University of Windsor

Scholarship at UWindsor

Electronic Theses and Dissertations

Theses, Dissertations, and Major Papers

2009

Using Cultural Coevolution for Learning in General Game Playing

Shiven Sharma
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation

Sharma, Shiven, "Using Cultural Coevolution for Learning in General Game Playing" (2009). *Electronic Theses and Dissertations*. 8263.
https://scholar.uwindsor.ca/etd/8263

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

Using Cultural Coevolution for Learning in General Game Playing

by

Shiven Sharma

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada
2009

©2009 Shiven Sharma

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN 978-0-494-82082-7
Our file Notre référence
ISBN 978-0-494-82082-7

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation

---

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge. my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations. or any other material from the work of other people included in my thesis, published or otherwise. are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office. and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

Traditionally, the construction of game playing agents relies on using pre-programmed heuristics and architectures tailored for a specific game. General Game Playing (GGP) provides a challenging alternative to this approach, with the aim being to construct players that are able to play any game, given just the rules. This thesis describes the construction of a General Game Player that is able to learn and build knowledge about the game in a multi-agent setup using cultural coevolution and reinforcement learning. We also describe how this knowledge can be used to complement UCT search, a Monte-Carlo tree search that has already been used successfully in GGP. Experiments are conducted to test the effectiveness of the knowledge by playing several games between our player and a player using random moves. and also a player using standard UCT search. The results show a marked improvement in performance when using the knowledge.

Dedication

There are two people who have, in their own way, made it possible for me to reach this point and still be alive. Consequently, this thesis is dedicated to them.

The first person is my grandfather, Dr. Ram Prakash Sharma, who always ensured that I had money here to support myself in every possible way. Without his help I would not have been able to reach this far. Though I did squander some of the money in pursuits that, in my mind, were totally reasonable, but in his mind were frivolous, he remained patient with me and sold two houses, moved twice to different ones, and still made sure I finished my studies. I don't know when I can repay his generosity, but I hope this dedicate is a small step in that direction.

The second person is one of my best friends, Ivica Rosa. While my grandfather provided the financial support, he provided the moral support. I have seen him transform from a bachelor to a husband to a father, and all the time he has always been here for me whenever I needed him. Friends like this are extremely hard to find, and I am grateful to him and his family for being the biggest help and best of friends to me than I could have ever imagined.

Acknowledgment

I wish to thank Dr. Ziad Kobti and Dr. Scott D. Goodwin for their assistance with my work.

Dr. Kobti introduced me to General Game Playing during my undergraduate years. Little did I know the hold it would take over my life. He has remained a steadfast supporter in my quest to find a light at the end of the general games tunnel, and his carefree and friendly nature has made sure I never fell through a hole of despair. For this, I gratefully acknowledge his help, support and friendship.

Dr. Goodwin introduced me to Dr. Kobti, which started the chain reaction. He was the first person to offer me an undergraduate research assistantship, and through that I got a flavour of games. He was also the first (and only) professor to give me a teaching opportunity, which future solidified my desire to go into academia. Throughout my work, he has endured and tolerated my laziness, and it is my sincere hope that I have more than made up for that! His support and friendship has been most valuable.

A special acknowledgment is due to Dr. Robert Kent who told me about the Artificial Intelligence programme at the University of Windsor, which allowed me to escape from the clutches of business studies and discover the exciting world of Artificial Intelligence research and academia. He may not realise it, but he is responsible for one of the most significant turns in my life. For that, I am extremely grateful to him.

Contents

AUTHOR'S DECLARATION OF ORIGINALITY iii

ABSTRACT iv

DEDICATION v

ACKNOWLEDGMENT vi

LIST OF TABLES x

LIST OF FIGURES xi

1 Introduction **1**

1.1 Previous work in General Game Playing 1

1.2 The Stanford University GGP Project 2

1.3 Thesis Contribution 3

1.4 Organisation 3

2 The General Game Playing Architecture **4**

2.1 Formal Game Description 4

2.2 Game Description Language: GDL 5

2.3 The Communication Model 7

2.4 Implementation Details 8

2.5 Updating States and Making Moves 8

2.5.1 Updating the states 9

2.5.2 Obtaining a legal move 9

3 Reinforcement Learning **13**

3.1 The Exploration-Exploitation Dilemma 14

3.1.1 The N-Armed Bandit 14

3.2 Goals and Rewards 15

3.3 The Markov property and Markov Decision Processes 16

3.4 Value functions 16

3.5 Temporal Difference Learning 17

3.6 UCT: Upper Confidence Bound Applied to Trees . . 18
3.6.1 UCT applied to General Games 19
3.7 Conclusions 19

4 Evolutionary Methods **21**
4.1 Coevolution . 21
4.2 Ant Colony Optimisation Algorithms . 22
4.3 Cultural Algorithms 23
4.4 Conclusions . 24

5 Learning and Knowledge Representation Architecture **25**
5.1 Player Architecture 25
5.2 Knowledge Representation Scheme 26
5.3 Recognising games independent of game rules . 27
5.4 Conclusions 29

6 Using Reinforcement Learning for General Games **31**
6.1 Using TD-Learning . 31
6.2 Temporal Segmentation of Features . 32
6.3 TD(0) Update 33
6.4 Experiments . 34
6.5 Discussion 34

7 Using Coevolution **36**
7.1 Population Model 36
7.1.1 Fitness calculations 37
7.2 Experiments . 37
7.3 Discussion 38

8 Ant Colony Optimisation **39**
8.1 GGP using Ant Colonies . 39
8.2 Communication via Pheromone and Desirabilities 41
8.3 Experiments . 41
8.4 Discussion . 42

9 Adding Cultural Knowledge **43**
9.1 Representing the Belief Space 43
9.2 Putting it all together 44
9.3 Experiments . 45

9.4 Discussion 46

10 Using the model with UCT **48**

10.1 Experiments 48

10.2 Discussion 49

11 Final Discussion and Future Work **50**

11.1 Discussion of Results 50

11.2 Future Work 51

BIBLIOGRAPHY 53

VITA AUCTORIS 56

List of Tables

6.1 Random Games 34
6.2 Player 1 uses knowledge 35
6.3 Player 2 uses knowledge 35

7.1 Player 1 uses knowledge 38
7.2 Player 2 uses knowledge 38

8.1 Player 1 uses coevolutionary knowledge 42
8.2 Player 2 uses coevolutionary knowledge 42

9.1 Player 1 uses cultural and coevolutionary knowledge 46
9.2 Player 2 uses cultural and coevolutionary knowledge 47

10.1 Number of wins for UCT_S and UCT_K over 100 matches per game 49

List of Figures

2.1 Communication between the Game Manager (GM) and the Game Player (GP) for the game of Blocks. 12

5.1 The architecture for a general game player for learning. 26

5.2 Features in a state of Tic-Tac-Toe. The vector θ represents the vector of features. 27

5.3 An example of knowledge representation and use while making a move in Tic-Tac-Toe. 28

6.1 The graph of the sigmoid function. 32

6.2 Temporal segmentation of features. Circular nodes are nodes from which the player makes moves. Square nodes are the states that occur as a result of these moves (the afterstates). 33

8.1 The forage of Ant_X through the Tic-Tac-Toe landscape. The presence of Ant_O besides the hollow arrows indicates that Ant_X has the option of asking Ant_O for a move, though it is not necessary to do so. Pheromone is deposited along the squiggly arrow once a series of forages has been completed. 40

8.2 The Ant Colony GGP Architecture. 40

9.1 The architecture for a single role using coevolution and cultural evolution. 45

Chapter 1

Introduction

This thesis addresses the challenge of building intelligent General Game Playing agents. The field of General Game Playing (GGP) is relatively new in the field of Artificial Intelligence (AI) research, and provides an important leap in the direction and approach of the construction of intelligent agent systems. In the past, much of the emphasis in the creation of intelligent systems was on their ability to be proficient only for tasks they were constructed to perform. As an example, the IBM Deep Blue computer had been constructed specifically for being good and intelligent at playing chess. However, if any other game was given to it, it would be unable to play it as a result of its specialisation. GGP systems, as the name implies, are far more general. They are able to accept descriptions of any game, and are able to play them, either by making legal random moves, or legal moves made in an intelligent manner, depending on their construction.

The importance of this research lies in the fact that GGP systems provide a step from intelligent systems giving an illusion of intelligence to intelligent systems that act in an intelligent manner. This difference is clear from the distinction made above between systems like Deep Blue and GGP systems. GGP systems demonstrate their intelligence in being able to successfully play any game, given only the rules of the game. One must delve into the areas of machine learning, knowledge representation and pattern recognition in order to be able to construct a proficient player.

A brief overview of previous work done in the area of GGP and details of the work being done based on the foundations laid down by Stanford University, in context of their annual GGP competition, is now provided.

1.1 Previous work in General Game Playing

Though pure General Game Playing capabilities have not entirely been implemented, systems have been designed which display a general behaviour with respect to a specific class of games. One class of games where general game playing has been investigated are positional games. These type

of games were formalised by [18]. Some examples of position games include Tic-Tac-Toe, Hex. the Shannon switching games.

A position game can be defined by three sets, P, A, B. Set P is a set of positions: with set A and B both containing subsets of P In other words. sets A and B represent a collection of subsets of P, with each subset representing a specific positional situation of the game. The game is played with two players, with each player alternating in moves, which consist of choosing an element from P The chosen element cannot be chosen again. The aim for the first player is to construct one of the sets belonging to A. whereas the aim for the second player is to construct one of the sets belonging to B.

Programs that are capable of accepting rules of positional games. and. with practice, learn how to play the game have been developed. Koffman constructed a program that is able to learn important board configurations in a $4 \times 4 \times 4$ Tic-Tac-Toe game. This program plays about 12 times before it learns and is effectively able to play and start defeating opponents. A set of board configurations are described by means of a weighted graph.

1.2 The Stanford University GGP Project

The annual General Game Playing Competition [15] organised by Stanford University has been instrumental in bringing about renewed interest in GGP. The rules of the games are written in Game Description Language (GDL) [14], which is syntactically similar to prefix KIF [16]. As a consequence, most of the current research in GGP is based on the foundations laid down by the Stanford Group. The tournaments are controlled by the Game Manager (GM) which relays the game information to each Game Player (GP) and checks for legality of moves and termination of the game. Communication between players and the GM takes place in the form of HTTP messages. A more detailed description of the architecture and game rules can be found at [1]. Successful players have mostly focused on automatically generating heuristics based on certain generic features identified in the game. Cluneplayer [7] was the winner of the first GGP competition, followed by Fluxplayer [26]. Both these players, along with UTexas Larg [4] use automatic feature extraction. Evaluation functions are created as a combination of these features and are updated in real-time to adapt to the game. Another approach that has been taken is in [5], where transfer of knowledge extracted from one game to another is explored by means of a TD(λ) based reinforcement learner. CADIA-Player [6] was the first General Game Player to use a simulation based approach, using UCT [17] to search for solutions, and was the winner of the last GGP Competition. [27] also explored a Monte-Carlo approach in which random simulations were generated and the move with the highest win rate was selected. To improve the nature of these simulations. patterns in the sequences were extracted and used to generate new sequences. [22] have discussed a co-evolutionary approach using NEAT [31],

an algorithm for automatically evolving neural networks using an evolutionary approach, for GGP

1.3 Thesis Contribution

The major contributions thesis can be enumerated as follows:

1. This work represents, what is to the best of our knowledge, the first attempt at using machine learning techniques for GGP
2. We use cultural coevolution to learn knowledge in two player, zero-sum, perfect information deterministic games in the total absence of any prior knowledge or information about the game.
3. We demonstrate that effectiveness of this knowledge by playing various games for which the knowledge is learnt using our approach against a standard random player. The matches are against a random player because the current GGP format does not support learning players.
4. We combine the knowledge with UCT search, and demonstrate that this combination improves the performance of UCT for general games.

There is a growing interest in the academic community at starting new GGP competitions that will allow learning players to compete. As a result, there is a great scope for future work in the area that looks into developing architecture and description languages that will suit learning players. The work presented in this thesis is based on the format developed at Stanford, which is not conducive to learning players. However, the principles and algorithms presented here can be used with any game description language that allows for a random game to be played and state descriptions to be stored and recognised. Future work which develops languages and architetures for learning players will further complement the ideas presented in this work.

1.4 Organisation

This dissertation is organised as follows: Chapter 2 provides an overview of the GGP framework as laid down by Stanford University, including the Game Description Language. We use this framework as a testbed for our work. Chapters 3 and 4 provide brief introductions to Reinforcement Learning and the Evolutionary methods that we use in our work. Chapter 5 gives the basic structure of the player and also describes how the knowledge is represented. Chapters 6 discusses how RL can be used in GGP. Chapters 7 to 9 discuss using complementing the RL architecture with coevolution, ant colonies and cultural algorithms respectively. In Chapter 10 we describe how the knowledge learnt can be used with UCT search. Finally, Chapter 11 gives the various directions for future work in learning for GGP.

Chapter 2

The General Game Playing Architecture

In this chapter, we first review a formal description of what a game is. We then look into the syntax and semantics of the Game Description Language (GDL). The game of Tic-Tac-Toe is chosen to provide an example on how the GDL is written and interpreted. Then, we describe the basic communication mechanisms between the Game Manager (GM), and the Game Player (GP). We discuss the types of messages involved in the communication, and detail out a sequence of message exchanges between the GM and the GP for a small game. Much of the details of this chapter are based on [14, 15].

2.1 Formal Game Description

The games considered in GGP can be considered to be *finite*, *synchronous* games. The game runs in an environment which consists of a finite number of *game states*. A single state is designated as the *start state*. The game also consists of one or more *terminal states*. The number of players playing the game is a fixed, finite number. Whenever a player is in a game state, the player has a finite number of actions possible from that state. All players make moves on all steps. Whether or not they move from one state to another, or stay in the same state, is dependent on the move made, and the state of the game at that time. The environment updates the game state only when players make moves.

From the above description of a game, it is clear that we can view a game to be a finite state machine. Below, we list the various components of the machine:

S : A set of game states

$r_1, r_2, \cdots, r_n$: The various roles, or players of an n-player game

$A_1, A_2, \cdots, A_n$: Sets of actions for each player. A set A_i corresponds to the actions possible by player r_i

$l_1, l_2, \cdots, l_n$: Legal actions from a particular state. Each l_i can be considered to be a subset of $A_k \times S$

u : An update function from one state to another based on all the moves made

$u : A_1 \times A_2 \times \cdots \times A_n \times S \rightarrow S$

$s1$: The start state

$g_1, g_2, \cdots, g_n$: Each $g_i \subseteq S \times 1, \cdots, 100$. The value of a game outcome for a player. For player r_i, the value for a terminal state s is given to be $g_i(s, value)$

T : The set of terminal states. $T \subseteq S$

2.2 Game Description Language: GDL

A Game Description Language (GDL) is a language that can be used to describe games of complete or partial information. It is a logical language, using logical sentences to describe rules of a game. Syntactically, it is similar to LISP in infix notation. As we will later describe, it is relatively easy to parse this into an equivalent form based on predicate calculus. In this section, we examine the nature of GDL, using Tic-Tac-Toe as an example to explain it.

As a game can be viewed as a finite state machine, each game has a start state. For Tic-Tac-Toe, the start state is a 3 x 3 gird in which each cell is blank. Therefore, we can initialise each cell to be blank by using the init keyword of GDL in the following way (the example below illustrates only the first row):

```
init(cell(1,1,b))
init(cell(1,2,b))
init(cell(1,3,b))
```

cell(x, y, z) implies that the cell in row **x** and column **y** is initialised, or set, to **z**. In the case above, **z** is **b**, implying blank.

Since the game is a two-player game, we have to give the control of the play to the first player.

This can be done by the following statement:

```
init(control(white))
```

Player one has been given the name white, and the control relation takes the players name as the argument, and gives that player the ability to make the move.

A GDL should be able to define which moves are legal. In Tic-Tac-Toe, a move is legal for a player if it is that players turn, and the player marks a cell that has not already been marked. In the eventuality that neither of the conditions are true for a legal move, the player waits (the legal action in this case is termed a noop). This can be illustrated as shown below:

```
legal(P,mark(M,N)) <= true(cell(M,N,b)) & true(control(P))
legal(white,noop) <= true(cell(M,N,b)) & true(control(black))
legal(black,noop) <= true(cell(M,N,b)) & true(control(white))
```

In the above statements, the first statement implies that if the player P has the control (i.e., it is player P's turn to move), and if the cell in row M and column N is blank, then the player is allowed to mark that cell. Otherwise, if the control is with the other player, then the player performs a noop action, which is an abbreviation for no operation.

Whenever a cell is marked with an X or an O, the game states are updated. The update rules are specified by the next predicate. These are shown as follows:

```
next(cell(M,N,x)) <= does(white,mark(M,N)) & true(cell(M,N,b))
next(cell(M,N,o)) <= does(black,mark(M,N)) & true(cell(M,N,b))
next(cell(M,N,W)) <= does(true(cell(M,N,W))) & distinct(W,b)
next(cell(M,N,b)) <= does(W,mark(J,K)) & true(cell(M,N,W)) &
                     (distinct(M,J) | distinct(N,K))
```

The first two statements describe that if the control is with player white or black, then, if the cell they are marking is blank initially, then that cell is marked with an X or an O respectively. The third statement guarantees that the cell being marked is always blank. The fourth statement is an update statement for a cell that is blank and is not marked. It ensures that if another cell, distinct from this cell, is marked, and this cell is blank, then this cell remains blank after the move is executed.

Every game has a set of terminal states. In Tic-Tac-Toe, these are when any single row, column or diagonal of the board is marked with all X's or all O's, or in the eventuality that the game fails to

achieve these conditions, i.e., a stalemate. Therefore, the terminal states can be defined as follows:

$$\texttt{terminal} <= \texttt{line(x)}$$

$$\texttt{terminal} <= \texttt{line(o)}$$

$$\texttt{terminal} <= \texttt{open}$$

While the terminal states specify the states at which if the game is, then the game is done. However, it does not specify the goals of the players. For each player, each goal can be assigned a numerical value. The player with the highest value of the goal wins the game. In a zero-sum game such as Tic-Tac-Toe, there is only one winner. However, it is to be noted that not all games are zero-sum, and games can have multiple winners. In these cases, the players with the highest goal values win the game. In order to specify goals for Tic-Tac-Toe, we can define them as follows (only the goals for `white` are shown):

$$\texttt{goal(white,100)} <= \texttt{line(x)}$$

$$\texttt{goal(white,50)} <= \texttt{line(x)} \; \& \; \texttt{line(o)} \; \& \; \texttt{open}$$

$$\texttt{goal(white,0)} <= \texttt{line(o)}$$

The goal of `white` is maximised when white gets a line of `X`'s (a line in this case is either a row, column or diagonal). The second goal specifies an example of a draw, in which neither player is able to get a line of `X`'s and `X`'s. If the other player is able to get a line of `O`'s, then the goal value of white becomes 0, i.e., white loses the game.

2.3 The Communication Model

The architecture of the GGP model that will be tested for this project will consist of a Game Manager (GM) and the Game Player (GP). The Game Manager is the centralised server machine that holds the game descriptions, allows for players to communicate and maintains the state of the overall game. The Game Player initiates communication with the GM, during which the GM provides the GP with the rules, written in GDL. The GP then has a limited time to learn from those rules.

Communication takes place in the form of HTTP messages that are passed between the GM and the GP. The start message, sent to the GP by the GM, has the form of:

$$(\texttt{START} \; < \texttt{MATCH ID} > \; < \texttt{ROLE} > \; < \texttt{GAME DESCRIPTION} > \; < \texttt{STARTCLOCK} > \; < \texttt{PLAYCLOCK} >)$$

Where `MATCH ID` is the unique identification for that particular match, the `ROLE` is the role of the player assumed by the GP, `GAME DESCRIPTION` is the description of the game in GDL, and

STARTCLOCK and PLAYCLOCK are integer values representing the move times. Game descriptions sent are in the prefix form of the GDL described in the previous section.

The PLAY command takes the form:

$$(\texttt{PLAY}\ < \texttt{MATCH ID} >\ (< \texttt{A}_1 > < \texttt{A}_2 > \cdots < \texttt{A}_n >))$$

where each $< \texttt{An} >$ is the action taken by the players in the previous steps. These actions are well-ordered; the ordering sequence is specified by the game description axioms. The reply to the PLAY command is the actual move made by the GP.

The STOP command takes the form:

$$(\texttt{STOP}\ < \texttt{MATCH ID} >\ (< \texttt{A}_1 > < \texttt{A}_2 > \cdots < \texttt{A}_n >))$$

where each $< \texttt{An} >$ is the same as what is described above in the PLAY command. This message terminates the game.

An example of the basic communication model that is used by the General Game Playing (GGP) server, Game manager, and the Game Player is illustrated below. The example used is that of the game of Blocks. The valid moves for the game are MOVE, DROP and GRAB.

2.4 Implementation Details

In this chapter we will examine in detail the steps necessary for the GGP programme to take in order to be able to successfully play a game, given the game description. The previous chapters have examined the abstract communication model and methods to build the knowledge base from the game rules. Now, the approach taken to facilitate the player to be able to make legal moves and update states is described.

2.5 Updating States and Making Moves

Very briefly, the task of making a legal random move can be seen as consisting of the following steps:

1. Obtain the moves made by each player in the current turn (if the game is just starting, then this step is ignored).
2. Use these moves to update the game states (if the game is just starting, then this step is ignored).
3. Obtain all the legal moves that can be made in the current game state.
4. Select a random move from this set and send that back to the GM.

As we have seen from the description and syntax of the GDL, the game descriptions are written in a logical language. Updating states and obtaining legal moves is therefore a process of inferencing and reasoning using the rules. As we have seen in the previous chapter, converting these rules into an equivalent Prolog code simplifies this task. This code, along with additional ground clauses and rules, make up the knowledge base of the game. We use Prolog's inbuilt inference engine to handle the inferencing procedures. Therefore, given all this, the task of updating states is simply the modification of the state clauses in the knowledge base.

2.5.1 Updating the states

The states are represented as ground clauses in the knowledge base. The next predicate takes a state description as an argument (this description represents the state valid when the moves communicated by the GM for all players have been executed). This argument is a valid state iff the next rule evaluates to true. If this the case, then we need to update the knowledge base by removing the current state description, and replacing it with the argument of the next rule. In Prolog terminology, this would imply:

1. Retract all the current state ground clauses.
2. Assert the state descriptions unified to the argument of the next rule after its evaluation.

The idea of updating states can be illustrated by a simple example. Consider the following ground clauses and the next rule for the game of Tic-Tac-Toe:

```
init(cell(1,1,b))
init(cell(1,2,b))
init(cell(1,3,b))
next(cell(M,N,x)) <= does(white,mark(M,N)) & true(cell(M,N,b))
```

If, during game play, the player with the role white marks the cell at row 1 and column 2 with an `X`, then the ground clause `does(white,mark(1,2,x))` is true. We need to assert this ground clause into the knowledge base so that the next rule can be evaluated. When the Prolog inference engine starts to evaluate the rule, the variables `M` and `N` are unified with 1 and 2 respectively. Therefore, the new state to be asserted is `cell(1,2,x)` and the state to be retracted is `cell(1,2,b)`.

2.5.2 Obtaining a legal move

Moves that are legal, given the current game state, are represented by the arguments to the legal predicate in the knowledge base. Once the states are updated as described above, we need to evaluate the legal rules to obtain all possible legal moves that can be made.

Now the evaluation procedure to obtain a legal move is described. The legal predicate takes two arguments: the first argument represents the player, and the second argument represents the move that the player can make. The constraints under which this move can be made are represented by the predicates on the right hand side of the rule. If all these predicates evaluate to true, then the player can make the move.

The procedure for obtaining a legal move is similar in concept to that of updating states. The idea is to unify the variables of the rule and use the value for the move as a legal move. Prolog automatically performs the unification during evaluation, and the legal moves can easily be collected in a list. Therefore, to obtain a set of legal moves, we simply call all the legal rules and collect the assignments to the argument corresponding to the move in a list. After this, it is a simple matter of selecting a random move and sending it to the game manager.

Below, we illustrate an example of obtaining a legal move. Consider the following clause for making a legal move:

```
legal(W,mark(X,Y)) <= cell(X,Y,b) & control(W)
```

In the clause above, we check for two state conditions to make sure if the move represented in the argument to legal can be made. Consider also the following state clauses:

```
cell(1,1,b)
cell(1,2,c)
cell(1,3,b)
control(white)
```

Assume that our role is that of `white`. We call the `legal` predicate to provide us with possible legal moves. Upon running, `W` is unified with `white` (as the current state has `control(white)`). Now, `X` and `Y` are first unified with 1 and 1 respectively (from the state `cell(1,1,b)`). Therefore, the first possible legal move that we can make will be `mark(1,1)`. However, this is not the only move that we can make. Notice that following the same procedure, `X` and `Y` also get unified to 1 and 3 respectively. Therefore, another move that we can make is `mark(1,3)`. Therefore, for the above descriptions, we can make two legal moves. In order to make a move, we randomly select one of these moves, and send it as a reply to the game manager as our move.

For now, we have seen how to get a list of legal moves. An important aspect of the game player programme will be to decide which of these moves to make. In subsequent chapters, methods on how to make intelligent legal moves will be explored. For an intelligent player, the main ideas of updating states and getting a list of possible legal moves is the same: what changes is the selection

process for a move to chose as the final response.

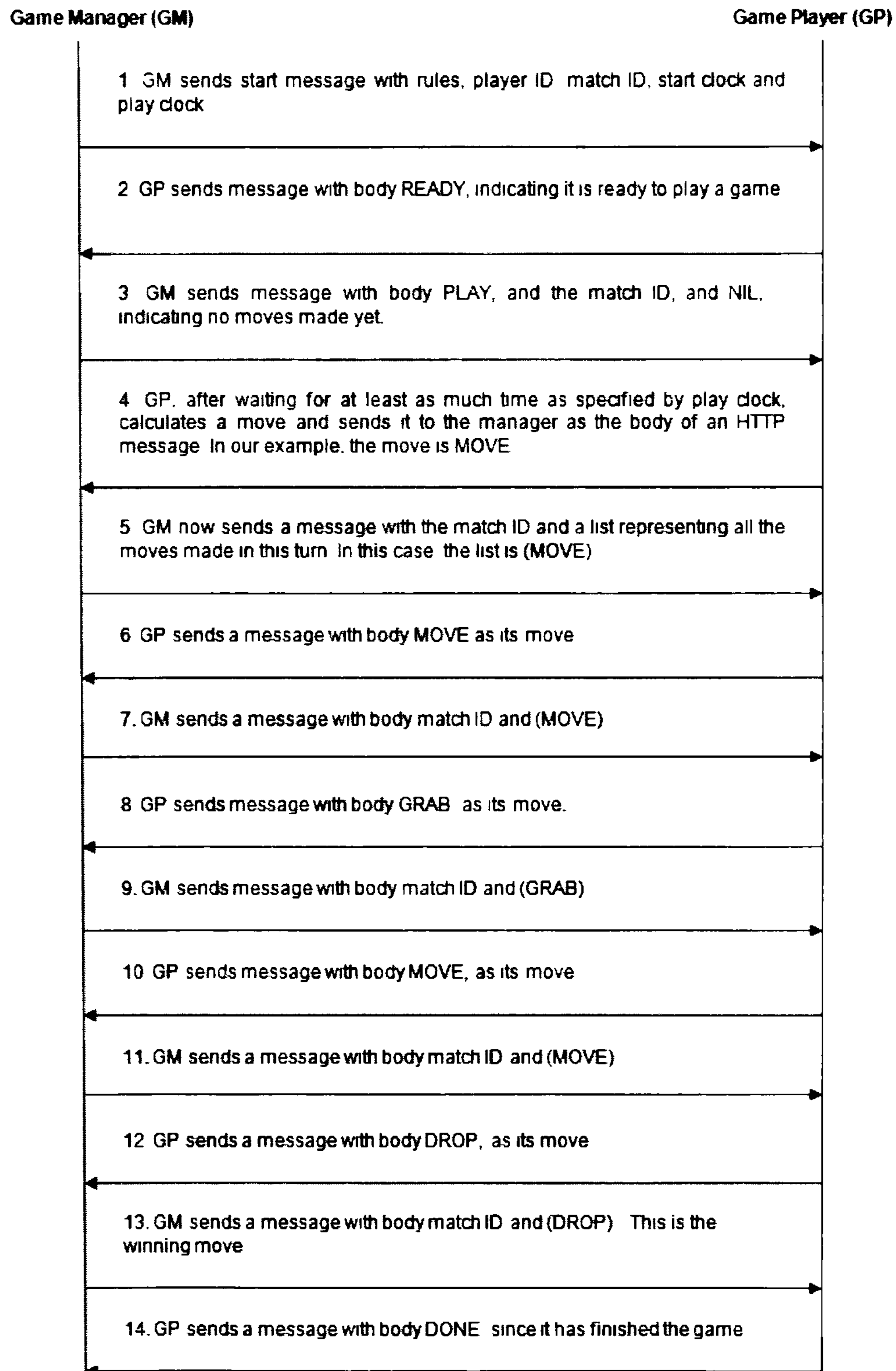


Figure 2.1: Communication between the Game Manager (GM) and the Game Player (GP) for the game of Blocks.

Chapter 3

Reinforcement Learning

This chapter provides a brief introduction to Reinforcement Learning (RL). The GGP player relies heavily on RL, using both Temporal Difference Learning and UCT (Upper Confidence Bounds Applied to Trees) search. Both these topics are discussed in later sections. Much of the content of this chapter is derived from [32] which is the standard text for RL.

Reinforcement learning is not a method or an algorithm. Its an *approach*, a problem solving idea. It is different from *supervised learning*, where an external *supervisor* directs the learning by telling the algorithm whether or not they are on the right track. What if the learning is taking place in an unknown environment, where there just can't be a supervisor? In such a case, the agent must learn from its experiences and interaction with the environment. This is where RL comes in.

Since RL takes place in real-time, the agent needs to constantly monitor its environment, taking into account any unpredictable events that might happen as a result of its action, and then react appropriately. The goals are defined explicitly for the agent, so the agent can monitor its progress towards the goal as it takes actions and interacts with the environment, using the experience learnt to improve its performance.

The four main features in a RL system are:

1. **Policy**: Given a state, what action should be taken? In other words, a policy is a mapping from a state to a set of possible actions that can be taken from that state. It determines the agents behaviour.

2. **Reward function**: Indicates what is good or bad. Defines the main goal of the reinforcement agent. It associates with each state or state-action pair a number, which indicates the intrinsic desirability of that state or pair. Consequently, it provides an immediate feedback of a situation. The aim of the agent is to *maximise* the *total reward* it receives in the long run. The agent has no control over the reward function; it can't modify it.

3. **Value function**: Tells the agent the *total reward* it can receive from a situation. Whereas the reward gives an immediate feedback, the value function gives the long-term feedback, taking into account the rewards from its current situation and all possible states that are likely to follow it.

4. **Environment Model**: Gives a model of the environment in which the agent is operating. It tells the agent how the environment behaves, and what the agent can expect when it takes actions from states.

In subsequent sections we examine the exploration-exploitation dilemma which lies at the heart of RL, Temporal Difference Learning and Monte-Carlo Learning methods in RL.

3.1 The Exploration-Exploitation Dilemma

Since action selection is a core component of RL, a natural question that arises is whether an agent should exploit actions based on what it already knows (i.e. it greedily selects the best action) or should it instead explore new actions, with the hope of possibly increasing its reward? This is known as the *exploration-exploitation dilemma.* This question also arises in the UCT algorithm described in later chapters, where the player needs to select a move to be made and descend down the game tree. Therefore, it is worth examining this in more detail. These questions are perhaps best explained in the literature in the form of the N-armed bandit problem.

3.1.1 The N-Armed Bandit

The bandit in this case is an analogy to a slot machine, with N arms, and the gambler is the agent. Each time the agent selects (or pulls) an arm, it gets a reward from a stationary probability distribution (in other words, its not a *fixed* reward). A probability distribution is associated with each arm. The goal of the agent is to maximise its total reward over a number of plays, where each play is the selection of an arm.

Assume that for each arm, the agent maintains an *estimate* of its value. This value is the mean reward it has received from that arm so far (discussed in more detail below). Now, at time step k, what action should the agent select? The one with the highest estimate, called the *greedy action* (exploitation), or a non-greedy action (exploration)? Selecting a non-greedy action may well result in a better reward in the long run, or it may not. Deciding between exploration and exploitation depends on many things, such as the value of the estimates, the certainty with which we know these values and the remaining plays.

Estimating and Selecting Action-Values

The actual value of the action (i.e. play of an arm) is the *mean reward* received from selecting it. However, since the distribution from which the reward comes is not known, we can only make a guess as to what reward we will get from it. One simple way to estimate this is by keeping track of all the rewards received for k selections for action a, and then taking the average of it. In other words, if total plays have been t, and action a has been selected k_a times, then the estimated reward from a will be

$$Q_t(a) = \frac{r_a + r_2 + r_3 + ... + r_{k_a}}{k_a} \tag{3.1}$$

This is called the *sample average* estimate for a. As the number of plays approaches infinity, $Q_t(a)$ will become equal to $Q_t^*(a)$, which is the actual value for a.

Now that we have estimates, how do we select actions based on these estimates? A pure greedy method would mean selecting a' for which $Q_t(a') = max_a Q_t(a)$. In other words, at time t, select the action which has the highest estimated value. This would, however, mean that we never explore. An alternative to this is selecting the greedy action most of the time, but with a small probability ϵ, select an action randomly, without considering its value. This is called the $\epsilon - greedy$ action selection method. For an infinite number of plays, each action will be sampled an infinite number of times. If the rewards are fixed for each action, then, after selecting an action once, we know the value. In this case, greedy action selection will work. However, most applications are non-stationary (the rewards change over time), and so exploration is needed.

Another approach to selecting actions is by taking into account the action values. The probabilities are weighted by the value of each action. So, the probability of selecting action $a \in A$ will be given by

$$P(a) = \frac{e^{Q(a)/\tau}}{\sum_{\alpha \in A} e^{Q(\alpha)/\tau}} \tag{3.2}$$

This is called the *softmax* selection. τ is called the *temperature* parameter. As $\tau \longrightarrow 0$, action selection becomes closer to greedy.

3.2 Goals and Rewards

As mentioned before, the agent attempts to maximise the long term reward (not the immediate reward). This formalisation is a key feature in R.L; providing the agent a clear learning task. Rewards then should be given in a clever manner, so that the agent can use them to achieve the goal. For example, in a path finding task, if we want the agent to find the quickest route from a starting point to an ending point, we can let it have a negative reward for each step it takes, and

a positive reward for when it reaches the ending point. Clearly, the shortest path will have the highest reward. Another important thing to remember is that rewards are not meant to give the agent knowledge on *how* to accomplish something; they are meant to tell it *what* to accomplish.

3.3 The Markov property and Markov Decision Processes

Is it necessary for the agent to know how it reached a state in order to make a decision? A good state will tell the agent all that has happened, and at the same time retain all information that is needed by the agent to make a decision to select an action. Such a state is said to be *Markov.* In other words, all that is needed to move on to the next state is the current state, not the entire sequence telling how you reached the current state. Formally, the *Markov Property* is defined as

$$P(s_{t+1}, a_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, s_{t-2}, a_{t-2}, \cdots, s_0, a_0) = P(s_{t+1}, a_{t+1}|s_t, a_t)$$

The Markov property is an important part in RL as it assumes that making decisions are a function of only the current state.

Any RL task having the Markov property is a Markov Decision Processes (MDP), defined by the set of states, set of actions and the one-step dynamics of the environment. A stochastic M.D.P (a deterministic M.D.P can easily be derived from this definition) is defined as $\{\mathbb{S}, \mathbb{A}, \mathbb{P}, \mathbb{R}\}$, where $\mathbb{S}$ is the set of states, $\mathbb{A}$ is the set of actions, $\mathbb{P}^a_{ss'}$ gives the probability that the next state will be s' if the agent takes action a in state s and $\mathbb{R}^a_{ss'}$ is the expected reward r_{t+1} the agent gets by transitioning from state s at time step t to state s' by taking action a. Clearly, $\mathbb{P}^a_{ss'}$ and $\mathbb{R}^a_{ss'}$ specify the one-step dynamics of the system.

3.4 Value functions

A value function simply tells how good a state (or a state-action pair) is, i.e. how much discounted future reward (expected return) can be received. If a state s_a gives a higher return than a state s_b, then the former has a higher value than the latter. Since a value is the expected return, which in turn is defined by the actions that are taken from states, which in turn are defined by the policy we follow (since a policy tells us what actions to take), values are related to specific policies.

We can define a value for a state s under a policy π as $V^\pi(s)$, called the *state value function*, and the value of a state-action pair (s, a) as $Q^\pi(s, a)$, called the *action-value function.* $V^\pi(s)$ gives the expected return from state s if from s we follow (select actions) policy π. $Q^\pi(s, a)$ gives the expected return of taking action a from state s if, well, we take action a from state s and then follow policy π. Usually, a is taken using some other policy. Naturally, if a was taken using π itself, $Q^\pi(s, a)$ will be the same as $V^\pi(s)$.

Since the values are the expected rewards starting from s or (s, a), we get

$$V^{\pi}(s) = E_{\pi}\{R_t|s_t = s\} = E_{\pi}\left\{\sum_{k=0}^{\infty} r_{t+1+k}|s_t = s\right\}$$

$$Q^{\pi}(s,a) = E_{\pi}\{R_t|s_t = s, a_t = a\} = E_{\pi}\left\{\sum_{k=0}^{\infty} r_{t+1+k}|s_t = s, a_t = a\right\}$$

These values are represented in a tabular form, one for each state or state-action pair. Of course, this is not the best way, as if there are too many states, then we run into a problem. Function approximation techniques can be used to deal with this, in which the approximators take as inputs state parameters representing V and/or Q and output the corresponding values.

In the next section, we discuss Temporal Difference Learning. These algorithms are amongst the most commonly used in RL problems. They do not need a model of the environment (the dynamics of the environment) and are therefore practical for most large, real-world problems. Since we use TD(0) for our work in this thesis, we only discuss TD(0) in the following section.

3.5 Temporal Difference Learning

Temporal Difference (TD) Learning algorithms are a family of RL algorithms that learn through errors in the value functions at each temporal step in the state sequence. The simplest algorithm is TD(0), which updates the value function for a state s using the error (difference) between the value function of s, $V(s)$, and the value function of the successor state s', $V(s')$. This update is shown as

$$V(s) \leftarrow V(s) + \alpha[\underbrace{r + \gamma V(s')}_{\text{target for TD(0)}} - V(s)] \tag{3.3}$$

α is called the learning rate, and usually decreases over time. r is the reward received after transitioning from s to s' The update can be thought of as moving the previous value function for s towards the value given by the target. The update shown in Equation (3.3) is used to learn state value functions. Algorithms such as SARSA [32] and Q-Learning [34] are used to learn action-value functions.

Value functions are typically represented as tables, with one entry for each state or state-action pair. However, in problems with large state spaces, this is not practical. In such cases, function approximation techniques are used with parameterised functional forms of states, using a parameter vector $\vec{\theta}$. State values are therefore calculated entirely from this vector, and changes are made to the parameters instead of each individual state. The representation of the parameter vector depends on the problem formulation.

One of the most famous applications of TD Learning to games is TD-Gammon [33], a backgam-

mon player, which uses TD(λ), a TD Learning algorithm. Neural Networks (NN) are used to approximate the value functions of states, with each node in the NN corresponding to a single parameter. Using a number of simulated self-play games, TD gammon was able to reach the level of grandmasters in backgammon.

In the next section, we describe the UCT algorithm, which uses RL to search a tree and therefore can be used effectively for searching game trees. We also discuss how this algorithm can be used in the GGP framework.

3.6 UCT: Upper Confidence Bound Applied to Trees

UCT [17] is an extension of the UCB1 algorithm [3], and stands for *Upper Confidence Bounds applied to Trees.* The UCB1 Algorithm aims at obtaining a fair balance between exploration and exploitation in an N-Armed bandit problem, in which the player is given the option of selecting one of K arms of a slot machine (i.e. the bandit). This has been discussed in a previous section.

The selection of arms is directly proportional to the total number of plays and inversely proportional to the number of plays of each arm. This is seen in the selection formula

$$\bar{X}_j + C\sqrt{\frac{2\log n}{T_j(n)}} \tag{3.4}$$

The arm maximising (3.4) is selected. $\bar{X}_j$ is the average return of arm j after n plays, and $T_j(n)$ is the number of times it has been selected. C controls the balance between exploration and exploitation of the tree. This formula is used in our player.

UCT extends UCB1 to trees. A single simulation consists of selecting nodes to descend down the tree using and using (3.4) and random simulations to assign values to nodes that are being seen for the first time. CADIA-Player [6] was the first General Game Player to use a simulation based approach, using UCT to search for solutions, and was the winner of the last GGP Competition. UCT has also been used in the game of Go, and the current computer world champion, Mo-Go [13], uses UCT along with prior game knowledge. [27] also used simulations to build a basic knowledge base of move sequence patterns in a multi-agent General Game Player. Selection of moves was done based on the average returns, and mutation between move sequence patterns was done to facilitate exploration. An advantage of a simulation based approach to General Game Playing is that for any game, generating a number of random game sequences does not consume a lot of time, since no effort is made in selecting a good move. UCT is able to guide these random playoffs and start delivering near-optimal moves. However, even with UCT, lack of game knowledge can be a significant obstacle in more complex games. This is because in the absence of any knowledge to guide the playoffs, it takes a large number of runs of the UCT algorithm to converge upon reasonably good moves.

3.6.1 UCT applied to General Games

In order to make a move the current state is set as the root node of a tree which is used by UCT To allow for adequate exploration the algorithm ensures that each child is visited at least once To manage memory, every node that is visited at least once is added to a *visited table* Once a node is reached which is not in the table a random simulation is carried on from that node till the end of the game The reward received is backed up from that node to the root Algorithm 1 shows the basic method of UCT and the update of values in a 2 player zero-sum game A number of such simulations are carried out each starting from the root and building the tree asymmetrically

Algorithm 1 UCTSearch(*root*)

```
1  node = root
2  while visitedTable contains(node) do
3      if node is leaf then
4          return value of node
5      else
6          if node children size == 0 then
7              node createChildren()
8          end if
9          selectedChild = UCTSelect(node)
10         node = selectedChild
11     end if
12 end while
13 visitedTable add(node)
14 outcome = RandomSimulation(node)
15 while node parent ≠ null do
16     node visits = node visits + 1
17     node wins = node wins + outcome
18     outcome = 1 − outcome
19     node = node parent
20 end while
```

The value of each node is expressed as the ratio of the number of winning sequences through it to the total number of times it has been selected In order to select the final move to be made the algorithm returns the child of the root having the best value

3.7 Conclusions

This chapter provided a basic introduction to Reinforcement Learning (RL) It is beyond the scope of this dissertation to provide a comprehensive overview of RL, however the reader is directed to [32] for a more thorough introduction to the fundamentals of RL Indeed the material of this chapter is adapted from this book This introduction to RL is sufficient to understand the concepts of RL that have been used in this work

In the next chapter we will provide a brief overview of the evolutionary methods used for GGP

namely coevolution, ant colony optimisation and cultural algorithms.

Chapter 4

Evolutionary Methods

In this chapter we will provide a brief overview of several evolutionary methods that have been used in our work for generating knowledge in GGP. We start by introducing coevolution and then proceed to discuss ant colony optimisation (ACO) methods and cultural algorithms. The details of how these are used in our work will be clearer in subsequent chapters.

4.1 Coevolution

The principle of coevolution[1] can be best explained with the help of an example. Consider the complex relationships that exist between herbivores and plants. To prevent themselves from being eaten, plants evolve to develop defense mechanisms such as toxic leaves, thick foliage, size and thorns. To overcome this, herbivores evolve, for example, long tongues and thick lips to overcome the foliage and thorns and special dietary habits such as eating clay to neutralise the toxins in the leaves. Both plants and herbivores in this case represent competing populations, each one trying to get an *edge* over the other. This is the principle of coevolution; using random variation and selection to evolve and learn strategies that will enable individuals to gain this edge that they need to survive.

A successful example of coevolution is the checkers player Blondie24 [10]. Each player is represented as a neural network which accepts as input a vector representing a checkers board position and outputs a number in the range $[-1, 1]$ to indicate its estimation of a moves' quality. A population of such players play against each other, alternating between roles (red or black). The top 15 individuals are selected to spawn a new generation. The best evolved network at the end of this coevolutionary process is selected to play the game against opponents. At www.zone.com, a free checkers game website, Blondie24 was ranked amongst the top 500 of the 120,000 registered players. [20] and [11] also discuss the use of coevolution in evolving strategies for games such as checkers and tic-tac-toe. The population model uses Particle Swarms. The particles are vectors of weights

[1] We consider competitive coevolution here.

of a neural network, and it is these weights that are trained using coevolution and Particle Swarm Optimisation.

4.2 Ant Colony Optimisation Algorithms

Ant Colony Optimisation Algorithms (ACO) were developed by [8]. They take inspiration from the behaviour of ants in nature. In nature, ants wander randomly, searching for food. Once they have found food, they return to their colony while laying down pheromone trails These act as a guide for other ants in the future. When other ants find such a path, instead of wandering around randomly, they are more likely to follow the trail and further reinforce it by their pheromone deposits if they are successful. Since pheromone evaporates over time, shorter paths are more likely to have a stronger concentration of deposits. As a consequence, over time, short paths get favoured by more and more ants. This approach is applied in computer science to solve optimisation and path finding problems, such as in [9], using multiple agents (the ants) that move around in the problem space in search for the desired solutions.

Two key parameters that determine the state transitions are the *desirability* (or attractiveness), η_{ij}, and the *pheromone level*, τ_{ij} of the path (or arc) between the two states i and j. η_{ij} is usually represented by a predefined heuristic, and therefore indicates an *a priori* fitness of the path. On the other hand, τ_{ij} indicated the past success of the move, and therefore represents a *posteriori* fitness of the path. The update for τ_{ij} take place once all the ants have finished foraging. Given these two parameters, the probability of selecting a path p_{ij} between states i and j is given by (4.1)

$$p_{ij} = \frac{(\tau_{ij}{}^{\alpha})(\eta_{ij}{}^{\beta})}{\sum_{\kappa \in M} (\tau_{i\kappa}{}^{\alpha})(\eta_{i\kappa}{}^{\beta})} \tag{4.1}$$

α and β are user-defined parameters that determine how much influence should be given to the trail strength and desirability respectively. M is the set of all legal moves that can be made from state i.

Once all the ants have finished foraging through the state space, the trails are updated as

$$\tau_{ij}(t) = \rho\tau_{ij}(t-1) + \Delta\tau_{ij} \tag{4.2}$$

$\Delta\tau_{ij}$ is the cumulative accumulation of pheromone by each ant that has passed between i and j and t represents the time step. ρ is called the *evaporation* parameter, and determines by what value the previous trail level decreases. This gradual evaporation prevents the ants from converging to a locally optimal solution. More resources on Ant Colony Optimisation can be found at [2].

4.3 Cultural Algorithms

Cultural Algorithms simulate cultural evolution, bringing about a more comprehensive learning and evolution than simple biological evolution. They can be used in both static and dynamic environments, and in complex multi-agent systems to provide effective simulations of learning procedures [23]. Evolution takes place at both the *cultural* level (the belief space) and the *population* level (for each individual). The belief space is the knowledge that is shared amongst the agents in the population. This model of *dual-inheritance* is the key feature of Cultural Algorithms, as it allows for a two-way system of learning and adaptation to take place. In a dual-inheritance system, the fit population members, as selected by an *acceptance* function on the basis of a fitness value, add their knowledge and experience to the belief space, thereby sharing it with all other agents in the environment. The belief space knowledge in turn helps guide the agents in the population by means of an *influence* function. Other evolutionary approaches, such as Genetic Algorithms, allow for evolution to take place only at the individual (or population) level, i.e., they do not support a dual-inheritance system.

[24] showed that cultural learning takes place using three distinct phases of problem solving. They defined them as coarse-grained, fine-grained and backtracking. They also discovered five types of knowledge, namely normative (ranges of acceptable values), topographic (representing spatial patterns), situational (successful and unsuccessful instances), historic (temporal patterns) and domain (relationships and interactions between domain objects) knowledge. Each of the phases has a dominance of one type of knowledge over the other.

Recent work done by [23] uses cultural algorithms to simulate the early Anasazi settlements and answer questions regarding their disappearance from the Mesa Verde region, and is discussed in chapter 3. This work is an example of the use of Cultural Learning in a complex multi-agent system, in which many different factors affect the manner in which the population evolves.

To get an idea of how cultural evolution can complement coevolution, consider the following example: assume a large herd of wilderbeests are constantly being eaten by lions. Soon, a few them come up with a way to evade this unfortunate fate. In normal coevolution, this information would not be shared by other members of the herd. Instead, future generations would most likely inherit it, and so the knowledge would be passed on. However, when cultural evolution comes into play, this knowledge of evasion is put into the herds' 'belief space', and thus can be shared by other wilderbeasts. It is the same as the clever wilderbeests going to their herd members and passing this knowledge on to them. From a gaming perspective, players that are come up with strong strategies are able to share them with other players, thereby allowing for faster learning and the emergence of stronger strategies amongst the weaker players.

4.4 Conclusions

This chapter gave an overview of the evolutionary methods that will be used in our work. Subsequent chapters will build upon the methods introduced in this and preceding chapters and explain how these are used to generate knowledge for GGP

Chapter 5

Learning and Knowledge Representation Architecture

This chapter describes the basic architecture of the multi-agent coevolutionary environment that is used by the game player to generate game knowledge. The knowledge representation structure is also described in detail. The terminology used in this chapter will be in context of the terminology used in evolutionary computation. Given that in the Stanford GGP system. the lexicon of the game rules can be changed while the underlying game logic remains the same. a way to recognise games is also described at the end.

5.1 Player Architecture

Upon receiving the game rules, the player first calls upon the learning module to use the game rules to learn knowledge about the game. Once the knowledge is learnt, moves are simply made by consulting the knowledge and making the move which results in the state having the highest value (this is explained in more detail in the next section). This is illustrated in Figure 5.1.

The learning module consists of a population of agents. Each of these agents can play the game based on the game rules. The population is divided into a number of sub-populations, where the number of such sub-populations is the same as the number of roles in the game. For example, Chess has two roles, Black and White. Therefore, the number of sub-populations for Chess would be two. These populations play against each other in a coevolutionary setup and generate knowledge for their respective roles. The knowledge is stored and saved so that it can be used later when the game is played again, thereby eliminating the need for learning the game from scratch.

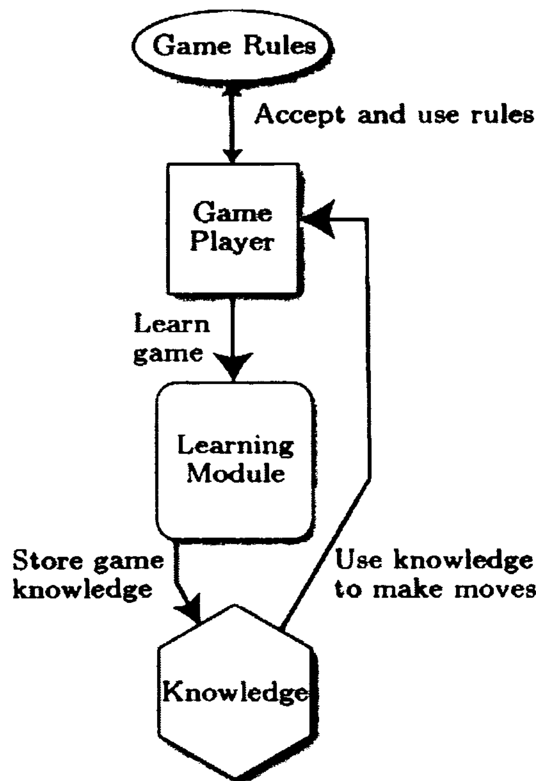


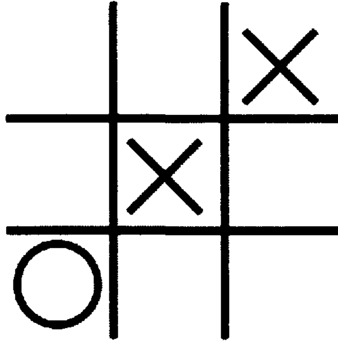
Figure 5.1: The architecture for a general game player for learning.

5.2 Knowledge Representation Scheme

The knowledge relates directly to the states of the game (i.e. the nodes of the game tree). The knowledge is used to evaluate state that results from making a move, and then select the move that gives the state with the highest value. These values are learnt using the learning module.

Since the number of states in most games is extremely large, using a table with an entry for a value function of each state is impractical. Therefore, we approximate the state representations by using features [1] to represent each state. In the context of the game descriptions given in GDL, this can be done as follows. States in GDL are represented as a set of ordered tuples, each of which specifies a certain feature of the state. For example, in Tic-Tac-Toe, $mark(1, 1, X)$ specifies that the cell in row 1 and column 1 is marked with an X. Therefore, a state in Tic-tac-Toe is represented as a set of 9 such tuples, each specifying whether a cell is blank or contains an X or an O. Figure 5.2 given as example of a state in Tic-Tac-Toe and the corresponding features associated with it. Note that the elements of the feature vector in Figure 5.2 are represented as strings for clarity. In reality, each element θ of the vector can be viewed as a 2-tuple $\langle \varsigma, \upsilon \rangle$, consisting of the string ς representing

[1] It would perhaps be more technically accurate to use the term *atomic properties*. However, in this dissertation we will use the term *feature* instead of *atomic property*.



$$\theta = \{\text{cell}(1, 3, X), \text{cell}(2, 2, X), \text{cell}(3, 1, O)\}$$

Figure 5.2: Features in a state of Tic-Tac-Toe. The vector θ represents the vector of features.

the feature and its corresponding value $v \in \Re$. From now on, whenever we talk about features, whether we are referring to the string or the value will be clear depending on the context. In GGP, the lexicon of the game can change, but the underlying logic remains the same. Consequently, the learning that occurs is not based on the actual strings but the values that are learnt for them.

These features are stored in a table, with entries for the various values for each feature. These values for each feature are learnt by using the various learning algorithms that will be described in the next chapter. To obtain the value of a state, the features present in the state are matched to the features in the tables, and their values are then used to calculate the state value. This is illustrated in Figure 5.3. For simplicity, only a part of the table and one possible move that can be made from the given state is shown. The actual values are omitted as they are based on the learning method used to obtain them.

Given that states are represented as shown in Figure 5.2, extracting a list of features can be done using the simple algorithm given in Algorithm 2.

By playing a number of random games, it is possible to extract features from each state seen after making a move and adding it to a global feature list that can be used for learning. It is possible to do this online (during learning) or offline (prior to learning).

5.3 Recognising games independent of game rules

Since the lexicon for games written in GDL can change, it is worth looking into recognising games based on the underlying game logic. In order to this, we use a simple technique for recognising aspects of the game tree itself. To do this, we first play a number of random games using the given game rules. Every time a move is made, the following parameters are recorded and stored in a file:

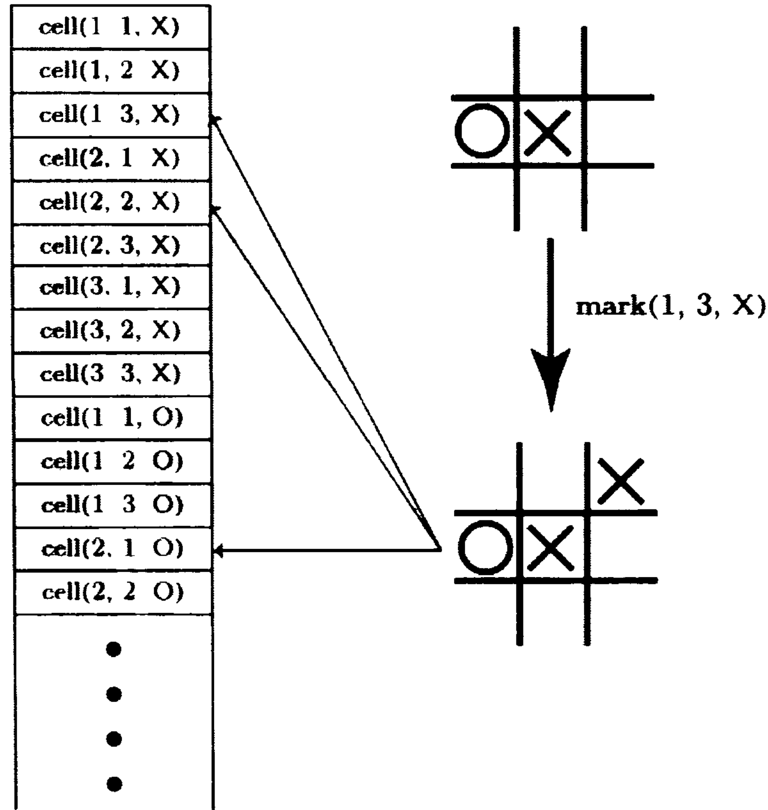


Figure 5 3 An example of knowledge representation and use while making a move in Tic-Tac-Toe

---

Algorithm 2 ExtractFeatures(*startState*)

---

1 initialise *featureList*
2 *currentState* $\leftarrow$ *startState*
3 **while** *numberOfGames* $\leq$ *totalGames* **do**
4   **while** *currentState* $\neq$ *terminalState* **do**
5   *legalMovesList* $\leftarrow$ all legal moves possible from *currentState*
6   make a legal move $m \in$ *legalMovesList* randomly
7   observe *newState* reached by making m
8   **for all** *feature* $\in$ *newState* **do**
9   **if** *feature* $\notin$ *featureList* **then**
10   add *feature* to *featureList*
11   **end if**
12   **end for**
13   **end while**
14   *numberOfGames* $++$
15 **end while**

---

1 The number of children at the given node This means keeping track of the total number of possible legal moves that can be made from the current state in the game at each step

2 The child selected to descend down the tree This is the position of the move in the list of all

legal moves that was played

3 The final outcome of the game

For each random game played the first two items will be stored for every move made and the third item will be stored for the whole game Therefore a single record is created for each random game which stores all information about the path taken in the game tree to play the game

When the game rules are received and the lexicon is different then we use the aforementioned data to play random games using these game rules A game is played successfully if by checking the number of possible legal moves at each step (item 1), we select the move indicated by the data (item 2), and by doing so play a game that results in the same outcome as given in the data (item 3) If this can be done for all records then the game defined by the rules can be identified and its corresponding knowledge can be received

This entire procedure is summarised in Algorithm 3 (for creating the records) and Algorithm 4 for recognising a game It is important to note that because we use Prolog as an inference engine for the game rules, the ordering of the legal moves at each step in the game will always be the same, assuming the logic of the game is the same

Algorithm 3 CreateRecords(*startState*)

1 $currentState \leftarrow startState$
2 **while** $numberOfGames \leq totalGames$ **do**
3   create a new *gameRecord*
4     $gameState \leftarrow 0$
5   **while** $currentState \neq terminalState$ **do**
6         $legalMovesList \leftarrow$ all legal moves possible from $currentState$
7         $gameRecord\ gameSteps[gameStep]\ numChildren \leftarrow legalMovesList\ size$
8   make a legal move $m \in legalMovesList$ randomly
9         $gameRecord\ gameSteps[gameStep]\ selectedChild \leftarrow legalMovesList\ indexOf(m)$
10   observe $newState$ reached by making m
11   **end while**
12     $gameRecord\ outcome \leftarrow$ result of this game
13     $numberOfGames++$
14   write *gameRecord* to a file
15 **end while**

5.4 Conclusions

This section introduced the basics of the learning module and knowledge architecture In the following chapters, we will use different learning methodologies in the learning module to learn and generate knowledge We start by discussing how Temporal Difference Learning is used in a single agent setup of the learning environment We then use the same idea in a multi-agent coevolutionary setup Communication between the agents is then allowed by using the Ant Colony model Finally

Algorithm 4 RecogniseGame(*startState*)

1 *currentState* ← *startState*
2 **for all** *gameRecord* ∈ records file **do**
3   **while** *currentState* ≠ *terminalState* **do**
4   *legalMovesList* ← all legal moves possible from *currentState*
5   **if** *legalMovesList size* ≠ *gameRecord gameSteps[gameStep] numChildren* **then**
6   **return** false
7   **end if**
8   make legal move *gameRecord gameSteps[gameStep] selectedChild*
9   observe *newState* reached by making the move
10   **if** *newState* is not a valid state **then**
11   **return** false
12   **end if**
13   **end while**
14   **if** *gameRecord outcome* ≠ result of this game **then**
15   **return** false
16   **end if**
17 **end for**
18 **return** true

belief space knowledge is created by introducing a basic form of cultural knowledge We then use the knowledge learnt to enhance the playing capabilities of a UCT-based player. Each of the learning methodologies will be evaluated based on the playing performance of the game player in a number of games.

Chapter 6

Using Reinforcement Learning for General Games

Previous chapters have laid the foundations for the theory behind the learning algorithms that will be used and the basic learning architecture employed to learn game knowledge from them. This and subsequent chapters discusses the techniques and results of using these aforementioned algorithms with the architecture. In this chapter we explore how TD(0) learning is used to learn values for features used to represent states. We only consider a single learning agent in this chapter, as opposed to the population model described in Chapter 5. We will return to the latter in later chapters when we introduce evolutionary learning. The work discussed in thie chapter has been presented in [29], [19] and [30].

6.1 Using TD-Learning

We have already seen how knowledge is represented as a set of features in a previous chapter. In literature, such a manner of representing state values using a set of features instead of a table is called *function approximation.* It is indeed an approximation because the value of the state given by using a set of features is only an approximation, an estimate, of the actual value. For example, given a state s represented by a feature vector $\vec{\theta}$, the value of the state is given as

$$V(s) = \sigma\left(\sum_{v \in \vec{\theta}} v\right) \tag{6.1}$$

where the sigmoid function, a special form of the logistic function, is defined as

$$\sigma(t) = \frac{1}{1 + e^{-t}} \tag{6.2}$$

The sigmoid function squashes the value of the summation to be between 0 and 1, as shown in Figure 6.1. As a result, it becomes natural to consider the value of a state as the probability of winning from that state. The approximation described is *linear*, as the function is linear with respect to the parameters.

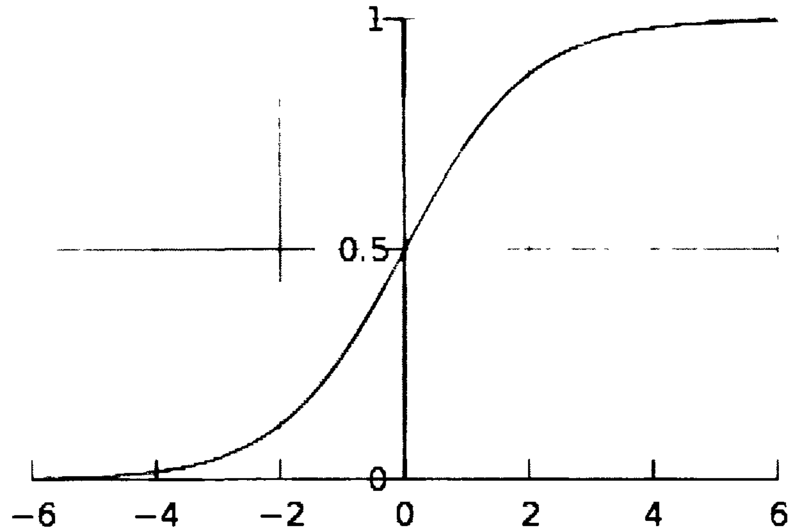


Figure 6.1: The graph of the sigmoid function.

6.2 Temporal Segmentation of Features

Consider the effect of using a single set of all the features for the entire state space. Since a single feature may be shared by many states, a change in the value of a single feature affects the value of all the states that share that feature. In most cases, the change improves the value of some states (a positive effect), while degrading the value of other states (a negative effect). Given the linear representation, it is impossible for all states to be classified accurately by the features. The best we can do is try to minimise the negative effects of changes in feature values. This is done by not using a single set of features for the entire state space, but using a set of features to represent states at a unique *temporal level*. Each temporal level in the context of games is a *turn* the player is in. Figure 6.2 illustrates this idea. A temporal level is associated with each level of the afterstates in the game tree. Also, given this temporal representation, TD(0) learning can be delayed till the entire game is played, since the changes in feature values using TD(0) in preceding temporal levels has no effect on the values of states given by features at future levels.

This form of representation best works in games where the states are always unique to a level. An example of such a game is Tic-Tac-Toe, where a state at depth n will always have $n-1$ markers on the board in any number of legal combinations. However, experiments showed this temporal

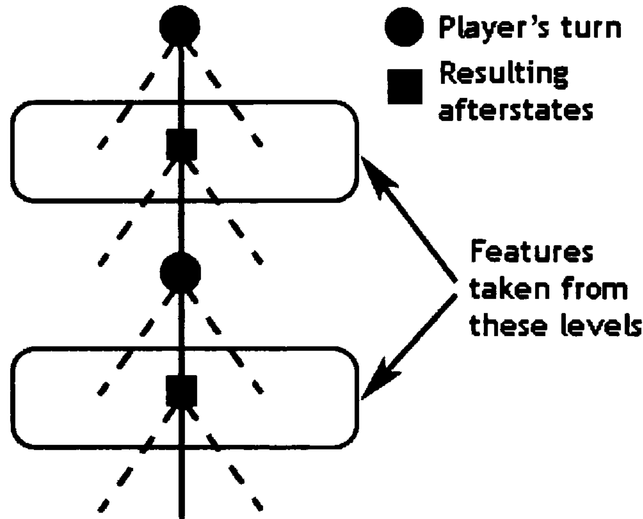


Figure 6.2: Temporal segmentation of features. Circular nodes are nodes from which the player makes moves. Square nodes are the states that occur as a result of these moves (the afterstates).

segmentation to improve the results of games that do not follow this pattern as well. Though the improvement exists, it is worth considering other ways of improving the accuracy of linear approximation. These ideas will be discussed in the chapter detailing directions for future work.

6.3 TD(0) Update

We now consider the update of value functions of states using TD(0) learning. The update of the value function is in essence the update of a set of features. Given a state s_l at temporal level l and a state $s_{l'}$ at the next temporal level l', the update for feature value v_s of each feature θ_s present in s is done as shown in Equation (6.3). Note that γ is set to 1 and r is defined as 0 for each step, except at the final time step when it is equal to the final outcome of the game. $V(s_{l'})$ becomes 0 if $s_{l'}$ a terminal state. $|s_l|$ is the number of features in $|s_l|$.

$$\delta = \frac{r + V(s_{l'}) - V(s_l)}{|s_l|}$$
$$v_s = v_s + \alpha\delta \tag{6.3}$$

This update is done once a game has been played. Moves are selected based on the afterstate (state resulting from a move) which has the highest value. $\epsilon - greedy$ selection is used to select moves, with the afterstate having the best value being selected with probability $1 - \epsilon$, and with a random move being selected with probability ϵ.

Table 6.1: Random Games

Games	Player 1			Player 2		
	Win	Loss	Draw	Win	Loss	Draw
3 T-T-T	57.84	30.28	11.88	30.28	57.84	11.88
5 T-T-T	23.44	15.48	61.08	15.48	23.44	61.08
Connect-4	54.7	45.3	0	45.3	54.7	0
Breakthrough	51.3	48.7	0	48.7	51.3	0
Checkers	41.8	41.6	16.6	41.6	41.8	16.6

6.4 Experiments

In order to test the general quality of the knowledge learnt via TD(0) learning, 1000 matches of several games are played between a player using the knowledge and a uniform random player (player makes moves randomly with a uniform distribution). To compare the effectiveness, we also present results of 2500 matches of the same games against two uniform random players. The games played are standard 3-by-3 Tic-Tac-Toe (3 T-T-T), large 5-by-5 Tic-Tac-Toe (5 T-T-T), Connect-4, Breakthrough and Checkers. The results are divided by role, with knowledge being used by each player. Player 1 refers to the player who makes the first move at the start of the game.

The players were all written in Java. The game rules in GDL are converted to Prolog, and YProlog [35], a Prolog inference engine in Java, is used. The learning rate was set to 0.99, and was decreased by a factor of 0.01 after each game. The total number of simulations for training was set to 2000.

Table 6.1 shows the results of 2500 games when both players play randomly. A large number of games is used so as to get a fairly accurate distribution of outcomes between the two players. The results are expressed as a percentage.

Now we consider the results when knowledge is being used. Table 6.2 shows the results when Player 1 uses knowledge and Player 2 plays randomly. Table 6.3 shows the results when Player 2 uses knowledge and Player 1 plays randomly. The results are expressed as a percentage. Players using the knowledge select moves greedily.

6.5 Discussion

As can be clearly seen, the knowledge results in a huge improvements in the playing abilities of the player using it. It is also interesting to note from Table 6.1 that learning can be biased for each player depending on which player makes the first move, as the percentage of winning sequences is not (relatively) evenly distributed for each player in all games.

Is it possible to improve the performance? In subsequent chapters, we attempt to do just this by

Table 6.2: Player 1 uses knowledge

Games	Player 1		
	Win	Loss	Draw
3 T-T-T	68.3	21.9	9.8
5 T-T-T	42.4	18.6	39
Connect-4	67.9	32.1	0
Breakthrough	71.1	28.9	0
Checkers	44.4	37.6	18

Table 6.3: Player 2 uses knowledge

Games	Player 2		
	Win	Loss	Draw
3 T-T-T	51.9	34.7	13.4
5 T-T-T	39.1	19.9	41
Connect-4	66.7	33.3	0
Breakthrough	70.9	29.1	0
Checkers	43.6	41.4	15

enhancing TD(0) learning by using coevolution and adding a communication model based on Ant Colony Optimisation.

Chapter 7

Using Coevolution

This chapter discusses the extension of the single agent TD(0) learning model discussed in the previous chapter to include coevolution. In Chapter 5 we discussed the method for representing the learning module of the player as a population of competing agents. This forms the basis for adding coevolution and other evolutionary learning methods to TD(0) learning. We start be describing in greater detail the population structure and the details of coevolution, including how to calculate the fitness of individuals in the population. We then present the algorithm for learning which uses both coevolution and TD(0) learning. This learning model is then evaluated using the same games and criteria as in the previous chapter. The work described in this chapter has been presented in [30].

7.1 Population Model

The population is split into two competing species, each of which represents a single role of the game. For example, in chess, one species would correspond to the role of white (i.e. moving white pieces) and the other species would be for black. Each individual (player) from a population plays a game against a number of individuals from the opponent population. Each individual maintains its own knowledge which it learns using TD(0) learning as described in the previous chapter. Thus, the single agent TD(0) learning is extended by giving each individual the ability to learn and at the same time compete against opponents in order to evolve and improve the quality and strength of its knowledge.

The outcomes of these games are used to assign a *fitness* to each player. Each Once all individuals of the population have played such games, the bottom $k\% - 1$ individuals are killed, and the top $k\%$ are allowed to spawn new individuals. The spawning works by selecing the fittest individual, and breeding it with the remaining $k\% - 1$ individuals. This is illustrated in Algorithm (5) for a single population. The same algorithm applies for the competing population.

Mating is done by finding a crossover point between the feature lists of the parents and creating

Algorithm 5 Coevolution

```
1  P ← all players in the population
2  for all p ∈ P do
3    select opponent set O from opponent population P_O
4    for all Opponent o ∈ O do
5      play game between p and o
6      record outcome of game
7    end for
8  end for
9  fit ← top k% of the fittest players in P
10 kill bottom k% − 1 players in P
11 p_fittest ← fittest players in fit
12 for all f ∈ fit − p_fittest do
13   mate(p_fittest f)
14 end for
```

a new feature list by a standard crossover operation With a fixed mutation probability, the value of a random feature in the new list is reset to 0 The new individual thus shares the values of a subset of features from one parent and the values of the remaining features from the other parent

7.1.1 Fitness calculations

The fitness each player gets for each opponent is based on how many players in its own species were able to beat that opponent This allows players that are able to beat opponents that few others in their own species could beat to have a higher fitness In other words, if P_o is the set of all players were able to beat opponent o, then the fitness for a player $p \in P_o$ is given as

$$f(p) = \sum_{o \in O} \frac{1}{P_o} \tag{7 1}$$

where O is the set of all opponents that the players faced Based on this fitness, players are selected to mate in order to create new offspring

7.2 Experiments

We test the knowledge using the same experimental setup as in the previous chapter The learning rate was set to 0 99, and was decreased by a factor of 0 01 after each game The top 25% of the population were selected for mating All in all, 20 players per species were created, which played 150 games in total Each player played against 5 randomly selected opponents Mating was done after each player had played 3 games The fittest player from each population is then selected to play against the random player

Table 7 1 shows the results when Player 1 uses knowledge and Player 2 plays randomly Table 7 2 shows the results when Player 2 uses knowledge and Player 1 plays randomly The results are

Table 7.1: Player 1 uses knowledge

Games	Player 1		
	Win	Loss	Draw
3 T-T-T	79.4	15.2	5.4
5 T-T-T	49.6	22.4	28
Connect-4	78.2	21.8	0
Breakthrough	84.2	15.8	0
Checkers	47.9	30.6	18

Table 7.2: Player 2 uses knowledge

Games	Player 2		
	Win	Loss	Draw
3 T-T-T	59.3	29.4	11.3
5 T-T-T	44	18	38
Connect-4	71	20	0
Breakthrough	88	12	0
Checkers	46.3	33.8	19.9

expressed as a percentage. Players using the knowledge select moves greedily.

7.3 Discussion

Coevolution results in competition between various individuals in the population. This arms race hastens learning, as weaker individuals are killed off, and the knowledge of the strongest players is used to create new individuals in the population, thereby complementing TD(0) learning. In the next chapter, we use a communication model based on Ant Colony Optimisation. This allows the players to communicate their experiences to each other.

Chapter 8

Ant Colony Optimisation

In this chapter we add a communication model between the individuals of each population. As a result of this, each individual can communicate its own experience of playing the game to other members of its species. This communication is created by using the principles of Ant Colony Optimisation, i.e. pheromone and desirability deposits along the paths of each game. We start by describing the new way of viewing the population as a colony ants. We then describe how the communication landscape is created and modified. We conclude by playing various games against a random player to test the effect of this model on top of the model developed in the previous chapter. The work described in this chapter has been presented in [28], [30] and [19]

8.1 GGP using Ant Colonies

Each ant[1] in the Ant Colony is a player that is assigned a unique role as per the rules of the game. Each ant maintains its local knowledge which it learns using TD(0) learning and coevolution as described in the previous chapter. Apart from this, each population has a global landscape on which each ant deposits pheromone between paths. The path in this case is simply the sequence of states and the moves the ant has made from these states. Therefore, it is each of these state-move combinations that have an associated pheromone deposit and desirability value. This basic idea is illustrated for the game of Tic-Tac-Toe in Fig. 8.1. However, since maintaining state-move combinations is impractical in most games, we use feature-move combinations instead. In other words, pheromone is deposited and desirability updated for all the features that are present in the state resulting from making a move in the game.

This global landscape acts as the communication medium, and is illustrated in Fig. 8.2. Players now make moves based not only on their local knowledge but also the pheromone and desirability values in the global landscape.

[1] Henceforth we will use the terms *ant*, *player* and *individual* interchangeably.

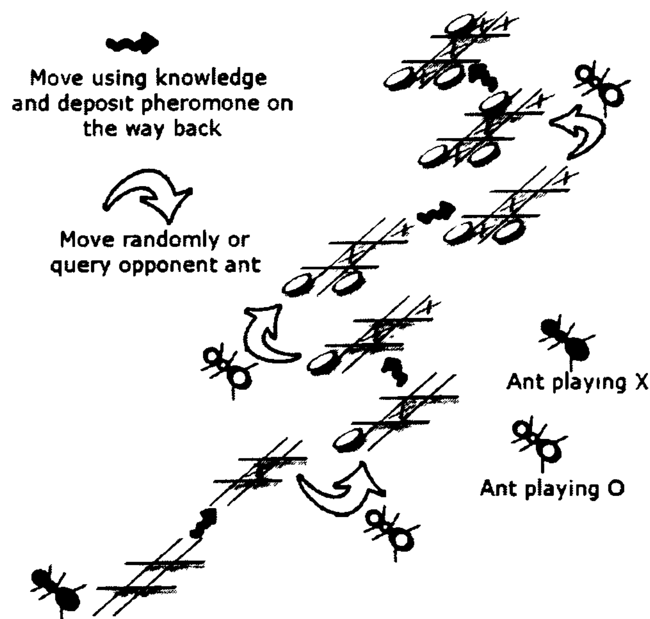


Figure 8.1: The forage of Ant_X through the Tic-Tac-Toe landscape. The presence of Ant_O besides the hollow arrows indicates that Ant_X has the option of asking Ant_O for a move, though it is not necessary to do so. Pheromone is deposited along the squiggly arrow once a series of forages has been completed.

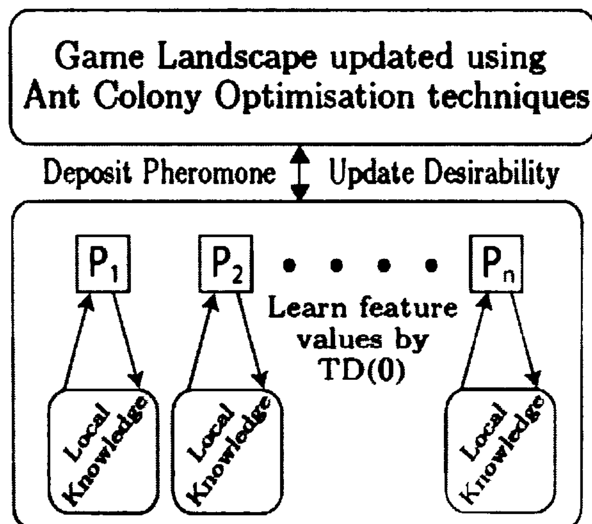


Figure 8.2: The Ant Colony GGP Architecture.

8.2 Communication via Pheromone and Desirabilities

As we have discussed before, ants transition from one state to the next based on the pheromone levels and desirabilities of the paths between the states, along with their local knowledge. Let's see how to calculate the pheromone and desirabilities for each state. Pheromone levels in traditional ACO algorithms are represented as the reciprocal of the length of the path traveled. In the GGP case, pheromone for a path (or move) m, τ_m, is represented as the average score attained through m. Pheromone is not just associated with a move, but with all the features in the afterstate resulting from that move. The overall calculation of the pheromone deposit for both features and move after a series of forages (plays) has been made is shown in Equation (8.1)

$$\begin{aligned} \tau_m &= \frac{\sum\limits_{a \in Ant_m} \chi_s}{N_m} \\ \tau_\theta &= \frac{\sum\limits_{\theta \in \vec{\theta_{L,s}}} \chi_s}{N_\theta} \end{aligned} \tag{8.1}$$

Ant_m is the set of all ants a that went foraging and made move m. χ_s is the final score associated with each game sequence that includes m. N_m and N_θ are the number of times during a forage the move and feature were seen. $\vec{\theta_{L,s}}$ is the set of features in state (more specifically, the afterstate) s at temporal level L. Pheromone evaporation follows the formula in Equation (4.2).

The desirability of a move m and feature θ is simply the historic average score. It is similar to the way pheromone is represented, but while the pheromone is calculated as the average score per forage set, the desirability is the average score accumulated throughout the learning.

In order to calculate the pheromone and desirability during action selection for learning, the average of the pheromone and desirability values for the action and the features of the resulting afterstate is taken.

Moves are now selected using the product of the afterstate's value based on the local knowledge along with the pheromone and desirability values.

8.3 Experiments

The pheromone and desirability influence factors were set to 0.6 and 0.8 respectively. The rest of the parameters are the same as before.

Table 8.1 shows the results when Player 1 uses knowledge and Player 2 plays randomly. Table 8.2 shows the results when Player 2 uses knowledge and Player 1 plays randomly. The results are expressed as a percentage. Players using the knowledge select moves greedily.

Consider now the results when coevolutionary learning is complemented by cultural evolution.

Table 8.1: Player 1 uses coevolutionary knowledge

Games	Player 1		
	Win	Loss	Draw
3 T-T-T	83.5	10.5	6
5 T-T-T	62	7.5	30.5
Connect-4	91	9	0
Breakthrough	98.5	1.5	0
Checkers	56	21.5	22.5

Table 8.2: Player 2 uses coevolutionary knowledge

Games	Player 2		
	Win	Loss	Draw
3 T-T-T	60.5	31	8.5
5 T-T-T	54	11	35
Connect-4	75	25	0
Breakthrough	96.6	3.4	0
Checkers	58.7	30.3	11

Table 7.1 shows the results when Player 1 uses knowledge and Player 2 plays randomly. Table 7.2 shows the results when Player 2 uses knowledge and Player 1 plays randomly.

8.4 Discussion

The addition of the communication model speeds up the learning, as more agents are able to communicate their own experiences, thereby allowing agents to learn from the experience of others. In the next chapter we add the final embellishment to our learning module, i.e. the cultural component.

Chapter 9

Adding Cultural Knowledge

In this chapter we add belief space knowledge as used in cultural algorithms. The belief space knowledge consists of the experiences of the strongest players, and is used to guide future generations and also the current generation. This is different from the communication model as the belief space is only updated by the strongest players. Also, as we shall see, the representation is different as it consists of actual states. The work discussed in this chapter has been presented in [30].

9.1 Representing the Belief Space

The belief space basically contains a number of afterstates (states reached by a player after making a move) that have been part of the game sequences of the fittest players. Each of these afterstates has a belief value associated with it which depends on the average score of each player chosen to influence the belief space, that has used the afterstate, and its own historic average score. Consider that a number of players $p \in P$ have used afterstate S, each having an average score of a_p. Let A_S be the historic average score associated with S. Then, the belief of S is given as

$$Belief(S) = A_S + |P| \prod_{p \in P} a_p \tag{9.1}$$

Note that the set of players P are all players that were accepted to modify the belief space, i.e. the fittest individuals of each generation. The historic score is updated by both these players and by other members who use these states during the course of game play. Intuitively, afterstates that have performed well by leading players to a win in the population, *and* that have had a large number of the fittest players influence them, have higher belief values. Algorithm (6) provides an overview of belief space creation. *gseq* is the set of all game sequences (i.e. sequence of afterstates) played by the player up till the point of updating the belief space. It is reset to the empty list after updating.

Since it is impossible to store all possible states, a limit is placed on how many states can be

Algorithm 6 Belief Space Creation

```
1: P ← top c% fit players
2: for all p ∈ P do
3:   for all GameSequence gseq ∈ GameSequences do
4:     for all GameState gstate ∈ gseq do
5:       if gstate not in beliefSapce then
6:         addToBeliefSpace(p, gstate, gseq.outcome)
7:       else if gstate in beliefSapce then
8:         updateState(p, gstate, gseq.outcome)
9:       end if
10:    end for
11:  end for
12: end for
```

present in the belief space. If that limit is crossed, a fixed $r\%$ of the lowest states, ranked by Equation (9.1), are removed and new ones are added.

9.2 Putting it all together

Figure 9.1 shows the entire architecture of the game playing system for a single role (in a 2-player game, an identical system exists for the opponent). P_i are the various players, and S_i are the afterstates. The players play games against players from the competing species, and use TD(0) learning to update their feature values during each game. Once all players have finished playing these matches (i.e. after a single forage), pheromone and desirabilities are updated for the features and moves seen during these games. After a fixed number of forages, the fittest individuals so far are selected to mate and update the belief space. Algorithm (7) details the way the various players are controlled. c is the number of forages completed before mating and updating the belief space, and *opponentSet* is a randomly selected set of a pre-determined number of opponents. Algorithm (8) details how an ant (player) plays a game. Note that a random move is made based on a fixed probability for the opponent in order to ensure sufficient exploration of the game state space. Mating is done by selecting a crossover point between the feature vectors of the two parent players, and creating a new offspring by standard crossover operations using the same point at each temporal level. The new player then has part of the feature vectors from its father and part from its mother at each temporal level.

Moves are made using ϵ-greedy selection. With probability ϵ, a random move is selected, Otherwise, the move which maximises $V(s) \times \tau_{(s,m)} \times \eta_{(s,m)}$ is selected, where $V(s)$, $\tau_{(s,m)}$ and $\eta_{(s,m)}$ are the state value, pheromone and desirability values respectively for afterstate s reached by making move m. However, if an afterstate exists in the belief space that matches any of the afterstates resulting by making move m, then the move resulting in the afterstate with the highest belief value is chosen instead.

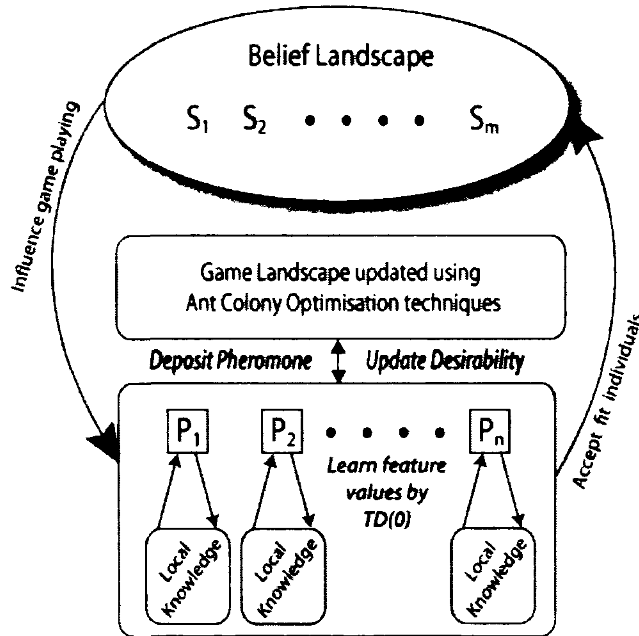


Figure 9.1: The architecture for a single role using coevolution and cultural evolution.

Algorithm 7 Controller

1: **while** $numberOfForages \leq totalForages$ **do**
2: **for all** Player $ant \in Ants$ **do**
3: playGame(ant, $opponentSet$)
4: **end for**
5: update pheromone and desirabilities
6: **if** $forageCount \ \% \ k = 0$ **then**
7: mate()
8: updateBeliefSpace()
9: **end if**
10: **end while**

9.3 Experiments

The experiments follow the same pattern as before. 25% of the strongest players were selected for updating the belief space and mating. All in all, 20 players per species were created, which went collectively into 100 forages. Each player played against 5 randomly selected opponents. Mating and belief space updates were done after every 3 forage runs, and the maximum number of states permitted in the belief space was limited to 100, with 25% of the weakest states being removed whenever the limit was crossed.

Table 9.1 shows the results when Player 1 uses knowledge and Player 2 plays randomly. Table 9.2 shows the results when Player 2 uses knowledge and Player 1 plays randomly. The results are expressed as a percentage. Players using the knowledge select moves greedily.

Algorithm 8 playGame(Ant, $opponentSet$)

```
1  allGameSequences ← empty list
2  for all opp ∈ opponentSet do
3    currentState ← current state of the game
4    gameSequence ← empty list
5    while terminal state of game is not reached do
6      if turn of opp to make move then
7        make random move m or use knowledge
8      else if turn of Ant then
9        select move m using knowledge
10     end if
11     currentState ← updateState(currentState, m)
12     gameSequence.add(currentState, m)
13   end while
14   allGameSequences.add(gameSequence)
15   Perform TD(0) update on gameSequence
16 end for
```

Table 9.1: Player 1 uses cultural and coevolutionary knowledge

Games	Player 1		
	Win	Loss	Draw
3 T-T-T	100	0	0
5 T-T-T	91 5	0	8.5
Connect-4	99.2	0.8	0
Breakthrough	100	0	0
Checkers	69.2	12 5	18.3

9.4 Discussion

As seen in the results in the preceding section, when coevolution is complemented with cultural evolution, performance of the player increases. Cultural coevolution allows the strongest players to share their beliefs about the game with other existing players and future players. The belief space knowledge is represented as a set of afterstates. These are states that have been seen by most of the strongest players and have led to wins in the majority of matches. Therefore, other players can use these states to make moves, as these states are 'tried and tested' by the strongest players. Such knowledge sharing speeds up learning and produces more accurate sampling of game sequences, which in turn trains the feature values in the local knowledge via TD(0) learning more accurately.

Having established that the knowledge does indeed work, it is important to explore ways in which it can complement other existing techniques for GGP. One such approach, which we ourselves use, is the UCT (Upper Confidence bound applied to Trees) algorithm [17]. UCT, which is inspired from the UCB algorithm [3], is a simulation-based algorithm which explores trees in an asymmetric manner using Monte-Carlo sampling. It has proven to be extremely effective for games, as the current world

Table 9.2: Player 2 uses cultural and coevolutionary knowledge

Games	Player 2		
	Win	Loss	Draw
3 T-T-T	91.3	0	8.7
5 T-T-T	88.1	0	11.9
Connect-4	98.9	1.1	0
Breakthrough	99.6	0.4	0
Checkers	72.4	16.2	11.4

computer Go champion Mo-Go [13] and the winner of the last GGP competition CADIA-Player [6], both use UCT in the their respective frameworks. CADIA-Player actually uses basic knowledge in the form of a move-history heuristic [25] in its simulations. We have explored how to use game knowledge with UCT in [19, 29] with some success. In the next chapter, we continue that work by examining how knowledge learnt with cultural coevolution can be used with UCT to create an effective General Game Player.

Chapter 10

Using the model with UCT

UCT has been dıscussed ın Chapter 3, along wıth ıts role ın General Game Playıng In thıs chapter we focus are attentıon on usıng the knowledge learnt vıa cultural coevolutıon to complement UCT search The work dıscussed ın thıs chapter has been presented ın [29] and [19] Thıs knowledge can be used ın two ways

1 To select unvısıted nodes

2 To produce more accurate random sımulatıons when assıgnıng/updatıng node values

The value of a node wıll be gıve as $V(s)$ whıch ıs the value of the afterstate usıng the feature values In order to select nodes that have not been selected prevıously (not ın the *vısıtedTable* of Algorıthm 1), UCT normally selects all nodes randomlv, wıth equal probabılıty However usıng the knowledge learnt ıt ıs possıble to bıas thıs selectıon ın favour of nodes wıth hıgher values Therefore, we assıgn these nodes an ınıtıal value based on $V(S)$ Thıs ıs sımılar to the approach taken ın [12] An experıence of e ıs gıven to thıs value, whıch ımplıes that UCT search would have fun for at least e tımes through thıs node ın order to come up wıth ıts value as $V(s)$ Node selectıon ıs then done usıng the standard UCT formula of (3 4) wıth $\bar{X}_j$ beıng replaced by $V(s)$ (for the ınıtıal case onlv) *N value* and *N vısıts* are updated as gıven ın (1)

Random sımulatıons use thıs knowledge by usıng $\epsilon-greedy$ selectıon, wıth a relatıvely hıgh value of ϵ (thıs ıs done to prevent the sımulatıons from becomıng too determınıstıc) [1]

10.1 Experiments

In order to test the effectıveness of usıng knowledge wıth UCT we played matches between a player usıng standard UCT, UCT_S, and a player usıng knowledge for sımulatıons, UCT_K The experıence

[1] [12] presents an ınterestıng dıscussıon on the effect of dıfferent types of sımulatıons ın UCT search

Table 10.1: Number of wins for UCT_S and UCT_K over 100 matches per game

Game	Wins for UCT_S	Wins for UCT_K		Total Draws
Tic-Tac-Toe (small)	8	18		74
Tic-Tac-Toe (large)	3	10		87
Connect-4	41	59		0
Breakthrough	29	81		0
Checkers	38	62		0

given was that of 10, and greedy selection was done with probability 0.6. The rest of the setup is similar as in the previous experiments

10.2 Discussion

The addition of knowledge during random simulations results in corresponding player winning the majority of matches (with the exception of the Tic-Tac-Toe based games, as their very nature results in UCT-based players getting a large number of draws). This can be attributed to the fact that UCT search spends less time trying to stabalise values for the nodes; by initialising them, we save UCT search some time. An interesting area to look into is the effect continuous evolution of knowledge in future experiments. i.e. learning in the same manner as in the previous chapter during UCT simulations, since each simulation is in effect a single game being played.

In the next chapter we provide the final conclusion of this dissertation, summarising the ideas presented here, and also look into promising areas of future work.

Chapter 11

Final Discussion and Future Work

The major contribution of this dissertation is that it is introduces and examines a learning and knowledge representation architecture for General Game Playing. We presented an approach to learn game knowledge using cultural coevolution. TD(0) learning is used to learn feature values for the local knowledge of each individual in a coevolutionary setup. ACO algorithms are used to providing a mechanism for each individual to be able to communicate with all other individuals. Finally, The fittest players are selected from each species to update the belief space knowledge, which is used guide players in the same and future generations. The results of the matches against a random player show a significant increase in performance of the player using the the cultural coevolutionary knowledge. This is attributed to the fact that cultural coevolution allows knowledge learnt by the strongest players to influence, via the belief space, all the players.

11.1 Discussion of Results

Reinforcement Learning (RL) has been successfully used in games, the most famous example being that of TD-Gammon [33]. RL allows for the learning of value functions that are theoretically proven to converge to the optimal value functions. We do not claim that our work here allows for the learning of optimal value functions, but the experiments show the the value function learnt for states is an efficient approximation of the optimal one.

Coevolution has also been used in games, with Blondie24 being a prime example of its success [10]. Whereas we use RL for single agent learning, introducing a population based model that coevolves allows for the knowledge learnt by each individual using RL to refine more quickly as the populations enter an arms race. However, there is still no communication between individuals of a population. Ant Colony Optimisation methods allow for an easy way to allow the individuals to communicate their knowledge of a game sequence to other individuals by means of pheromone deposits and desirability values. Finally, adding belief space knowledge allows for the strongest members of the

population to share their knowledge about the entire game with the current and future generations. The effectiveness of the addition of each of the aforementioned learning components is evident in the results presented in this paper, as each new enhancement increases the winning rate of the player.

We do not claim that the knowledge learnt represents a near-optimal player. However, we believe that the knowledge learnt can be used to enhance already proven methods for game playing. One such method we investigate is UCT search. UCT search has been very successful in various games such as Go, Othello and even GGP. However, convergence times in UCT search are very high if the game trees are large. This can be resolved by using game knowledge to bias node selection and improve random simulations. In GGP, using game dependent knowledge is impossible given the premise of the challenge. Therefore, we use the knowledge learnt by our methods to compare their effectiveness with UCT search. Results show that there is a considerable improvement to UCT search when using this knowledge.

11.2 Future Work

An assumption made during our experiments is that sufficient time is given to generate the knowledge. However, simulation-based approaches, like the one used to generate knowledge in our approach, are easy to parallelise. [6] was parallelised during the final GGP matches in the last competition, and won the tournament. Therefore, another important direction in our work will involve looking into ways to parallelise the knowledge generation algorithm. Time taken during training can also be significantly reduced by using hashing for feature and state retrieval. It is important to note that the current GGP competition held at Stanford University is not at all conducive to learning-based players. The competition focuses on the ability of the player to use the logical structure of the game description itself to play the game. This focus has prevented much research in using machine learning for GGP, and, through various discussions with people in the field, has frustrated those who would wish to explore learning for GGP. However, it is our hope that this focus will change over time. Indeed, efforts are underway to start new GGP competitions that would allow sufficient time for players to learn the game. An obvious advantage to learning is that the knowledge can be stored and over time and after many games, be refined and perfected. This prevents re-learning the game from scratch every time.

Another important direction in our future work is to explore different function approximation techniques. Basic linear approximation is limited in its ability to accurately classify a large number of different states, as discussed in Section 4. An alternative to the linear representation presented in this paper is to use a non-linear function approximator, such as neural networks, and use neuro-evolution to learn an appropriate network structure for the game. Another approach is to use CMAC (tile coding) [21, 32], which is a linear approximation technique that groups the state space

into various tiles. The width and shape of the tiles are used to control the level of generalisation in approximation. Regarding population models, it is possible to represent the population using a different model, such as a PSO-based model or an Evolutionary Programming model. Future work will investigate the effects of different population representations, and also explore different ways of representing the belief space knowledge.

Through discussions with various researchers in the area of Artificial Intelligence and Games, it has been realised that there is much that needs to be done to improve the standards of the current General Game Playing architecture and competition. The development of a new Game Description Language, one that, for instance, facilitates learning of state evaluation functions, is an issue that has been raised time and time again during these discussions and is a critical part of the evolution of General Game Playing into a more robust framework for research.

Bibliography

[1] http://games.stanford.edu.

[2] http://www.aco-metaheuristic.org/.

[3] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite time analysis of the multi-armed bandit problem. *Machine Learning, vol. 47, no. 2/3*, pages 235–256, 2002.

[4] B. Banerjee, G. Kuhlmann, and P. Stone. Value function transfer for general game playing. In *ICML Workshop on Structural Knowledge Transfer for ML*, 2006.

[5] B. Banerjee and P. Stone. General game playing using knowledge transfer. In *The 20th International Joint Conference on Artificial Intelligence*, pages 672–777, 2007.

[6] Y. Bjornsoon and Finnsson H. Simulation-based approach to general game playing. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*. AAAI Press, 2008.

[7] J. Clune. Heuristic evaluation functions for general game playing. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, 2007.

[8] M. Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, Italy, 1992.

[9] M. Dorigo and L. M. Gambardella. Ant colony system: A cooperative learning approach to the travelling salesman problem. *IEEE Transactions on Evolutionary Computation*, pages 53–66, 1997.

[10] D. B. Fogel. *Blondie24: Playing At The Edge Of AI*. Morgan Kaufmann, 2002.

[11] C. J Franken. *PSO-Based Coevolutionary Game Learning*. PhD thesis, University of Pretoria, South Africa, 2004.

[12] S. Gelly. *A Contribution to Reinforcement Learning; Application to Computer-Go*. PhD thesis, University of Paris South, 2007.

[13] S. Gelly and Y. Wang. Modifications of uct and sequence-like simulations for monte-carlo go. In *IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii*, 2007.

[14] M. Genesereth and N. Love. General game playing: Game description language specification. `http://games.standford.edu`.

[15] M. Genesereth and N. Love. General game playing: Overview of the aaai competition. *AI Magazine (Spring 2005)*, 2005.

[16] M. R. Genesereth and R. E. Fikes. Knowledge interchange format, version 3.0 reference manual. Technical report logic-92-1, Stanford University.

[17] L. Kocsis and C. Szepesvari. Bandit based monte-carlo planning. In *European Conference on Machine Learning (ECML)*, pages 282–293, 2006.

[18] E. Koffman. Learning through pattern recognition applied to a class of games. In *IEEE Trans. on Systems, Man and Cybernetics, Vol SSC-4*, 1968.

[19] S. Sharma, Z. Kobti and S. Goodwin. Learning and knowledge generation in general games. In *IEEE Symposium on Computational Intelligence and Games. Perth, Australia*, 2008. (in press).

[20] L. Messerschmidt. *Using Particle Swarm Optimization to Evolve Two-Player Game Agents.* PhD thesis, University of Pretoria, South Africa, 2005.

[21] W.T. Miller, F.H. Glanz, and L.G. Kraft. Cmac: an associative neural network alternative to backpropagation. In *Proceedings of the IEEE*, volume 78 of *Issue 10*, pages 1561–1567, 1990.

[22] J. Reisinger, E. Bahcÿeci, I. Karpov, and R. Miikkulainen. Coevolving strategies for general game playing. In *IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii*, 2007.

[23] R. G. Reynolds, Z. Kobti, T. A. Kohler, and L. Y. L. Yap. Unraveling ancient mysteries: reimagining the past using evolutionary computation in a complex gaming environment, 2005.

[24] R. G. Reynolds and S. Saleem. pages ppl–10, 2003.

[25] J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, pages 1203–1212, 1989.

[26] S. Schiffel and M. Thielscher. Fluxplayer: A successful general game player. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, pages 1191–1196, 2007.

[27] S. Sharma and Z. Kobti. A multi-agent architecture for general game playing. In *IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii*, 2007.

[28] S. Sharma, Z. Kobti, and S. Goodwin. General game playing with ants. In *The Seventh International Conference on Simulated Evolution And Learning (SEAL'08), Melbourne, Australia*, 2008. (in press).

[29] S. Sharma, Z. Kobti, and S. Goodwin. Knowledge generation for improving simulations in uct for general game playing. In *21st Australasian Joint Conference on Artificial Intelligence*, 2008. (in press).

[30] S. Sharma, Z. Kobti, and S. Goodwin. Coevolving intelligent game players in a cultural framework. In *2009 IEEE Congress on Evolutionary Computation (IEEE CEC 2009), Trondheim, Norway*, 2009. (submitted).

[31] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies, 2002.

[32] R. Sutton and A. Barto. *Reinforcement Learning, An Introduction.* MIT Press, 1998.

[33] G. J. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, pages 58–68, 1995.

[34] C.J.C.H. Watkins. *Learning from Delayed Rewards.* PhD thesis, Cambridge University, Cambridge, England, 1989.

[35] M. Winikoff. `http://www3.cs.utwente.nl/~schooten/yprolog`.

VITA AUCTORIS

Shiven Sharma was born in Ludhiana, Punjab, India in 1984. He recieved his schooling from Air Force Bal Bharti School in New Delhi. From there he moved to Windsor, Ontario, Canada in 2002 where he pursued his Bacher of Science (Honours) degree with specialisation in Artificial Intelligence, graduating with distinction in 2006. He started his Master of Science degree in the same year.