agent aims at maximizing a delayed reward, and is typically trained by some form of gradient based method. While Co-Evolutionary Learning (CEL) follows the competitive evaluation scheme and typically starts with generating a random initial population of player individuals. Individuals play games with each other, and the outcomes of these confrontations determine their fitness values. The best performing strategies are selected, undergo genetic

modifications such as mutation or crossover, and these resultant strategies replace some of former strategies.

The basic idea is to create an Artificial Neural Network, a densely interconnected massive network of simple nodes representing different states of the game (neuron) with weights to represent favourability of that particular state. Learning agent observes an input neuron, selects an appropriate action and sends an output signal to environment which creates a link to new neuron in the network, and then it receives a scalar feedback signal from the environment. The goal of learning is to generate the optimal actions leading to maximal reward.

A game of Othello has 10^{27} states and hence it is not possible to maintain a neuron corresponding to each state in the neural network since memory is limited. Thus, the Co-evolutionary algorithm decides which states to store as 'Survival of the Fittest' is pursued, i.e. States which lead to a favourable result are kept while the states below a certain threshold are eliminated.

Learning uses the feedback to modify the weights and network connectivity by addition and deletion of neurons and links. At the initial stages of development, random action is selected by the learning agent. The desirability of the states is adjusted using TD- λ techniques which impart different credibility to different game states according to their contribution in the final win or loss. Learning will be divided into gradual phases corresponding to a Learning Factor. A low learning factor in initial stages of learning would correspond to a low feedback signal and vice versa.

Table of Contents

Certificate	2
Acknowledgement	3
Abstract	4
Table of Figures	8
Chapter 1: Introduction	9
1.1 Reinforcement Learning	9
1.2 Importance of Reinforcement Learning.....	11
1.2 Implementing Board games	11
Chapter 2: Literature Survey	13
2.1 Temporal Difference Learning and TD-Gammon By Gerald Tesauro (Article).....	13
2.2 Reinforcement learning in board games (Research Paper)	14
Chapter 3: System Design	15
3.1 Choice of Game	15
3.1.1 Tic-Tac-Toe	16
3.1.2 Othello	16
3.2 Methodology	17
3.2.1 Artificial Neural Network	18
3.2.2 Learning	20
Chapter 4: Implementation	24
4.1 Programming Language	24
4.2 Role of Temporal Difference Learning	24
4.3 Role of Co-Evolutionary Learning.....	25
4.4 Artificial Neural Network Implementation.....	26
4.5 Populating Database	28
4.5.1 Database play	28
4.5.2 Random Play	28
4.5.3 Fixed opponent	29
4.5.4 Self-play	30
4.6 Class Overview of Tic-Tac-Toe.....	30
4.6.1 GameTicTacToe.cs.....	31
4.6.2 Simulator.cs.....	31

4.6.3 Program.cs	31
4.6.4 HumanPlayer.cs	31
4.6.5 Player2.cs	32
4.6.6 Player3.cs	32
4.7 Class Overview of Othello.....	32
4.7.1 GameOthello.cs	33
4.7.2 Emulator.cs	34
4.7.3 Program.cs	34
4.7.4 Player1.cs	34
4.7.5 Player2.cs	34
4.7.6 Player3.cs	34
4.7.7 Enum GridEntry	34
4.7.8 Enum PlayerType	35
Chapter 5: Results and Future Improvements	36
5.1 Results	36
5.1.1 Tic-Tac-Toe	36
5.1.2 Othello	36
5.2 Future Developments	38
References	40
Appendix A: Strategic elements of Othello.....	41
A.1 Openings	41
A.2 Corners.....	41
A.3 Mobility	42
A.4 Edges	42
A.5 Parity.....	43
A.6 Endgame	43
Appendix B: Minimax Algorithm with α , β Pruning	44
B.1 Pseudocode for α , β Pruning	45

Table of Figures

Figure 1.1 Basic Paradigm of Reinforcement Learning.....	9
Figure 3.1 Game Tree for Tic-Tac-Toe.....	16
Figure 3.2 Initial Othello board Configuration.....	17
Figure 3.3 Number of States in different Board Games.....	18
Figure 3.4 Neural Network.....	19
Figure 4.1 Database of Tic-Tac-Toe	26
Figure 4.2 Database of Othello	27
Figure 4.3 Class Diagram of Tic-Tac-Toe	31
Figure 4.4 Class Diagram of Othello	32
Figure 4.5 Game of Othello in progress	33
Figure B.1 Minimax tree with apha beta pruning	45

Chapter 1: Introduction

1.1 Reinforcement Learning

The easiest way to understand reinforcement learning is by comparing it to supervised learning. In supervised learning an agent is taught how to respond to given situation, in reinforcement learning the agent is not taught how to behave rather it has a free choice in how to behave. However once it has taken its actions it is then told if its actions were good or bad (this is called the reward - normally a positive reward indicates good behavior and a negative reward bad behavior) and has to learn from this how to behave in the future. However very rarely is an agent told directly after each action whether the action is good or bad. It is more usual for the agent to take several actions before receiving a reward. This creates what is known as the temporal credit assignment problem that is if our agent takes a series of actions before getting the award how can we take that award and calculate which of the actions contributed the most towards getting that award.

The basic paradigm of reinforcement learning is as follows: The learning agent observes an input state, it produces an output signal, and then it receives a scalar feedback signal from the environment. The goal of learning is to generate the optimal actions leading to maximal reward. Fig. 1.1 demonstrates this paradigm.

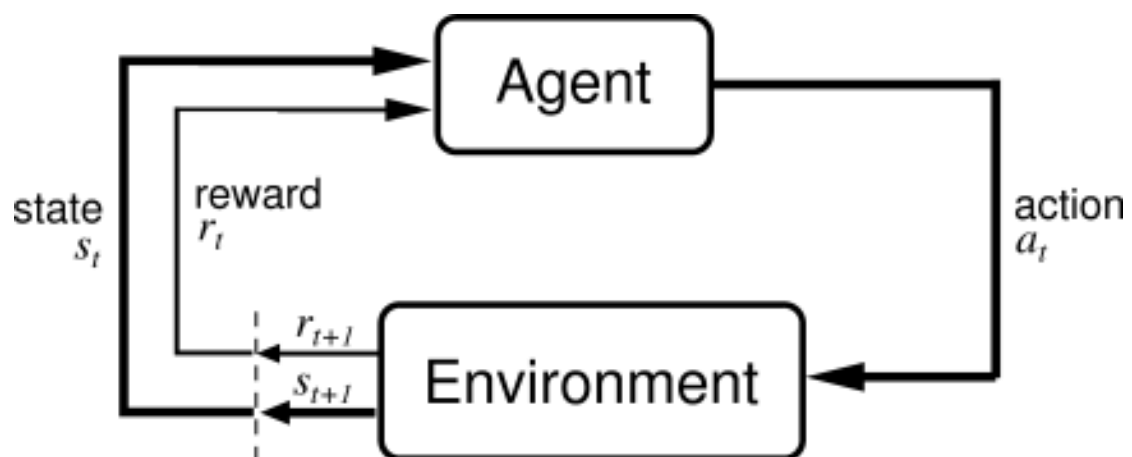


Figure 1.1 Basic Paradigm of Reinforcement Learning

This problem is obviously one that humans have to deal with when learning to play board games. Although adults learn using reasoning and analysis, young children learn differently. If one considers a young child who tries to learn a board game, the child attempts to learn in much the same way as we want our agent to learn. The child plays the board game and rather than deducing that they have won because action X forced the opponent to make action Y, they learn that action X is correlated with winning and because of that they start to take that action more often. However if it turns out that using action X results in them losing games they stop using it. This style of learning is not just limited to board games but extends many activities that children have to learn (i.e. walking, sports, etc).

The techniques we intend to concentrate on, one called $TD(\lambda)$ (which belongs to the Temporal Difference class of methods for tackling the temporal credit assignment problem) and Co-evolutionary Learning.

Temporal Difference Learning is a canonical variant of reinforcement learning, where the playing agent aims at maximizing a delayed reward, and is typically trained by some form of gradient based method.

Co-Evolutionary Learning (CEL) follows the competitive evaluation scheme and typically starts with generating a random initial population of player individuals. Individuals play games with each other, and the outcomes of these confrontations determine their fitness values. The best performing strategies are selected; undergo genetic modifications such as mutation or crossover, and these resultant strategies replace some of former strategies.

The essential difference between TDL and CEL is that TDL guides the learning using the whole course of the game while CEL uses only the final game outcome. As a result, TDL in general learns faster than CEL. However, for some domains a properly tuned CEL can eventually find strategies that outperform those generated by TDL.

1.2 Importance of Reinforcement Learning

The reinforcement learning has held great intuitive appeal and has attracted considerable interest for many years because of the notion of the learner being able to learn on its own, without the aid of an intelligent "teacher," from its own experience at attempting to perform a task.

Reinforcement Learning, due to its generality, can be applied to many disciplines, such as operation research, information theory, genetic algorithms, simulation based optimization, economics and game theory. In the operation research reinforcement learning methods are applied to field called Approximate Dynamic Programming. The problem has been studied in the theory of optimal control, though most studies there are concerned with existence of optimal solutions and their characterization, and not with the learning or approximation aspects. In economics and game theory, reinforcement learning may be used to explain how equilibrium may arise under bounded rationality.

Two components make reinforcement learning powerful: The use of samples to optimize performance and the use of function approximation to deal with large environments.

Reinforcement learning can be used in large environments in any of the following situations:

- A model of the environment is known, but an analytic solution is not available
- Only a simulation model of the environment is given (the subject of simulation-based optimization). The only way to collect information about the environment is by interacting with it.

1.2 Implementing Board games

Board games form an integral part of civilization representing an application of abstract thought by the masses; the ability to play board games well has long been regarded as a sign of intelligence and learning. Generations of humans have taken up the challenge of board

games across all cultures, creeds and times, the game of chess is fourteen hundred years old, backgammon two thousand years but even that pales to the three and half millennia that tic-tac-toe has been with us. So it is very natural to ask if computers can equal us in this very human pastime.

However there is a far more practical reason as well. Board games have defined starting and ending stages with clear winner and loser. This simplifies implementation of machine learning and temporal difference reward calculation.

Chapter 2: Literature Survey

Different research papers and articles referred by us are mentioned below:

2.1 Temporal Difference Learning and TD-Gammon By Gerald

Tesauro (Article)

This article was originally published in *Communications of the ACM*, March 1995 / Vol. 38, No. 3. Copyright © 1995 by the Association for Computing Machinery.

Ever since the days of Shannon's proposal for a chess-playing algorithm and Samuel's checkers learning program the domain of complex board games such as Go, chess, checkers, Othello, and backgammon has been widely regarded as an ideal testing ground for exploring a variety of concepts and approaches in artificial intelligence and machine learning. Such board games offer the challenge of tremendous complexity and sophistication required to play at expert level. At the same time, the problem inputs and performance measures are clear-cut and well defined, and the game environment is readily automated in that it is easy to simulate the board, the rules of legal play, and the rules regarding when the game is over and determining the outcome.

This article presents a game-learning program called TD-Gammon. TD-Gammon is a neural network that trains itself to be an evaluation function for the game of backgammon by playing against itself and learning from the outcome. Although TD-Gammon has greatly surpassed all previous computer programs in its ability to play backgammon that was not why it was developed. Rather, its purpose was to explore some exciting new ideas and approaches to traditional problems in the field of reinforcement learning.

2.2 Reinforcement learning in board games (Research Paper)

Author: Imran Ghory

Date: May 4, 2004

This project investigates the application of the $TD(\lambda)$ reinforcement learning algorithm and neural networks to the problem of producing an agent that can play board games. It provides a survey of the progress that has been made in this area over the last decade and extends this by suggesting some new possibilities for improvements (based upon theoretical and past empirical evidence). This includes the identification and formalization (for the first time) of key game properties that are important for TD-Learning and a discussion of different methods of generate training data. Also included is the development of a TD-learning game system (including a game-independent benchmarking engine) which is capable of learning any zero-sum two-player board game. The primary purpose of the development of this system is to allow potential improvements of the system to be tested and compared in a standardized fashion. Experiments have been conduct with this system using the games Tic-Tac-Toe and Connect 4 to examine a number of different potential improvements.

Chapter 3: System Design

Reinforcement Learning has attracted considerable interest for many years because of the notion of the learner being able to learn on its own, without the aid of an intelligent "teacher" signal, from its own experience at attempting to perform a task. In contrast, in the more commonly employed paradigm of supervised learning, learner is explicitly told what the correct output is for every input pattern. We intend to explore the domain of reinforcement learning by developing game playing algorithm which has no prior knowledge of any tactics or strategies and is capable of consistently beating human players and other artificial intelligences.

Board games offer the challenge of tremendous complexity and sophistication required to play at expert level. At the same time, the problem inputs and performance measures are clear-cut and well defined, and the game environment is readily automated in that it is easy to simulate the board, the rules of legal play, and the rules regarding when the game is over and determining the outcome.

3.1 Choice of Game

We selected Tic-Tac-Toe and Othello for developing game playing algorithms although the system is game independent given fulfillment of hardware constraints. Tic-Tac-Toe was selected for its simplicity and limited number of states possible. It has a low Divergence Rate, i.e. it only adds a single piece to board at a time with no effect on pieces already on the board. The database created is small and hence can be tested manually. It acted as the testing grounds for further development. Othello has medium level of number of states and has average branching factor. It has the highest Divergence Rate of all known board games. Further both the games are symmetric in nature and hence a transformation and board inversion is possible.

3.1.1 Tic-Tac-Toe

Tic-Tac-Toe is a classical board game (dating from the second millennium BC) which involves a 3x3 board on which alternate players place their pieces in an attempt to get three pieces in a line. It has a state-space complexity of around 5000 and can be trivially solved using a minmax search of depth nine. Players soon discover that best play from both parties leads to a draw. Tic-Tac-Toe is often used to learn a branch of artificial intelligence that deals with the searching of game trees. Figure 3.1 illustrates a game tree for Tic-Tac-Toe.

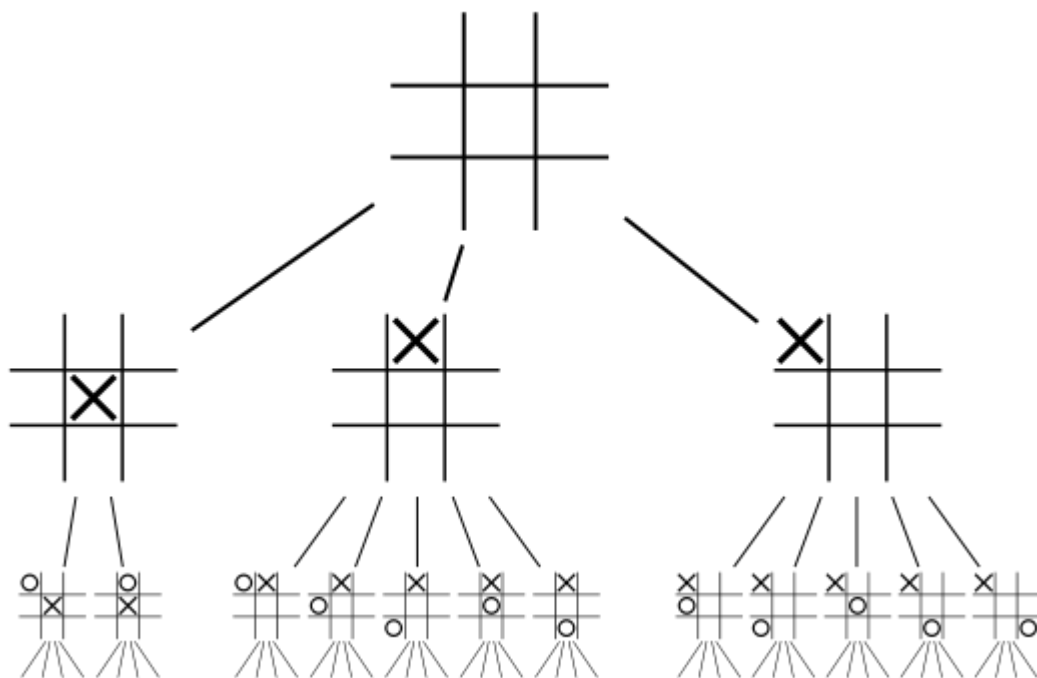


Figure 3.1 Game Tree for Tic-Tac-Toe

3.1.2 Othello

Othello is a two-player game which has existed in its current incarnation since the 1970s, and was popularized in Japan. The game is played on an 8 X 8 board. All playing pieces are identical and reversible, with one side black, and the other white. Pieces are usually referred to as black or white, according to the side showing face-up. The starting board configuration is shown in Figure 3.2, below.

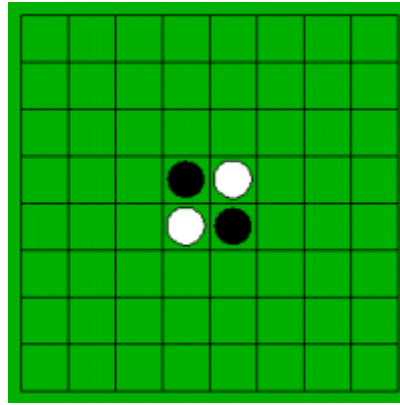


Figure 3.2 Initial Othello board Configuration

On a player's go, they must place one piece of their colour on the board, in a square which encloses one or more pieces of the opponent's colour between one or more of their own, horizontally, vertically, or diagonally. All these enclosed pieces are then 'captured', whereby they are turned over to show the opposite colour. If a player cannot make a legal move, they simply miss their go. The game ends when there are no legal moves available for either player. The winner of the game is the player with the most pieces of their colour on the board at the end of the game.

3.2 Methodology

The basis technique applied can be broken down into two parts: Artificial Neural Network and Learning. For most board games it is not possible to store all possible board configurations, hence neural networks are used instead. Another advantage of using neural networks is that they allow generalization, i.e. if a board position has never been seen before they can still evaluate it on the basis of knowledge learnt when the network was trained on similar board positions.

For Learning we apply a mixture of Temporal Difference and Co-evolutionary Approach. Temporal Difference (λ) is used to reward different stages traversed in between. Storing states is the most challenging part in this project. A Game of Chess has approximately 10^{47} states, a

game of Othello has 10^{27} states, and a game of backgammon has 10^{20} states. Refer to Figure 3.3 to see number of states in different board games.

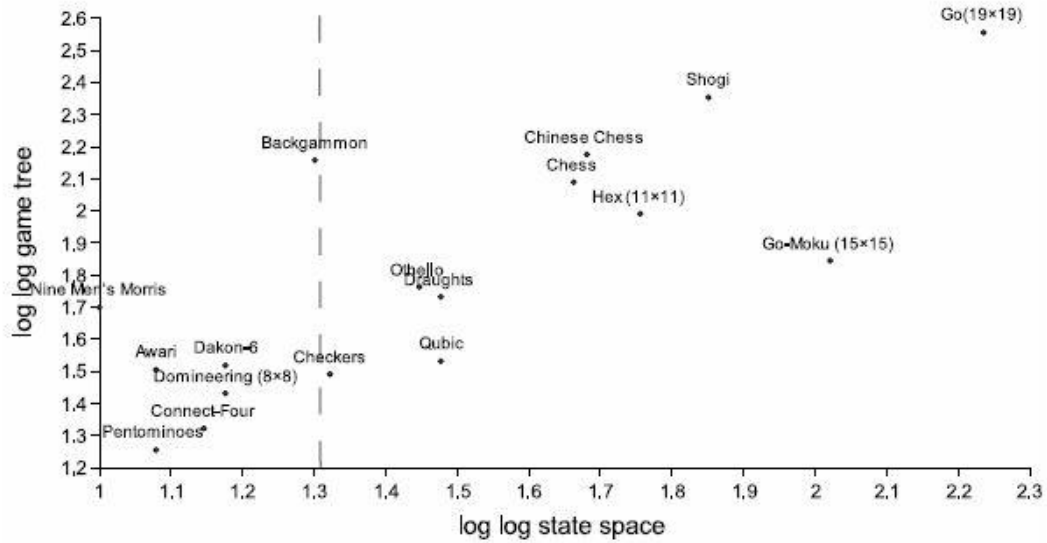


Figure 3.3 Number of States in different Board Games

It is not possible to store information regarding all the states since memory is limited. Thus, the genetic algorithms come into play, and ‘Survival of the Fittest’ is pursued, i.e. States which lead to a favorable result are kept while the states below a certain threshold are eliminated.

3.2.1 Artificial Neural Network

The basic idea is to create an Artificial Neural Network, a densely interconnected network of nodes representing different states of the game with weights to represent favorability of that state.

Our neural network saves different states of the board configuration. The network has a number of input nodes (in our case these are the raw encoding of the board position) and an output node(s) (our evaluation of our board position). A two-layered network will also have a hidden layer which will be described later. All of the nodes on the First layer are connected by

a weighted link to all of the nodes on the second layer. For a diagram of a network see Figure 3.3.

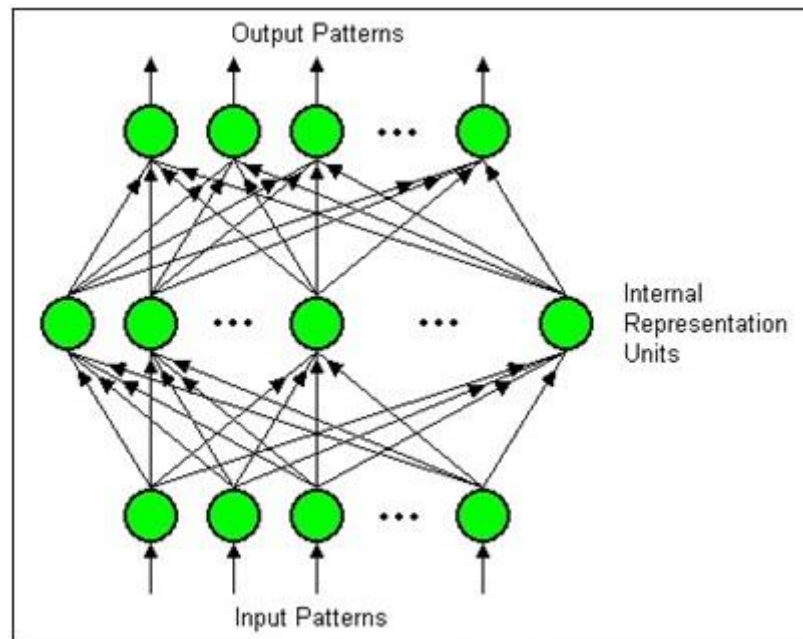


Figure 3.4 Neural Network

In the one-layer network the output is simply a weighted sum of the inputs. However this can only be used to represent a linear function of the input nodes, hence the need for a two-layer network. The two-layer network is the same as connecting two one-layer networks together with the output of the first network being the input for the second network, with one important difference - the hidden nodes. The input is passed through a non-linear function before being outputted.

At the heart of our implementation is a neural network that is organized in standard multilayer perception (MLP) architecture. The MLP architecture, also used in the widely known backpropagation algorithm for supervised training may be conveniently thought of as a generic nonlinear function approximator. Its output is computed by a feed-forward flow of activation from the input nodes to the output nodes, passing through one or more layers of internal nodes called "hidden" nodes. Each of the connections in the network is parameterized

by a real valued "weight." Each of the nodes in the network outputs a real number equal to a weighted linear sum of inputs feeding into it, followed by a nonlinear sigmoidal "squashing" operation that maps the total summed input into the unit interval. The nonlinearity of the squashing function enables the neural network to compute nonlinear functions of its input, and the precise function implemented depends on the values of the weights. In both theory and in practice, MLPs have been found to have extremely robust function approximation capabilities. In fact, theorists have proven that, given sufficient hidden units, the MLP architecture is capable of approximating any nonlinear function to arbitrary accuracy.

3.2.2 Learning

The process of "learning" consists of applying a formula for changing the weights so that the desirability of different neurons of the neural network are more closely approximated to actual values. For large networks with thousands of connections, it is generally not feasible to find the globally optimal set of weights that gives the best possible approximation. Nevertheless, there are many reasonably efficient learning rules, such as backpropagation, that can find a locally optimal set of weight values. Such locally optimal solutions are often satisfactory for many classes of real-world applications.

The learning process in our project is as follows: the system observes a sequence of board positions starting from initial configuration of board to some terminal board configuration with either of the player winning. Input to the neural network is the current state of the board. The system examines the current state and tries to determine the best move possible by using different tools like: Heuristics, Minimax algorithms and Neural Network storing the moves with their desirability. Based on these tools next best possible move is determined.

The process of learning i.e. developing the neural network is tricky. In the initial stages of learning, system needs to explore as much state space as possible. To implement this we need a probabilistic function which returns a random move from a set of good moves, based on their desirability. Learning Factor, a factor determining the change in weights of different

neurons based on different stages of learning, has to be controlled. For initial self learning it is kept low as moves chosen by the system are random with negligible knowledge, and is gradually increased with rising knowledge. Learning Factor is kept high when system is playing against a human expert as an expert.

Our system is using a hybrid of Temporal Difference (λ) and Co-Evolutionary Learning. Almost all reinforcement learning systems are based on either of these techniques, a hybrid has never been tried in our knowledge.

3.2.2.1 Temporal Difference Learning

Temporal difference (TD) learning is a prediction method. It has been mostly used for solving the reinforcement learning problem. TD resembles Monte Carlo method because it learns by sampling the environment according to some policy. TD is related to dynamic programming techniques because it approximates its current estimate based on previously learned estimates. The TD learning algorithm is related to the temporal difference model of animal learning.

As a prediction method, TD learning takes into account the fact that subsequent predictions are often correlated in some sense. In standard supervised predictive learning, one learns only from actually observed values: A prediction is made, and when the observation is available, the prediction is adjusted to better match the observation. The core idea of TD learning is that we adjust predictions to match other, more accurate, predictions about the future. This procedure is a form of bootstrapping, as illustrated with the following example:

Suppose you wish to predict the weather for Saturday, and you have some model that predicts Saturday's weather, given the weather of each day in the week. In the standard case, you would wait until Saturday and then adjust all your models. However, when it is, for example, Friday, you should have a pretty good idea of what the weather would be on Saturday.

Mathematically speaking, both in a standard and a TD approach, we would try to optimize some cost function, related to the error in our predictions of the expectation of some random variable, $E[z]$. However, while in the standard approach we in some sense assume $E[z] = z$

(the actual observed value), in the TD approach we use a model. For the particular case of reinforcement learning, which is the major application of TD methods, z is the total return and $E[z]$ is given by the Bellman equation of the return.

3.2.2.2 Co-Evolutionary Learning

In biology, co-evolution is "the change of a biological object triggered by the change of a related object." Co-Evolution is primarily a biological concept, but has been applied by analogy to fields such as computer science and astronomy.

Co-Evolutionary algorithms are variants of evolutionary computation where individual's fitness depends on other individuals. The evaluation of an individual takes place in the context of at least one other individual, and may be of cooperative or competitive nature. In the former case, individuals share the fitness they have jointly elaborated, whereas in the latter one, a gain for one individual means a loss for the other.

Co-Evolutionary Learning (CEL) follows the competitive evaluation. A Game of Chess has approximately 10^{47} states, a game of Othello has 10^{27} states, and a game of backgammon has 10^{20} states. It is not possible to store and maintain information regarding all the states in Neural Network since memory is limited. Thus, the genetic algorithms come into play, and 'Survival of the Fittest' is pursued, i.e. States which lead to a favorable result are kept while the states below a certain threshold are eliminated.

Co-Evolutionary Learning typically starts with generating a random initial population of neurons in Neural Networks. Individual neurons compete with each other, and the outcomes of these confrontations determine their fitness values. Neurons with low desirability are removed from the Neural Network.

3.2.2.3 Hybrid of Temporal Difference and Co-Evolutionary Learning

The essential difference between Temporal Difference Learning (TDL) and Co-Evolutionary Learning (CEL) is that TDL guides the learning using the whole course of the game while CEL uses only the final game outcome. As a result, TDL in general learns faster than CEL.

However, for some domains a properly tuned CEL can eventually find strategies that outperform those generated by TDL.

The past results of learning strategies for Othello and small-board Go demonstrate that TDL and CEL exhibit complementary features. TDL learns much faster and converges within several hundreds of games, but then sticks, and, no matter how many games it plays; eventually it fails to produce a well-performing strategy. CEL progresses slower, but, if properly tuned, eventually outperforms TDL. Therefore, it sounds reasonable to combine these approaches into a hybrid algorithm exploiting advantages revealed by each method.

Chapter 4: Implementation

4.1 Programming Language

C# was the language chosen for this project—its strong object oriented nature does lend itself to the construction of well designed programs. C# is a multi-paradigm programming language encompassing strong typing, imperative, declarative, functional, generic, object-oriented, and component-oriented programming disciplines. We used Microsoft’s Visual Studio 2010 as programming environment. Visual Studio includes a code editor supporting IntelliSense as well as code refactoring. The integrated debugger works both as a source-level debugger and a machine-level debugger. Other built-in tools include a forms designer for building GUI applications, class designer, and database schema designer.

4.2 Role of Temporal Difference Learning

We are using Temporal Difference Learning and Co-Evolutionary Approach in our system. Temporal Difference (λ) is used to solve the temporal credit problem. Board configuration at every stage of the game is examined. Based on heuristic score, minimax tree and neural network, a move is guessed. Reward signal to the neural network can only be generated at the end of the game when a clean winner has been determined. Delay in reward (i.e. after a long sequence of moves) leads to “temporal credit problem”.

Temporal Difference (λ) provides solution to this problem. Every state visited by the system is maintained in the neural network. Each neuron saves the board configuration, number of times it was visited, number of times that state led to a win, number of times that state led to lost game, and the desirability of that state. Desirability of a state is used to determine best move possible while examining the neural network.

Temporal Difference (λ) is used to calculate the error signal which modifies the desirability of every state visited. This error signal is base on the Learning Factor, move number and the score of the winner as seen in the Equation 4.1.

$$Error = Learning\ Factor \times \frac{Winning\ Score}{No.\ of\ states - Winning\ Score + 1} \times \frac{Move\ Number}{5}$$

Equation 4.1

Here, Learning Factor controls the error signal. Learning Factor is kept low in initial stages of self learning and is gradually increased as knowledge of the neural network increases. It is kept high when learning is done against an expert human. Winning Score is the final score of the winner, for example in case of Othello it is the number of pieces of the winner finally has on the board. Move Number controls the error signal for different states in a single game. Contribution of states at the end of a game must be higher than states at beginning.

4.3 Role of Co-Evolutionary Learning

It is not possible to store and maintain information regarding all the states in Neural Network since memory is limited. Storing states with low desirability has no effect on the game playing algorithm. After the neural network has developed a good level of knowledge, we select the best move with the maximum desirability and maximum score from minmax algorithm. Thus low desirability moves will never be used as viable candidates. Eliminating these low desirability moves will have no adverse effect. Benefit of elimination is twofold. Firstly, it limits the size of the database thereby satisfying the memory constraints. Secondly, searching and updation of states in database becomes fast.

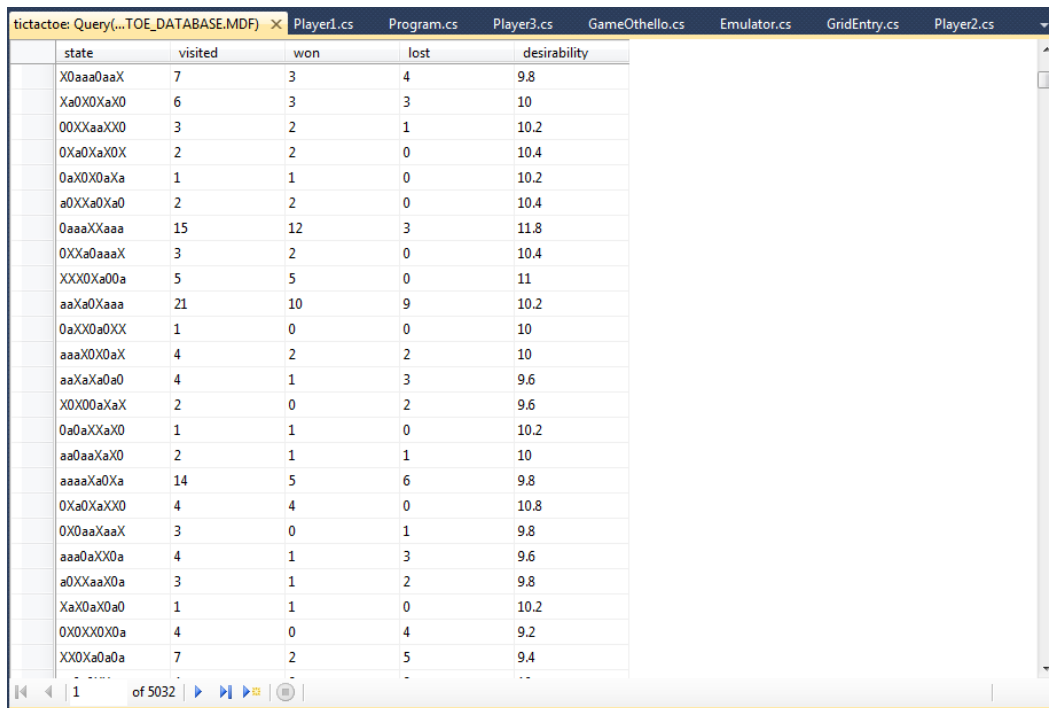
The genetic algorithms come into play, and ‘Survival of the Fittest’ is pursued, i.e. States which lead to a favorable result are kept while the states below a certain threshold are eliminated. Co-Evolutionary Learning typically starts with generating a random initial

population of neurons in Neural Networks. Individual neurons compete with each other, and the outcomes of these confrontations determine their fitness values. Desirability of states which lead to winning will slowly increase while that of states leading to losing will decrement. Neurons with low desirability are removed from the Neural Network.

4.4 Artificial Neural Network Implementation

Artificial Neural Networks are also similar to the biological neural networks in the sense that functions are performed collectively and in parallel by the units, rather than there being a clear delineation of subtasks to which various units are assigned (see also connectionism). Currently, the term Artificial Neural Network (ANN) tends to refer mostly to neural network models employed in statistics, cognitive psychology and artificial intelligence.

We implemented Artificial Neural Network using database. Figure 4.1 illustrates the database for Tic-Tac-Toe and Figure 4.2 that of Othello.



state	visited	won	lost	desirability
X0aaa0aaX	7	3	4	9.8
Xa0X0XaX0	6	3	3	10
00XXaaXX0	3	2	1	10.2
0Xa0XaX0X	2	2	0	10.4
0aX0X0aXa	1	1	0	10.2
a0XXa0Xa0	2	2	0	10.4
0aaaXXaaa	15	12	3	11.8
0XXa0aaaX	3	2	0	10.4
XXX0Xa00a	5	5	0	11
aaXa0Xaaa	21	10	9	10.2
0aX0a00XX	1	0	0	10
aaaX0X0aX	4	2	2	10
aaXaXa0a0	4	1	3	9.6
X0X0aXaX	2	0	2	9.6
0a0aXXaX0	1	1	0	10.2
aa0aaXaX0	2	1	1	10
aaaaXa0Xa	14	5	6	9.8
0Xa0XaXX0	4	4	0	10.8
0X0aaXaaX	3	0	1	9.8
aaa0aXX0a	4	1	3	9.6
a0XXaaX0a	3	1	2	9.8
XaX0aX0a0	1	1	0	10.2
0X0XX0X0a	4	0	4	9.2
XX0Xa0a0a	7	2	5	9.4

Figure 4.1 Database of Tic-Tac-Toe

SELECT	board, won, lost, desirability, visited				
FROM	Table8				
	board	won	lost	desirability	visited
	BBBBBBEEEEBW...	13	0	133	13
	BBBBBBEEEEBW...	4	1	112	5
	BBBBBBEEEEWB...	10	0	114	10
	BBBBBBEEEEWB...	34	2	158	36
	BBBBBBEEEEWE...	41	0	212	41
	BBBBBBEEEEWE...	7	1	117	8
	BBBBBBEEEEBW...	4	1	109	5
	BBBBBBEEEEBB...	5	1	108	6
	BBBBBBEEEEBB...	5	0	108	5
►	BBBBBBEEEEBW...	8	1	108	9
	BBBBBBEEWW...	4	0	105	4
	BBBBBEEEEWB...	4	0	110	4
	BBBBBEEWW...	7	1	108	8
	BBBBEEEEBBB...	8	0	151	8
	BBBBEEEEBBB...	3	1	103	4
	BBBBEEEEBW...	6	1	122	7
	BBBBEEEEBB...	7	0	109	7
	BBBBEEEEBB...	9	0	113	9
	BBBBEEEEBB...	6	2	105	8
	BBBBEEEEBB...	8	0	114	8
	BBBBEEEEBB...	3	2	103	5
	BBBBEEEEBEB...	6	1	109	7
	BBBBEEEEERR	2	2	101	5

Figure 4.2 Database of Othello

Each state in database stores: the board configuration, number of times that particular configuration was visited, number of wins from that configuration, number of losses from that configuration, and the desirability of that configuration. As number of states increase, searching and modification in the database becomes complex and time consuming. To overcome this, indexing was done. Multiple tables were created, each storing board configurations with a particular number of moves. For example, there are total 64 different board positions and therefore maximum 64 moves are possible in a single game of Othello, and hence 64 different tables were created.

4.5 Populating Database

For learning we needed a collection of games which could be processed with the $TD(\lambda)$ and Co-Evolutionary algorithms. An important task was to obtain that collection of games. As like with humans, the games the agent learns from have an impact on the style and ability of play the agent will adopt. Three different ways were used to populate out databases.

4.5.1 Database play

One way for our technique to learn is from a database of (normally human played) games, one of the major advantages of which is that of speed. Learning from a database game only takes a fraction of the time taken when actually playing. This is because playing involves a large number of evaluations, when the AI is playing one side - double that if it is playing both sides (i.e. self-play).

Database play is stronger than random play but weaker than when playing against an expert. In recent years the amount of interest into using databases for training has increased, undoubtedly related to the growth in availability of large databases of human played games caused by the increasing popularity of Internet gaming servers which record games.

Other approaches involving databases include NeuroChess by Sebastian Thrun which was trained by using a unusual combination of database play and self-learning. Games were started by copying moves from a database of grandmaster games but after a set number of moves the agent stopped using the database and then continued the game using self-play.

4.5.2 Random Play

Random play methods fall into two types: firstly a method in which we produce training data by recording games played between our agent and a random move generator. The second type of random play is where we play a random-player against itself to produce a database of games which we use for training. Although only limited work has been undertaken in this area

there is some evidence that it may be beneficial for initial training to be performed using random player data before moving onto another training method.

Random Play has a great advantage of very small computation time for the next move. As no major evaluation needs to be done before selecting a move, time to play games reduces significantly in games which have a constant maximum number of moves possible (Othello and Tic-Tac-Toe). Other major benefit is randomness in selection of moves helps in exploring the state space of games. TD-Gammon by Gerald Tesauro has discovered many moves which were not considered very effective before but are now one of the best moves possible.

While this takes approximately half the time of self-play the disadvantages seem in general to outweigh any benefits. Among the disadvantage are,

- In many games there exist strategies which can beat random-play but fail against even the most novice human players.
- Random play is often far different from the way normal players (both computer and human) play the game, and valid techniques learned against random players may not serve a practical purpose.
- Dependent on the game in question, random play may not cover important basic knowledge of the game.
- In some games random play may result in a game that lasts longer (by several orders of magnitude) than a human game.

4.5.3 Fixed opponent

Fixed opponent play is where our agent plays one side and a single specific opponent (be it human or computer) plays the other side. This has the problem that our agent may learn to beat the opponent but still fail to develop a good general evaluation function. One of the reasons for this is because standard TD-learning technique were developed on the basis of an agent learning to improve their behavior in a fixed environment, so when we introduce an opponent into the environment it is just treated as part of that environment. That is, the agent

has no concept that there is a separation between environment and opponent, so the game and the opponent are treated as the same.

4.5.4 Self-play

Self-play is when the agent plays against itself to generate games to train from. This technique has proved highly-popular among researchers. One of the major reasons for this is that it doesn't need anything apart from the learning system itself to produce the training data. This has several advantages including that the agent will not be biased by the strategy of other players (that is it will be able to learn strategies that work effectively rather than strategies that other players think work effectively). Research with fixed opponents has shown that learning appears to be most effective when the opponent is roughly the same strength as the agent; with self-play the agent obviously satisfies this criterion.

However the technique also has several disadvantages which contribute to making learning take place at a slower rate. The most important of these disadvantages is that takes a long time to play a large number of games as for each game we have to perform a high number of board evaluations given by Equation 4.2.

$$game\ length \times branching\ factor^{search\ depth} \quad \text{Equation 4.2}$$

A related disadvantage is that when we start learning for the first time the agent has no knowledge and plays randomly and so it may take a substantial number of training games before the agent can correctly evaluate basic concepts in the game.

4.6 Class Overview of Tic-Tac-Toe

Figure 4.3 displays the class diagram for Tic-Tac-Toe. Each other class represents a concrete implementation of an object. Suffice to say; what follows is a brief overview of each class and its function within the project.



Figure 4.3 Class Diagram of Tic-Tac-Toe

4.6.1 GameTicTacToe.cs

It is the main class of the project which maintains the game. Different types of games may be created using different player classes. It maintains the current state of the board configuration with which player's turn it is. Move selected by the player is verified to be valid. After validation of the move board configuration is updated and turn is passed to the other player.

4.6.2 Simulator.cs

Simulator is an emulator which controls the GameTicToe Class. It creates a game using GameTicTacToe object. Game is continued till there is a winner or the game is a draw.

4.6.3 Program.cs

This class contains the main() method. It executes the Simulator Class multiple times.

4.6.4 HumanPlayer.cs

HumanPlayer Class is used to provide an interface for acquiring moves from human. This class gets the position from the human player and passes this value back to GameTicTacToe in desired format for board updation.

4.6.5 Player2.cs

This class is used while Random Play. Based on Heuristic score and Minimax Tree a combination of viable moves is created. A random move is selected from the combination with selection probability based on their Heuristic score.

4.6.6 Player3.cs

Player3 class is used in Self Play. Selection criterion is based on the desirability of the configuration from the Artificial Neural Network, Heuristic score and Minimax Tree (in case move is not found in the database). At initial stages of the learning, a random function is used to find a move from viable candidates with selection probability based on above mentioned criterion.

4.7 Class Overview of Othello

Figure 4.4 displays the class diagram for Othello. The main Class contained within the Othello project is GameOthello. Each class represents a concrete implementation of an object, with the exception of GridEntry and PlayerType, which are enumeration to represent a board entry and player respectively. Brief overview of each class and its function within the project are mentioned ahead. Figure 4.5 illustrates a game of Othello in progress.



Figure 4.4 Class Diagram of Othello

	0	1	2	3	4	5	6	7
0	-	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-
2	-	-	-	B	W	-	B	-
3	-	-	-	B	W	B	-	-
4	-	-	B	B	B	B	-	-
5	-	-	-	W	W	B	-	-
6	-	-	-	-	-	-	-	-
7	-	-	-	-	-	-	-	-

White Player's Turn
Enter your move
Enter Row No. :
3
Enter Column No. :
2

	0	1	2	3	4	5	6	7
0	-	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-
2	-	-	-	B	W	-	B	-
3	-	-	W	W	W	B	-	-
4	-	-	B	W	B	B	-	-
5	-	-	-	W	W	B	-	-
6	-	-	-	-	-	-	-	-
7	-	-	-	-	-	-	-	-

Figure 4.5 Game of Othello in progress

4.7.1 GameOthello.cs

This is the main class in this project and maintains the game being played. It maintains the board configuration, turn of players, number of pieces of each player, and heuristic score of the board. It validates the move chosen by the player, creates a list of possible moves, update the board configuration, check if current configuration of board is terminal and tells which player won. If no move is possible for the current player, turn is passed to the other player.

4.7.2 Emulator.cs

Emulator Class controls the GameOthello Class. It creates a game using its object. Game is continued till there is a winner or the game is a draw. Added functionality includes displaying board configuration after each move.

4.7.3 Program.cs

This class contains the main() function. It executes the Emulator Class multiple number of times.

4.7.4 Player1.cs

Human inputs are provided to the game using this class. This class is used to Expert Play. It asks user for inputs and checks if that particular board position is empty. If not, user is told so and asked to move again. A list of moves selected is maintained and later weights of neurons are altered accordingly.

4.7.5 Player2.cs

This class can be used to implement Random Play and Self Play. Moves can be chosen using two ways: firstly, a random move is selected from the pool of valid moves, no heuristic is applied in move selection process. Second way is using heuristic score, minimax tree and desirability of the configuration from the Artificial Neural network. Desirability of selected moves is changed according to the result of the game.

4.7.6 Player3.cs

This class implements Database Play. Moves from documented transcripts of well known games and world championship games are feed to engine. Results of the game are used to modify the Artificial neural Network.

4.7.7 Enum GridEntry

This enumerator type is used to denote different possible value of a board configuration. A position may either be Empty, or have a Black piece on it, or a White piece.

4.7.8 Enum PlayerType

This enumerator is used to define two player type. A player may either be Black or White

Chapter 5: Results and Future Improvements

5.1 Results

The games of Tic Tac Toe and Othello were successfully simulated and the results observed.

The observations are as follows

5.1.1 Tic-Tac-Toe

- The game of Tic-Tac-Toe developed basic intelligence after 500 games of self play. It was still losing games and hence required further learning games.
- After about 2500 games of self play for Tic-Tac-Toe, the AI player showed significant intelligence and didn't lose a single game against a control player which was developed using minimax and heuristic techniques.
- The database was populated and contained entries for 5032 states after 2500 games of self play and about 90% of the moves were available in the database.

5.1.2 Othello

- The following steps were followed for the learning of Othello.
 - Database Play – The AI player learned by observing about 2780 famous games taken from World Othello championships through the transcripts of the games and populated its databases.
 - Random Play – The next step was random play. Two players were produced that played according to heuristics but in a probabilistically random manner for about 2500 games. The players developed basic intelligence and was winning approximately 20-25% of the matches against Level 5 of our control opponent which was a known AI player of Othello.
 - Fixed opponent – The database was further populated by playing a player using random moves with a player employing heuristics and a 3-ply minimax

algorithm. About 7500 games were simulated between these two and the player showed continuous progress learning complex strategies and evolving. The progress has been documented in the table below.

- After 15000 odd training games, the player had a high win rate of about 86% (1718 games out of 2000) against our control opponent playing with heuristics and a 3-ply minimax algorithm and a win rate of about 80% (1596 games out of 2000) against one with a 4-ply minimax algorithm.
- The database was a success and about 30-40% moves in each game were obtained from the database which would only improve as it plays more and more training games. Table 5.1 shows the different stages while development of the database for Othello.

Table 5.1 Different Stages of Development of Database for Othello

Opponent	Learning Factor	Number of games played	Strategies developed
Database Play	1.0	2000	Basic strategies, low win rate
Random Play	0.2	2500	Attacking corners, staying to the inside squares as long as possible
Fixed Opponent	0.5	2500	Attacking the sides, sticking to the sides
Fixed Opponent	0.5	2500	Strategies to avoid giving corners as much as possible.
Fixed Opponent	0.5	2500	Much Complex strategies such as forcing the other player to give up corners and sides

5.2 Future Developments

One feature of many programs is notably absent from this project—a graphical user interface. Whilst it would be nice to have the use of a GUI, it is not really a crucial element of the project, since games can be perfectly displayed and interacted with in the console in the form of an ASCII-based interface. Since the main focus of this project was to evolve a Tic-Tac-Toe and Othello playing agent, and not to create a game to be released to the masses, it was felt that the inclusion of a GUI would be given a low priority.

Development of Board Game playing systems was done to test the ability of Reinforcement Learning in simulated environment. Board games offered the challenge of tremendous complexity and sophistication required to play at expert level. At the same time, the problem inputs and performance measures are clear-cut and well defined, and the game environment is readily automated in that it is easy to simulate the board, the rules of legal play, and the rules regarding when the game is over and determining the outcome. After successful testing of our game playing agents we are hopeful that Reinforcement Learning can be used in a large environment.

One of the major improvements that can be made is to obtain some sort of similarity between the states so that multiple states can be mapped to one single state. We have used this partially by implementing Board Inversion. Few other techniques that can be used are –

- Use a position based hash based representation of states.
- Use a combination of a position based hash and heuristics to distinctly represent board states which may be similar to look but have altogether different impact.
- Use the symmetric pattern of boards to save lesser states. For example, in case of Othello the board can be divided into 4 symmetrical pieces and store each of them as 1 state.

Two components make reinforcement learning powerful: the use of samples to optimize performance and the use of function approximation to deal with large environments. Reinforcement learning can be used in an environment with any of the following situations:

- A model of the environment is known, but an analytic solution is not available;
- Only a simulation model of the environment is given (the subject of simulation-based optimization);
- The only way to collect information about the environment is by interacting with it.

Reinforcement learning has proved particularly well suited to problems which include a long-term versus short-term reward trade-off. Due to its generality, Reinforcement Learning can be applied to many disciplines, such as operation research, information theory, genetic algorithms, simulation based optimization, economics and game theory. In the operation research reinforcement learning methods are applied to field called Approximate Dynamic Programming. The problem has been studied in the theory of optimal control, though most studies there are concerned with existence of optimal solutions and their characterization, and not with the learning or approximation aspects. In economics and game theory, reinforcement learning may be used to explain how equilibrium may arise under bounded rationality. It has successfully been applied to various fields like robot control, elevator scheduling, telecommunications, backgammon and checkers (Sutton and Barto 1998).

References

1. **Temporal Difference Learning and TD-Gammon By Gerald Tesauro**
Originally published in *Communications of the ACM*, March 1995 / Vol. 38, No.
3. Copyright © 1995 by the Association for Computing Machinery
2. **Reinforcement learning in board games by Imran Ghory (May 4, 2004)**
citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.4.7712
3. **Reinforcement Learning: An Introduction by Richard S. Sutton and Andrew G. Barto**

The MIT Press, Cambridge, Massachusetts, London, England
4. <http://www.wikipedia.org/>
5. **Development of an Artificial Neural Network to Play Othello by Alexander J. Pinkney (May 2009)**

Appendix A: Strategic elements of Othello

Strategic concepts in Othello include openings, corners, mobility, edge play, parity, endgame play and looking ahead.

A.1 Openings

It is generally the case that for relatively inexperienced players the opening (early) part of a game of Othello is not something that is very easy to make sense of. A general—and somewhat, but not entirely, meaningless—rule of thumb on the opening is that a good opening is one that leads to a good middle game. By and large, good moves in the very earliest stages are determined by whether there is a *refutation* to a move only and few other truly general considerations aside from what exactly constitutes a refutation. The first move (by dark) is no choice at all other than for the purpose of the player's possible sense of ideal visualization. The first move by light gives three choices, and, in fact, it is generally accepted at the highest level that one of these actually may be successfully refuted, that being what is known as the *Parallel opening*. The other two choices by light are called the *Diagonal opening* and the *Perpendicular opening*, and these three in the order mentioned with f5 as dark's first move (See discussion on notation above) are f4, f6 and d6. As this sub-topic is generally broad and complex—at high levels of play it is routine for ten or more moves to be rattled off from rote memorization by both players, travelling down well-worn paths, and there is a common naming system for diverse early-game move choices—it is probably best to leave expansion of this topic to a separate article specifically dedicated to the issue or to say no more.

A.2 Corners

Corner positions, once played, remain immune to flipping for the rest of the game, being termini of horizontal, vertical and diagonal lines. More generally, a piece is *stable* when,

along all four axes (horizontal, vertical, and each diagonal), it is in terminal position or if from it along the axis one reaches a terminal disk passing only through disks the same color. These are not the only kinds of stable disk, however, and occupying a corner may often be a grave error if one allows ones opponent to create a *wedge* that results in him or her gathering more stable disks. This can render occupying the corner largely useless, and often much worse than that because of loss of *tempo* (Where it is an issue of running out of desirable moves and being forced to make undesirable ones, the grabbing of a corner may give the opponent not only the wedging response but also a follow-up move which one cannot respond to practically).

A.3 Mobility

An opponent playing with reasonable strategy will not so easily relinquish the corner or any other good moves. So to achieve these good moves, a player must force his or her opponent to play moves that relinquish those good moves. One of the ways to achieve this involves reducing the number of moves available to the player's opponent. Ideally, this will eventually force the opponent to make an undesirable move.

A.4 Edges

Edge pieces can anchor flips that influence moves to all regions of the board. If played poorly, this can poison later moves by causing players to flip too many pieces and open up many moves for the opponent. However, playing on edges where an opponent cannot easily respond drastically reduces possible moves for that opponent.

The square immediately diagonally adjacent to the corner (called the X-square), when played in the early or middle game, typically guarantees the loss of that corner. Nevertheless, such a corner sacrifice is sometimes played for some strategic purpose (like retaining mobility). Playing to the edge squares adjacent to the corner (called the C-squares) can also be dangerous if it gives the opponent powerful forcing moves.

A.5 Parity

Parity is one of the most important parts of the strategy. In short, the concept of parity is about getting the last move in every empty region in the end-game, and thereby increasing the number of stable disks.

The concept of parity led to a change in the perception of the game, as it led to distinct strategies for playing black and white. It forced black to play more aggressive moves and gave white the opportunity to stay calm and focus on keeping the parity. As a result the opening books and mid-game were focused on black being the "attacker" and white being the "defender".

The concept of parity also controls how edge positions are played and how edges interact.

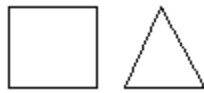
A.6 Endgame

For the endgame (the last 20 or so moves of the game) the strategies will typically change. Special techniques such as sweeping, gaining access, and the details of move-order can have a large impact on the outcome of the game. Actual counting of disks in the very final stages is often critical, but sometimes in human play an inaccurate choice for disk differential can be better than an accurate one in terms of the expected outcome (and can be essential in lost positions).

Appendix B: Minimax Algorithm with α , β

Prunning

The minimax algorithm is a way of finding an optimal move in a two player game. *Alpha-beta pruning* is a way of finding the optimal minimax solution while avoiding searching subtrees of moves which won't be selected. In the search tree for a two-player game, there are two kinds of nodes, nodes representing *your* moves and nodes representing *your opponent's* moves. Nodes representing your moves are generally drawn as squares (or possibly upward pointing triangles):



These are also called *MAX* nodes. The goal at a MAX node is to maximize the value of the subtree rooted at that node. To do this, a MAX node chooses the child with the greatest value, and that becomes the value of the MAX node.

Nodes representing your opponent's moves are generally drawn as circles (or possibly as downward pointing triangles):



These are also called *MIN* nodes. The goal at a MIN node is to minimize the value of the subtree rooted at that node. To do this, a MIN node chooses the child with the least (smallest) value, and that becomes the value of the MIN node.

Alpha-beta pruning gets its name from two bounds that are passed along during the calculation, which restrict the set of possible solutions based on the portion of the search tree that has already been seen. Specifically, β is the *minimum upper bound* of possible solutions and α is the *maximum lower bound* of possible solutions. Figure B.1 demonstrate minimax with alpha-beta pruning:

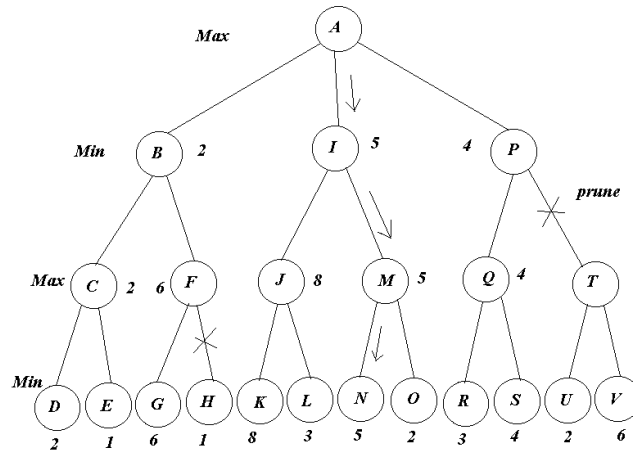


Figure B.1 Minimax tree with apha beta pruning

B.1 Pseudocode for α , β Pruning

```

function alphabeta(node, depth,  $\alpha$ ,  $\beta$ , Player)
  if depth = 0 or node is a terminal node
    return the heuristic value of node
  if Player = MaxPlayer
    for each child of node
       $\alpha := \max(\alpha, \text{alphabeta}(\text{child}, \text{depth}-1, \alpha, \beta, \text{not}(\text{Player}))$ 
      if  $\beta \leq \alpha$ 
        break (* Beta cut-off *)
    return  $\alpha$ 
  else
    for each child of node
       $\beta := \min(\beta, \text{alphabeta}(\text{child}, \text{depth}-1, \alpha, \beta, \text{not}(\text{Player}))$ 
      if  $\beta \leq \alpha$ 
        break (* Alpha cut-off *)
    return  $\beta$ 
  (* Initial call *)
  alphabeta(origin, depth, -infinity, +infinity, MaxPlayer)

```