Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Omkar Jayant Tilak

Entitled
Decentralized and Partially Decentralized Multi-Agent Reinforcement Learning

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Dr. Snehasis Mukhopadhyay
Chair

Dr. Mihran Tuceryan

Dr. Luo Si

Dr. Jennifer Neville

Dr. Rajeev Raje

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Dr. Snehasis Mukhopadhyay

Approved by: Dr. William Gorman 12/08/2011
Head of the Graduate Program Date

Graduate School Form 20
(Revised 9/10)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation:

Decentralized and Partially Decentralized Multi-Agent Reinforcement Learning

For the degree of Doctor of Philosophy

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Executive Memorandum No. C-22,* September 6, 1991, *Policy on Integrity in Research.**

Further, I certify that this work is free of plagiarism and all materials appearing in this thesis/dissertation have been properly quoted and attributed.

I certify that all copyrighted material incorporated into this thesis/dissertation is in compliance with the United States' copyright law and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of the law. I agree to indemnify and save harmless Purdue University from any and all claims that may be asserted or that may arise from any copyright violation.

Omkar Jayant Tilak

Printed Name and Signature of Candidate

12/08/2011

Date (month/day/year)

*Located at http://www.purdue.edu/policies/pages/teach_res_outreach/c_22.html

DECENTRALIZED AND PARTIALLY DECENTRALIZED MULTI-AGENT REINFORCEMENT LEARNING

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Omkar Jayant Tilak

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2012

Purdue University

West Lafayette, Indiana

To the Loving Memory of My Late Grandparents : Aniruddha and Usha Tilak

To My Late Father : Jayant Tilak : Baba, I'll Always Miss You!!

ACKNOWLEDGMENTS

Although the cover of this dissertation mentions my name as the author, I am forever indebted to all those people who have made this dissertation possible.

I would never have been able to finish my dissertation without the constant encouragement from my loving parents, Jayant and Surekha Tilak, and from my fiancee, Prajakta Joshi. Their continual love and support has been a primary driver in the completion of my research work. Their never-ending interest in my work and accomplishments has always kept me oriented and motivated.

I would like to express my deepest gratitude to my advisor, Dr. Snehasis Mukhopadhyay for his excellent guidance and providing me with a conducive atmosphere for doing research. I am grateful for his constant encouragement which made it possible for me to explore and learn new things. I am deeply grateful to my co-advisor Dr. Luo Si for helping me sort out the technical details of my work. I am also thankful to him for carefully reading and commenting on countless revisions of this manuscript. His valuable suggestions and guidance were a primary factor in the development of this document.

I would like to thank Dr. Ryan Martin, Dr. Jennifer Neville, Dr. Rajeev Raje and Dr. Mihran Tuceryan for their insightful comments and constructive criticisms at different stages of my research. It helped me to elevate my own research standard and scrutinize my ideas thoroughly.

I am also grateful to the following current and former staff at Purdue University for their assistance during my graduate study – DeeDee Whittaker, Nicole Shelton Wittlief, Josh Morrison, Myla Langford, Scott Orr and Dr. William Gorman. I'd also like to thank my friends – Swapnil Shirsath, Pranav Vaidya, Alhad Mokahi, Ketaki Pradhan, Mihir Daptardar, Mandar Joshi, and Rati Nair. I greatly appreciate their

friendship which has helped me stay sane through these insane years. Their support has helped me overcome many setbacks and stay focused through this arduous journey.

It would be remiss of me to not mention other family members who have aided and encouraged me throughout this journey. I would like to thank my cousin Mayur and his wife Sneha who have helped me a lot during my stay in the United States. Last, but certainly not the least, I would also like to thank Dada Kaka for his constant encouragement and support towards my education.

PREFACE

Multi-Agent systems naturally arise in a variety of domains such as robotics, distributed control and communication systems. The dynamic and complex nature of these systems makes it difficult for agents to achieve optimal performance with predefined strategies. Instead, the agents can perform better by adapting their behavior and learning optimal strategies as the system evolves. We use Reinforcement Learning paradigm for learning optimal behavior in Multi Agent systems. A reinforcement learning agent learns by trial-and-error interaction with its environment. A central component in Multi Agent Reinforcement Learning systems is the intercommunication performed by agents to learn the optimal solutions. In this thesis, we study different patterns of communication and their use in different configurations of Multi Agent systems. Communication between agents can be completely centralized, completely decentralized or partially decentralized. The interaction between the agents is modeled using the notions from Game theory. Thus, the agents could interact with each other in a in a fully cooperative, fully competitive, or in a mixed setting. In this thesis, we propose novel learning algorithms for the Multi Agent Reinforcement Learning in the context of Learning Automaton. By combining different modes of communication with the various types of game configurations, we obtain a spectrum of learning algorithms. We study the applications of these algorithms for solving various optimization and control problems.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABBREVIATIONS

LA	Learning Automaton
LAs	Learning Automata
MARL	Multi Agent Reinforcement Learning
DPLA	Decentralized Pursuit Learning game Algorithm
PDGLA	Partially Decentralized Games of Learning Automata
HOGLA	Homogeneous Games of Learning Automata
HEGLA	Heterogeneous Games of Learning Automata

ABSTRACT

Tilak, Omkar Jayant Ph.D., Purdue University, May 2012. Decentralized and Partially Decentralized Multi-Agent Reinforcement Learning. Major Professors: Snehasis Mukhopadhyay and Luo Si.

Multi-agent systems consist of multiple agents that interact and coordinate with each other to work towards to certain goal. Multi-agent systems naturally arise in a variety of domains such as robotics, telecommunications, and economics. The dynamic and complex nature of these systems entails the agents to learn the optimal solutions on their own instead of following a pre-programmed strategy. Reinforcement learning provides a framework in which agents learn optimal behavior based on the response obtained from the environment. In this thesis, we propose various novel decentralized, learning automaton based algorithms which can be employed by a group of interacting learning automata. We propose a completely decentralized version of the estimator algorithm. As compared to the completely centralized versions proposed before, this completely decentralized version proves to be a great improvement in terms of space complexity and convergence speed. The decentralized learning algorithm was applied; for the first time; to the domains of distributed object tracking and distributed watershed management. The results obtained by these experiments show the usefulness of the decentralized estimator algorithms to solve complex optimization problems. Taking inspiration from the completely decentralized learning algorithm, we propose the novel concept of partial decentralization. The partial decentralization bridges the gap between the completely decentralized and completely centralized algorithms and thus forms a comprehensive and continuous spectrum of multi-agent algorithms for the learning automata. To demonstrate the applicability of the partial decentralization, we employ a partially decentralized team of learning

automata to control multi-agent Markov chains. More flexibility, expressiveness and flavor can be added to the partially decentralized framework by allowing different decentralized modules to engage in different types of games. We propose the novel framework of heterogeneous games of learning automata which allows the learning automata to engage in disparate games under the same formalism. We propose an algorithm to control the dynamic zero-sum games using heterogeneous games of learning automata.

1 INTRODUCTION

Human beings, and indeed all sentient creatures, learn by interacting with the environment in which they operate. When an infant begins playing and walking around at a young age, it has no explicit teacher. However, but it does receive a sensory feedback from its environment. A child collects information about cause and effect associated with different actions,. Based on this information gathered over an extended period of time, a child learns about what to do in order to achieve goals. Even during adulthood, such interactions with the environment provide knowledge about the environment and direct a person's behavior. Whether we are learning to drive a car or to interact with another human being, we learn by using this interactive mechanism.

Reinforcement learning (RL) is modeled after the way human beings learn in an unknown environment. Reinforcement learning involves an agent acting in an environment and interacting with it. The goal of the agent is to maximize a numerical reward signal based on the experience it has of the interaction with the environment. During the learning process, the agent is not instructed on which actions to take, but instead must explore the action space by trying different actions and by taking into account the response from the environment for those actions. The exploration of the action space based on the trial-and-error method and the ultimate goal of selecting the most optimal action are two important features of reinforcement learning.

1.1 Reinforcement Learning Model

The reinforcement learning problem is represented as the problem of learning from interaction with an environment to achieve certain optimization goal. The learner (also called as an *agent*) decides which actions should be performed based on certain

criteria. The part of the universe comprising of everything that is outside the agent is called as the *environment*. The agent interacts continually with the environment. The environment responds by giving rewards. Rewards are special numerical values that the agent tries to maximize over time. For simplicity, the agent and environment interaction can be viewed over a sequence of discrete time steps $t = 0, 1, 2, \ldots$. At each time step, the agent receives a representation of the state of the environment, $s_t \in \mathcal{S}$ where $\mathcal{S}$ is the set of all possible environment states. Based on this information, the agent selects an action $a_t \in \mathcal{A}_{s_t}$, where $\mathcal{A}_{s_t}$ is the set of actions available in state s_t. Based on the action selected, at the next time instant $t + 1$, the agent receives a numerical reward, $r_{t+1} \in \mathcal{R}$, where $\mathcal{R}$ is the set of real numbers. The agent transitions to a state s_{t+1} based on the previous state s_t and the selected action a_t. The agent implements a mapping from states to probabilities of selecting each possible action in that state. This mapping is called the agent's *policy*, $\pi(s, a)$. Reinforcement learning techniques specify how the agent changes and *learns* its policy as a result of its experience so that it can maximize the total amount of reward it will receive over the long run.

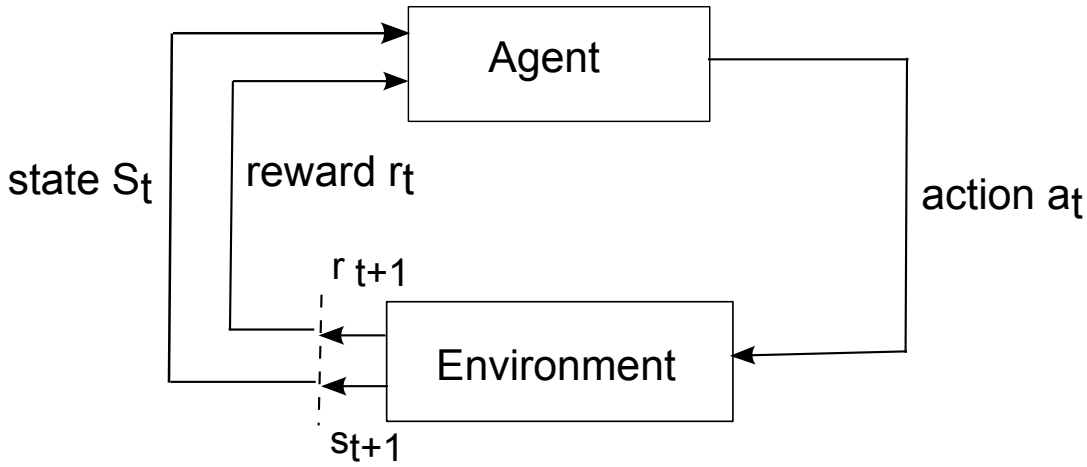


Figure 1.1. Reinforcement Learning Model

Reinforcement learning differs significantly from supervised learning in these aspects. In supervised learning, the agent learns the optimal behavior based on the examples provided by an external supervisor. Thus the active interaction between agent and environment, which is a hallmark of reinforcement learning, is not present in the supervised learning. Since complex and dynamic systems evolve with time, it often makes it impractical to obtain representative examples that are accurate representative of their behavior. Thus,it is beneficial for an agent to be able to learn and adapt its behavior from its own experience and interacting actively with the environment.

A reinforcement learning algorithm tries to incorporate a balance between exploration and exploitation. Both exploration and exploitation are necessary for the agent to select an optimal strategy in the given environment. Exploitation involves the agent selecting actions produced good reward during previous interactions. However, to gain this information about various actions, it has to try actions that were not selected before. This involves exploration. However, the agent has to strike a balance between these two seemingly contradictory tasks. Thus agent need to stochastically select different actions many times to gain a reliable estimate about their rewards. All learning algorithms take into account this exploration-exploitation dilemma while exploring the action space and interacting with the environment. In supervised learning, the agent does not need to worry about exploration and exploitation as the learning is done based on the examples provided by the supervisor.

1.1.1 Markov Decision Process Formulation

For a RL problem, it is typically assumed that the environment has Markov property. If the environment has the Markov property, then the environment's response at time step $t+1$ depends only on the state and action selected at the previous time instant t. A reinforcement learning task that satisfies the Markov property is called

a Markov Decision Process (MDP). If the state and action spaces are finite, then it is called a finite Markov decision process (finite MDP).

A particular finite MDP is defined by its state and action sets and by the one-step dynamics of the environment. Given any state s and action a, the probability of possible transition to the next state s' is given by the *transition probability function*:

$$P^a_{ss'} = Pr(s_{t+1} = s' | s_t = s, a_t = a)$$

The corresponding expected value of the reward is given by the *reward probability function*:

$$R^a_{ss'} = E(r_{t+1} | s_t = s, a_t = a, s_{t+1} = s')$$

The functions $P^a_{ss'}$ and $R^a_{ss'}$ completely specify the dynamics of a finite MDP. Most of the RL algorithms implicitly assumes the environment is a finite MDP. Various types of RL learning algorithms have been proposed in the literature [1] for a single agent to learn optimal action in an MDP environment. Here, we will describe them in a brief manner. Almost all RL algorithms are based on estimating value function $V(s)\text{or}Q(s,a)$ for different states or state-action pairs of a MDP. These functions estimate how good it is for the agent to be in a given state or how good it is to perform a given action in a given state. The *goodness* is defined in terms of future expected return of the rewards. These value functions are defined with respect to particular policies π. They are defined as follows:

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

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

where E_π is the expected value obtained under policy π and $0 < \gamma < 1$ is a small learning parameter. V^π is called as the *state-value function* while Q^π is called as the

action-value function. The RL algorithms learn or compute these functions and use them to find the optimal policy.

1.1.2 Dynamic Programming Algorithm

The Dynamic Programming (DP) algorithm updates the value function for all $s \in \mathcal{S}$ as follows:

$$V_{k+1}(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}(R^a_{ss'} + \gamma V_k(s'))$$

where the policy π is initialized arbitrarily and is improved as follows:

$$\pi(s) \leftarrow \text{argmax}_a \sum_{s'} P^a_{ss'}(R^a_{ss'} + \gamma V(s'))$$

DP algorithm assumes that all the transition and reward probability values of the MDP are known and uses this information to *compute* the optimal policy.

1.1.3 Q-learning Algorithm

The Q-learning algorithm *learns* the value function by trying various actions and uses this information to iteratively calculate the optimal policy. In its simplest form, the Q-learning algorithm is defined by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

where α and γ are two learning parameters. By iteratively updating the value function in this manner, the optimal policy is calculated during each iteration till convergence.

1.1.4 Temporal Difference Learning Algorithm

Temporal Difference (TD) learning is a combination of Monte Carlo (MC) technique and DP ideas. Like MC methods, TD methods can learn directly from raw experience without a model of the environment. Like DP, TD methods update bootstrap the estimates based in part on other learned estimates, without waiting for a final outcome. In its simplest form, TD algorithm updates the value function as follows:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_(t+1) + \gamma V(s_{t+1}) - V(s_t)]$$

Using the above equation, an arbitrary policy π can be evaluated or an optimal policy can be learned dynamically.

1.2 n-armed Bandit Problem

n-armed bandit problem consists of a player making an action selection and receives different rewards. Through repeated plays, the player is supposed to maximize the winnings by concentrating the plays on the best possible action. Each action has an expected or mean reward (also called as value) associated with it. If one knew the value of each action, then it would be trivial to solve the n-armed bandit problem: the player would always select the action with highest value. It is assumed that the player does not know the action values with certainty, although the player may have estimates.

If the player maintains estimates of the action values, then at any time there is at least one action whose estimated value is greatest. By selecting action in such a greedy manner, the player can exploit the current knowledge of the values of the actions. If instead the player selects one of the non-greedy actions, then we say that the player is exploring. Exploitation is the right thing to do to maximize the expected reward on the one play, but exploration may produce the greater total reward in the long run. For example, suppose the greedy action's value is known with certainty,

while several other actions are estimated to be nearly as good but with substantial uncertainty. In such cases, it may be better to explore the non-greedy actions and discover which of them are better than the greedy action. Because it is not possible both to explore and to exploit with any single action selection, one often refers to the "conflict" between exploration and exploitation.

Various mechanisms can be used to devise precise values of the estimates, uncertainties. There are many sophisticated methods for balancing exploration and exploitation. Learning Automaton provides a framework to solve the n-armed bandit problem.

1.3 Learning Automaton

The Learning Automaton was modeled based on mathematical psychology models of animal and child learning. The learning automaton attempts to learn long-term optimal action through the use of reinforcement. These actions are assumed to be performed in an abstract environment. The environment responds to the input action by producing an output (also called as reinforcement) which is probabilistically related to the input action. The reinforcement refers to an on-line performance feedback from a teacher or environment. The reinforcement, in turn, may be qualitative, infrequent, delayed, or stochastic. The interaction between automaton and the environment is as shown below.

Stochastic learning automata operating in stationary as well as nonstationary random environments have been studied extensively [2], [3]. Learning automaton (LA) uses reinforcement learning paradigm to choose the best action from a finite set. An LA A consists of a finite set of actions $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_r\}$. On every trial n, LA performs one action $\alpha(n) = \alpha_i \in \alpha$ by sampling its action probability vector and obtains a reinforcement $\beta(n)$. LA then updates its action probability vector $P_j(n), 1 \leq j \leq r$; based on this reinforcement. The manner in which $P(n)$ is updated is governed by the learning algorithm T. The environment E is described by a set

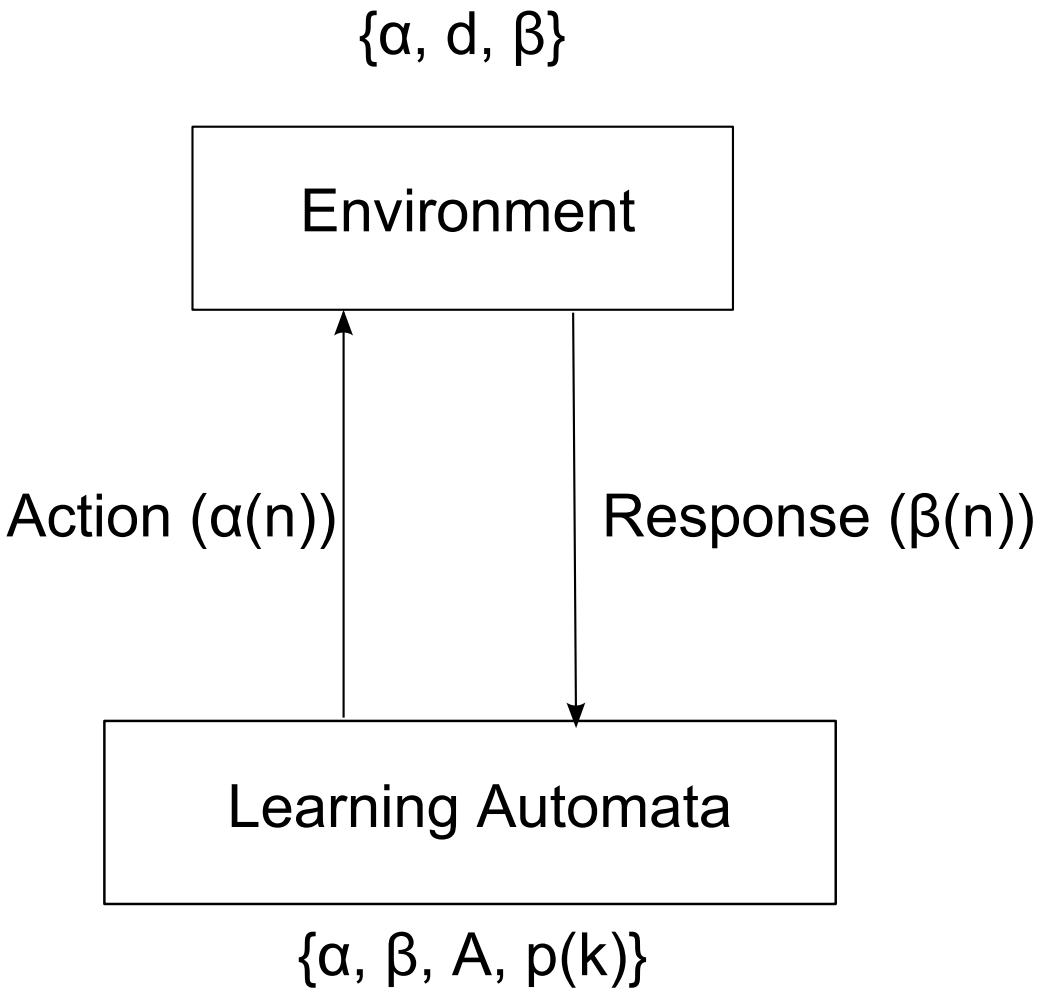


Figure 1.2. Interaction between Learning Automaton and Environment

of reward probabilities $\{d_j\}$ where, $d_j = \Pr[\beta(n) = 1 \mid \alpha(n) = \alpha_j]$. Various learning algorithms (e.g. L_{R-I}, L_{R-P} algorithm) have been proposed in the literature for the automaton to update its action probability vector [2]. If action selected at n-th time instant is α_i, then the general reward-penalty LA algorithm is given by:

$$p_i(n+1) = p_i(n) + a\beta(n)(1 - p_i(n)) - b(1 - \beta(n))p_i(n)$$

$$p_j(n+1) = p_j(n) - a\beta(n)p_j(n) + b(1 - \beta(n))(\frac{1}{r-1} - p_i(n))j \neq i$$

where $0 < a < 1$ and $0 < b < 1$ are constants called the reward and penalty parameters, respectively. If $b = a$, the scheme is called linear reward-penalty (L_{R-P}) and if $b = 0$, it is called linear reward-inaction (L_{R-I}).

The L_{R-I} and L_{R-P} algorithms are called as model-free algorithms because they do not use a model of the environment in the learning process. Pursuit algorithm [4], on the other hand, is a model-based learning algorithm. It incorporates a model of the environment in the form of the estimates of the reward probabilities (denoted as $\hat{d}$). The automaton maintains a vector $\hat{d}_i(n)$ where n refers to the current iteration. Let $\hat{d}_M(n)$ be the highest estimated value in vector $\hat{d}(n)$. Let e_i represent a unit vector with i^{th} component set to unity and all other components set to zero. The automaton also maintains two vectors $(Z_1(n), Z_2(n), \ldots, Z_r(n))^T$ and $(R_1(n), R_2(n), \ldots, R_r(n))^T$. The number of times an action α_i is chosen till trial n is given by $Z_i(n)$ while $R_i(n)$ gives the total reinforcement obtained in response to action α_i till trial n. The automaton uses $\alpha(n)$ and $\beta(n)$ to update $R_i(n)$ and $Z_i(n)$ and they are used to obtain $\hat{d}_i(n)$. The details are given below: Let $\alpha(n) = \alpha_i$. Then the automaton updates $Z_i(n)$, $R_i(n)$ and obtains the estimates $\hat{d}_i(n)$ as follows:

$$R_i(n) = R_i(n-1) + \beta(n)$$

$$R_j(n) = R_j(n-1), \forall j \neq i$$

$$Z_i(n) = Z_i(n-1) + 1$$

$$Z_j(n) = Z_j(n-1), \forall j \neq i$$

$$\hat{d}_i(n) = \frac{R_i(n)}{Z_i(n)}, \forall i$$

The Pursuit algorithm proceeds as follows:

1. At every time step n, the automaton chooses an action by sampling its action probability vector.

2. The automaton obtains a payoff $r(n)$ based on the action chosen.

3. Based on the response, the automaton updates R, Z and $\hat{D}$ matrices as described above. Then based on this information, the automaton updates its action probability vector as follows:

$$p(n+1) = p(n) + \lambda(e_M - p(n))$$

where $0 < \lambda < 1$ is the learning parameter and index M is determined by $\hat{d}_M(n) = \max_i \hat{d}(n)_i$

To study the convergence properties of the learning automata, various norms such as expediency, optimality, ϵ-optimality, and absolutely expediency have been defined in the literature [2]. In this paper, we propose novel algorithms for multi-agent Markov chain control that are based on (model-free) the L_{R-I} algorithm and the (model-based) Pursuit algorithm.

1.3.1 Games of LA

A LA acting alone represents a single learning agent operating in an environment. However, such simple paradigm is not adequate to model a lot of real-world systems. More interesting learning schemes can be designed by allowing multiple learning agents to interact and interconnect with each other. An automata game involves N learning automata $A^i (i = 1, 2, \ldots, N)$, each with an action set $\alpha^i = \{\alpha_1^i, \alpha_2^i, \ldots, \alpha_{a_i}^i\}$

interacting through a stationary random environment. At each instant n, each individual automaton A^i selects one action $\alpha^i_{s_i}$ by sampling its current action probability vector $P^i = \{P^i_1, P^i_2, \ldots, P^i_{a_i}\}$. The resultant action tuple $\{\alpha^1_{s_1}, \alpha^2_{s_2}, \ldots, \alpha^N_{s_N}\}$ determines the random environment response $\beta(n)$ received by the automata for the current iteration of the game. If all the automata of the team receive the same response $\beta(n)$, then the game is called as an identical-payoff game. However, if one automaton in the team receives $\beta(n)$ and the other one receives $-\beta(n)$, then such game is called as a zero-sum game of LA [2]. Each individual LA can use any suitable learning scheme (L_{R-I}, L_{R-P}, Pursuit learning etc) to update its own action probabilities.

1.4 Motivation

Multi-agent systems appear very frequently and in various domains such as robotics, distributed control and telecommunications. The complex and dynamic nature of these systems makes it difficult to control them with predetermined agent behavior. Instead, the agents must discover and adapt a solution on their own using learning. In a multi-agent systems, agents may want to (or need to) interact with each other thus leading to various communication configurations. Also, since agents need to adapt to the changing environment, the learning process needs to track the changes in the environment and guide the agents appropriately. These factors complicate the learning algorithm and makes its analysis harder.

The games of LAs paradigm represents the multi-agent interaction model for LAs. ***In this thesis, we focus on multi-agent systems that are modeled as games of LAs.*** As described earlier, model-based (such as Pursuit algorithm) techniques or model-free (such as L_{R-I} algorithm) techniques can be used to learn optimal strategies for the games of LAs. However, the Pursuit learning algorithms proposed for this model remain centralized in nature. The L_{R-I} game algorithm is decentralized in nature. However, it displays very slow convergence and converges to one of the many

equilibrium points. Thus, there is a need for a LA game algorithm that possesses fast convergence speed and is yet decentralized in nature.

The LA game algorithms proposed so far in the literature deal with either completely centralized or completely decentralized configurations. However, the configurations where only a subset of the automata communicate with each other have not been studied or proposed yet. One can imagine a gamut of game algorithms for configurations ranging from completely decentralized to completely centralized. This leads to the proposal of partially centralized configurations of LAs.

Also, the LA game configurations proposed so far require that all the automata in the group participate in a single type of game: either a zero-sum game or an identical-payoff game. However, the configurations where a subset of automata participate in identical-payoff game while others participate in a zero-sum game need further investigation. Towards this end, we proposed the heterogeneous games of LAs. Under this paradigm, different local groups of LAs participate in a zero-sum (or identical-payoff) game while the automata across the groups participate in an identical-payoff (or zero-sum) game.

1.5 Contributions

The salient contributions of this thesis are as follows:

1. We propose a novel algorithm called Decentralized Pursuit Learning (DPL) algorithm for learning optimal strategies in games of LAs. DPL algorithm combines fast convergence speed with the decentralized memory storage and distributed learning mechanism.

2. We propose partially centralized configurations of LAs. This paradigm has the power to model a vast range of LA game configurations. We applied this paradigm to the multi-agent Markov chains and proposed various novel algorithm to control the multi-agent Markov chains.

3. The thesis also explores the possibility of combining different types of games (namely zero-sum and identical-payoff games) for a group of interacting LAs. We propose a novel framework of the heterogeneous games of LAs. This allows a group of LAs to participate in one type of game (say identical-payoff game) while the other group participates in a different type of game (namely zero-sum game). A novel algorithm is proposed which models the dynamic zero-sum games as a heterogeneous games among LAs. The algorithm then uses this framework to control the dynamic zero sum games.

4. We applied the games of LAs framework to solve optimization problems in different domains. In particular, we applied the DPL algorithm to solve the sensor subset selection problem in object tracking systems. To our knowledge, it is the first time a reinforcement learning algorithm was applied for the object tracking domain. We also applied the DPL algorithm to solve the watershed management problem. The results from these two experiments demonstrate the the power and flexibility of the LA and its applicability in various disparate domains.

1.6 Outline

This thesis is organized as follows: In chapter 2, we discuss various MARL algorithms that have been proposed in the literature. In chapter 3, we describe the novel distributed Pursuit learning game algorithm and analyze its convergence mathematically. In chapter 4, we propose the novel framework of the partially decentralized games of LA and use it to control multi-agent Markov decision process. Chapter 5 discusses the novel paradigm of heterogeneous games of LA and its use to control dynamic zero sum games. In chapter 6, we describe some of the applications of the games of LA to solve various real-world problems. In particular, we discuss the sensor subset selection and watershed management problem. Finally, chapter 7 discuss the possible future extensions of this work and concludes the thesis.

2 MULTI-AGENT REINFORCEMENT LEARNING

A learning automaton acting alone represents a single learning agent operating in an environment. Along with the LA algorithms described earlier, a single agent can learn using a plethora of other algorithms. If the agent interacts with a Markovian environment, then various Reinforcement Learning (RL) algorithms such as Q-Learning, Temporal Difference (TD)-learning [1] can be used to learn optimal policy. If the parameters of the environment model are completely known, then optimal policy can be calculated using Dynamic Programming (DP) approaches [1].

However, such simple paradigm is not adequate to model a lot of real-world systems. More interesting learning schemes can be designed by allowing multiple learning agents to interact and interconnect with each other. A multi-agent system is defined as a group of autonomous, interacting learning agents sharing a common environment, which they receive response from and upon which they act by performing certain actions. However, several new challenges arise for RL in multi-agent systems. One challenge involves defining a good learning goal for the multiple RL agents. Furthermore, it is sometimes required for each learning agent to keep track of the other learning agents. The helps the agent to coordinate its behavior with other agents, such that a coherent joint behavior emerges [5]. However, this makes the learning process nonstationary. The nonstationarity also invalidates the convergence properties of most single-agent RL algorithms. In addition, the scalability of algorithms to realistic problem sizes is also a cause for concern in MARL. The Multi-Agent Reinforcement Learning (MARL) field is rapidly expanding, and a wide variety of approaches to exploit its benefits and address its challenges have been proposed over the last few decades. Various algorithms and approaches have been proposed which integrate developments in the areas of single-agent RL, game theory, and various

other policy search techniques. In this section, we describe few relevant algorithms and techniques that highlight different approaches towards MARL.

2.1 A-Teams

An A-Team [6] is a multi-agent framework in which autonomous agents cooperate by modifying results produced by other agents. These results circulate continually in a graph which represents interconnection between agents. Convergence is said to occur if and when a persistent solution appears. A-Team results in a type of asynchronous organization that combines features from various learning paradigms such as insect societies, genetic algorithms, blackboards and simulated annealing.

An A-Team consists of a set of autonomous agents and a set of memories that are interconnected to form a strongly cyclic network. Thus, every agent is in a closed loop with other agents in the system. Agents may include all manner of problem-solving entities, including computer-based agents and humans. An agent is defined to consist of three components: an operator (algorithm), a selector and a scheduler. The operator creates and modifies the solutions stored in memories, the selector determines which solutions the operator will work on, and the scheduler does the resource management. An autonomous agent has completely self-contained selector and scheduler components.

An A-Team can be visualized as a directed data-flow hypergraph. Each node of the graph represents a complex of overlapping memories. Each arc represents an autonomous agent. Results or trial-solutions accumulate in the memories to form populations (like those in genetic algorithms). These populations change as new members are continually added by construction agents, while older members are being erased by destruction agents. All the agents in an A-Team act in an autonomous manner. Each agent makes decisions for itself regarding what it is going to do and when it is going to do it. There is no centralized control. Agents cooperate by working on the results produced by the other agents. Because the agents are autonomous,

this cooperation is asynchronous. All the agents can work in parallel thus potentially increasing the convergence speed. Thus, an A-Team is modeled as a strongly cyclic network of memories and autonomous agents. Each memory is dedicated to one problem. Collectively, the memories represent the problem that the agents try to solve together. Various possible solutions for the parts of the problem are produced by the agents and stored in the memories to form populations. Agents cooperate by working on the solutions produced by the other agents.

2.2 Ant Colony Optimization

Swarm intelligence is an approach to problem solving that takes inspiration from the social behaviors of insects and of other animals. Ant colony optimization (ACO) [7] takes inspiration from the foraging behavior of ants. The ants deposit pheromone on the ground in order to mark some favorable path that should be followed by other members of the colony. Ant colony optimization exploits a similar mechanism for solving optimization problems. In ACO, a number of artificial agents (called ants) build solutions to an optimization problem at hand and exchange information on the quality of these solutions via a communication scheme that is similar to the one adopted by real ants. ACO solves the optimization problem by simulating a number of artificial ants moving on a graph that encodes the problem. The nodes of the graph represent *solution components* which represent possible assignment of values to the decision variables of the optimization problem. Edges between the node represent a variable called as a pheromone and it can be read and modified by ants. So far, ACO has been applied on variety of different NP-hard problems, stochastic optimization problems and multi-objective optimization problems [7].

ACO proceeds in is an iterative manner. At each iteration, a number of artificial ants are considered to be active. Each of them builds a solution by walking from vertex to vertex on the graph with the constraint of not visiting any vertex that she has already visited in her walk. At each step of the solution construction, an ant

selects the next vertex to be visited according to a stochastic mechanism that is based on the pheromone. In particular, when in vertex i, if vertex j has not been previously visited, it can be selected with a probability that is proportional to the pheromone associated with edge (i, j). At the end of an iteration, on the basis of the quality of the solutions constructed by the ants, the pheromone values are modified in order to bias ants in future iterations to construct solutions similar to the best ones previously constructed.

The behavior of any ACO algorithm is governed mainly by the way in which the pheromone update is done. Different algorithms have been proposed in the literature which update the pheromone values between nodes in different ways. Ant System (AS) was the first ACO algorithm proposed in the literature [8]. Its main characteristic is that, at each iteration, the pheromone values are updated by all the ants that have built a solution in the current iteration. The pheromone θ_{ij} , associated with the edge joining nodes i and j is updated as follows:

$$\theta_{ij} \leftarrow (1-\rho)\theta_{ij} + \sum_{k=1}^{m} \Delta\theta_{ij}^{k}$$

where ρ is the evaporation rate, m is the number of ants participating in the current iteration and $\Delta\theta_{ij}^{k}$ is the quantity of pheromone laid on edge (i, j) by the ant k.

Under the Max-Min Ant System (MMAS) algorithm, only the best ant updates the pheromone trails and that the value of the pheromone is bound. The pheromone update is implemented as follows:

$$\theta_{ij} \leftarrow [(1-\rho)\theta_{ij} + \sum_{k=1}^{m} \Delta\theta_{ij}^{best}]_{\theta_{min}}^{\theta_{max}}$$

where θ_{min} and θ_{max} are the lower and lower bounds imposed on the pheromone values respectively the operator $[x]_a^b$ returns x if and only if $a < x < b$ otherwise it returns the suitable lower or upper bound value.

Local pheromone update algorithm updates the pheromone values in addition to the pheromone updates performed at the end of the construction process (called offline pheromone update). The local pheromone update is performed by all the ants after each construction step. The main goal of the local update is to diversify the search performed by subsequent ants during an iteration. By decreasing the pheromone concentration on the traversed edges, local pheromone update encourage subsequent ants to choose other edges and, hence, to produce different solutions. This makes it less likely that several ants produce identical solutions during one iteration. Each ant applies the local pheromone update only to the last edge traversed in the following manner:

$$\theta_{ij} = (1 - \varphi)\theta_{ij} + \varphi\theta_0$$

where $\varphi \in (0, 1]$ is the pheromone decay coefficient, and θ_0 is the initial value of the pheromone.

2.3 Colonies of Learning Automata

In [9], authors discuss the similarities between ACO model and the graphical formulation of the MDP framework. Authors state that MDP can be modeled as a graph. Since ACO problems are also modeled as graphs, a particular ACO can be modeled as an interconnected network of LAs which is capable of controlling an MDP. Thus authors state that ACO model can be mapped onto the framework introduced by Wheeler-Narendra [10]. The Wheeler-Narendra framework deploys one LA at each state of the MDP. Authors state that these LAs act as ants in ACO and the links between the states of ACO act as the links between different states of MDP.

Thus, an ant in ACO can be viewed as a dummy mobile agent that walks around in the graph of interconnected LAs, makes states and the LAs that reside in that state active and brings information so that the LAs involved can update their action probabilities. The only difference is that, in ACO, several ants are walking around

simultaneously in a parallel and autonomous manner. Thus under the new formulation, several LAs can be active at the same time. In the model of Wheeler and Narendra, there is only one LA active at a time. However, authors state that adding multiple mobile agents to the system will not harm the convergence. The automata will use the same update scheme and the environment response calculation as the one used for Markov chain control by Wheeler-Narendra.

By connecting LA and ACO in this manner, the authors give a formal justification for the use of ant algorithms in the cases where graph is static. Therefore, LAs give insight into why ACO algorithms work. Authors predict that in the case when the graph is dynamic (meaning the transition probabilities in the MDP may depend on the action probabilities of the other nodes), the model of LA colonies can still be used. Therefore, these two frameworks may influence each other in a positive way.

2.4 Dynamic or Stochastic Games

The generalization of the Markov Decision Process (MDP) to the multi-agent interaction is called a stochastic game or a dynamic game. A dynamic game can be represented by a tuple $\langle S_1, S_2, \ldots, S_N; A_1, A_2, \ldots, A_M; T; R_1, R_2, \ldots R_M \rangle$ where $S = \{S_i\}, i = 1, 2, \ldots, N$ is the discrete set of states of the Markov chain, $A_j, j = 1, 2, \ldots, M$ are the discrete sets of actions available to the agent j ($j = 1, 2, \ldots, M$). The joint action set is then given by $\mathcal{A} = A_1 \times A_2 \times \ldots \times A_M$. The transition probability function is defined as $T : S \times \mathcal{A} \times S \to [0, 1]$. The reward functions are defined as $R_i : S \times \mathcal{A} \times S \to \mathcal{R}$.

For the dynamic games, the state transitions are the result of the joint action of all the agents. The action tuple at k^{th} instant is given by $a_k \in \mathcal{A} = [a_{1k}, a_{2k}, \ldots, a_{mk}]^T$ where $a_{ik} \in A_i$, for $i = 1$ to M and T denotes vector transpose operator. Consequently, the rewards r_{ik+1} also depend on the joint action. If $R_1 = R_2 = \ldots = R_M$ then all the agents try to maximize the same expected common return, and the dynamic is fully cooperative. It describes a dynamic identical-payoff game. If $M = 2$

and $R_1 = -R_2$, the two agents have opposite goals, and the dynamic game is fully competitive. It describes a dynamic zero-sum game. Mixed games are stochastic games that are neither fully cooperative nor fully competitive. In this thesis, we focus on the identical-payoff and zero-sum games of the learning agents (in particular, Learning Automata).

2.4.1 RL Algorithm for Dynamic Zero-Sum Games

In [11], Littman proposes a novel learning algorithm called minimax-Q learning algorithm for systems where there are only two agents and they have diametrically opposed goals (in other words, a dynamic zero-sum game). The algorithm is very similar to the traditional Q-learning algorithm used for single agent RL with *minimax* operator replacing the *max* operator in Q-learning. In equation form:

$$V(s) = \max_{\pi \in PD(S)} \min_{o \in O} \sum_{a \in A} Q(s, a, o)\pi_a$$

$$Q(s, a, o) = r(s, a, o) + \gamma \sum_{s'} T(s, a, o, s')V(s')$$

where o represents the action selected by the opponent and γ is the learning parameter. Author proves that the minimax-Q algorithm learns to optimal *minimax* strategy of the underlying game matrix.

2.4.2 RL Algorithm for Dynamic Identical-Payoff Games

In a Dynamic Identical Payoff Game (DIPG), all the agents have the same reward function ($R_1 = R_2 = \ldots = R_M$) and the learning goal is to maximize the expected value of the common payoff. If a centralized entity was available who knows the actions selected by all the agents, the DIPG will reduce to a MDP, the action space of which would be the joint action space of the SG. In this case, the goal could be achieved by learning the optimal joint-action values with Q-learning:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

where $a_t \in \mathcal{A} = [a_{1k}, a_{2k}, \ldots, a_{mk}]^T, a_{ik} \in A_i$, for $i = 1$ to M is the joint action tuple.

Since, the agents are autonomous decision makers, a coordination problem arises in this particular situation. Even if all the agents update their Q-values in a synchronous manner, the optimal Q-value learned by the individual agents might be sub-optimal. Agents can learn by applying a greedy strategy to the Q-function as follows:

$$\text{argmax}_{a_i} \max_{a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_n} Q(s, a)$$

Since the greedy action selection procedure breaks ties randomly, in the absence of additional coordination procedures, different agents may break ties in different ways and the resulting joint action may be suboptimal. This is termed as the *coordination problem*.

The Team Q-learning algorithm [12] avoids the coordination problem by assuming that the optimal joint actions are unique. Then, if all the agents update the common Q-function in parallel then they can safely use the greedy policy to select the optimal joint actions and maximize their return. Since the optimal joint action is assumed to be unique, even if each individual agents breaks the ties arbitrarily, each agent will converge to the unique optimal action.

The Distributed Q-learning algorithm [13] solves the cooperative task without assuming coordination and its complexity is similar to that of single-agent Q-learning. However, the algorithm only works for cases where the optimal joint policy is deterministic. Each agent i maintains a local optimal policy π_i and a local Q-function $Q_i(s, a_i)$. This Q-function depending only on the action set of the agent i. The local Q-values are updated only when the update leads to an increase in the Q-value. This implies:

$$Q_{i,t+1}(s_t, a_{i,t}) = \max\{Q_{i,t}(s_t, a_{i,t}), r_{t+1} + \gamma \max_{a_i} Q_{i,t}(s_{t+1}, a_{i,t})\}$$

This ensures that the local Q-value are always equal to the maximum of the joint-action Q-values:

$$Q_{i,t}(s_t, a_i) = \max_{a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_n} Q_t(s, a)$$

Similarly, the local optimal policy π is updated only if the update leads to an improvement in the local Q-values:

$$\pi_{i,t+1}(s_t) = \begin{cases} a_{i,t} & \text{if } \max_{u_i} Q_{i,t+1}(s_t, a_i) > \max_{u_i} Q_{i,t}(s_t, a_i) \\ \pi_{i,t}(s_t) & \text{otherwise} \end{cases}$$

This ensures that the joint policy $[pi_{1,t}, pi_{2,t}, \ldots, pi_{n,t}]^T$ is always optimal with respect to the global Q-function. Under the condition that initial values of local Q-functions are set to zero, then it is proven that the local policies of the agents converge to an optimal joint policy.

Coordination graphs [14] paradigm can be applied to cases where the global Q-function can be additively decomposed into local Q-functions that only depend on the actions of a subset of agents. The decomposition might be different for different states. Typically the local Q-functions have smaller dimensions than the global Q-function and these dimensions are independent of each other. Maximization of the joint Q-value is done by solving simpler task of maximizing local Q-functions. The the individually optimized solutions are aggregated to calculate the optimized value of the global Q-function. Under certain conditions, coordinated selection of an optimal joint action is guaranteed [15].

2.5 Games of Learning Automata

An extension of a single learning automaton is the game scenario where a team of automatas receive a reinforcement whose probability depends on the actions of all the automatas. The game we consider here is a discrete stochastic game played by N automatas (representing N players). Each of the automatas has finitely many

actions. At each instant, every automaton stochastically selects an action to be played. After each play, the automatas receive reinforcement from the environment. These reinforcements are treated as the payoffs to individual automatas. The game is one of incomplete information. Thus, nothing is known regarding the distributions of elements of the random payoff matrix. The game is played repeatedly and the goal of the game is for each automaton, to asymptotically learn and converge to Nash equilibrium strategies with respect to the expected value of the payoff. The games of automata models have been used in telephone traffic routing [16] and control of Markov chains [10], among several applications. Learning automata models have also been proposed for non-stationary environments where the reward probabilities of the environment change in specific manners (see, e.g., [17]). A specific model of such non-stationarity leads to the so-called Associative Learning problem [18, 19] where the reward probabilities are functions of an exogenous context vector and the learning problem is to determine a map (e.g., a linear map) from the context space to the optimal actions. However, the context changes in this model are not controlled by the agent's actions.

Each automaton i is assumed to have a finite set of actions or pure strategies, R_i, $1 \leq i \leq N$. Let Each play of the game then consists of each of the automatas choosing an action. The result of each play is a random payoff to each automaton. Let r_i denote the random payoff to automaton i, $1 \leq i \leq N$. The functions

$$d^i : \prod_{j=1}^{N} S_j \rightarrow [0, 1]$$

where

$$d^i(\alpha_1, \alpha_2, \ldots, \alpha_N) = E[r_i | \text{automaton} j \text{chose action} \alpha_j, \alpha_j \in S_j, 1 \leq j \leq N]$$

defines the payoff or utility matrix of automaton i. A strategy for automaton i is defined to be its probability vector $p_i = [p_{i1}, p_{i2}, \ldots, p_{im}]$. Each of the pure strategies or actions of the i^{th} automaton are considered as a strategy. Let e_i be a

unit probability vector (of appropriate dimension) with i^{th} component unity and all others zero. Then e_i is the strategy corresponding to the action i.

A slight variation of the above game formulation is a zero-sum where there are *two* LAs in the group ($N = 2$). They payoff one automaton receives is the opposite of he payoff received by the other automaton. If one automaton receives a reward then the other one gets a penalty and vice versa. Thus, we have $r_1 = r$, and $r_2 = -r$. Following algorithm learns the optimal minimax *pure* strategy for a zero-sum game.

2.5.1 L_{R-I} Game Algorithm for Zero Sum Game

L_{R-I} game algorithm for zero-sum game proceeds as follows:

1. At every time step, each automaton chooses an action according to its action probability vector. Thus, the i^{th} automaton ($i = 1$ or 2) chooses action α_i at instant k, based on the probability distribution $p_i(k)$.

2. First automaton (max or row player) obtains a payoff $r_1(k)$ based on the set of all actions. The second automaton (min or column player) obtains a payoff $r_2(k) = -r_1(k)$ based on the set of all actions.

3. Each automaton updates its action probability as follows

$$p_i(k+1) = p_i(k) + \lambda r_i(k)(e_{\alpha_i} - p_i(k)), i = (l, 2)$$

where $0 < \lambda < 1$ is a parameter and e_{α_i} is a unit vector of appropriate dimension with α_ith component unity. This is the Linear Reward-Inaction (L_{R-I}) game algorithm. It has been shown [2] that, if each automaton uses the L_{R-I} algorithm for playing the game, the automata team converges to one of the Nash equilibrium points (local maximum or locally optimal strategy) of the underlying game matrix.

A slightly modified version of the game setup is a game with common payoff where all the automatas receive the same payoff after each play of the game. It is called the

identical-payoff game. Thus, we have $r_i = r$, for all i and hence $d^i(\alpha_1, \alpha_2, \ldots, \alpha_N) = d^j(\alpha_1, \alpha_2, \ldots, \alpha_N)$, $\forall i, j$. Hence the reward structure of the game can be represented by a single hyper matrix D of dimension $m_l \times m_2 \times \ldots \times m_N$, whose elements are $d_{\alpha_l \alpha_2 \ldots \alpha_N} = E[r_i|$ Player j played action α_j] ,$1 \leq j \leq N$ Various algorithms have been proposed to learn optimal strategies for identical-payoff games of LAs.

2.5.2 L_{R-I} Game Algorithm for Identical Payoff Game

L_{R-I} game algorithm for identical-payoff game proceeds as follows:

1. At every time step, each automaton chooses an action according to its action probability vector. Thus, the i^{th} automaton chooses action α_i at instant k, based on the probability distribution $p_i(k)$.

2. Each automaton i obtains a common payoff $r_i(k)$ based on the set of all actions.

3. Each automaton updates its action probability as follows $p_i(k+1) = p_i(k) + \lambda r_i(k)(e_{\alpha_i} - p_i(k))$, $i = l, 2, \ldots, N$.

where $0 < \lambda < 1$ is a parameter and e_{α_i} is a unit vector of appropriate dimension with α_ith component unity. This is the Linear Reward-Inaction (L_{R-I}) game algorithm. It has been shown [2] that, if each automaton uses the L_{R-I} algorithm for playing the game, the automata team converges to one of the Nash equilibrium points (local maximum or locally optimal strategy) of the underlying game matrix.

2.5.3 Pursuit Game Algorithm for Identical Payoff Game

The idea behind Pursuit Learning algorithm is to keep estimates of reward probabilities and use them in updating action probabilities. In case of game formulation, we have one payoff hypermatrix entry for every N-tuple of actions. The Pursuit Game Algorithm estimates these entries of payoff matrix using the reinforcement received

at each instant. These estimates are denoted by $\hat{d}_{i_1 i_2 \ldots i_n}$. Similar to the implementation of Pursuit Learning Algorithm, the pursuit game algorithm maintains two hypermatrixes, Z and R who have the same dimensions as that of the payoff matrix D. The entry $Z_{i_1 i_2 \ldots i_n}(k)$ of hypermatrix $Z(k)$ gives the number of times the action tuple $(\alpha_{1 i_1}, \alpha_{2 i_2}, \ldots, \alpha_{N i_N})$ is selected till trial k, while the element $R_{i_1 i_2 \ldots i_N}(k)$ of the hypermatrix $R(k)$ gives total reinforcement obtained for this action tuple till trial k . These hypermatrices are updated at each instant and are used to get the estimates of the entries of payoff matrix.

At each instant k, each automaton selects an action at random based on its current action probability vector. The action selected by j^{th} automaton is denoted by $\alpha_{j i_j}$. Now various hypermatrices are updated as follows:

$\forall (j_1, j_2, \ldots, j_N) \neq (i_1, i_2, \ldots, i_N)$

$$R_{i_1 i_2 \ldots i_N}(k) = R_{i_1 i_2 \ldots i_N}(k-1) + \beta(k)$$

$$R_{j_1 j_2 \ldots j_N}(k) = R_{j_1 j_2 \ldots j_N}(k-1)$$

$$Z_{i_1 i_2 \ldots i_N}(k) = Z_{i_1 i_2 \ldots i_N}(k-1) + 1$$

$$Z_{j_1 j_2 \ldots j_N}(k) = Z_{j_1 j_2 \ldots j_N}(k-1)$$

$$\hat{d}_{i_1 i_2 \ldots i_N}(k) = \frac{R_{i_1 i_2 \ldots i_N}(k)}{Z_{i_1 i_2 \ldots i_N}(k)}$$

$$\hat{d}_{j_1 j_2 \ldots j_N}(k) = \hat{d}_{j_1 j_2 \ldots j_N}(k-1)$$

Based on the estimated playoff matrix, the j^{th} automaton $(1 \leq j \leq N)$ calculates a vector, $\hat{E}^j(k) = [\hat{E}^j_1, \hat{E}^j_2, \ldots, \hat{E}^j_{r_j}]$, which will be used for updating the action probabilities. These vectors are obtained as follows:

$$\hat{E}^j_{i_j}(k) = \max_{l_i} \{\hat{d}_{l_1 l_2 \ldots l_{j-1} i_j l_{j+1} \ldots l_N}(k)\}$$

$$\hat{E}^j_l(k) = \hat{E}^j_l(k-1), \forall l \neq i_j$$

Now, j^{th} automata updates its probability distribution as follows:

$$p_j(k+1) = p_j(k) + \lambda(e_{M_j}(k) - p_j(k))$$

where $e_{M_j}(k)$ is a unit vector with $M_j(k)$-th component unity and the index $M_j(k)$ is defined by:

$$M_j(k) = \arg\max_l \hat{E}_l^j(k)$$

It has been shown [20] that if the automata team employs Pursuit Learning Game then the automata team converges to the global maximum of the underlying game matrix. Since the computation of estimation matrices (namely $\hat{d}, \hat{R}$ and $\hat{Z}$ is done in a centralized manner, we call this algorithm as the Centralized Pursuit Learning Algorithm (CPLA).

Learning automaton and games of learning automata have been used various applications like multiple access channel selection [21], congestion avoidance in wireless networks [22], channel selection in radio networks [23], model a student's behavior [24], clustering in wireless ad-hoc networks [25], power system stabilizers [26], backbone formation in ad-hoc wireless networks [27] and spectrum allocation in cognitive networks [28], graph partitioning problem [29], capacity assignment problem [30] and keyboard optimization problem [31, 32].

The L_{R-I} identical-payoff game algorithm is decentralized. However, it converges to one of the many Nash equilibria in the game matrix. Also, it is slower to converge. The Pursuit algorithm for the identical-payoff game (CPLA) exhibits faster convergence and it converges to the global maxima in the game matrix. However, it has one serious drawback. Since it is a centralized algorithm, its memory requirement grows exponentially with the the number of automaton in the group. Thus, in the next chapter, we propose and analyze an algorithm for identical-payoff games that is decentralized in nature and yet exhibits faster convergence.

3 COMPLETELY DECENTRALIZED GAMES OF LA

In this chapter, we discuss the application of indirect learning method (namely Pursuit learning algorithm) for zero-sum as well as identical-payoff games of learning automata. We propose a novel decentralized version of the Pursuit learning algorithm. We call this algorithm the Decentralized Pursuit Learning Algorithm (DPLA) [33]. Such a decentralized algorithm has significant computational advantages over its centralized counterpart. The theoretical study of such a decentralized algorithm requires the analysis to be carried out in a nonstationary environment. We use a novel bootstrapping argument to prove the convergence of the algorithm. To our knowledge, this is the first time such analysis has been carried out for zero-sum and identical-payoff games. Extensive simulation studies are described that demonstrate the fast and accurate convergence of the algorithm in a variety of game scenarios.

CPLA extends the single agent Pursuit learning algorithm to the multi-agent case. But as discussed earlier, the CPLA is a *centralized* algorithm in the sense that each automaton is aware of the actions taken by all the other automata in the team. The problem with the centralized algorithms is that they can be computationally expensive and require maintenance of estimate matrices whose size grows exponentially with the number of automata in the team. We describe below a *decentralized* version in which the Pursuit learning algorithm is applied by each automaton, independently of the other(s). We call this algorithm the Decentralized Pursuit Learning Algorithm (DPLA). The advantage of the DPLA algorithm over the CPLA is its computationally efficiency. The DPLA obviates the need for an individual automaton to communicate its choice of action to the other automata in the system at each instant of time. Also, the size of the estimate matrices grows linearly with the number of automata in the system. Suppose P automata $A_1, \ldots, A_P$ are involved in a game of identical payoff, with A_p having r_p possible actions. Then the size of each estimate matrix of CPLA

is $O(r_1 \times r_2 \times \cdots \times r_P)$. This exponential space complexity becomes untenable for systems with even a moderate number of automata. The DPLA algorithm, on the other hand, has space complexity that grows linearly with the number of automata in the team. The total size of all the estimate matrices of DPLA is $O(r_1 + r_2 + \cdots + r_P)$.

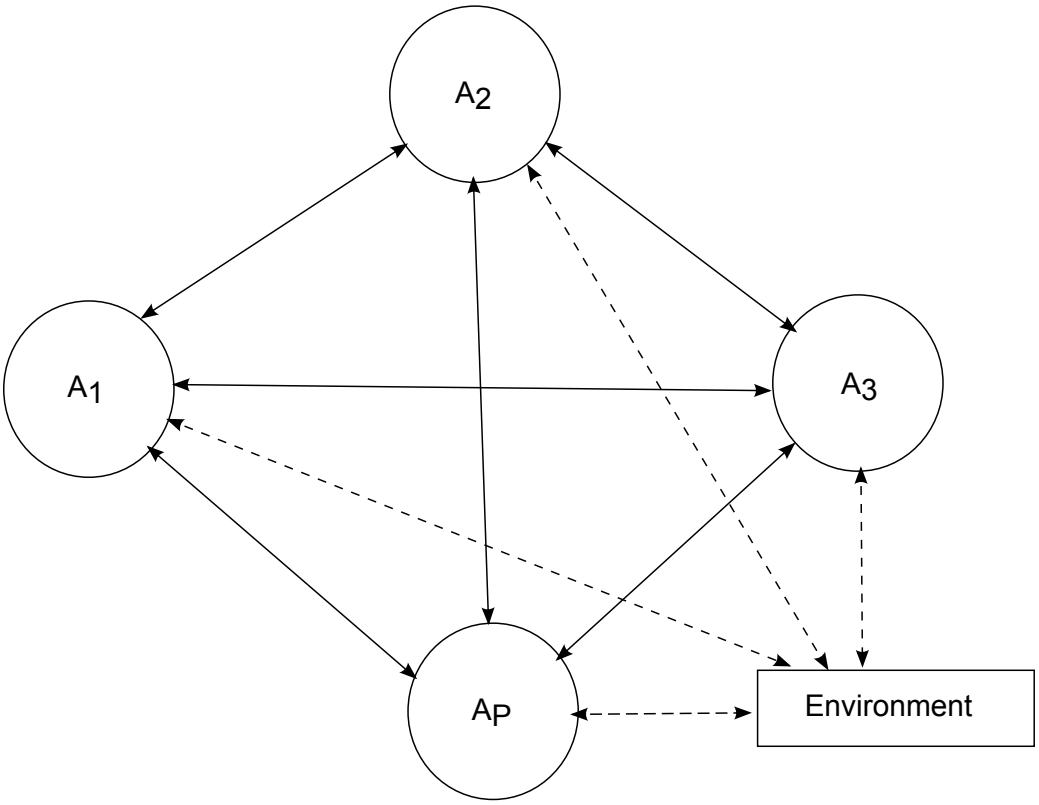


Figure 3.1. Schematic of CPLA - Figure 1

Figures (3.1) and (3.2) demonstrate the CPLA schematically. There are two mechanisms for implementing CPLA. As indicated in Figure (3.1), each automaton can receive action choices from all the other automata in the system. Each automaton can then compute the estimate matrices of CPLA. Alternatively, each automaton can send its selected action information to a central controller which in turn, calculates the estimate matrices and then all the other automata use this information to execute the CPLA. It is evident from the figures that under both these options, CPLA causes lot of communication overhead and requires exponential memory space.

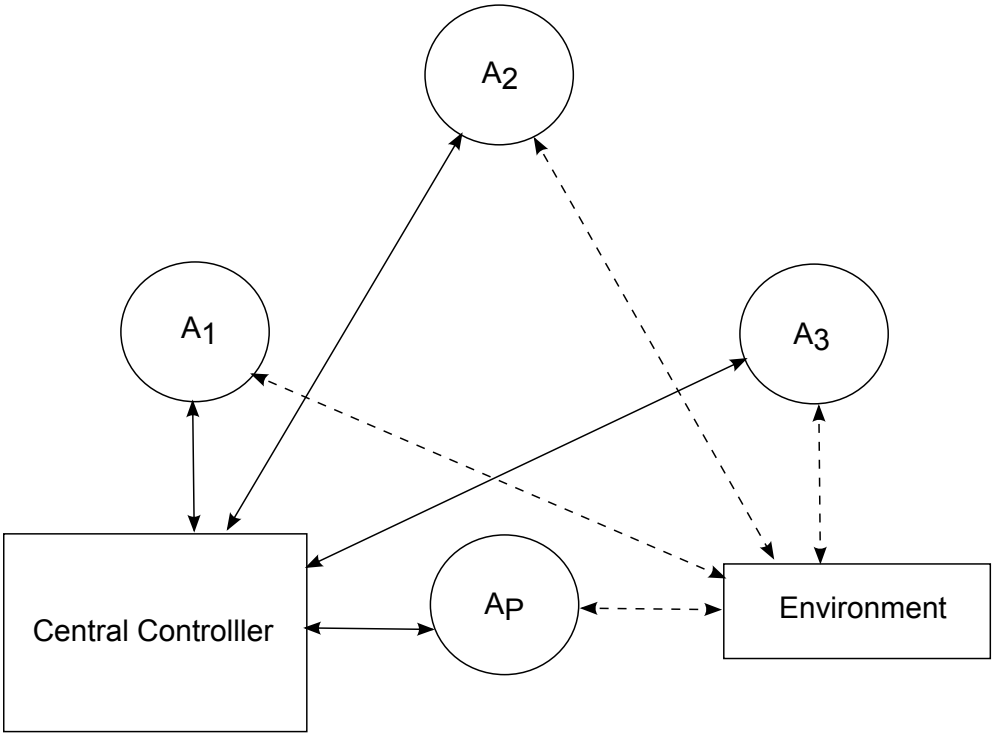


Figure 3.2. Schematic of CPLA - Figure 2

Figure (3.3) shows the corresponding schematic for the DPLA. As the schematic indicates, the DPLA does not require any communication between the participating automata nor does it need a centralized controller. The DPLA combines fast convergence of indirect learning techniques with the smaller memory and communication requirements of decentralized learning algorithms. However, the DPLA causes the environment to exhibit non-stationary properties, thus making the theoretical analysis more challenging.

3.1 Games of Learning Automaton

As described earlier, when multiple learning automaton interact with each other, this system can be modeled by using concepts from the Game theory [34]. The LAs can participate in various types of games. In this thesis, we will focus on the zero-sum

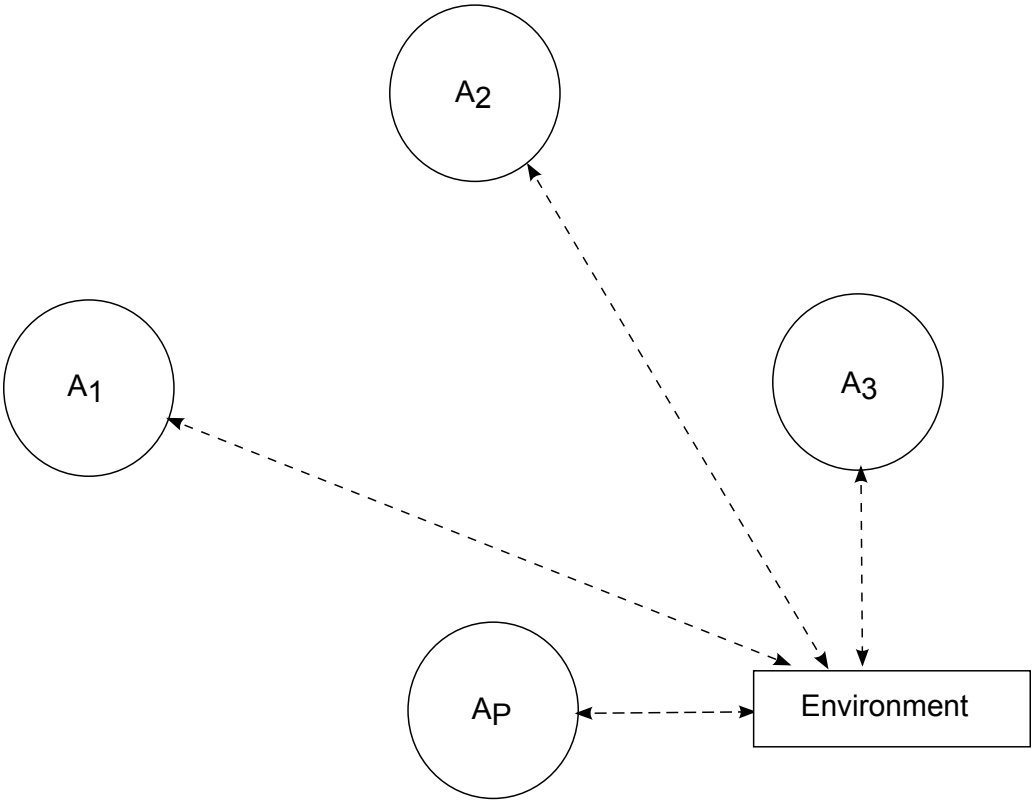


Figure 3.3. Schematic of DPLA

and identical-payoff games of LAs. We will describe these game setups in detail and in the process, also formalize the notation used in this chapter.

3.1.1 Identical Payoff Game

In an identical payoff game, all the automata participating in the game get the same payoff at the end of each iteration of the game. Suppose P automata $A_1, \ldots, A_P$ are involved in a game of identical payoff, with A_p having r_p possible actions (or strategies). Each play consists of each of the automata choosing an action and then the team getting a common payoff. This payoff will form the environmental response for each of the automata. The game is stochastic. For simplicity, we assume that the payoff is a random variable X taking values 0 or 1, with 1 indicating reward and 0 indicating penalty.

Such identical-payoff game is characterized by a hypermatrix $\mathbb{D} = [d_{i_1 i_2 \cdots i_P}]$ of dimension $r_1 \times r_2 \times \ldots \times r_P$. The elements of the said hypermatrix $\mathbb{D}$ (also called as a payoff matrix) are defined as:

$$d_{i_1 i_2 \cdots i_P} = \mathsf{E}[X \mid A_p \text{ chooses action } a_{pi_p},\ p = 1, 2, \ldots, P].$$

Suppose there is a choice of strategies, m_p^e by automaton A_p, forming an P-tuple of actions. Such action tuple is called a Nash equilibrium in pure strategies if for each p, $1 \leq p \leq P$

$$d_{m_1^e \ldots m_{p-1}^e m_p^e m_{p+1}^e \ldots m_P^e} \geq d_{m_1^e \ldots m_{p-1}^e m_p m_{p+1}^e \ldots m_P^e}, \forall m_p \in S_p$$

where S_p is the set of pure strategies of the player A_p. The m_p^e is called the Nash equilibrium strategy of player A_p.

3.1.2 Zero-sum Game

A zero-sum game consist of two automata. One automaton is called as the Row player while the other automaton is called as the Column player. Assume that the two automata, A_1 and A_2, have r_1 and r_2 actions (i.e. strategies) respectively. Let these actions be denoted by $a_{p1}, \ldots, a_{pr_p}$ for $p = 1, 2$. A single play of the zero-sum game consists of each automaton choosing an action, and then both the automata getting their respective payoffs. These payoffs will form the environmental response for each of the automata. The game is stochastic. At the end of every play, the automata receive a payoff which is a random variable. For simplicity, we assume that the random payoff X takes values 0 or 1, with 1 indicating reward and 0 indicating penalty.

The zero-sum game is characterized by a bimatrix $\mathbb{D} = [(d^1_{i_1 i_2}, d^2_{i_1 i_2})]$ of dimension $r_1 \times r_2$, representing the reward probabilities each automaton. The elements of this payoff bimatrix are defined as:

$$d^1_{i_1 i_2} = \mathsf{E}[X \mid A_p \text{ chooses action } a_{pi_p},\ p = 1, 2],$$

where E denotes mathematical expectation. Since it is a zero-sum game,

$$d^2_{i_1 i_2} = 1 - d^1_{i_1 i_2}.$$

The element $d^1_{i_m i_n}$ is said to be a saddle point of $\mathbb{D}$ if

$$d^1_{i_k i_n} < d^1_{i_m i_n} < d^1_{i_m i_l}, \forall k \neq m, \forall l \neq n.$$

Then i_m is the saddle point (or the equilibrium point) strategy of the Row player and i_n is the saddle point strategy of the Column player.

3.2 Decentralized Pursuit Learning Algorithm

Consider a sequence of probability distributions $\pi_p(t)$, for $t \geq 0$, on the action space of automaton A_p, $p = 1, 2, \ldots, P$. We assume that for each t, $\pi_p(t)$ is an r_p-dimensional probability vector in the probability simplex

$$\mathcal{S}_{r_p - 1} = \left\{ (s_1, \ldots, s_{r_p}) : \ s_k \geq 0 \text{ and } \sum_{k=1}^{r_p} s_k = 1 \right\} \subset \mathbb{R}^{r_p}.$$

These probability vectors are initialized to uniform initial value as follows:

$\pi_{p i_p}(0) = 1/r_p$, $i_p \in \mathcal{A}_p$, where $\mathcal{A}_p = \{1, 2, \ldots, r_p\}$.

DPLA [33] is a type of Pursuit learning (or estimator type) algorithm. Thus it makes use of the estimates of the environment parameters in the learning process. For this purpose, the DPLA uses the estimate vectors $\widehat{D}_p(t)$ that keep track of the empirical averages of rewards. The initial values of these estimate vectors, $\widehat{D}_p(0)$, are set to zero.

At iteration $t \geq 1$, each LA p samples its own action probability vector to select an action $\alpha_p(t) \sim \pi_p(t-1)$. Then based on this action choice, a reward $X(t) \sim \mathsf{P}_{\alpha(t)}$ is observed. Here $\mathsf{P}_{\alpha(t)}$ denotes the conditional distribution of the reward, given $\alpha(t) = [\alpha_1(t), \ldots, \alpha_P(t)]$ is the action tuple selected by the automata team. For $p = 1, 2, \ldots, P$, we define the sets

$$I_{p i_p}(t) = \{1 \leq s \leq t : \alpha_p(s) = i_p\}, \quad i_p \in \mathcal{A}_p,$$

These sets keep track of the iterations in which A_p played action i_p. Then the DPLA for both the zero-sum game and identical-payoff games can be described as follows:

1. At iteration $t \geq 1$, sample $\alpha_p(t) \sim \pi_p(t-1)$ for $p = 1, 2, \ldots, P$, and observe $X(t) \sim \mathsf{P}_{\alpha(t)}$.

2. Update
$$I_{pi_p}(t) = \begin{cases} I_{pi_p}(t-1) \cup \{t\} & \text{if } \alpha_p(t) = i_p \\ I_{pi_p}(t-1) & \text{if } \alpha_p(t) \neq i_p. \end{cases}$$

3. Update
$$\widehat{D}_{pi_p}(t) = \begin{cases} \widehat{D}_{pi_p}(t-1) + \frac{X(t) - \widehat{D}_{pi_p}(t-1)}{\#I_{pi_p}(t)} & \text{if } \alpha_p(t) = i_p \\ \widehat{D}_{pi_p}(t) = \widehat{D}_{pi_p}(t-1) & \text{if } \alpha_p(t) \neq i_p, \end{cases}$$
where $\#E$ denotes cardinality of the set E.

4. Compute: For $p = 1, 2, \ldots, P$,
$$\omega_p(t) = \arg\max_{i_p \in \mathcal{A}_p} \widehat{D}_{pi_p}(t)$$

5. Update: For $p = 1, 2, \ldots, P$,
$$\pi_p(t) = \pi_p(t-1) + \lambda\{\delta_{\omega_p(t)} - \pi_p(t-1)\},$$
where δ_x denotes a unit vector of suitable dimension with mass 1 at component with index x and all other values are set to zero. The parameter $\lambda \in [0, 1]$ is the learning parameter.

6. If convergence criteria are met, then stop; otherwise, set $t = t + 1$ and return to step 1.

The convergence criteria can be different depending on the application at hand. However, typically, it is defined based on a certain action probability value threshold. When the action probability value (the π values) of the automaton reach a certain threshold, that particular automaton is termed as a converged automaton. Intuitively, one would expect that under DPLA, each automaton would learn the optimal action, so if there exists an optimal action tuple in $\mathbb{D}$, then we would expect that the automata team running the DPLA to converge to this optimal action tuple. But the analysis is more complicated than in the case of the CPLA algorithm because the DPLA algorithm causes the environment to display non-stationary characteristics.

3.3 Convergence Analysis

This section gives a convergence analysis for the DPLA applied to the zero-sum as well as the identical-payoff games. For this analysis we shall consider an infinite time-horizon and show almost sure convergence of the DPLA under certain constraints imposed on the game matrix $\mathbb{D}$. We also assume that the $\lambda = \lambda_t$ vanishes at a certain rate as $t \to \infty$.

3.3.1 Vanishing λ and The ε-optimality

For the DPLA algorithm, we define ε-optimality as follows: *for any* $\varepsilon, \delta > 0$*, there exists* $T^* = T^*(\varepsilon, \delta)$ *and* $\lambda^* = \lambda^*(\varepsilon, \delta)$ *such that*

$$\Pr\left[\min_p \pi_{pm_p}(t) > 1 - \varepsilon\right] > 1 - \delta$$

for all $t > T^*$ *and* $\lambda < \lambda^*$. This is a "finite-time and fixed λ" notion of convergence. We argue that both "infinite-time" and "decreasing λ" are implicit in the definition of ε-optimality.

- $T^*(\varepsilon, \delta)$ increases as ε and/or δ decreases, and if the number iterations increases, the upper bound λ^* must decrease. Therefore, in ε-optimality, λ is indirectly linked to the number of iterations through ε, δ.

- The critical part of proving ε-optimality is establishing a monotonicity in the dominant action equilibrium sampling probability. This boils down to showing that the $\widehat{D}$'s are correctly ordered *forever* after a certain number of iterations with probability $> 1-\delta$; see the proof of Theorem 3.1 in [35] and, in particular, the definition of the event $E_2(k)$ on p. 594. Therefore, despite the fact that the approach is "finite-time" in nature, an implicit control of the estimates over an infinite time-horizon is generally needed to prove ε-optimality.

However, the recommendation for a fixed $\lambda \in (0, 1)$ in the DPLA algorithm description and a vanishing λ in theoretical analysis is not a contradiction. The theory requires only that $\lambda = \lambda_t$ vanish at a certain rate as $t \to \infty$, and this can still be satisfied if λ_t is constant over a finite initial sequence of iterations. Moreover, in practical problems where resources are fixed and a specified number of iterations T are available, the rate of decay for λ_t can be used to determine a suitably small fixed $\lambda \approx \lambda_T$.

3.3.2 Preliminary Lemmas

In this section, we define some additional notation and the preliminary lemmas. We define the following increasing sequences of σ-algebras:

$$\{\mathscr{A}_t^p : t \geq 0\}, \quad p = 1, 2, \ldots, P, \tag{3.1}$$

where $\mathscr{A}_t^p$ tracks the information accumulated by the automaton A_p, up to and including iteration t; $\mathscr{A}_0^p$ is the trivial σ-algebra. We also define:

$$\lambda_t = 1 - \theta^{1/t}, \quad t \geq 1, \tag{3.2}$$

where $\theta \in (e^{-1}, 1)$ is selected arbitrary. Note that $\lambda_t \downarrow 0$ for each θ.

Lemma 1. For λ_t in (3.2) with $\theta \in (e^{-1}, 1)$,

$$\Pr\left[\lim_{t\to\infty} \#I_{pi_p}(t) = \infty\right] = 1$$

for all $p = 1, \ldots, P$ and $i_p \in \mathcal{A}_p$.

Proof: Write $\#I_{pi_p}(t) = \sum_{s=1}^{t} \xi_{pi_p}(s)$, where

$$\xi_{pi_p}(s) = \begin{cases} 1 & \text{if } \alpha_p(s) = i_p \\ 0 & \text{otherwise.} \end{cases}$$

For simplicity, drop the p and i_p subscripts. Then the goal is to show that $\sum_{t=1}^{\infty} \xi(t) = \infty$. The sequence $\{\xi(t) : t \geq 1\}$ is adapted to $\{\mathscr{A}_t : t \geq 1\}$, and according to Lemma 3.1 of [35],

$$\begin{aligned} \mathsf{E}[\xi(t) \mid \mathscr{A}_{t-1}] &= \Pr[\text{this action taken at time } t] \\ &\geq \pi(0) \prod_{s=1}^{t} (1 - \lambda_s) \\ &= \pi(0)\theta^{\gamma(t)}, \end{aligned}$$

where $\gamma(t) = \sum_{s=1}^{t} \frac{1}{s}$. The claim will follow from Lévy's extension of the Borel-Cantelli lemma if $\sum_{t=1}^{\infty} \theta^{\gamma(t)} = \infty$; see also [36] , [37, p. 96]. But since $\gamma(t)$ is asymptotically equivalent to $\ln(t)$, and $\ln(t) = \ln(\theta)\log_\theta(t)$, where $\log_\theta$ denotes log base θ, we have

$$\theta^{\ln(t)} = \theta^{\ln(\theta)\log_\theta(t)} = t^{\ln(\theta)}.$$

Therefore, since $e^{-1} < \theta < 1$, $\sum_{t=1}^{\infty} t^{\ln(\theta)} = \infty$, proving the claim.

It follows immediately from Lemma 1 and Bonferroni's inequality that *all* actions are tried infinitely often with probability 1. So we get:

$$\begin{aligned} &\Pr\Big[\bigcap_p \bigcap_{i_p} \Big\{\lim_{t\to\infty} \#I_{pi_p}(t) = \infty\Big\}\Big] \\ &\quad \geq \sum_p \sum_{i_p} \Pr\left[\lim_{t\to\infty} \#I_{pi_p}(t) = \infty\right] - \sum_p r_p + 1 \qquad (3.3) \\ &\quad = 1. \end{aligned}$$

Lemma 1 and its extension (3.3) justifies the law of large numbers-type of reasoning upon the convergence argument for DPLA is based. $\widehat{D}(t)$ is an empirical average of some unknown theoretical quantity. But here, unlike in the CPLA scenario, the theoretical quantity being estimated is itself random and changing with t. The DPLA algorithm is a constrained version (where action sampling is forced to be independent across players) of the CPLA, but there is a loss of information in the sense that no automaton is aware of the actions taken by the others at a given iteration.

We now describe a precise mathematical formulation of this "loss of information" of the DPLA. By definition, the entries in the game matrix $\mathbb{D}$ are the expected rewards under *complete information*; that is,

$$d_{i_1 \cdots i_P} = \mathsf{E}[X(t) \mid \alpha_p(t) = i_p, p = 1, \ldots, P].$$

But in the DPLA, no automaton is aware of the actions taken by the others, so from A_p's perspective, the observed $X(t)$ is a proxy for the marginal expectation at iteration t, namely,

$$\begin{aligned} D_{pi_p}(t) &= \mathsf{E}[X(t) \mid \alpha_p(t) = i_p] \\ &= \sum_{i_1, \cdots, i_{p-1}, i_{p+1}, \cdots, i_P} d_{i_1 \cdots i_{p-1} i_p i_{p+1} \cdots i_P} \prod_{q \neq p} \pi_{q i_q}(t-1), \end{aligned} \tag{3.4}$$

a weighted average of the possible awards A_p could earn for playing action i_p. Therefore, one automaton not knowing the actions taken by the other automaton results in a loss of information. It has the effect of marginalizing over appropriate dimensions of the game matrix. Also, it is evident that the environment reward probabilities $D_{pi_p}(t)$, are random and changing with t as the corresponding π's change. This causes the automata environment to exhibit non-stationary properties and makes the convergence analysis more challenging compared to that of CPLA.

For the σ-algebras $\mathscr{A}_t^p$ in (3.1), define:

$$\mathscr{F}_t^p = \sigma(\mathscr{A}_{t+1}^p, \{\mathscr{A}_t^q : q \neq p\}),$$

where $\sigma(\mathscr{C})$ denotes the smallest σ-algebra containing the events in $\mathscr{C}$. Also, define:

$$\tau_{pi_p}(t) = \text{iteration when } A_p \text{ plays } i_p \text{ for the } t^{\text{th}} \text{ time.} \tag{3.5}$$

Although, $\tau_{pi_p}(t)$ is a random variable, it is $\mathscr{F}_{t-1}^p$-measurable. This variable indicates the *sampling times* in the sense of Breiman [37], Definition 5.9.

It is straightforward to check that

$$\Delta_{pi_p}(t) := X(\tau_{pi_p}(t)) - D_{pi_p}(\tau_{pi_p}(t))$$

forms a martingale difference sequence with respect to $\mathscr{F}_t^p$. So we get:

$$\mathsf{E}[\Delta_{pi_p}(t) \mid \mathscr{F}_{t-1}^p] = 0 \quad t \geq 1.$$

Therefore, it follows immediately that:

$$M_{pi_p}(t) = \sum_{s \in I_{pi_p}(t)} \Delta_{pi_p}(t) \tag{3.6}$$

is a martingale with respect to $\mathscr{F}_t^p$. Alternatively, one could construct these martingales by applying Doob's optional sampling theorem; see [38] or [37, Theoremm 5.10].

Now if we define

$$\overline{D}_{pi_p}(t) = \frac{1}{\#I_{pi_p}(t)} \sum_{s \in I_{pi_p}(t)} D_{pi_p}(s),$$

then we see that the difference between $\widehat{D}$'s and $\overline{D}$'s is a function of the martingale M. This observation allows us to analyze the fluctuations in the estimates $\widehat{D}(t)$ in Lemma 2 below.

The event that a sequence of events $\{B(t) : t \geq 1\}$ occurs infinitely often (i.o.) will written as

$$\{B(t) \text{ i.o.}\} = \bigcap_{t\geq 1} \bigcup_{s\geq t} B(s);$$

This indicates that for any t there exists an $s \geq t$ such that $B(s)$ occurs.

Lemma 2. If $\lambda = \lambda_t$ is as in (3.2) with $e^{-1} < \theta < 1$, then for all $\varepsilon > 0$,

$$\Pr\left[\max_p \max_{i_p \in \mathcal{A}_p} |\widehat{D}_{pi_p}(t) - \overline{D}_{pi_p}(t)| > \varepsilon \text{ i.o.}\right] = 0.$$

Proof: Let $B(t) = B_\varepsilon(t)$ denote the sequence of events given by:

$$\begin{aligned} B(t) &= \left\{\max_p \max_{i_p} |\widehat{D}_{pi_p}(t) - \overline{D}_{pi_p}(t)| > \varepsilon\right\} \\ &= \bigcup_p \bigcup_{i_p} \{|\widehat{D}_{pi_p}(t) - \overline{D}_{pi_p}(t)| > \varepsilon\} \end{aligned}$$

The goal is to show $\Pr[B(t) \text{ i.o.}] = 0$, but since

$$\Pr[B(t) \text{ i.o.}] \leq \sum_p \sum_{i_p} \Pr[|\widehat{D}_{pi_p}(t) - \overline{D}_{pi_p}(t)| > \varepsilon \text{ i.o.}], \tag{3.7}$$

it is clear that we need only show $\Pr[|\widehat{D}_{pi_p}(t) - \overline{D}_{pi_p}(t)| > \varepsilon \text{ i.o.}] = 0$ for each (p, i_p) combination. For simplicity, we drop the (p, i_p) subscripts in the following discussion. The difference $|\widehat{D}(t) - \overline{D}(t)|$ changes only at the sampling times $\tau(t)$, and by Lemma 1 we know that there are infinitely many such sampling times. Since the difference can be more than ε infinitely often if and only if it is more than ε at infinitely many sampling times, for our purposes we can (without loss of generality) modify the time scale so that $|\widehat{D}(t) - \overline{D}(t)| = t^{-1}|M(t)|$, where $M(t)$ is the martingale defined in (3.6). The summands in (3.6) are bounded by 2, so by Azuma's inequality [39] we have (for the modified time scale)

$$\begin{aligned} \Pr[|\widehat{D}(t) - \overline{D}(t)| > \varepsilon] &= \Pr[|M(t)| > t\varepsilon] \\ &\leq 2\exp\{-t\varepsilon^2/8\}. \end{aligned}$$

By the Borel-Cantelli lemma,

$$\sum_{t=1}^{\infty} \Pr[|\widehat{D}(t) - \overline{D}(t)| > \varepsilon] \leq \sum_{t=1}^{\infty} 2e^{-t\varepsilon^2/8} < \infty$$

implies

$$\Pr[|\widehat{D}(t) - \overline{D}(t)| > \varepsilon \text{ i.o.}] = 0,$$

and so the lemma now follows from (3.7).

Lemma 2 implies that $\widehat{D}(t)$ will be close to $\overline{D}(t)$ for all sufficiently large t. However, convergence of the Pursuit Learning algorithm requires that $\widehat{D}(t)$ be close to $D(t)$. This would follow from the previous lemma if it could be shown that $\overline{D}(t)$ is close to $D(t)$. **We will establish the necessary ordering of the $\overline{D}$'s in the case of $P = 2$ and $r_1 = r_2 = 2$.**

The next lemma follows immediately from the definition of dominating strategy equilibrium.

Lemma 3. If the game matrix $\mathbb{D}$ is 2×2, and there exists a unique dominating strategy equilibrium, then one automaton will always have a clear preference ordering between its actions, independent of the other player's actions.

3.3.3 Bootstrapping Mechanism

Suppose Player 2 has the clear preference ordering indicated in the Lemma 3. Then mathematically this means that for any number $\pi \in [0, 1]$,

$$\pi d_{21} + (1 - \pi)d_{22} \gtrless \pi d_{11} + (1 - \pi)d_{12}$$

The "$>$" symbol means the automaton (or Player) 1 prefers Action 1, "$<$" means the automaton 1 prefers Action 2. For example, consider the following 2×2 game matrix:

$$\mathbb{D} = \mathbb{D}_{2\times 2} = \begin{bmatrix} 0.4 & 0.9 \\ 0.2 & 0.6 \end{bmatrix}$$

Note that the dominating action equilibrium is d_{12}. In this case, automaton 2 has the clear preference. Indeed, we see that automaton 2's $D(t)$'s, and hence $\overline{D}(t)$'s are clearly separated for all t. Therefore, Lemma 2 implies that, eventually, automaton 2's $\widehat{D}$'s will be correctly ordered and, after this point, its $\pi_{22}(t)$ will be monotonically increasing. Once $\pi_{22}(t)$ is sufficiently close to 1, the automaton 1's $\widehat{D}(t)$'s will be correctly ordered, and its $\pi_{11}(t)$ will likewise be monotonically increasing. This is the essence of the novel ***bootstrapping argument***. The bootstrapping mechanism means that when one automaton converges, the other is then forced to converge. It is easy to see that similar result applies when the dominating strategy equilibrium resides at other locations in the game matrix.

3.3.4 2×2 Identical Payoff Game

Theorem 1. Let $\mathbb{D}$ be the game matrix of an 2×2 identical payoff stochastic game with incomplete information. Assume that there is a unique dominating strategy equilibrium in the game matrix. If λ_t is as in (3.2) with $e^{-1} < \theta < 1$, then $\pi_{pm_p}(t) \to 1$ almost surely for $p = 1, 2$.

Proof: Without loss of generality, assume that d_{11} is the dominating action equilibrium point of the 2×2 game matrix $\mathbb{D}$, and that Player 1 has the clear preference ordering indicated in Lemma 3. Thus, the entries $d_{i_1 i_2}$ satisfy

$$d_{22} < d_{21} < d_{12} < d_{11}$$

Let $\delta = d_{12} - d_{21}$ be the minimum separation between $\overline{D}_{11}(t)$ and $\overline{D}_{12}(t)$, and set $\varepsilon = \delta/2$. Then by using Lemma 2, the selected value of ε guarantees that there exists a time T such that $\widehat{D}_{11}(t) > \widehat{D}_{12}(t)$ for all $t > T$. Once this event occurs, $\pi_{11}(t)$ is monotonically increasing. For notational simplicity, assume $T = 0$. If the estimates are correctly ordered (i.e., $\omega_1(t) = 1$ for all $t > 0$), then it follows that, for $t \geq 1$,

$$\pi_{11}(t) = \pi_{11}(0)\prod_{s=1}^{t}(1-\lambda_s) + \sum_{s=1}^{t}\lambda_s \prod_{r=s+1}^{t}(1-\lambda_r),$$

with the conventions that a sum and product over an empty index set is 0 and 1, respectively. For λ_t in (3.2) this simplifies to

$$\begin{aligned}
\pi_{11}(t) &= \pi_{11}(0)\theta^{\gamma(t)} + \sum_{s=1}^{t}(1-\theta^{1/s})\theta^{\gamma(t)-\gamma(s)} \\
&= \theta^{\gamma(t)}\left[\pi_{11}(0) + \sum_{s=1}^{t}\left(\theta^{-\gamma(s)} - \theta^{-\gamma(s-1)}\right)\right] \\
&= \theta^{\gamma(t)}\left[\pi_{11}(0) + \theta^{-\gamma(t)} - 1\right] \\
&= 1 - \theta^{\gamma(t)}[1-\pi_{11}(0)].
\end{aligned}$$

Since $e^{-1} < \theta < 1$ and $\gamma(t) \sim \ln(t) \to \infty$, it is clear that $\pi_{11}(t) \uparrow 1$ as $t \to \infty$. Now once $\pi_{11}(t)$ becomes sufficiently close to 1, automaton 2 will have separation between its $\overline{D}(t)$'s, so eventually $\widehat{D}_{21}(t) > \widehat{D}_{22}(t)$ for all $t > T'$, where $T' > T$. After this point, $\pi_{21}(t)$ is monotonically increasing, and the previous argument may be applied to show that $\pi_{21}(t) \to 1$ as $t \to \infty$.

3.3.5 Zero-sum Game

Theorem 2. Let $\mathbb{D}$ be the game matrix of a 2×2 zero-sum stochastic game with incomplete information. Let the saddle point for the game be unique. If λ_t is as in (3.2) with $e^{-1} < \theta < 1$, then $\pi_{pm_p}(t) \to 1$ almost surely for $p = 1, 2$.

Proof: Without loss of generality, assume that d_{11} is the saddle point of the game matrix $\mathbb{D}$. This means that d^1_{11}, d^1_{12} and d^1_{21} are ordered. The ordering will be $d^1_{21} < d^1_{11} < d^1_{12}$. Now depending upon the value of d^1_{22}, we will consider four cases and show that in all these cases, the automata team will converge to the saddle point.

Case 1: $d^1_{22} < d^1_{21} < d^1_{11} < d^1_{12}$. By using the arguments given in the Section 3.3.4, we can prove that since automaton 1 has clear preference among its actions, it will

first converge to the saddle action. This in turn will cause the other automaton to converge to its saddle action. Hence the whole team converges to the saddle point.

Case 2: $d^1_{21} < d^1_{11} < d^1_{22} < d^1_{12}$. Since this is a zero-sum game, the ordering for the Player 2 will be $d^2_{21} > d^2_{11} > d^2_{22} > d^2_{12}$. Now, by using the arguments given in the Section 3.3.4, we can prove that since automaton 2 has clear preference among its actions, it will first converge to the saddle action. This in turn will cause automaton 1 to converge to its saddle action. The whole team thus converges to the saddle point.

The other two cases, namely,

$$d^1_{21} < d^1_{22} < d^1_{11} < d^1_{12}$$

and

$$d^1_{21} < d^1_{11} < d^1_{12} < d^1_{22}$$

are the same as cases 1 and 2 above, respectively.

3.4 Simulation Results

We implemented the novel DPLA for both zero-sum and identical payoff games. The simulation results obtained are in confirmation with the theoretical convergence proofs discussed in the earlier section.

3.4.1 2×2 Identical-Payoff Game

The following 2×2 game matrix was used in the simulation of the decentralized zero-sum game:

$$\mathbb{D} = \mathbb{D}_{2\times 2} = \begin{bmatrix} 0.4 & 0.9 \\ 0.2 & 0.6 \end{bmatrix}$$

This game matrix has the dominating action equilibrium at $\mathbb{D}(1, 2)$.

Figure (3.4) shows the change in action probabilities of the two automata as they converge to the dominating action equilibrium. Figure (3.5) shows the expected value

of the marginal reward ($D(t)$'s) and its estimate ($\widehat{D}(t)$'s) for every action of both the automata. The non-stationarity of the environment is exhibited by the varying $D(t)$'s. It is evident that a correct separation between the $\widehat{D}(t)$'s of two actions occurs as the automata begin to converge. This separation is achieved fairly quickly, even though there may be some gaps between the $\widehat{D}(t)$'s and corresponding $D(t)$'s. Lemma 3 says that the $\widehat{D}(t)$'s will eventually find the true $D(t)$'s, but the decentralization causes a nontrivial reduction in the rate of convergence, explaining the gaps between the $\widehat{D}(t)$'s and $D(t)$'s present in some cases.

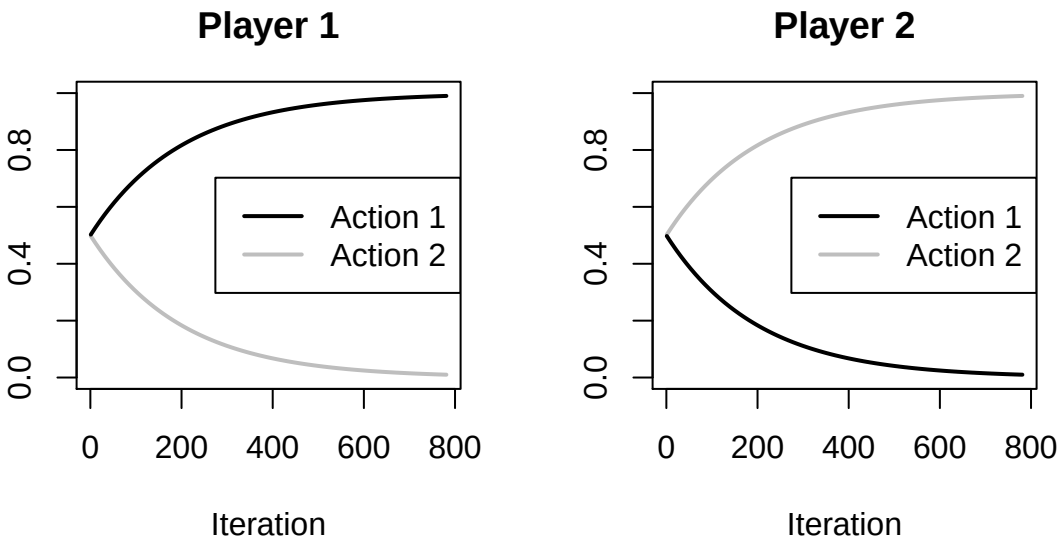


Figure 3.4. Action Probabilities $\pi_{pi_p}(t)$ for the Decentralized Pursuit Algorithm in the 2×2 Identical Payoff Game in Section 3.4.1

3.4.2 Identical-Payoff Game for Arbitrary Game Matrix

Although this convergence proofs of DPLA given earlier apply to a $\mathbb{D}_{2\times 2}$ game matrix, *the same dominating equilibrium convergence is observed when the game matrix* $\mathbb{D}_{r_1\times r_2\times \ldots \times r_P}$ *has arbitrary number of players and each player has arbitrary number of*

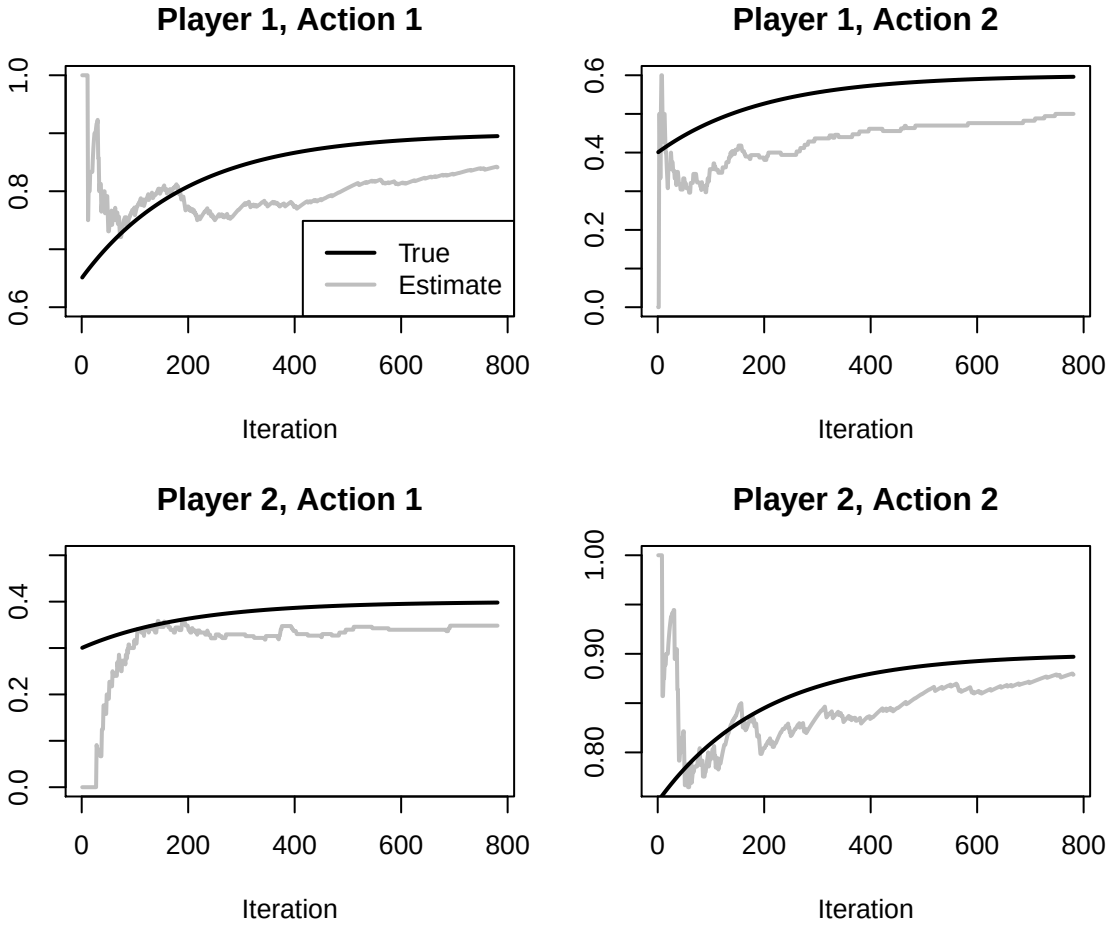


Figure 3.5. $D(t)$ (Black line) and $\widehat{D}(t)$ (Gray Line) for the Decentralized Pursuit Algorithm in the 2×2 Identical Payoff Game in Section 3.4.1

actions. If $\mathbb{D}_{r_1 \times r_2 \times \ldots \times r_P}$ has no dominance structure, then the algorithm converges to one of the Nash equilibria (modes) of the game matrix.

We consider following general $\mathbb{D}_{2\times 2}$ game matrix:

$$\mathbb{D} = \mathbb{D}_{2\times 2} = \begin{bmatrix} 0.4 & 0.6 \\ 0.8 & 0.2 \end{bmatrix}$$

This game matrix has two Nash equilibrium points at $\mathbb{D}(1,2)$ and $\mathbb{D}(2,1)$ respectively. In the simulations, it was observed that the automata team always converges to one of the Nash equlibria. Figure (3.6) shows the change in action probabilities of both the automata and Figure (3.7) shows the expected value of the marginal reward ($D(t)$'s) and its estimates ($\widehat{D}(t)$'s) for every action of both the automata.

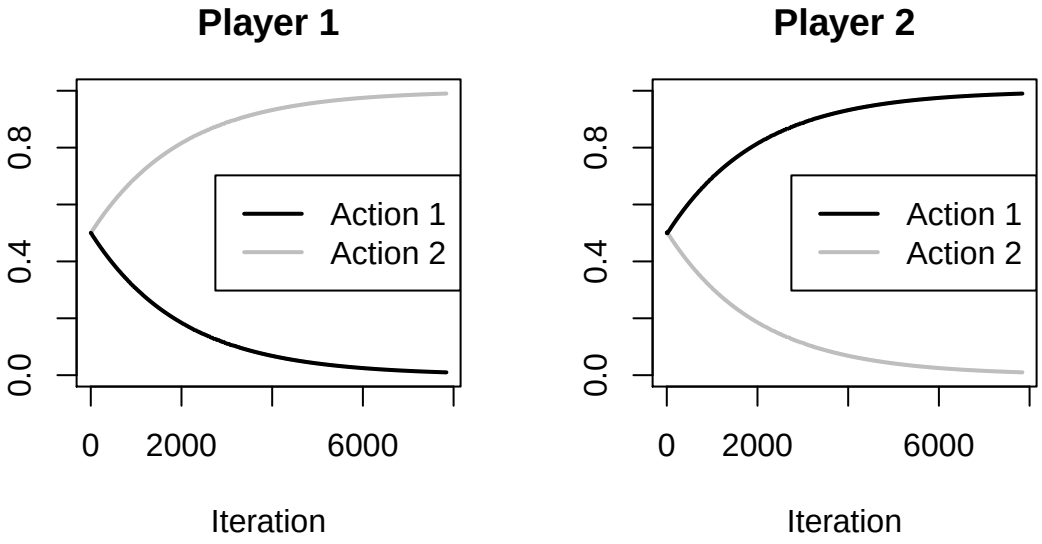


Figure 3.6. Action Probabilities $\pi_{pi_p}(t)$ for the Decentralized Pursuit Algorithm in the 2×2 Identical Payoff Game in Section 3.4.2

3.4.3 2×2 Zero-Sum Game

The following 2×2 game matrix was used in the simulation of the decentralized zero-sum game:

$$\mathbb{D} = \mathbb{D}_{2\times 2} = \begin{bmatrix} 0.6 & 0.8 \\ 0.3 & 0.7 \end{bmatrix}$$

This game matrix has saddle point at $\mathbb{D}(1,1)$.

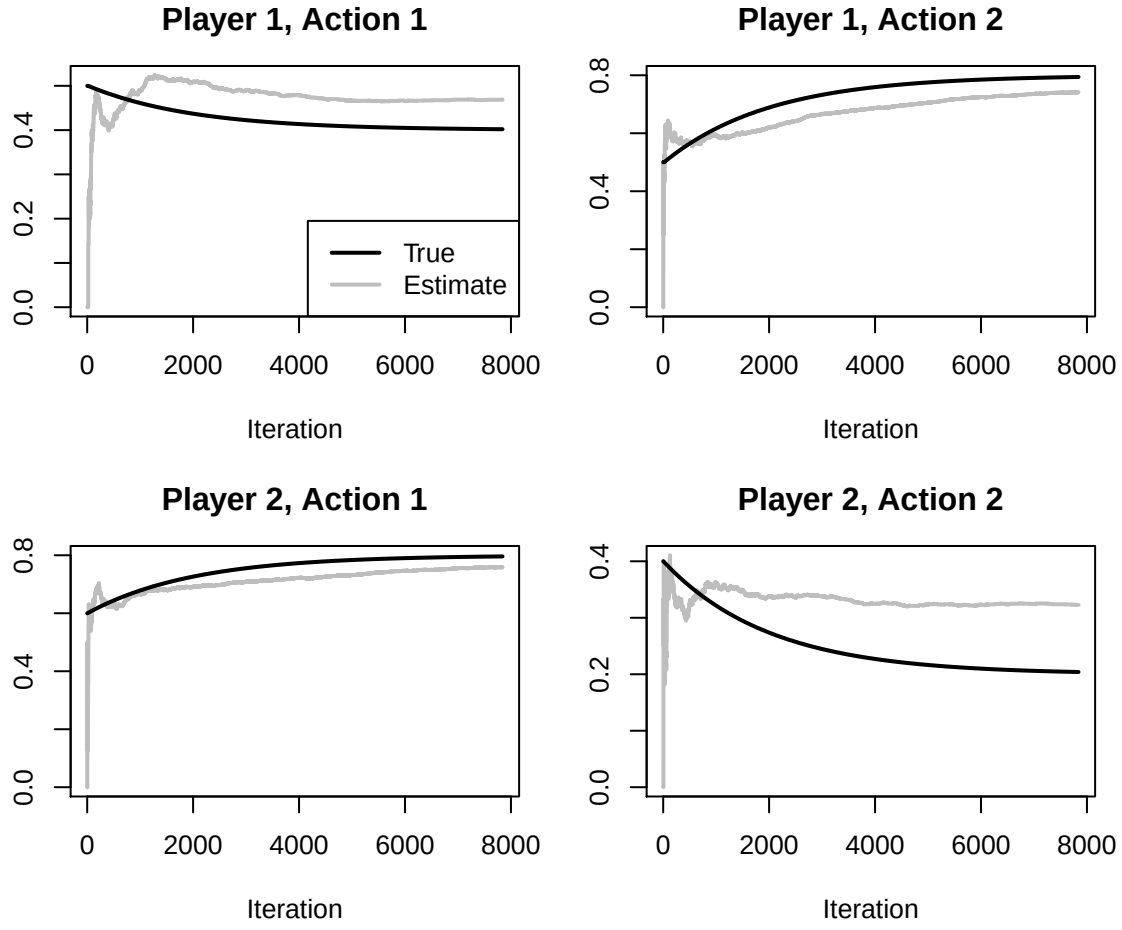


Figure 3.7. $D(t)$ (Black line) and $\widehat{D}(t)$ (Gray Line) for the Decentralized Pursuit Algorithm in the 2×2 Identical Payoff Game in Section 3.4.2

Figure (3.8) shows the trajectory of action probabilities of the two automata in the team. As the figure indicates, the action probability for the saddle action increases and that of the non-saddle action decreases monotonically.

Figure (3.9) shows the expected value of the marginal reward ($D(t)$'s) and its estimate ($\widehat{D}(t)$'s) for every action of both the automata. As explained earlier, the $D(t)$'s keep changing with time (shown by the black line), thus making the environment

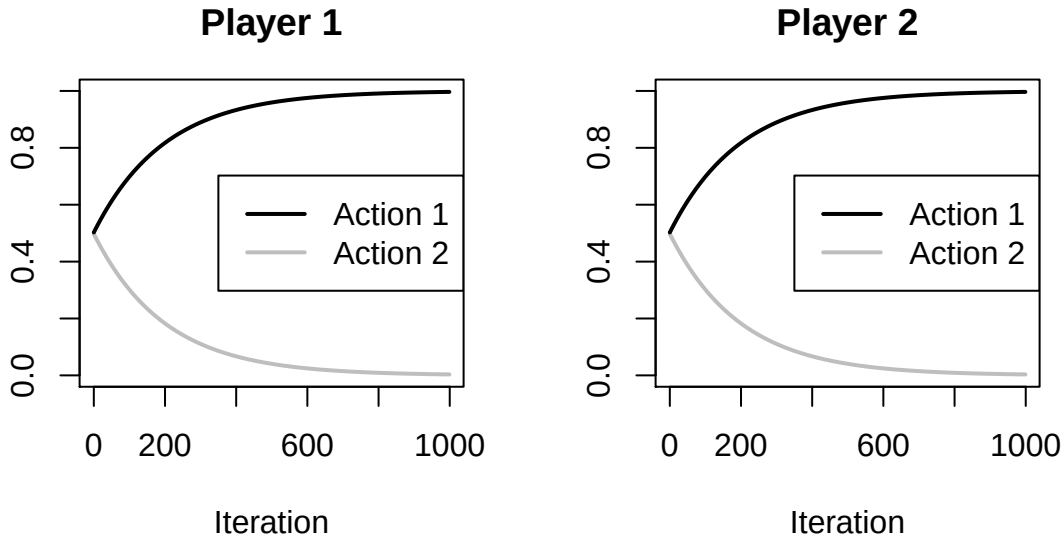


Figure 3.8. Action Probabilities $\pi_{pi_p}(t)$ for the Decentralized Pursuit Algorithm in the 2×2 Zero-sum Game in Section 3.4.3

non-stationary. The gray line indicates the running average for each action ($\widehat{D}(t)$'s). As expected, the $\widehat{D}(t)$'s get into proper ordering as the automata begin to converge.

3.4.4 Zero-sum Game for Arbitrary Game Matrix

Although, this paper gives a saddle-point convergence proof for zero-sum game consisting of $\mathbb{D}_{2\times2}$ game matrix, *the same saddle point convergence is obtained when both the automata have arbitrary number of actions.* Consider following $\mathbb{D}_{4\times4}$ game matrix:

$$\mathbb{D} = \mathbb{D}_{4\times4} = \begin{bmatrix} 0.7 & 0.3 & 0.2 & 0.5 \\ 0.9 & 0.4 & 0.6 & 0.5 \\ 0.4 & 0.1 & 0.6 & 0.7 \\ 0.2 & 0.3 & 0.5 & 0.8 \end{bmatrix}$$

This game matrix has the saddle point at $\mathbb{D}(2, 2)$.

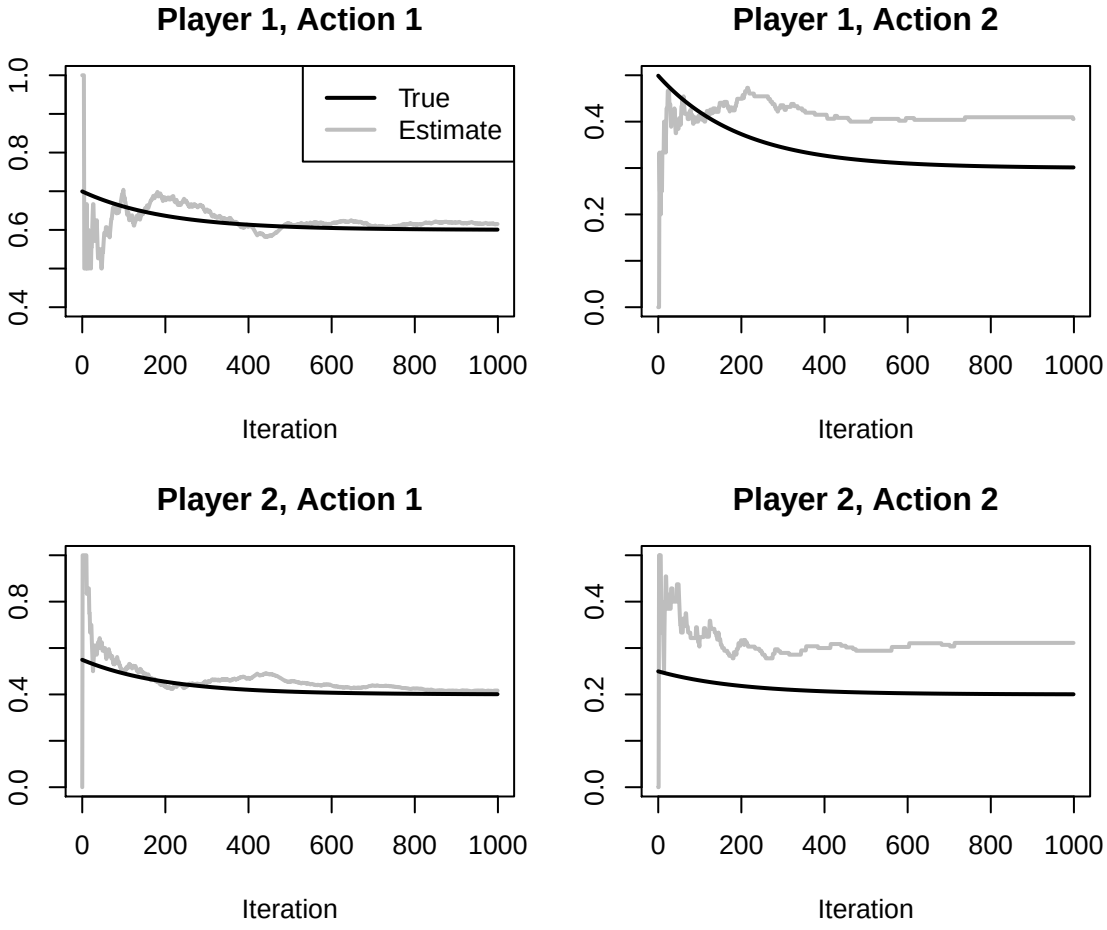


Figure 3.9. $D(t)$ (Black line) and $\widehat{D}(t)$ (Gray Line) for the Decentralized Pursuit Algorithm in the 2×2 Zero-sum Game in Section 3.4.3

Figure (3.10) shows the change in action probabilities of the two automata during the run of the algorithm until the convergence is reached. Figures (3.11) and (3.13) show the marginal reward ($D(t)$'s) and its estimate ($\widehat{D}(t)$'s) for every action of both the automata. As expected, the $\widehat{D}(t)$'s get into proper ordering as the automata begin to converge.

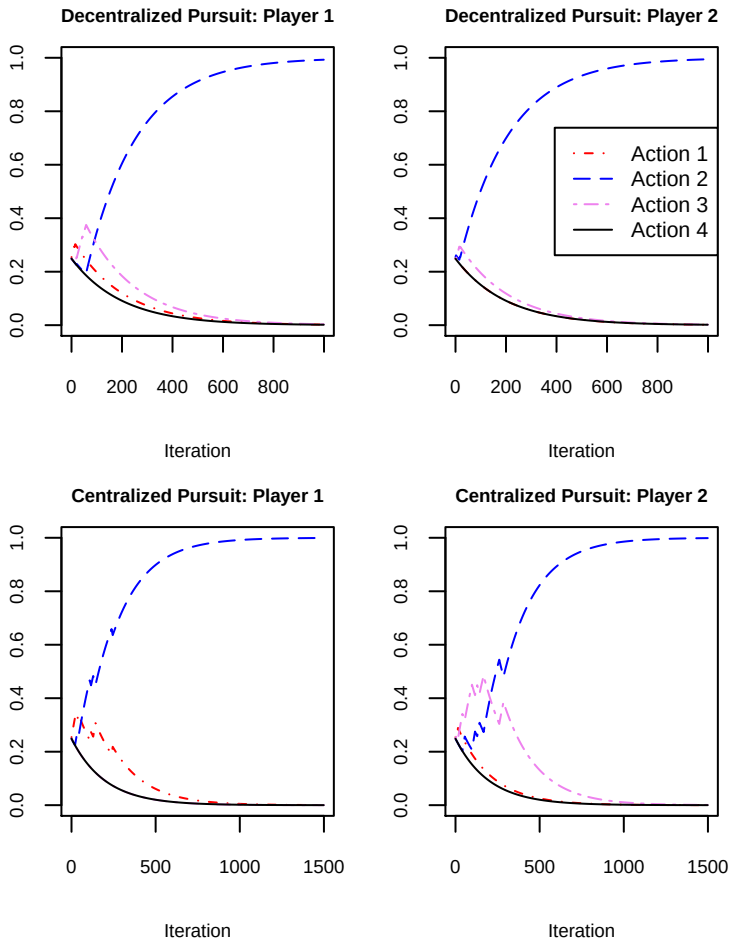


Figure 3.10. Comparison of Various Algorithms : Trajectory of Action Probabilities $\pi_{pi_p}(t)$

3.4.5 Zero-sum Game Using CPLA

For comparison, we also present the action probability plots for the zero-sum CPLA using the same $\mathbb{D}_{4\times4}$ matrix.

As the figures indicate, the DPLA algorithm has faster convergence than the CPLA. Although in Figure (3.10), the convergence speed of the DPLA appears to be only slightly faster (or comparable to) than the centralized Pursuit learning game algorithm, this difference increases with the increase in the number of automata in the team. This may be due to the fact that a large number of trials were needed for the large hypermatrix estimate $(\widehat{D})$ to stabilize to sufficiently accurate values.

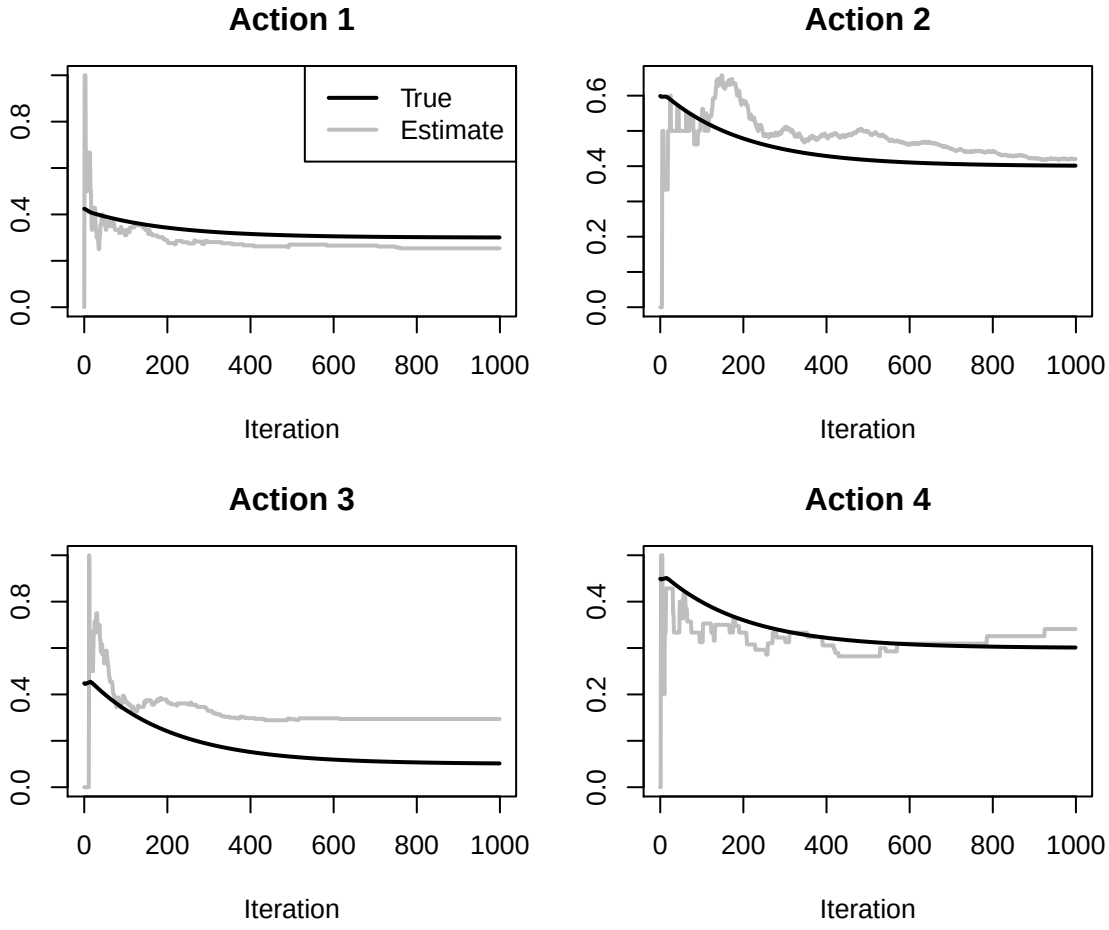


Figure 3.11. $D(t)$ (Black line) and $\widehat{D}(t)$ (Gray Line) of Player 1 for the Decentralized Pursuit Algorithm in the 4×4 Zero-sum Game in Section 3.4.5

However, in case of an identical payoff game, the DPLA converges to one of the modes of the game matrix (local maxima) whereas the centralized pursuit learning algorithm converges to the maximum among the modes of the game matrix (global maxima). However, as explained earlier, the CPLA incurs lot of communication overhead and has exponential memory requirement. These factors make it impractical for applications consisting of moderate or large number of automata. The DPLA does

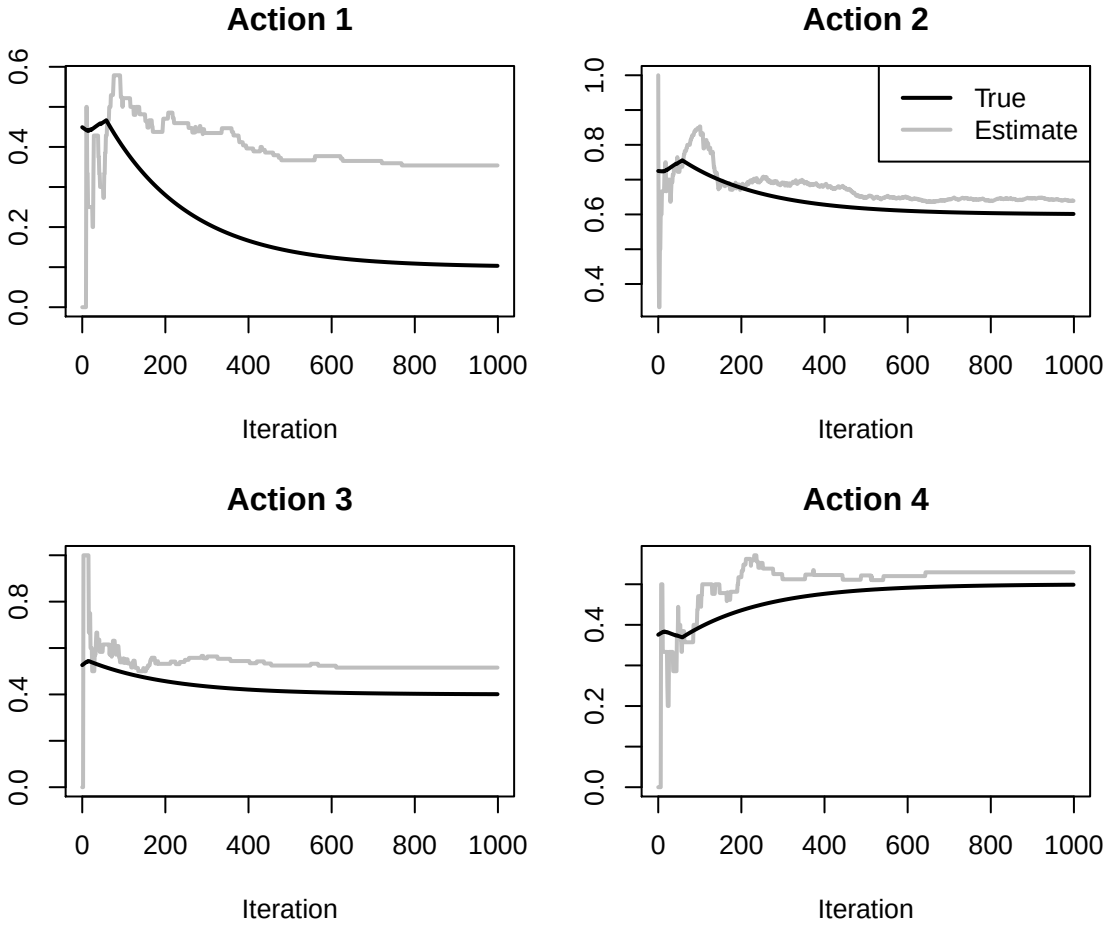


Figure 3.12. $D(t)$ (Black line) and $\widehat{D}(t)$ (Gray Line) of Player 2 for the Decentralized Pursuit Algorithm in the 4×4 Zero-sum Game in Section 3.4.5

not suffer from these drawbacks. This combination of properties makes the DPLA algorithm a better candidate for application in a game scenario.

3.5 Partially Decentralized Identical Payoff Games

As discussed earlier, the CPLA requires all the automata in the team to communicate their action choice to each other. The DPLA algorithm, on the other hand

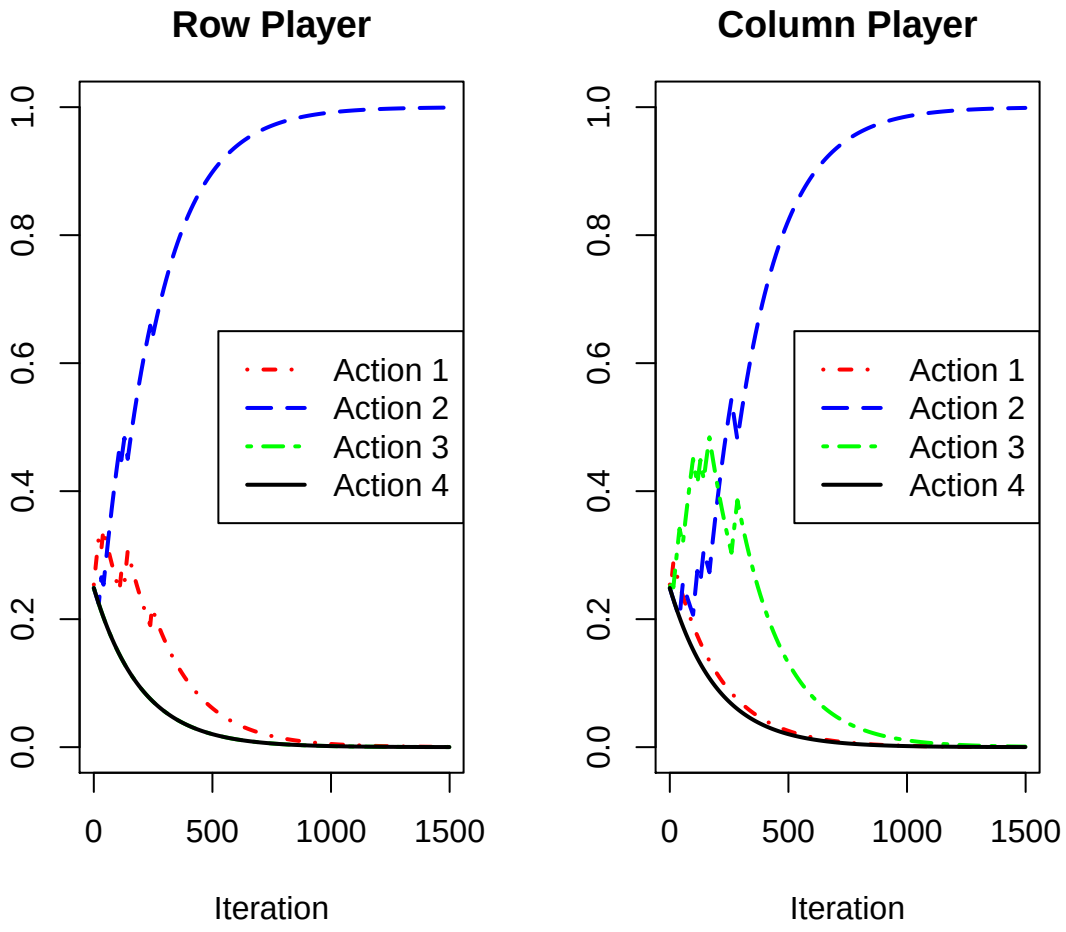


Figure 3.13. Comparison of Various Algorithms : Trajectory of Action Probabilities $\pi_{pi_p}(t)$

obviates this need and each automaton in the team operates in isolation. However, there is a middle ground where different configurations can be imagined so that only a subset of automata communicate with each other. We call such class of configurations Partially Decentralized games of LAs. In the next chapter, we will formally describe Partially Decentralized games and use them to control multi-agent Markov chains.

4 PARTIALLY DECENTRALIZED GAMES OF LA

As described in the previous chapter, the CPLA requires all the automata in the team to communicate their action choice to each other. This results in a lot of communication overhead. Also, since the size of estimate hypermatrices grows exponentially with the number of automata in the team, the CPLA also requires a large amount of memory. Although the advantage of centralized algorithm is that it guarantees convergence to the global maxima in the game matrix, it comes at a heavy memory cost.

On the other hand, the DPLA [33] described in the previous chapter obviates the need for automata to communicate their action choice to each other. Also, the size of the estimate hypermatrices grows linearly with the number of automata in the system. Thus the decentralized version offers a great improvement over the centralized version of the algorithm. However, the decentralized algorithm converges to one of the Nash equilibria of the game matrix(local maxima), instead to the global maxima (unlike centralized algorithm).

However, it might be useful if the automata in the team decide to communicate their action choices with some other chosen automaton (or automata) in the system. Then the team may converge to an action tuple which has higher payoff than the one offered in the case when there is no communication (as in the case of DPLA). In certain scenarios, it may even be possible for the team to converge to the global maxima and still communicate in a partial manner. To model this type of grouping among LAs, we propose a novel paradigm called Partially Decentralized Games of Learning Automata (PDGLA) [40].

4.1 Partially Decentralized Games

Under PDGLA, a group of learning automata are subdivided into various subgroups. The automata (or automaton) existing in each subgroup communicate with all the other automata in the same group and maintain the estimate matrices necessary to implement a Pursuit algorithm mechanism. Thus PDGLA [40] results in locally centralized groups. However, the entire automata team is not centralized. The choice of communication partner(s) could be made based on various constraints and criteria or combinations thereof. One possible constraint is memory available with an agent. If an agent accepts action choice input from n other automata, then it has to maintain estimate matrices of dimension $n+1$ and size $r_1 \times r_2 \times \ldots \times r_{n+1}$ where r_i is the size of action set associated with automata i. So it is clear that the maximum number of automata a single automaton (agent) can communicate with is determined by the available memory. The other possible criteria for selecting communication partner is communication cost. In the systems where communication has a lot of cost (e.g. in sensor systems where communication drains the power of the agent and reduces its lifetime) and where bandwidth is at a premium, it may be sensible to communicate with minimum number of other agents and try to get a better payoff. By taking all these factors into account, the local groups of automata can be formed.

Figure (4.1) depicts the schematic of a system in which learning automata participate in partially decentralized games. The system consists of *ten* leaning automata who are are engaged in PDGLA. The automata team is subdivided into three subgroups. The automata within each subgroup learn using a centralized learning mechanism (like CPLA). However, as the figure indicates, there is no communication or coordination between the automata (or automaton) across different subgroups. Each subgroups exists as an island and is oblivious to the activities in other subgroups. The centralization within each subgroup can be done by using any desired mechanism. The automata within a subgroup can interact with a central controller which maintains the estimate matrices necessary to implement CPLA. On the other hand, each automaton in the subgroup may communicate its action choice to every other

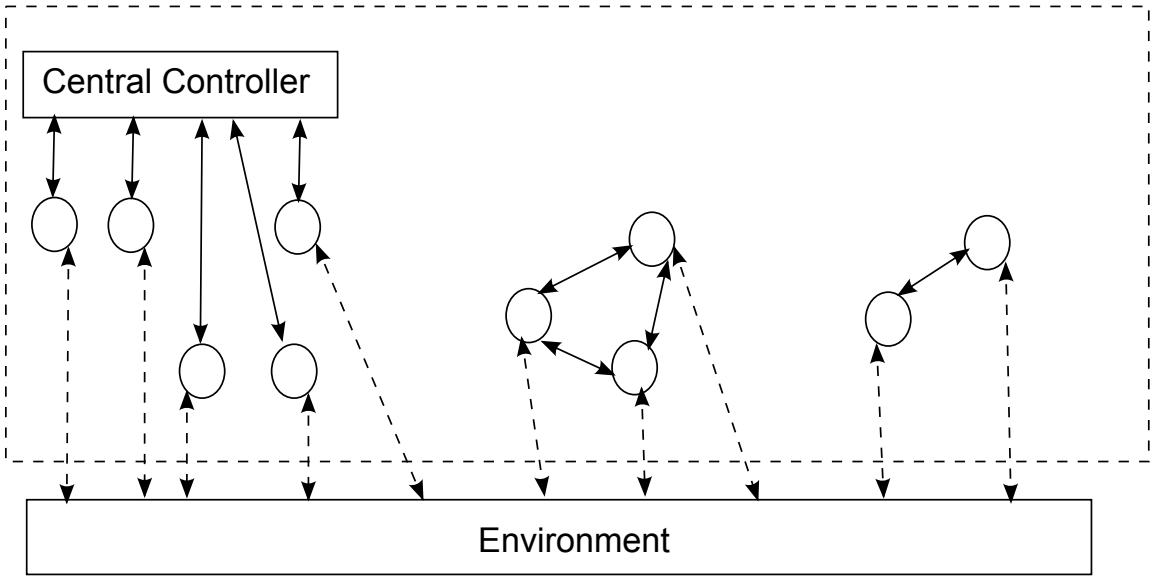


Figure 4.1. Schematic for Partially Decentralized Games of Learning Automata

automata in the group. This way, each automaton maintains the estimate matrices needed to run the CPLA and this obviates the need of a central controller.

We will demonstrate with an example, that partial communication (i.e., a subset of automata communicating with each other) may remove some Nash equilibria and may even reduce the resultant game matrix to one which has a unique Nash equilibrium. Consider a system of three automata A_0, A_1, A_2 each having two actions i_0, i_1. Let the game matrix be:

$\mathbb{D}[i_0, i_0, i_0] = 0.7$, $\mathbb{D}[i_0, i_0, i_1] = 0.6$, $\mathbb{D}[i_0, i_1, i_0] = 0.6$, $\mathbb{D}[i_1, i_0, i_0] = 0.6$, $\mathbb{D}[i_0, i_1, i_1] = 0.8$, $\mathbb{D}[i_1, i_0, i_1] = 0.6$, $\mathbb{D}[i_1, i_1, i_0] = 0.6$, $\mathbb{D}[i_1, i_1, i_1] = 0.9$.

If the DPLA algorithm was used, the game matrix $\mathbb{D}_{DPLA} = \mathbb{D}$ remains unchanged and it manifests two Nash equilibrium points. They are $\mathbb{D}[i_0, i_0, i_0]$ and $\mathbb{D}[i_1, i_1, i_1]$, with $\mathbb{D}[i_1, i_1, i_1]$ being the global maximum.

Now consider A_1 and A_2 communicating to form a resultant automaton B_0. Then, the game matrix between A_0 and B_0 is:

$$\mathbb{D}_{PDGLA} = \mathbb{D}_{2\times 4} = \begin{bmatrix} 0.7 & 0.6 & 0.6 & 0.8 \\ 0.6 & 0.6 & 0.6 & 0.9 \end{bmatrix}$$

This game matrix has a unique mode with value 0.9. So, it is possible to convert a multimodal decentralized game matrix to a unimodal game matrix with partial communication.

4.1.1 Description of PDGLA

Each automaton A^p, that participating in the PDGLA, maintains three matrices $\hat{\mathbb{D}}^p, R^p$ and Z^p. If automata A^p gets action choice communication from k other automata $\{A^{q_1}, A^{q_2}, \ldots, A^{q_k}\}$, then the estimate matrices of A^p will be of size $a_{q_1} \times a_{q_2} \times \ldots \times a^{q_k}$. The partially decentralized pursuit learning game algorithm works in two phases: (1) Action selection and communication phase and (2) Update phase.

Action selection and communication phase: Each automaton A^p chooses action $\alpha^p_{i_p}$ at instant k by sampling the its action probability distribution $P^p(k)$. Then each automaton A^p then communicates this action choice to other automaton(automata) it is connected to. Each automaton in the team uses its own selected action in conjunction with the actions received from other automaton(automata). This collective action information is used by each automaton to update its estimate hypermatrices.

Update phase: Once the communication phase is over, all the automata in the team update their estimate hypermatrices. Each automata A^p gets action choice communication from k other automata $\{A^{q_1}, A^{q_2}, \ldots, A^{q_k}\}$. So each automaton forms an action tuple $\{\alpha^p_{i_p}, \alpha^{q_1}_{i_{q_1}}, \alpha^{q_2}_{i_{q_2}}, \ldots, \alpha^{q_k}_{i_{q_k}}\}$. Then the estimate matrices are updated as follows:

$$R^p_{i_p,i_{q_1},i_{q_2},\ldots,i_{q_k}}(k) = R^p_{i_p,i_{q_1},i_{q_2},\ldots,i_{q_k}}(k-1) + \beta(k)$$

$$R^p_q(k) = R^p_q(k-1), \forall q \neq \{i_p, i_{q_1}, i_{q_2}, \ldots, i_{q_k}\}$$

$$Z^p_{i_p,i_{q_1},i_{q_2},\ldots,i_{q_k}}(k) = Z^p_{i_p,i_{q_1},i_{q_2},\ldots,i_{q_k}}(k-1) + 1$$

$$Z^p_q(k) = Z^p_q(k-1), \forall q \neq \{i_p, i_{q_1}, i_{q_2}, \ldots, i_{q_k}\}$$

$$\hat{\mathbb{D}}^p(k) = \frac{R^p(k)}{Z^p(k)}$$

Each automaton then updates its action probability vector as follows:

$$P^p(k+1) = P^p(k) + \lambda(e_{M_p} - P^p(k))$$

where $0 < \lambda < 1$ is the learning parameter and index M_p is determined by

$$M_p = \max_{j_p,j_{q_1},j_{q_2},\ldots,j_{q_k}} \hat{\mathbb{D}}^p_{j_p,j_{q_1},j_{q_2},\ldots,j_{q_k}}(k)$$

If automaton A^p gets action choice communication from k other automata

$$A^{q_1}, A^{q_2}, \ldots A^{q_k}$$

then it will need $O(a_{q_1} \times a_{q_2} \times \ldots \times a_{q_k})$ memory to store estimate hypermatrices.

It is easy to see from the above discussion that the CPLA and DPLA are the degenerate cases of a more general concept of PDGLA. Thus theoretical study of such systems holds great potential and will bridge the gap between the two extremes. We will describe how the PDGLA framework can be used to adaptively control a Multi-Agent Markov Decision Process (MAMDP).

4.2 Multi Agent Markov Decision Process

A large number of distributed real-world systems can be modeled as multi-agent systems [41, 42]. A Multi-agent Markov Decision Process (MAMDP) framework is a flexible formalism that can be used to model such systems for control and decision-making problems. Multi-robot systems, unmanned air vehicle (UAV) systems, sensor networks, computer networks, smart power grids, intelligent vehicle highway (IVH) systems, Massively Multi-player Online Role-Playing Games (MMORPGs), defense simulations and economic systems are some examples of systems that can be modeled as MAMDPs [43].

Many of these real-world multi-agent systems possess significant sources of uncertainty, making it impossible to pre-compute the optimal decision and control rules in an off-line manner. The complex, non-linear nature of these systems, coupled with the inherent uncertainty requires a mathematical framework that is powerful yet simple in nature. It also requires practically feasible algorithms to act on this model to compute the decision and control rules in an on-line manner. MAMDP and Multi-Agent Reinforcement Learning (MARL) provide the framework and the algorithms, respectively, for such distributed control problems.

An MAMDP framework expresses these distributed, multi-agent systems in a convenient mathematical formulation for decision and control problems. For tractabil-

ity, various approximations can be used while expressing multi-agent systems in an MAMDP framework. An MAMDP framework can also deal with the inherent non-linear and stochastic nature of the decision and control rules used to optimize long term performance criteria in a distributed manner. MARL integrates seamlessly with MAMDP and uses the rewards obtained during the execution of the system to update the decision variables in an on-line manner. In this paper, we propose novel MARL algorithms for the control of MAMDP systems.

A finite state Markov Decision Process (MDP) consists of multiple states. The MDP performs state transitions that generate rewards which depend on actions taken by the agents acting in different states of the chain. The control of finite, *multi-agent* MDP for which transition and reward probabilities are known can be stated as follows: Let $\Phi = \{\varphi_1, \varphi_2, \ldots, \varphi_N\}$ be the state space of a finite MAMDP with N states. For notational simplicity, let M be the number of agents present in the MAMDP. Let $\|R\|$ denote cardinality of the set R. We use R_k^i to denote action set of the agent k residing in state i of the MAMDP and $\bigotimes$ to denote Cartesian product. An action tuple is formed by one action from each agent. Let $\alpha^i = \{R_1^i \bigotimes R_2^i \bigotimes \ldots \bigotimes R_M^i\}$ be the finite set of *action tuples* available in state φ_i. Transition probabilities $t_j^i(\mathscr{A})$ and corresponding rewards $r_j^i(\mathscr{A})$ depend on the source state φ_i, sink state φ_j and the action tuple $\mathscr{A} \in \alpha^i$. Thus we define transition probability function for state Φ_i as $t^i : \Phi \times R_1^i \times R_2^i \times \ldots \times R_k^i \rightarrow PD(\Phi)$ where PD operator represents a set of discrete probability distributions. The corresponding associated reward function $r^i : \Phi \times R_1^i \times R_2^i \times \ldots \times R_k^i \rightarrow \mathfrak{R}$ where $\mathfrak{R}$ is the set of real numbers that lie in the interval $[0, 1]$. The goal is to choose a set of actions, or policy, $\psi \in \alpha^1 \bigotimes \alpha^2 \bigotimes \ldots \bigotimes \alpha^N$ that maximizes the long term expected reward

$$J(\psi) \equiv \lim_{n\rightarrow\infty} \frac{1}{n} E[\sum_{t=0}^{n-1} r^{x(t)}(x(t), x(t+1), \psi)]$$

where $x(t)$ and $x(t+1)$ represent the states the MAMDP visits as time t and $t+1$ respectively.

4.3 Previous Work

Dynamic programming methods can be used to determine optimal policy for a MDP [44]. However, the dynamic programming approaches become computationally intensive when the number of states and transitions increase. Also, dynamic programming approaches require the knowledge of transition probabilities and reward values associated with different actions. Depending on the constraints of the problem, this information may be unknown or may change during system operation.

Wheeler and Narendra proposed a reinforcement learning solution for the control of a single-agent MDP problem [10]. It is shown that by associating one learning automaton with each state of the MDP and treating the problem as an identical-payoff game of learning automata, the automata team will converge to the optimal policy tuple. In case of a single-agent MDP, the resultant game matrix is shown to have a unique equilibrium point and thus the automata team converges to the optimal policy corresponding to the unique equilibrium. The Wheeler-Narendra solution uses the framework of identical-payoff game of learning automata to solve the control problem. In the configuration proposed by the authors, one learning automaton is associated with each state of the MDP. The learning automaton acts as the decision maker and makes action selection in each state of the system. Each decision maker uses simple L_{R-I} learning scheme [2] to update its action probabilities. It is proven that in case of identical-payoff games of learning automata, if the game matrix has a unique equilibrium point, and the learning agents do learning in sufficiently small steps, then the automata team will converge to the unique equilibrium point. Learning automata associated with the states of MDP participate in an asynchronous, identical-payoff game. The resultant game matrix has a unique equilibrium and thus the automata team converges to optimal policy represented by the equilibrium.

An important feature of this control scheme is that the automata acting in a state are not informed of the one-step reward resulting from their actions. The algorithm assumes presence of a central controller which keeps track of the cumulative reward generated by the chain so far and global time which counts number of transitions

performed by the chain so far. When the MDP transitions to a state, the automaton acting within that state receives information about the cumulative reward generated by the chain so far and the current global time from the central controller. From these, the automaton calculates the average reward value which is used as the payoff β for the learning process. As described earlier, L_{R-I} algorithm is used to update the action probabilities of the automaton. Thus action probabilities are updated as follows:

$$p(k+1) = p(k) + \lambda\beta(e_\alpha - p_i(k)), i = (l, 2, \ldots, N)$$

where $0 < \lambda < 1$ is a parameter. α is the action selected by this automaton during previous time when MDP was in the current state and e_α is a unit vector of appropriate dimension with α-th component unity. After updating the action probabilities, the automaton then samples its action probability vector to select the next action which is used by the MDP to transition to an appropriate next state and the process repeats.

We use a similar technique to calculate the environment response for the proposed PDGLA approach described later. The environment response for the proposed PDGLA algorithm(s) is calculated as the average reward generated so far by the MAMDP. The central controller keeps track of cumulative reward and number of transitions and the automata use this information to calculate the reward values.

4.4 An Intuitive Solution

MAMDP problems can be thought of as an extension of single-agent problem with each state consisting of multiple agents instead of a single agent. An intuitive way to extend the single-agent solution proposed by Wheeler-Narendra [10] to the multi-agent problem will be to assign one learning automaton for each agent in the chain. However, the resultant game matrix in such cases may have multiple equilibrium points. Since the game matrix does not have a unique equilibrium point, such decen-

tralized game of learning automata may not converge to the global maximum among the possible equilibrium points. The convergence in such decentralized case will be to one of the equilibrium points in the game matrix [45]. While each such equilibrium represents local maximum in the game matrix, the corresponding control policy may be a *locally optimal policy*. Among all possible equilibrium points of the game matrix, the equilibrium point with maximum value represents global maximum. The policy corresponding to this equilibrium point is the *globally optimal control policy*.

The problem of non-optimal policy convergence can be addressed by allowing the automata in the system to communicate their action choices with each other. Such action communication constitutes *centralized* game of learning automata. Depending on the number of automata involved in the communication, the game can be completely or partially centralized. Centralization can also be achieved by combining actions of different automata into a *superautomaton*. The superautomaton then acts as representative of the group of automata and participates in the game on behalf of the group. However, such superautomaton construction reduces the degree of autonomy in the system. Since, superautomaton makes actions selection for all the automata it represents, the individual automaton loses its own autonomy and surrenders it to the superautomaton. Thus depending on how centralization is performed (with or without the use of superautomaton approach), the system will possess different degree autonomy. Thus, we motivate the discussion of our algorithms based on two factors: degree of communication and autonomy. Depending on the availability of the resources and the domain constraints (dictating the memory capacity and autonomy of an agent), a suitable algorithm can be chosen from a gamut of possible algorithms.

In the following algorithms, we use the strategy described used by Wheeler-Narendra [10] to calculate the environment response to the learning automata operating in the states of the MAMDP. The learning automata are not aware of the one-step reward values $t_j^i(\mathscr{A})$ resulting from their selected action tuple $\mathscr{A}$. The learning agent(s) representing a state receive information about the effect of their se-

lected actions from a bookkeeper only when the process returns to that state. When the Markov process returns to state φ_i; the automata representing that state receive two pieces of data from a central controller: 1) the cumulative reward generated by the process up to time n and 2) the current global time $\mathcal{N}$. From these, the learning agent(s) in the state computes reward generated since the last time this state was visited and the corresponding elapsed global time Δn. These increments are added to the current cumulative reward $\rho_i(n_i)$ and cumulative time $\theta_i(n_i)$, resulting in new totals $\rho_i(\Delta n+1)$ and $\theta_i(\Delta n+1)$. The environment response β^i is given as an input to all the learning agent(s) operating in the state φ_i and it is calculated as $\beta^i = \frac{\rho_i(\Delta n+1)}{\theta_i(\Delta n+1)}$. Thus, β^i represents the average turnaround reward generated by the chain and it is used in the learning process. The bookkeeper merely stores the information about the cumulative reward and does not make action selection. The action selection, and hence the control function, is performed by the automata that reside in the individual states of the chain.

4.5 Superautomaton Based Algorithms

We propose two partially decentralized *Superautomaton*-based algorithm for the control of MAMDPs. Each state of the MAMDP is represented by a superautomaton. This superautomaton is formed by combining actions of the subautomata present in individual states. The action space of each superautomaton consists of Cartesian product of the action space of individual subautomata present in the state of MAMDP. Assume that state φ_i of the MAMDP has M number of agents. The action set of j^{th} agent is denoted by R^i_j. Thus, the action set of the superautomaton $\mathcal{A}^i$ representing state φ_i is $\{R^i_1 \bigotimes R^i_2 \bigotimes \ldots \bigotimes R^i_M\}$.

Figure (4.5) describes the Superautomaton configuration every state i of a multi-agent MAMDP. As the figure indicates, the multiple learning automata that reside in the state are substituted by one Superautomaton. Dashed lines indicate that these subautomata surrender their autonomy to the superautomaton who participates in

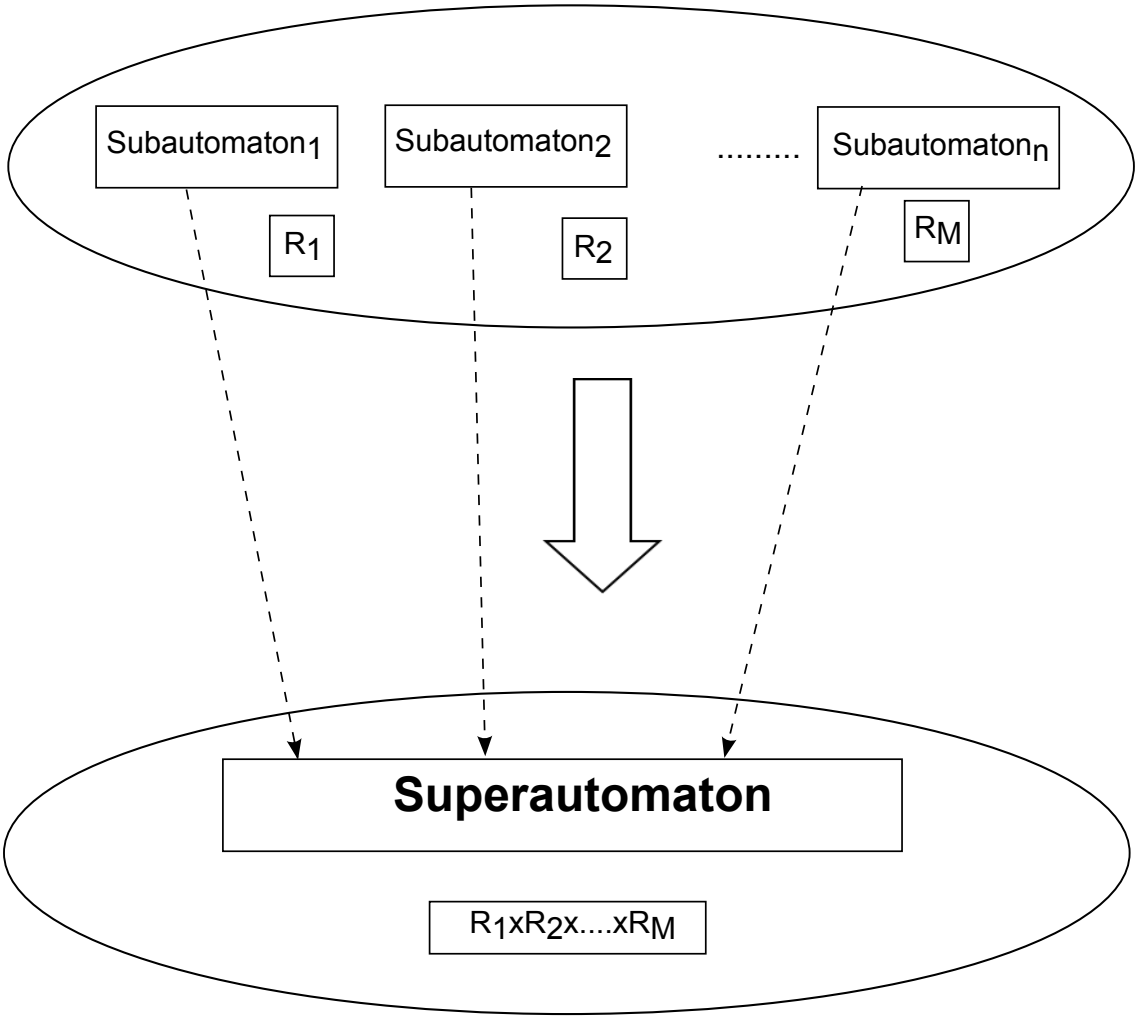


Figure 4.2. Superautomaton Configuration for Any State i

the learning process on behalf of all the subautomata. As denoted in the figure, the action space of the superautomaton is Cartesian product of the action spaces of each individual subautomaton. Thus Superautomaton essentially replaces all the subautomata that reside in a state of the MAMDP.

4.5.1 L_{R-I}-Based Superautomaton Algorithm

Under L_{R-I}-based superautomaton algorithm, each superautomaton uses L_{R-I} algorithm to update its action probabilities. The algorithm proceeds as follows:

1. When the MAMDP is in state φ_i, the superautomaton $\mathcal{A}^i$ representing that state selects an action tuple by sampling its action probability vector. Let this action tuple be $\{r_1^i, r_2^i, \ldots, r_M^i\} \in \{R_1^i \bigotimes R_2^i \bigotimes \ldots \bigotimes R_M^i\}$

2. Based on the selected action tuple, MAMDP makes probabilistic transition to a new state.

3. When the MAMDP returns to the state φ_i, the β^i is calculated as described earlier. The superautomaton $\mathcal{A}^i$ uses this environment response to update its action probabilities using the L_{R-I} algorithm. Thus:

$$p^i(n+1) = p^i(n) + \lambda\beta^i(e_{\alpha(n)} - p(n))$$

where $\alpha(n) = \{r_1^i, r_2^i, \ldots, r_M^i\}$ and $e_{\alpha(n)}$ is a unit vector of appropriate dimension with $\alpha(n)$-th component set to unity.

Since the action space of the superautomaton $\mathcal{A}^i$ is a Cartesian product of the action space of individual agents acting in the state φ_i, this configuration is similar to the one described in [10]. Thus using the analysis described in [10], it can be proven that the resultant game matrix formed by using superautomaton approach has a unique equilibrium. Hence, the team of superautomata will converge to the optimal policy represented by this unique equilibrium point.

4.5.2 Pursuit-Based *Superautomaton* Algorithm

We propose another version of the *Superautomaton* algorithm that makes use of the Pursuit algorithm to update action probabilities. As described earlier, each state of the MAMDP is represented by a superautomaton. Each superautomaton maintains an estimate matrix $\hat{d}^i$ of dimension $|R_1^i| \times |R_2^i| \times \ldots \times |R_M^i|$. All the values of $\hat{d}^i$ matrix are initialized to zero. The algorithm proceeds as follows:

1. When the MAMDP is in state φ_i, the superautomaton $\mathcal{A}^i$ representing that state selects an action tuple by sampling its action probability vector. Let this action tuple be $\{r_1^i, r_2^i, \ldots, r_M^i\} \in \{R_1^i \bigotimes R_2^i \bigotimes \ldots \bigotimes R_M^i\}$

2. Based on the selected action tuple, the MAMDP makes probabilistic transition to a new state.

3. When the MAMDP returns to the state φ_i, the β^i is calculated as described earlier. The superautomaton $\mathcal{A}^i$ uses this environment response to update its action probabilities using the Pursuit algorithm. First, the algorithm uses the β^i value to set the values of the $\hat{d}^i$ matrix. The superautomaton $\mathcal{A}^i$ sets $\hat{d}_{r_1^i, r_2^i, \ldots, r_M^i} = \beta^i$ and all other values are left unchanged. Then algorithm selects maximum element (and the corresponding action tuple) in the $\hat{d}^i$ matrix and increases action probability of that particular element (i.e. action tuple) by a small value. Thus:

$$p(n+1) = p(n) + \lambda(e_M - p(n))$$

where $0 < \lambda < 1$ is the learning parameter and e_M represents a unit vector of appropriate dimensions with M^{th} component set to unity and all other components set to zero. The index M is determined by $\hat{d}^i_M(n) = \max_{a_1^i, a_2^i, \ldots, a_M^i} \hat{d}^i_{a_1^i, a_2^i, \ldots, a_M^i}(n)$. In other words, index M is the action tuple $a_1^i, a_2^i, \ldots, a_M^i$ that represents the maximum value in the matrix $\hat{d}^i$ at the n-the iteration of the algorithm.

Since the action space of the superautomaton $\mathcal{A}^i$ is a Cartesian product of the action space of individual agents acting in the state φ_i, this configuration is also similar to the one described in [10]. Thus using the analysis described in [10], it can be proven that if the MAMDP is ergodic then ordering in each of the $\hat{d}^i$ matrices becomes same as the ordering in the identical-payoff game matrix d^i. Here, $\hat{d}^i$ can be thought of as the estimate matrix that tries to estimate entries in the actual game matrix d^i. Since β^i represents average reward obtained by the selected action tuple, the values of β^i approach the values in d^i asymptotically. Thus by using the convergence argument from [4], it can be proven that each Superautomaton will converge to the globally optimal policy tuple.

4.5.3 Drawbacks of Superautomaton Based Algorithms

The drawback of both the L_{R-I}-based and Pursuit-based superautomaton approach is that it provides very little autonomy and fault tolerance. Various subautomata present in the state surrender their autonomy (regarding action selection) to the superautomaton which represents them in the learning framework. This reduces the level of autonomy in the system. Depending on the problem at hand, it may not be possible for individual agents to surrender their autonomy in this manner. Also, since each state is controlled by only one superautomata, it represents a single point of failure and thus provides very little in terms of fault-tolerance. Failure of one superautomaton will hinder the working of the entire system. Ideally, one would like to provide individual agents complete autonomy in making action choices and also make the system more robust by providing a mechanism for fault-tolerance. At the same time, one would like the learning agents to learn the optimal policy.

To this end, we propose two more novel algorithms which mantain the autonomy of individual agents. First algorithm is called Distributed Pursuit algorithm. The Pursuit-based Superautomaton algorithm employs a Superautomaton configuration. The Distributed Pursuit algorithm, on the other hand, preserves the autonomy of individual learning agents by allowing them to select the action independently while keeping the learning process centralized. The second algorithm uses a Master-Slave configuration to mimic the behavior of superautomaton in every state while still maintaining the autonomy of the individual agents in the system. Since the Distributed Pursuit algorithm and Master-Slave algorithm maintain the autonomy of individual subautomata operating inside the states, they provide greater fault-tolerance.

4.6 Distributed Pursuit Algorithm

We propose another version of the Pursuit based algorithm that makes use of the Pursuit algorithm to update action probabilities of the individual automata. Each agent in the state is represented by a LA and each learning automaton uses Pursuit

learning algorithm to update its action probabilities. For each state, the algorithm also maintains an estimate matrix $\hat{d}^i$ of dimension $\mid R_1^i \mid \times \mid R_2^i \mid \times \ldots \times \mid R_M^i \mid$. All the values of $\hat{d}^i$ matrix are initialized to zero. The algorithm proceeds as follows:

1. When the MAMDP is in state φ_i, *each subautomaton* present in the state selects an action by sampling its action probability vector. The individual actions selcted by the individual automaton are combined to form an action tuple. Let this action tuple be $\{r_1^i, r_2^i, \ldots, r_M^i\} \in \{R_1^i \bigotimes R_2^i \bigotimes \ldots \bigotimes R_M^i\}$

2. Based on the selected action tuple, MAMDP makes probabilistic transition to a new state.

3. When the MAMDP returns to the state φ_i, the β^i is calculated as described earlier. The algorithm uses this environment response to update the action probabilities of the individual LA using the Pursuit algorithm. First, the algorithm uses the β^i value to set the values of the $\hat{d}^i$ matrix. The algorithm sets $\hat{d}_{r_1^i, r_2^i, \ldots, r_M^i} = \beta^i$ and all other values are left unchanged. Then algorithm selects maximum element (and the corresponding action tuple) in the $\hat{d}^i$ matrix. Let this action tuple be $m_1^i, m_2^i, \ldots, m_M^i$. The action probability of each individual LA in state φ_i is changed as follows:

$$p^i(n+1) = p^i(n) + \lambda(e_{m_j^i} - p(n)), 1 \leq j \leq M$$

where $0 < \lambda < 1$ is the learning parameter and $e_{m_j^i}$ represents a unit vector of appropriate dimensions with m_j^i-th component set to unity and all other components set to zero.

Using the analysis described in [10], it can be proven that the if the MAMDP is ergodic then the ordering in each of the $\hat{d}^i$ estimate matrices becomes same as the ordering in the identical-payoff game matrix d^i. Since β^i represents average turnaround reward obtained by the selected action tuple, the values of β^i approach the values

in d^i asymptotically. Thus by using the convergence argument from [20], it can be proven that each Superautomaton will converge to the optimum policy tuple.

4.7 Master-Slave Algorithm

We propose and Master-Slave configuration for the control of the MAMDPs. In Master-Slave formulation, each agent present in the state of the MAMDP is represented by a learning automaton. One of these automata acts as a Slave automaton while one or more of the remaining automata act as Master automata. The Master and Slave automata together simulate the behavior of the superautomaton. If $M-1$ automata act as Master automata then the Master-Slave configurations will simulate the behavior of superautomaton whose action space is the Cartesian product of the action space of $M-1$ Masters as well as *one* Slave automaton.

Figure (4.7) describes the Master-Slave configuration for every state i of a MAMDP. As described in the figure, all except one learning agent become Master automata and one learning automaton acts as Slave automata. Solid lines indicate that original subautomata merely act as Master or Slave automata but still keep their autonomy intact (unlike the Superautomaton configuration). Also, since Slave automaton simulates the behavior of the hypothetical superautomaton, its action space is the Cartesian product of the action spaces of each individual subautomaton.

Master and Slave automata select actions with complete autonomy. Master automata communicate their selected actions to the Slave automaton which uses this information to simulate the behavior of the superautomaton. The Slave automata then sends its action probability vector to the Master automata. Master automata use the action probability vector of the Slave automata to set the values of their action probability vector. Although the Master automata update their own action probability based on the action probability vector communicated by the Slave automaton, Master and Slave automata are still autonomous with regard to action selection. Both Master and Slave select actions completely independently of each other.

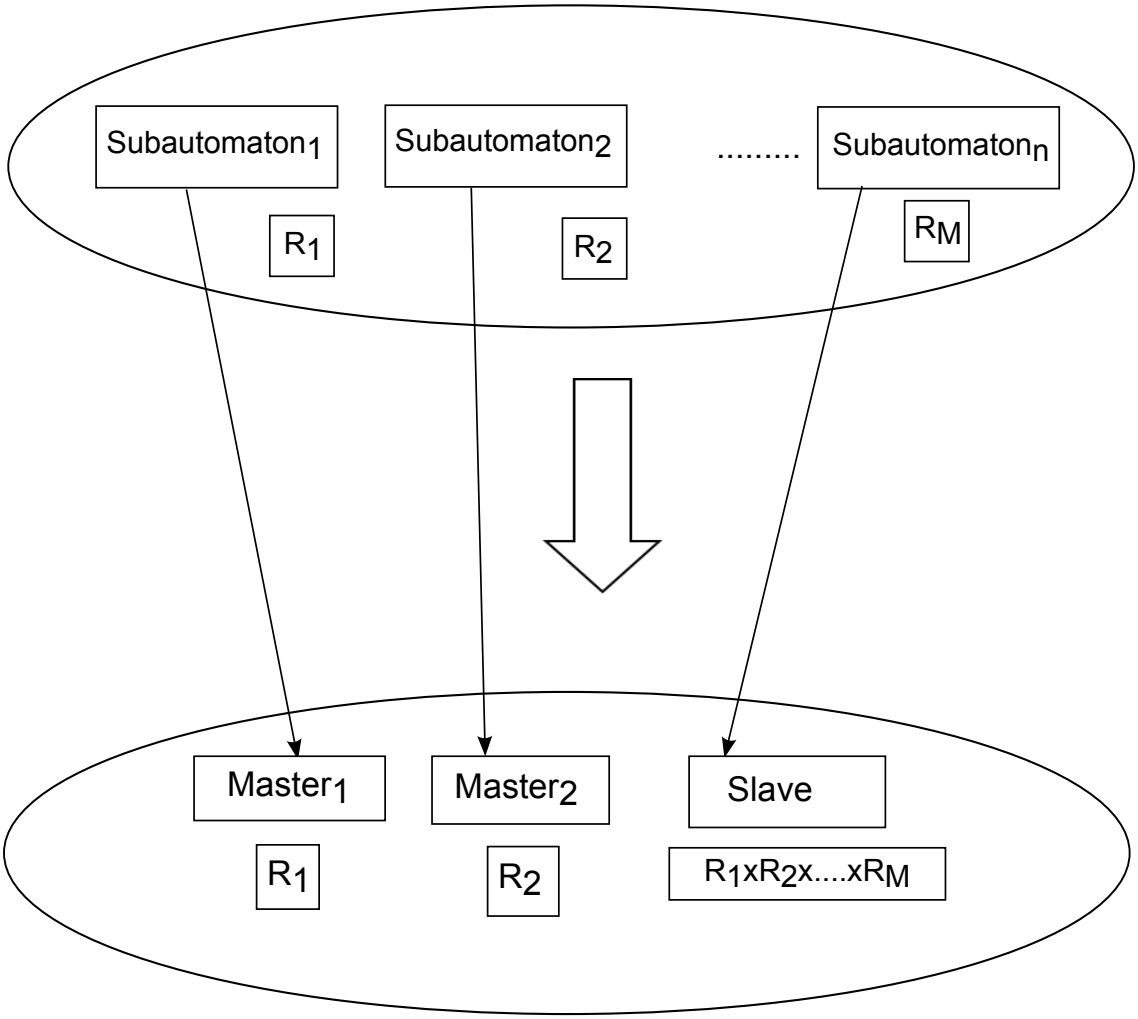


Figure 4.3. Master-Slave Configuration for Any State i

4.7.1 Master-Slave Equations

Assume that Master-Slave configuration in state i of the MAMDP consists of $M-1$ Master automata and the M^{th} automaton is the Slave automaton $(A^1, A^2, \ldots, A^M)$. The corresponding superautomaton is denoted by $\mathcal{A}^i$. Let R_j denote the set of actions for automaton j. We also use $\|R_j\|$ to denote the cardinality of the set R_j. Suppose i_k is the action chosen by k^{th} automaton. Let action probabilities of $M-1$ Master automata be denoted by $\{p_1^1, p_2^1, \ldots, p_{\|R_1\|}^1\}$, $\{p_1^2, p_2^2, \ldots, p_{\|R_2\|}^2\}$, ..., $\{p_1^{M-1}, p_2^{M-1}, \ldots, p_{\|R_{M-1}\|}^{M-1}\}$. M^{th} automaton acts as the Slave automaton. The actions

probabilities of M^{th} automaton are denoted by $\{p^M_{R_1 \otimes R_2 \otimes R_M}\}$. Let $\{q_{R_1 \otimes R_2 \otimes R_M}\}$ denote action probabilities of the (hypothetical) *superautomaton* $\mathcal{A}^i$. Master-Slave team will together try to simulate the behavior of this Superautomaton. In the following discussion, let $T \parallel i$ indicate i^{th} component of tuple T. $\{T \biguplus t\}$ operation adds element t to the tuple T. The idea of Master-Slave configuration is to keep the product of the action probabilities of Master-Slave automata same as the corresponding action tuple of the superautomaton. Thus, the action probabilities of Master-Slave configuration are always adjusted such that following invariant is satisfied:

$$p^1_{i_1} \times p^2_{i_2} \times \ldots \times p^{M-1}_{i_{M-1}} \times p^M_{i_1 i_2 \ldots i_{M-1} i_M} = q_{i_1 i_2 \ldots i_{M-1} i_M} \tag{4.1}$$

The invariant in equation (4.1) states that for any action tuple $\{i_1 i_2 \ldots i_{M-1} i_M\}$, the product of action probabilities of $M-1$ Master automata and the Slave automata is equal to the corresponding action probability of the superautomaton. The basic idea behind the Master-Slave configuration is to ensure that Master-Slave together emulate the behavior of the hypothetical superautomaton exactly. Thus it is necessary that product of individual action probabilities of Master-Slave configuration (p-values) equals the corresponding action probability of the superautomaton (q-values). Equation (4.1) states this criteria in a mathematical form.

Rewriting the above invariant, we get the following constraint:

$$q_{R_1 \otimes R_2 \otimes R_M} = \prod_{i=1}^{M-1} \{p^i_{\{R_1 \otimes R_2 \otimes R_M\} \parallel i}\} \times p^M_{R_1 \otimes R_2 \otimes R_M} \tag{4.2}$$

Another invariant of the algorithm relates action probabilities within each learning automaton. Action probabilities of an individual learning automaton must always sum to 1. So, for all the Master automata ($j = 1$ to $M-1$), we have following constraint:

$$\sum_{i \in R_j} p^j_i = 1 \tag{4.3}$$

The action probabilities of the Slave automaton must also always sum to 1. Thus we get the following invariant:

$$\sum_{i \in R_M} p^M_{\{R_1 \otimes R_2 \otimes \ldots \otimes R_{M-1}\} \uplus i} = 1 \tag{4.4}$$

Similarly for the superautomaton, we have the following invariant:

$$\sum q_{R_1 \otimes R_2 \otimes R_M} = 1 \tag{4.5}$$

We can rewrite invariant (4.2) to get following relationship between superautomaton and the Master-Slave automata:

$$\sum_{j \in R_M} q_{\{R_1 \otimes R_2 \otimes \ldots \otimes R_{M-1}\} \uplus j} = \prod_{i=1}^{M-1} \{p^i_{\{R_1 \otimes R_2 \otimes R_M\} \| i}\} \times \sum_{j \in R_M} \{p^M_{\{R_1 \otimes R_2 \otimes \ldots \otimes R_{M-1}\} \uplus j}\} \tag{4.6}$$

But from equation (4.4), the summation term in the RHS of 4.6 must sum to 1. So we get:

$$\sum_{j \in R_M} \{p^M_{\{R_1 \otimes R_2 \otimes \ldots \otimes R_{M-1}\} \uplus j}\} = 1 \tag{4.7}$$

Thus equation (4.6) now becomes:

$$\sum_{j \in R_M} q_{\{R_1 \otimes R_2 \otimes \ldots \otimes R_{M-1}\} \uplus j} = \prod_{i=1}^{M-1} \{p^i_{\{R_1 \otimes R_2 \otimes R_M\} \| i}\} \tag{4.8}$$

Also, rearranging the terms in equation (4.2), we get:

$$p^M_{R_1 \otimes R_2 \otimes \ldots \otimes R_M} = \frac{q_{R_1 \otimes R_2 \otimes \ldots \otimes R_M}}{\prod_{i=1}^{M-1} p^i_{\{R_1 \otimes R_2 \otimes R_M\} \| i}} \tag{4.9}$$

But according to equation (4.8), the denominator term in the above equation (4.9) is

$$\prod_{i=1}^{M-1} \{p^i_{\{R_1 \otimes R_2 \otimes R_M\} \| i}\} = \sum_{j \in R_M} q_{\{R_1 \otimes R_2 \otimes \ldots \otimes R_{M-1}\} \uplus j} \tag{4.10}$$

Thus, the equation (4.9) now becomes,

$$p^M_{R_1 \otimes R_2 \otimes \ldots \otimes R_M} = \frac{q_{R_1 \otimes R_2 \otimes \ldots \otimes R_M}}{\sum_{j \in R_M} q_{\{R_1 \otimes R_2 \otimes \ldots \otimes R_{M-1}\} \uplus j}} \tag{4.11}$$

The above equation (4.11) establishes the relationship between the action probabilities of the Slave automaton (the M-th automaton) and a hypothetical superautomaton. Thus if Slave automaton could calculate the q-values (which is same as simulating the behavior of the hypothetical superautomaton), then it can in turn calculate its own action probabilities (the p^M-values). Since the q-values are based on the actions selected by all the automata in a particular state of the MAMDP, Slave automata needs information about action selected by all the other automata in the state. Towards this end, Master automata send their selected actions to the Slave automaton. The Slave automaton uses this information along with its own selected action to calculate superautomaton action probabilities (the q-values). The $beta^i$ value needed to compute q-values for $\mathcal{A}^i$ is calculated as described earlier. The Slave automata forms an action tuple based on the actions communicated by the $M-1$ Master automata ($r^i \in R^i, i = 1$ to $M-1$) and its own selected action $r^M \in R^M$. Then Slave automaton calculates the q-values using the L_{R-I} learning algorithm as follows:

$$q(n+1) = q(n) + \lambda\beta(n)(e_{\alpha(n)} - q(n))$$

where $\alpha(n) = \{r_1^i, r_2^i, \ldots, r_M^i\}$ and $e_{\alpha(n)}$ is a unit vector of appropriate dimension with $\alpha(n)$-th component set to unity.

Thus after calculating the q-values, the Slave automaton sends the q-vector back to all the Master automata. Now it is the turn of the Master automata to use the q-vector information to calculate and set their own action probabilities.

From equation (4.12), we get:

$$p_j^i = \sum_{R_1 \otimes R_2 \otimes \ldots \otimes R_{i-1} \otimes R_{i+1} \otimes \ldots \otimes R_M} q_{R_1 \otimes R_2 \otimes \ldots \otimes R_{i-1} \otimes i_j \otimes R_{i+1} \otimes \ldots \otimes R_M} \qquad (4.12)$$

where p_j^i is the j-th action probability of the i-th Master automata. Thus once Master automata receive the q-values, they can calculate their own action probability vectors using equation (4.12). All the Master automata then send their calculated p-values back to the Slave automaton. The Slave automaton then calculates its own

action probability vector using equation (4.9). The denominator in this equation consists of p-values of the Mater automata. So once Slave automaton receives p-values of the Master automata, it can calculate its own action probabilities.

Since these action probability values are calculated by taking all the invariants into account, it ensures that the sum of action probabilities of individual automaton will always sum to 1.

To summarize, the Master-Slave configuration works as follows:

1. When the MAMDP is in a particular state φ_i, the Master and Slave automata residing in that state select an action individually and autonomously. The chain then transitions to the next state based on the action tuple formed by the actions selected by Master and Slave automata.

2. When the chain returns to the state φ_i, the Master automata in this state send their selected actions (from step 1) to the Slave automaton that resides in the current state of the MAMDP.

3. Slave automata calculates the q-values of the hypothetical superautomata using the L_{R-I} update equation.

4. Slave automaton sends q-value information back to all the Master automata that reside in the state φ_i. The Master automata then calculate their action probabilities using the values in the q-vector.

5. All the Master automata send their action probability values back to the Slave automaton. Slave automaton then calculates its own action probability vector using the q-vector and the action probability vectors of the Master automata.

6. Goto step 1 (i.e. the Master and Slave automata then each select an action by sampling their updated probability vectors and the process repeats).

4.8 Simulation Results

In this section, we will present simulation results for different configurations discussed in this paper. Here is a brief overview of various algorithms that will be discussed in this section.

1. **Multi-agent Wheeler-Narendra Algorithm:** First algorithm is a simple extension of Wheeler-Narendra (single agent Markov-chain control) algorithm to the MAMDP case. It works exactly the same way as the Wheeler-Narendra algorithm. Under this configuration, we represent each of the agents present in each state with a LA and use L_{R-I} algorithm as Learning algorithm. If there are N states in the MAMDP and M agents per state, then there will be $N \times M$ LAs. The learning within each state is completely decentralized as automata with a state and across the states do not share any information with each other. Also, all the automata make action selections in an autonomous manner. However, since the automata team may converge to the local maxima, the solution obtained this way might be sub-optimal. We use this algorithm as the baseline for the comparison with the novel algorithms proposed by us.

2. **L_{R-I}-based Superautomaton Algorithm:** All the LA within a state are replaced by a Superautomaton. If there are N states in the MAMDP, there will be N Superautomata. Each Superautomaton uses L_{R-I} algorithm. Each Superautomaton makes action selection in autonomous manner. The Superautomata converge to globally optimal policy. However, individual agents within a state lose their autonomy since they are all replaced by one Superautomaton which selects action on their behalf.

3. **Pursuit-based Superautomaton Algorithm:** It is similar to L_{R-I}-based Superautomaton algorithm with the exception that each Superautomaton uses Pursuit algorithm for learning.

4. **Centralized Pursuit Algorithm:** All the LAs within a state learn by using Pursuit algorithm in a centralized manner. The action selection is done autonomously while the learning is done in a centralized manner.

5. **Master-Slave Algorithm:** The learning agents within each state are replaced by Master and Slave automata. If there are N states in the MAMDP and M agents per state, then there will be $(M-1) \times N$ Master automata and N Slave automata in total. Each state individually has $M-1$ Master and *one* Slave automaton. The Master and Slave automata together converge to the globally optimal policy. Also, each Master and Slave automaton chooses its action in an autonomous manner. The action selection is done autonomously by Masters and Slave while the learning is done in a centralized manner.

We simulated these algorithms on a 2-agent, 2-state MAMDP. In the following table, we list the performance of these algorithms measured in terms of the number of iterations needed for convergence. The convergence value was set as 0.95. Whenever action probability value of any action belonging to an automaton reaches 0.95, we consider that particular automaton as a converged one. If a system has multiple automata (as is the case with all the algorithms under consideration in this paper), when all the individual automaton in the system converge, the whole system is regarded as a converged system and the algorithm execution stops. Suppose the states are numbered 1 and 2 and the actions for each automaton are also numbered 1 and 2. The transition and reward probabilities of the chain are summarized below:

$$\begin{aligned} t^1 \;=\; & \{[1,\{1,1\}] = 0.15; [2,\{1,1\}] = 0.85; [1,\{1,2\}] = 0.41; [2,\{1,2\}] = 0.59; \\ & [1,\{2,1\}] = 0.22; [2,\{2,1\}] = 0.78; [1,\{2,2\}] = 0.38; [2,\{2,2\}] = 0.62\} \end{aligned}$$

$$\begin{aligned} t^2 \;=\; & \{[1,\{1,1\}] = 0.68; [2,\{1,1\}] = 0.32; [1,\{1,2\}] = 0.73; [2,\{1,2\}] = 0.27; \\ & [1,\{2,1\}] = 0.56; [2,\{2,1\}] = 0.44; [1,\{2,2\}] = 0.35; [2,\{2,2\}] = 0.65\} \end{aligned}$$

$$r^1 = \{[1,\{1,1\}] = 0.2; [2,\{1,1\}] = 0.3; [1,\{1,2\}] = 0.4; [2,\{1,2\}] = 0.7;$$
$$[1,\{2,1\}] = 0.2; [2,\{2,1\}] = 0.9; [1,\{2,2\}] = 0.2; [2,\{2,2\}] = 0.3\}$$

$$r^2 = \{[1,\{1,1\}] = 0.7; [2,\{1,1\}] = 0.2; [1,\{1,2\}] = 0.4; [2,\{1,2\}] = 0.6;$$
$$[1,\{2,1\}] = 0.2; [2,\{2,1\}] = 0.9; [1,\{2,2\}] = 0.5; [2,\{2,2\}] = 0.8\}$$

Here t^i and r^i represent transition and reward functions for the i-th state of the MAMDP. The individual entries of these functions give the values of these parameters for various states and action tuples. The entry $t^i = [j,\{k,l\}]$ (or $r^i = [j,\{k,l\}]$) gives the transition (or reward) probability for the transition from state i to state j when the action tuple $\{k,l\}$ is selected (i.e. when one LA selects action k and the other selects action l). This 2-agent, 2-state MAMDP has *four* equilibrium points. The equilibrium points are listed in Table (4.1).

Table 4.1
Equlibrium Points

Equilibrium Tuple	Value
{1, 2, 1, 1}	0.55
{1, 2, 2, 2}	0.66
{2, 1, 1, 1}	0.64
{2, 1, 2, 2}	0.72

The equilibrium tuple $\{1,2,1,1\}$ indicates the optimal action for all the four automata in the system (2-states and 2-agents means $2 \times 2 = 4$ automata. First two actions correspond to the first two agents in state 1 and the last two correspond to the two agents in state 2. As indicated in the table, there are multiple equilibrium points (Nash equilibria) which represent multiple local maxima. Only one action tuple, $\{2,1,2,2\}$ corresponds to the global maximum.

Each algorithm was executed 50 times and the iteration values reported in the table are averaged over these 50 runs. The table also lists the converged policy value obtained by these algorithms and indicates whether it is a global maximum or a local maximum. We also list communication and storage complexity required *per state* by each algorithm. The storage (or space) complexity indicates the space required to store all the action probability values and other algorithm parameters (for example, estimate matrices in case of the Pursuit-based algorithm). For simplicity of the analysis, we assume that each automaton in the system has r number of actions, there are n number of states and m number of agents in the MAMDP.

Table 4.2
Performance Comparison

Configuration	Average Number of Iterations	Converged Policy Value	Communication Complexity per State	Space Complexity per State
Multi-agent Wheeler-Narendra	38149	0.627 (Local Maxima)	No Communication	$O(m \times r)$
L_{R-I}-based Superautomaton	40425	0.72 (Global Maxima)	No Communication	$O(m^n)$
Pursuit-based Superautomaton	23126	0.72 (Global Maxima)	No Communication	$O(m^n)$
Centralized Pursuit Algorithm	21174	0.72 (Global Maxima)	$O(m \times m)$	$O(m^n)$
Master-Slave	14518	0.72 (Global Maxima)	$O(m \times m)$	$O(m^n)$

The Multi-Agent Wheeler-Narendra algorithm requires no communication and has a modest space requirement. The space requirement grows linearly with the increase in number of automata in a state. However, this algorithm converges to one of the (possibly many) local maxima. Thus it can not guarantee a globally optimal solution. The L_{R-I}-based Superautomaton algorithm requires no communication between LAs and guarantees convergence to the global maxima. However, the space requirement grows exponentially with the number of automata in a state. Thus it has much higher space complexity than Multi-agent Wheeler-Narendra algorithm. One can view higher space requirement as a necessary trade-off for converging to the globally optimal policy.

The Pursuit-based Superautomaton algorithm has the same memory requirement as the L_{R-I}-based Superautomaton algorithm and it converges to the global maxima just like the L_{R-I}-based Superautomaton algorithm. However, it converges much

faster than the L_{R-I}-based Superautomaton algorithm. Centralized Pursuit algorithm, on the other hand, requires the LAs within a state to communicate their action choices plus it has same memory requirement as the Pursuit-based and L_{R-I}-based Superautomaton algorithms. Its convergence speed is comparable to the Pursuit-based Superautomaton algorithm and is much faster than L_{R-I}-based Superautomaton algorithm. The Master-Slave configuration requires same communication and memory capacity as the Centralized Pursuit algorithm and its convergence speed is also comparable to the Centralized Pursuit algorithm.

It is evident from the table that the new algorithms proposed in this paper do significantly better than the previously proposed Wheeler-Narendra algorithm. They converge to the global maxima, thus improving the quality of solution obtained. The amount of extra memory space needed can be justified in the light of the fact that the algorithm always converges to the globally optimal solution. This shows an interesting trade-off between the quality of the solution and memory requirement.

We also show the evolution of action probabilities of Master and Slave automaton to demonstrate the stochastic and temporal behavior of this algorithm. The convergence value for action probability was set at 0.9 in both Superautomaton and Master-Slave configuration. The average number of iterations for convergence were obtained by averaging over run of 50 simulations.

4.9 Heterogeneous Games

PDGLA allows for more decentralization in the games learning automata paradigm. The original team of automata can be split into various subgroups and each subgroup can run CPLA. We call this type of decentralization a HOmogeneous Games of Learning Automata (HOGLA). Although the automata are divided in different subgroups, each subgroup runs a single *type* of algorithm: namely the identical-payoff game. However, it is possible for each subgroup to run a different type of game. One subgroup may want to run identical-payoff game while the other subgroup uses

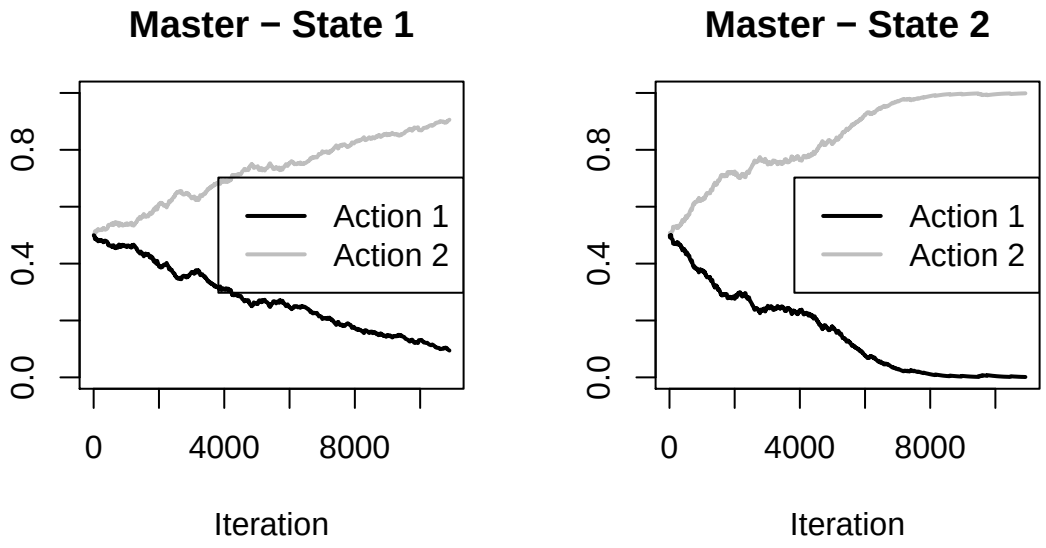


Figure 4.4. Action Probabilities for Master Automaton - 2-agent, 2-state MAMDP

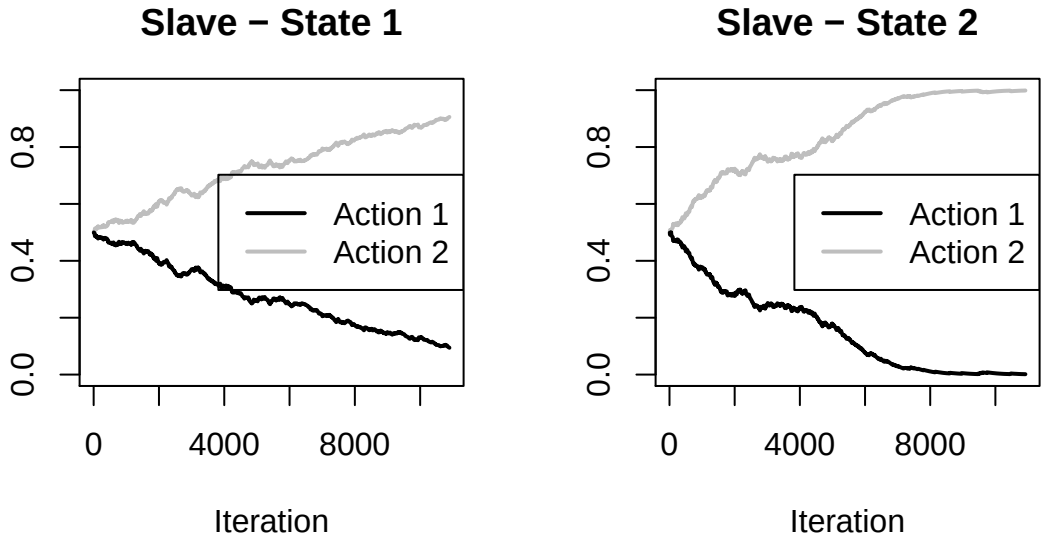


Figure 4.5. Action Probabilities for Slave Automaton - 2-agent, 2-state MAMDP

the zero-sum game. Such mixed-type game formulations are very interesting from a theory and application point of view. In the next chapter, we propose and explore

the application of HEterogeneous Games of Learning Automata (HEGLA). We use HEGLA to control the dynamic zero-sum games.

5 LEARNING IN DYNAMIC ZERO-SUM GAMES

The PDGLA framework allows a group of learning automata to be subdivided into multiple subgroups. Each subgroup acts as a locally centralized unit, although there is no centralization of the entire group. However, the PDGLA formalism requires the learning algorithms used across all the subgroups to be of the same *type*. Each subgroup can alternatively use a centralized or decentralized approach. But each subgroup participates in one and only one type of game: namely an identical-payoff game. Thus we termed this configuration as HOmogeneous Games of Learning Automata (HOGLA).

However, the PDGLA framework can be made more expressive by allowing different subgroups to participate in different types of games. In this thesis, we focus on two types of games: identical-payoff games and zero-sum games. Thus, one can imagine a configuration where the automata group is divided into two subgroups. While the automata residing in a one subgroup participate in an identical-payoff game. However, the automata in the other subgroup participate in a zero-sum game. This concept can be easily extended to scenarios where there are more than two subgroups present. In such case, automata in a few subgroups will participate in an identical-payoff game while the automata belonging to the other subgroups will participate in a zero-sum game. We call this framework HEterogeneous Games of Learning Automata (HEGLA) [46].

Figure (5.1) describes a system of *twelve* learning automata arranged and interacting using the HEGLA framework. One subgroup consists of five automata who are involved in an identical-payoff game using a centralized controller. Another subgroup consists of three learning automata who are involved in an identical-payoff game by communicating their action choice with every other automata in the system. System also consists of two automata (shown in black) who are involved in a centralized,

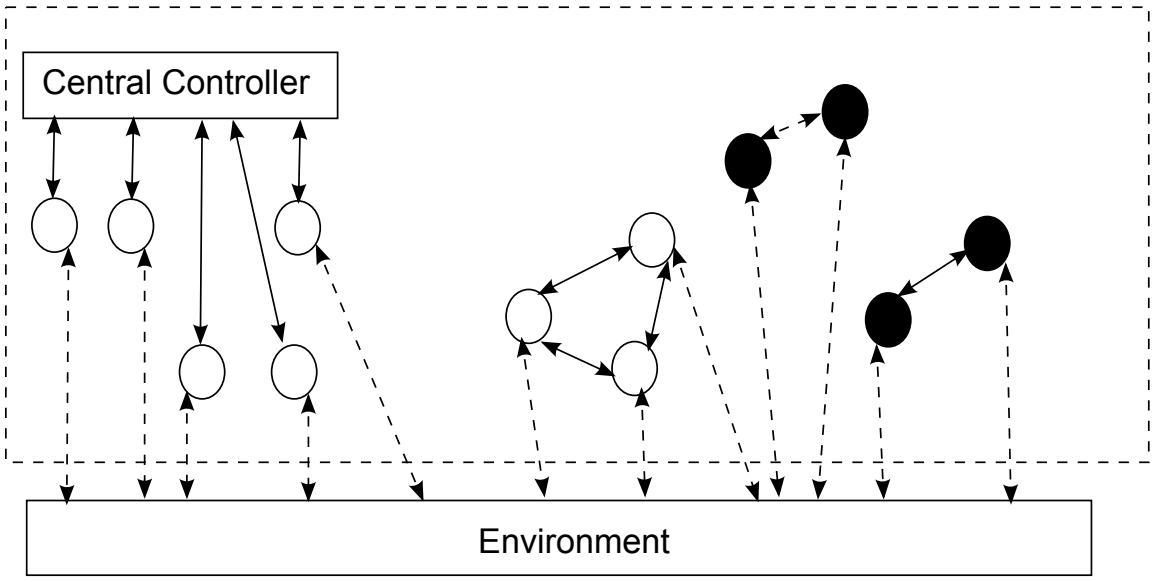


Figure 5.1. Heterogeneous Games of Learning Automata

zero-sum game (shown by a solid two-way arrow). Two other automata (shown in black) are involved in a zero-sum game which is completely decentralized (shown in wavy, dotted, two-way arrow). The system as a whole consists of automata involved in zero-sum game as well as identical-payoff game. In this chapter, we will describe a HEGLA-based solution for the control of Dynamic Zero-Sum Games (DZSGs).

In addition to the novel HEGLA solution [46], we propose an Adaptive Shapley Recursion (ASR) solution for the control of DZSG problem. We use the dynamic programming based Shapely recursion algorithm [47] to estimate optimal policies for given configuration of DZSG. This optimal policy is then used to estimate the optimal action in a given state. State transition and reward probabilities are estimated based on the frequency with which different transitions are performed and the feedback obtained during these transitions. These estimated parameters of the chain are used as the input to the Shapely algorithm in a recursive manner. In addition, we also propose a Temporal Difference (TD) style algorithm for the control of DZSGs. We call this algorithm Minimax-TD algorithm. Minimax-TD algorithm estimates the Q-values of every state and action tuple pair using TD formula and these estimates are

used for the control operation. The actions for each agent are selected by calculating saddle point in the Q-matrix and the Q-values are then updated based on the reward obtained by performing the selected actions.

5.1 Dynamic Zero Sum Games

A DZSG can be represented by a tuple $\langle S_1, S_2, \ldots, S_N; A_1, A_2; T; R_1, R_2 \rangle$ where $S = \{S_i\}, i = 1, 2, \ldots, N$ are the discrete set of states of the Markov chain, $A_j, j =$ 1or2 are the discrete sets of actions available to the agent j ($j = 1$ or 2). The two agents are called as Maximizing (or Row) player and Minimizing (or Column) player respectively. The joint action set is then given by $\mathbb{A} = A_1 \times A_2$. The transition probability function is defined as $T : S \times \mathbb{A} \times S \to [0, 1]$. The reward functions are defined as $R_i : S \times \mathbb{A} \times S \to \mathbb{R}$. For DZSG, we have $R_1 = -R_2$. The state transitions in DZSGs are the result of joint action of both the agents acting in state S_i. In turn, the instantaneous rewards also depend on the selected joint actions.

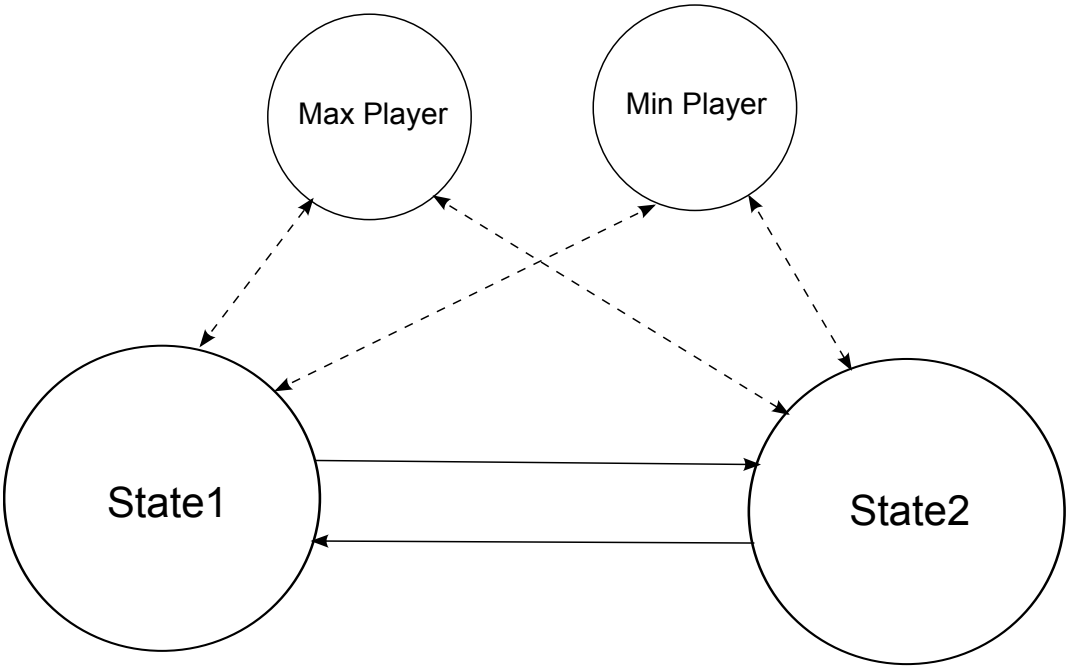


Figure 5.2. Dynamic Zero Sum Game

Figure (5.2) outlines a simple 2-state DZSG. As shown in the figure, two players play the DZSG. Each player picks an action independently (there is no communication between the two players) and then depending on the state in which DZSG finds itself, it transitions to a new state based on the selected action tuple.

A dynamic programming solution that optimizes the expected reward for the DZSG when complete information regarding transition and reward probabilities are available was given by Shapley [47]. However, the computational cost of this solution increases dramatically with increasing in number of states and actions. Thus the problem introduced by the presence of many states remains a major practical limitation for this dynamic programming based approach. In this regard, decentralization is a priority and serves to reduce the debilitating effect of large state space. Towards that end, we present a novel, completely decentralized learning algorithm using HEGLA. No knowledge of either transition probabilities or reward values is assumed, other than that these values are normalized to lie in the interval $[0, 1]$. We use L_{R-I} learning algorithm for LA to update its action probabilities so no explicit parameter estimation is needed. Also, the control scheme is implemented in a decentralized fashion.

5.2 Wheeler-Narendra Control Algorithm

The Wheeler-Narendra control algorithm [10] for a single-agent Markov chain has been described in detail in the previous Chapter. It uses the framework of identical payoff game of learning agents to solve the control problem. In the configuration proposed by the authors, one learning automaton is associated with each state of the Markov chain. Each LA uses simple L_{R-I} learning scheme [2] to update its action probabilities.

The algorithm assumes presence of a central bookkeeper which keeps track of the cumulative reward generated by the chain so far and global time which counts number of transitions performed by the chain so far. When the Markov chain transitions

to a state, the automaton acting within that state receives information about the cumulative reward generated by the chain so far and the current global time from the central controller. From these, the automaton calculates the average reward value which is used as the payoff β for the learning process. The L_{R-I} algorithm is used to update the action probabilities of the automaton in the following manner:

$$p(k+1) = p(k) + \lambda\beta(e_\alpha - p_i(k)), i = l, 2, \ldots, N$$

where $0 < \lambda < 1$ is a parameter. α is the action selected by this automaton during previous time when Markov chain was in the current state and e_α is a unit vector of appropriate dimension with α-th component unity.

We use a similar technique for the control of DZSGs. Two LAs are associated with each state of the DZSG (one for the ROW player or the MAX player and other for the COLUMN player or the MIN player) which learn using L_{R-I} algorithm. Also, the environment response for the the proposed HEGLA-based control scheme is calculated as the average reward generated so far by the Markov chain. The central bookkeeper keeps track of cumulative reward and number of transitions and the automata use this information to calculate the reward values.

5.3 Shapley Recursion

Shapely recursion [47] uses Dynamic Programming in a recursive manner. During each iteration, an estimated value matrix for every state of the Markov chain is calculated. The minimax value corresponding to this value matrix is calculated which is then used in the next iteration. It is proven that the ordering in this value matrix becomes same as the ordering in the actual game matrix in an asymptotic manner. Author proves that this algorithm converges to the optimal strategy asymptotically. Assume that the DZSG consists of N states. We use $\|A\|$ to denote the cardinality of the set A. We use the operator *minimax* to denote minimax value obtained by the maximizing player. It is easy to prove that for two matrices X and Y,

$$|\operatorname{minmax}[X] - \operatorname{minmax}[Y]| \leq \max_{i,j} |x_{ij} - y_{ij}|$$

Let $\overrightarrow{\alpha}$ be the N-dimensional, numeric vector where. For each state s of the Markov chain, the algorithm calculates a value matrix $A^s(\overrightarrow{\alpha})$ as

$$R(s,i,j) + \sum_{l=1}^{N} T(s,i,j,l)\alpha^l$$

where $i = 1, 2, \ldots, \|A_1\|$. At the start of the recursion, values in $\overrightarrow{\alpha}(0)$ are set arbitrarily. As mentioned earlier, $\overrightarrow{\alpha}(t)$ is calculated recursively using a dynamic programming technique as follows:

$$\overrightarrow{\alpha}^s(t) = \operatorname{minmax}[A^s(\overrightarrow{\alpha}(t-1)]$$

It is shown that the limit of $\overrightarrow{\alpha}(t)$ as $t \to \infty$ exists and is independent of $\overrightarrow{\alpha}(0)$, and its components are the optimal values of the stochastic game.

Shapley recursion is a dynamic programming technique which *computes* optimal strategy given all the parameters of the game. Thus, Shapely recursion needs complete information about the state of the chain with all the transition and reward probabilities known at the start of the algorithm. However, for many systems, these values are not known in advance. Thus there is a need to learn these values dynamically as the algorithm evolves. Also, the computational cost of Shapley recursion increases dramatically with increase in the number of states of the DZSG. Thus decentralization is a desirable characteristic of the algorithm used to control DZSGs. Towards this end, we propose our novel HEGLA-based algorithm which operates in a completely decentralized manner.

5.4 HEGLA Based Algorithm for DZSG Control

As described earlier, in HEGLA, a subset of LAs participate in identical-payoff game while another subset of LAs participate in zero-sum game. We assign one LA with each agent/player present in the state of the DZSG. Thus one LA is associated for maximizing (Row) player and one LA is associated with the minimizing (Column)

player in every state of the DZSG. The algorithm for learning optimal strategies in stochastic games is inspired from the Wheeler-Narendra algorithm described in [10].

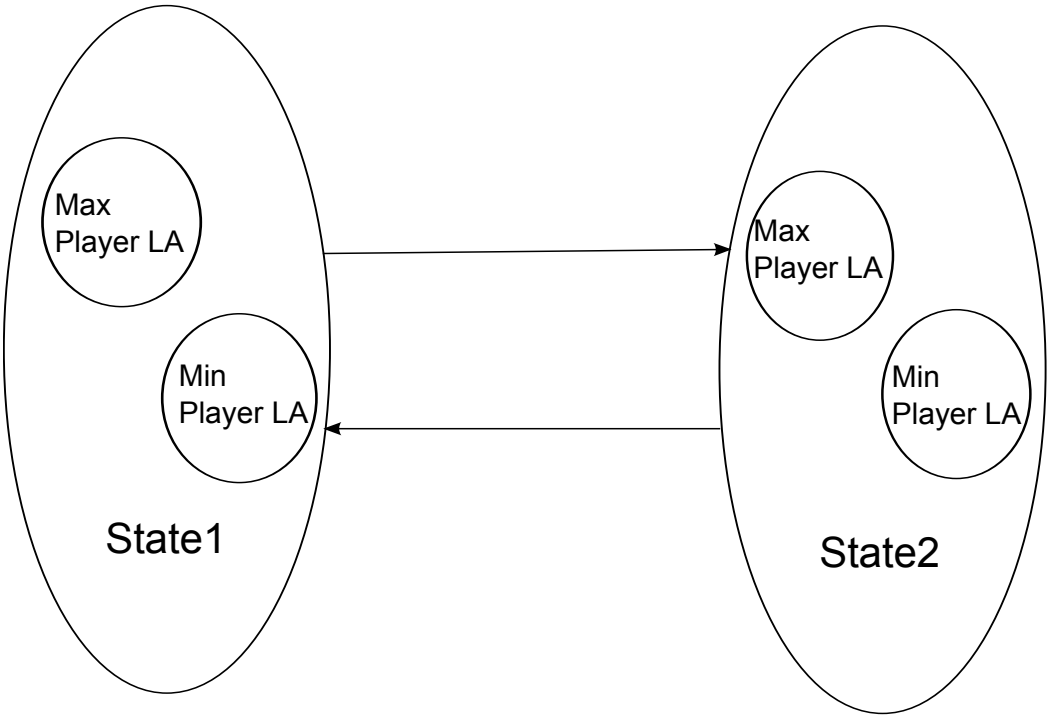


Figure 5.3. HEGLA Configuration for DZSG

The HEGLA arrangement for DZSG is described in the Figure (5.3). As depicted in the figure, each state of the HEGLA DZSG is inhabited by two learning automata. One automaton plays the part of the Maximizing or Row player (denoted by Max Player LA) while the other automata acts as the Minimizing or Column player(denoted by Min Player LA).

The LAs acting in a state s_i are not aware of the one-step reward values $R(s_i, A)$ resulting from their selected action tuple $\mathscr{A} \in \mathbb{A}$. The LAs in a state s_i receive information about the effect of their selected actions only when the Markov chain returns to that state. At each state s_i, we maintain two matrices: $REW^{s_i}(A,T)$ stores the cumulative reward obtained for action tuple $\mathscr{A} \in \mathbb{A}$ upto time T and $TIME^{s_i}(A,T)$ stores time passed since the action tuple $\mathscr{A} \in \mathbb{A}$ was last tried. The global time is incremented each time a state transition occurs. So the elapsed time

essentially indicates the number of state transitions that occurred since this state s_i was last visited. We assume that there is a central bookkeeper which provides following information when the control returns to state s_i:

1. the cumulative reward generated by the process up to time T.

2. the current global time $\mathcal{T}$.

From these, the algorithm computes Δ_T as the elapsed global time since the current state was last visited and Δ_R as the corresponding change in the global reward. Then we compute

$$REW^{s_i}(A, T+1) = REW^{s_i}(A, T) + \Delta_R$$

and

$$TIME^{s_i}(A, T+1) = TIME^{s_i}(A, T) + \Delta_T$$

Based on this information, the reward for the maximizing player is calculated as: $\beta^{Max} = \frac{REW^{s_i}(A,T+1)}{TIME^{s_i}(A,T+1)}$. The reward for minimizing player is given by: $\beta^{Min} = 1 - \beta^{Max}$. Note that the coordinator merely acts as a bookkeeper and not the decision maker. All the control functions are performed by the decentralized learning automata.

Both maximizing player and minimizing player LAs update their action probabilities as follows:

$$p^{Max/Min}(k+1) = p^{Max/Min}(k) + \lambda\beta^{Max/Min}(e_{\alpha_j} - p^{Max/Min}(k))$$

where α_j is the action chosen by Max (or Min) player and e_{α_j} is a unit vector of appropriate dimension with α_j-th component set to unity and all other components are set to zero.

The maximizing player and the minimizing player acting in every state are playing a zero-sum game with each other. However, using the same argument given by Wheeler-Narendra in [10], it is evident that all the maximizing player from all the states of the Markov chain form a logical team which is engaged in an identical payoff

game. Similarly, all the minimizing players from all the states form a logical group which participate in an identical payoff game. Thus we can formulate the resultant stochastic game as a game of zero-sum game between two teams of LAs. One team is composed of maximizing players from all the states of the Markov chain. The other team is composed of all the minimizing players from all the states of the Markov chain. These two teams form two virtual superautomata who play a zero-sum game between themselves. Thus the system of DZSG can be modeled as a heterogeneous game of LAs.

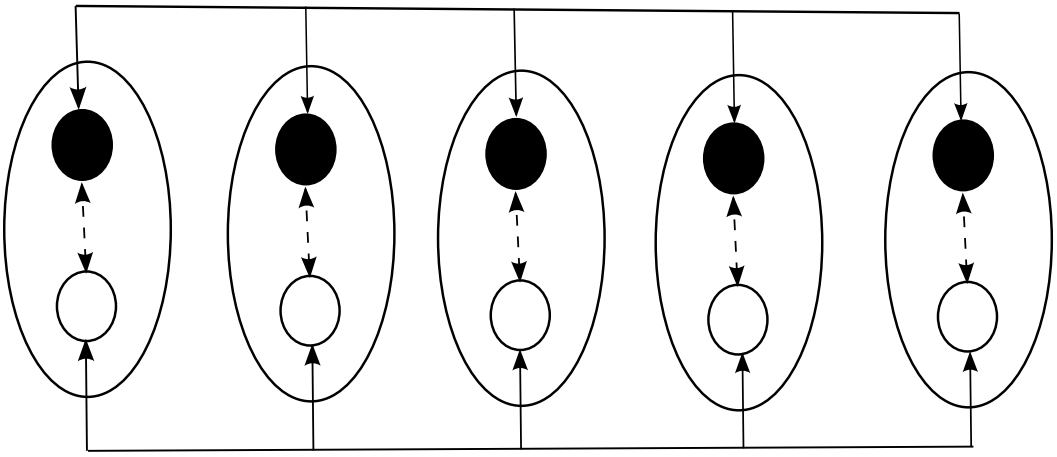


Figure 5.4. HEGLA Interaction in DZSG

Figure (5.4) depicts how learning automata that participate in a DZSG behave in a HEGLA-manner. The Max player learning automaton (denoted by the black circle) and the Min player learning automaton (denoted by the while circle) within each state participate in a zero-sum game among themselves (denoted by the dotted, two-way arrrow). However, the Max players across all the states for a team such that all the Max players are involved in an identical-payoff game. This is indicated by a solid arrow line that connect all the Max players (black circles) in the system. Similarly, all the Min player learning automata are involved in an identical-payoff game. Thus we have teams of automata who are involved in identical-payoff game

and teams of automata who are involved in a zero-sum game. Thus the system of learning automata that control the DZSG form a HEGLA configuration.

We now prove that the configurations of LAs proposed in this heterogeneous game algorithm will converge to the optimal policy for a given DZSG. Let M_i^k indicate the LA associated with each agent i (Max-player and Min-player) in the state k. Let A_i^k be the corresponding action set. Then $\Gamma = (N, A^k, J(\alpha^k))$ denotes a finite, zero-sum game among M_1^k and M_2^k in which the play $A^k = A_1^k \times A_2^k$ results in the payoff $J(A^k)$, where $J(A^k)$ is given by Shapley as

$$J(A^k) = R^k(A^k) + \sum_l T(k, A^k, l)\Pi^l$$

where Π^l represents the limiting reward values. We first state and prove some necessary lemmas.

Lemma 1. Γ has a unique equilibrium.

Proof: Follows directly from Theorems 1 and 2 of Shapley.

Here, we assume that the unique equilibrium exists in pure strategies. The proof and the algorithm works for the DZSGs where the saddle point exists in pure strategies.

Lemma 2. Saddle point in Γ represents optimal policy for the control of the zero-sum Markov chain.

Proof: We represent the controlled Markov chain using a game Γ, which is shown to have a unique equilibrium. We further assume that the equilibrium exists in pure strategies. If Γ were an automata game, then players using L_{R-I} learning scheme would be ϵ-optimal (based on [48], Theorem on Page 4). Thus each automata team acting within the state will converge to the saddle point of the game matrix D^k. However, a payoff in Γ is obtained asymptotically by using a fixed policy. Thus at each time step n, the Γ is approximated by $\Gamma(n)$. Since each decision maker uses L_{R-I} updating procedure, Γ becomes a limiting game $\Gamma = \lim_{n\to\infty} \Gamma(n)$. The elements of $\Gamma(n), d^k(A^k, n) = E[\beta(n)|\alpha(n) = A^k]$, depend on n. Thus the Zero-Sum Markov chain updating is not the same as the updating in the zero-sum automata game described

in Lakshmivarahan-Narendra [48]. However, since we assume that the Markov chain is ergodic, it follows that for a sufficiently large n, the ordering among the $d^k(A^k, n)$ will be identical to that among the $J(A^k)$ in Γ. Therefore, it is sufficient to analyze the automata game Γ.

Based on these two leamms, the main result can be stated as follows.

Theorem 1. Let an automaton M_i^k using L_{R-I} learning scheme, having A_i^k actions, be associated with each agent of an N-state DZSG. If the chain is ergodic, then for any $\epsilon > 0$, there exists an $0 < \lambda^* < 1$ such that for any $\lambda < \lambda^*$ in L_{R-I} learning scheme, $\lim_{n\to\infty} > J(A^k) - \epsilon$.

Proof: The proof follows immediately from lemma 1 and lemma 2.

5.5 Adaptive Shapley Recursion

The idea behind indirect adaptive methods is to estimate the unknown model parameters based on the data obtained during the execution of the model. For the DZSGs, the task is to estimate unknown transition and reward probabilities based on the history of transitions and immediate costs observed as the Markov chain goes through various transitions. The Adaptive Shapley Recursion (ASR) algorithm for *learning* the optimal strategies for a DZSG proceeds as follows. To initialize, the estimated model parameters of the DZSG are set to arbitrary values. At each time step t during the execution of the algorithm, we use Shapley recursion algorithm to determine optimal action values based on the latest estimated parameter values of the DZSG. Using *certainty equivalence principle*, we assume this to be optimal action of the DZSG and execute that action. The information obtained during the execution (reward values, action taken during transition etc.) of this action is further used to update the estimates of the model parameter and this process repeats.

The action based on certainty equivalence principle appears to be best based on information obtained upto time t and hence it is pursued. However, since the estimated model at time t may not be the same as actual model parameter values

we are trying to estimate, we must occasionally also pursue actions that are different from the one given by the certainty equivalence principle. A simple way of doing it is by using randomized policies where actions are selected according to a probability distribution. We use ϵ-greedy action selection technique. Thus with probability ϵ, we select the optimal action as suggested by Shapley recursion and we select some other random action with probability $1 - \epsilon$. The objective of this strategy is to achieve a balance between exploration and optimization.

Following description explain the proposed ASR algorithm for learning optimal control policy for a DZSG.

1. For each state of the DZSG Markov chain, we estimate the state transition probabilities and reward values based on the past experience of state-transitions and rewards observed by the agents acting in that state. Let $n_k(s_i, s_j, a_{lm})$ represents the number of transitions observed from state s_i to state s_j until time step k when action tuple a_{lm} was chosen. Then we calculate the total number of times the action a_{lm} was executed in state s_i as follows:

$$N_k(s_i, a_{lm}) = \sum_{s_j \in S} n_k(s_i, s_j, a_{lm})$$

   From this, the transition probabilities at step k are estimated as follows:

$$\hat{P}_k(s_i, s_j, a_{lm}) = \frac{n_k(s_i, s_j, a_{lm})}{N(s_i, a_{lm})}$$

2. On the basis of the rewards obtained for every transition, transition rewards estimates $\hat{R}_k(s_i, s_j, a_{lm})$ can be directly determined.

3. On the basis of the model parameters estimated at time step k (namely $\hat{P}_k(s_i, s_j, a_{lm})$ and $\hat{R}_k(s_i, s_j, a_{lm})$ time, we then execute Shapley recursion to obtain the optimal policy for given configuration of model parameters.

4. Once the Shapley recursion algorithm has converged, we use ϵ-greedy technique to choose an action tuple $a^{s_i}_{l'm'}$ in current state s_i.

5. Based on the selected action tuple, the DZSG estimate parameters (namely $\hat{P}$ and $\hat{R}$) are updated as explained in step 1 and the process repeats.

5.6 Minimax-TD

Temporal Difference (TD) learning combines the dynamic programming methodology with Monte Carlo paradigm. Like Monte Carlo methods, TD methods learn directly from online experience without a model of the dynamics of the environment. Like dynamic programming, temporal difference method updates estimates of the current state based on the previously learned estimates which are calculated during previous states of the system evolution. We propose a Temporal Difference (TD) algorithm to control the DZSG. It is called as Minimax-TD algorithm. Minimax-TD uses action tuples for the indices of the Q-matrix. The action tuple selection is done by calculating the minimax value in the Q-matrix. The Minimax-TD algorithm proceeds as follows:

1. Initialize $Q(s, \alpha_1^s, \alpha_2^s)$ arbitrarily $\forall s \in N$ and $\forall \alpha \in A_1 \times A_2$.

2. Arbitrarily select the starting state $curr \in N$ and call it the current state.

3. Repeat forever

   (a) Select (α_1^t, α_2^t) by performing $minmax(Q(curr))$. Then select α_1^{curr} and α_2^{curr} by using ϵ-greedy technique.

   (b) Based on action tuple (α_1^{curr}, α_2^{curr}), transition to state $next \in N$ and observe the reward r.

   (c) Select $(\alpha_1^{t'}, \alpha_2^{t'})$ by performing $minmax(Q(next))$. Then select α_1^{next} and α_2^{next} by using ϵ-greedy technique.

(d) Calculate

$$\begin{aligned} Q(curr, \alpha_1^{curr}, \alpha_2^{curr}) &= Q(curr, \alpha_1^{curr}, \alpha_2^{curr}) \\ &\quad +\delta[r + Q(next, \alpha_1^{next}, \alpha_2^{next}) - Q(c, \alpha_1^{curr}, \alpha_2^{curr})] \end{aligned}$$

(e) Set $curr = next$ and goto step 1.

Minimax-TD algorithm is analogous to the SARSA-TD algorithm [1] for the control of a single agent Markov chain with minimax operator used in conjunction with ϵ-greedy technique for action selection.

5.7 Simulation Results

In this section, we will present simulation results for different algorithms discussed in this chapters. Here is a brief overview of various algorithms that will be discussed in this section.

1. **HEGLA Based Algorithm:** First algorithm is a learning automata based algorithm where all the learning automata in the DZSG participate in a HEGLA. Under this configuration, we represent each of the agents present in each state with a LA and use L_{R-I} for updating action probabilities. If there are N states in the DZSG, then there will be $N \times 2$ LAs. The learning within each state is completely decentralized as automata with a state and across the states do not share any information with each other. Also, all the automata make action selections in an autonomous manner.

2. **Adaptive Shapley Recursion (ASR) Algorithm:** The adaptive Shapley recursion algorithm *learns* the parameters of the DZSG (namely the transition and reward probabilities) and further improves these learned estimates using a dynamic programming algorithm (namely Shapley recursion).

3. **TD Based (Minimax-TD) Algorithm:** The Q-values are updated using a TD-style equation. The action selection is done based on the minimax value in the Q-matrix.

We simulated these algorithms on a 2-state DZSG. Suppose the states of the DZSG are numbered 1 and 2 and the actions for each agent are also numbered 1 and 2. The transition probabilities of the DZSG are summarized below:

$$\begin{aligned} t^1 \;=\; & \{[1,\{1,1\}]=0.5;[2,\{1,1\}]=0.5;[1,\{1,2\}]=0.5;[2,\{1,2\}]=0.5; \\ & [1,\{2,1\}]=0.5;[2,\{2,1\}]=0.5;[1,\{2,2\}]=0.5;[2,\{2,2\}]=0.5\} \end{aligned}$$

$$\begin{aligned} t^2 \;=\; & \{[1,\{1,1\}]=0.5;[2,\{1,1\}]=0.5;[1,\{1,2\}]=0.5;[2,\{1,2\}]=0.5; \\ & [1,\{2,1\}]=0.5;[2,\{2,1\}]=0.5;[1,\{2,2\}]=0.5;[2,\{2,2\}]=0.5\} \end{aligned}$$

Here t^i represent transition function for the i-th state of the Markov chain. The entry $t^i = [j,\{k,l\}]$ gives the transition probability for the transition from state i to state j when the action tuple $\{k,l\}$ is selected (i.e. when one LA selects action k and the other selects action l).

The reward probabilities are as follows:

$$r^1 = \{[\{1,1\}]=0.1;[\{1,2\}]=0.2;[\{2,1\}]=0.3;[\{2,2\}]=0.4;\}$$

$$r^2 = \{[\{1,1\}]=0.5;[\{1,2\}]=0.6;[\{2,1\}]=0.7;[\{2,2\}]=0.8;\}$$

Here r^i represent reward function for the i-th state of the Markov chain. The entry $r^i = [\{k,l\}]$ gives the reward probability for the state i when the action tuple $\{k,l\}$ is selected (i.e. when one LA selects action k and the other selects action l).

The action tuple $\{2,1,2,1\}$ is the optimal action tuple for this particular DZSG. There are four agents in the system (two states and two agents per state means

$2 \times 2 = 4$ agents). First two actions of the optimal action tuple give the optimal actions for the first two agents in state 1 and the last two correspond to the two agents in state 2.

For the HEGLA-based DZSG control algorithm, the convergence value was set as 0.95. Whenever action probability value of any action belonging to a LA reaches 0.95, that automaton is termed as converged. If a system has multiple learning automata (as is the case with the heterogeneous game algorithm presented in this paper), when all the individual automaton in the system converge, the whole system is regarded as a converged system and the algorithm execution stops. In the Figures (5.5) and (5.6), we show the evolution of action probabilities of the Maximum (Row) and Minimum (Column) automaton that reside in the first state of the DZSG. As indicated in the figure, the Row player converges to action 2 and Column player converges to action 1. These are the optimal policies for these two player. This demonstrates the stochastic and temporal behavior of the LA based DZSG control algorithm.

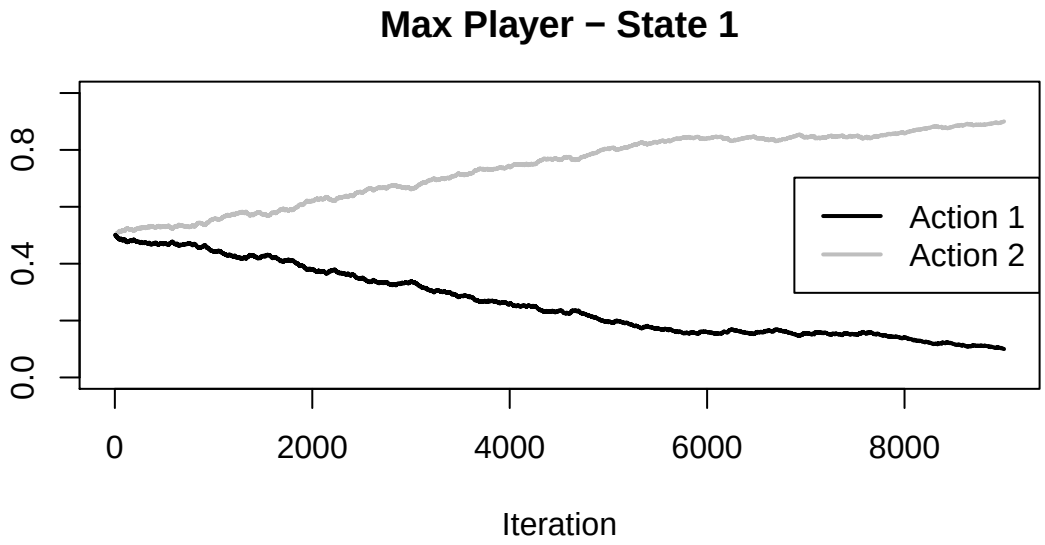


Figure 5.5. Evolution of Action Probabilities for the Maximum (Row) Automaton In A 2-state DZSG

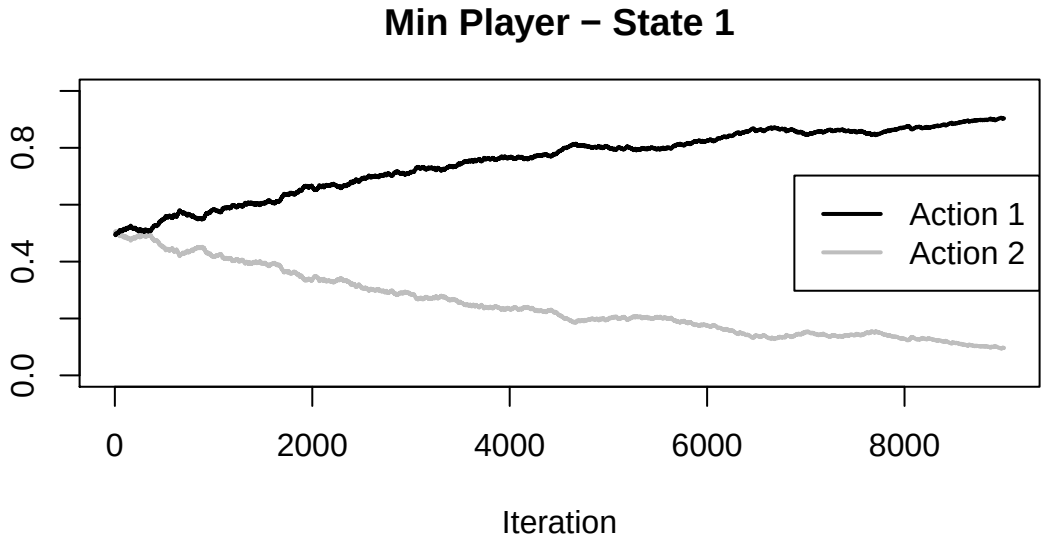


Figure 5.6. Evolution of Action Probabilities for the Minimum (Column) Automaton In A 2-state DZSG

For the Minimax-TD algorithm, the Figure (5.7) demonstrates the behavior of the Q values of the for the agents in the first state of the DZSG. The figure depicts the variation in Q-values for a 4000 iteration size window during the execution of the algorithm. As it is evident from the figure, the Q-value for action $\{2, 1\}$ represent the minimax value in the matrix formed by the four action tuples. Thus the algorithm will select the $\{2, 1\}$ action tuple for the control of the DZSG.

Figure (5.8.a) depicts the temporal behavior of Shapley recursion algorithm. We plot A matrix (value matrix) for a 4000 iteration size window. As demonstrated in the figure, all the matrix values always increase monotonically over the execution of the algorithm. Although it appears that all the matrix values are superimposed on a single line, actually very small differences exist between the individual values in the matrix. However, these differences have much lower resolution that then resolution of the vertical axis. However, the minimax value in the A matrix always resides in the entry $\{2, 1\}$.

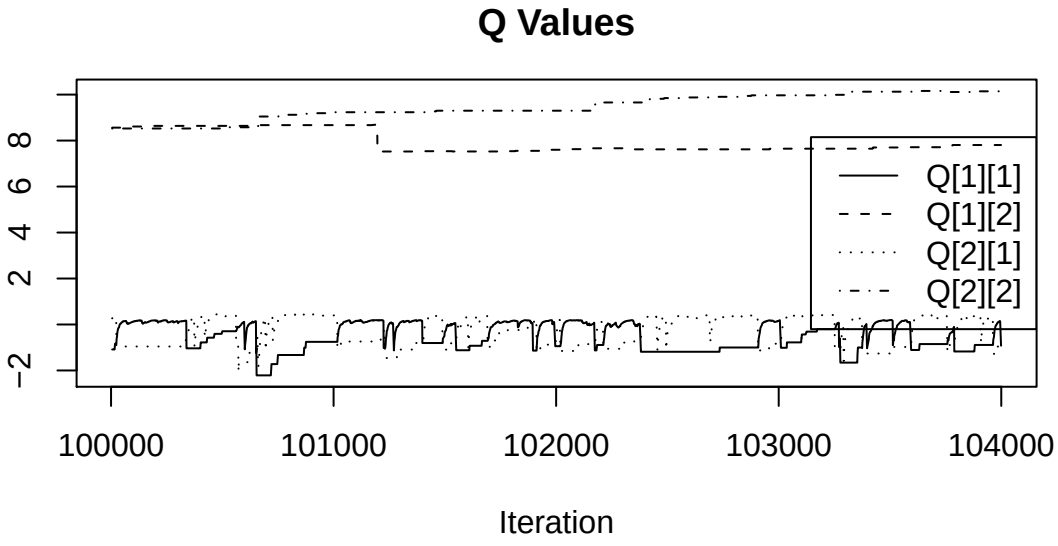
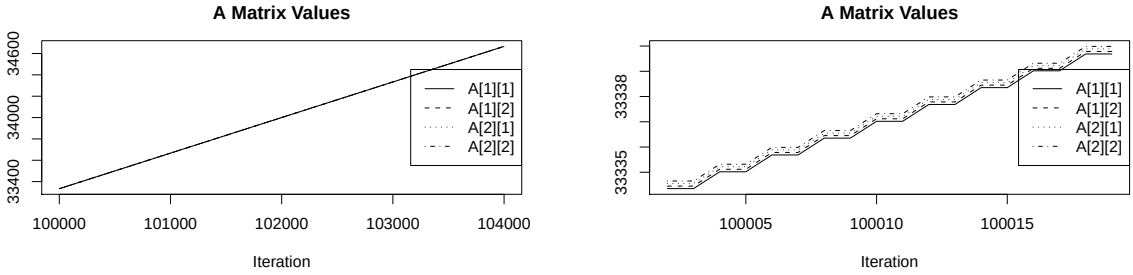


Figure 5.7. Evolution of Action Probabilities for the Minimum (Column) Automaton In A 2-state DZSG



(a) (b)

Figure 5.8. The value matrix (A matrix) entries for the Shapley recursion. (a) and (b) show these values at different scales and resolution.

Figure (5.8.b) shows the behavior of the Shapley recursion algorithm for the first 20 iterations. As demonstrated in the figure, all the matrix values always increase monotonically over the execution of the algorithm. However, at this scale and resolution, we can clearly see how the individual A matrix (value matrix) entries are ordered

with respect to each other. It is clear that the minimax value in the A matrix resides in the entry $\{2, 1\}$.

These simulations show that the HEGLA approach is applicable to DZSG where optimal control policy exists in the pure strategies. If the optimal policies exist in mixed strategies, other learning automata algorithms (e.g. L_{R-P} algorithm or some modification of L_{R-I} algorithm) might be useful. However, this particular issue needs more investigation. The TD-minimax algorithm uses the *minimax* operator which finds optimal policies in pure strategies. However, it will be straightforward to extend this technique to include cases where optimal policy exists in mixed strategies. We can use linear programming technique used in [11] for learning mixed strategies under TD-minimax formalism. The use of linear programming in learning algorithm can lead to large computational complexity and thus slow convergence speed. However, it might be possible to use approximate algorithms to find linear solutions which might be sufficient to obtain optimal convergence. In the next chapter, we describe some applications of the decentralized learning methods using the DPLA.

6 APPLICATIONS OF DECENTRALIZED PURSUIT LEARNING ALGORITHM

In this chapter, we will describe a framework for solving computationally hard, distributed function optimization problems using reinforcement learning techniques. In particular, we model a function optimization problem as an identical-payoff game played by a team of learning automata. The team performs a stochastic search through the domain space of the parameters of the function. We use the novel Decentralized Pursuit Learning Automata (DPLA) game algorithm. We describe a formulation of the NP-Hard sensor subset selection problem and watershed management problem as an identical-payoff game of learning automata. We then apply the DPLA algorithm to compute optimal solutions for these problems, thus demonstrating the viability of the DPLA.

6.1 Function Optimization Using Decentralized Pursuit Algorithm

In this section, we describe a generic procedure for learning the parameter values that *maximize* a given function [49]. The procedure given here is based on the one described in [20]. Assume a function $Y = f(X_1, X_2, \ldots, X_N)$. We will assume that each parameter $\{X_i, \forall i\}$ can have real values ($X_i \in \mathbb{R}, \forall i$). Assume that each parameter $\{X_i, 1 \leq i \leq N\}$ can take values from range R^i_{min} and R^i_{max} ($X_i \in [R^i_{min}, R^i_{max}]$). Each parameter range is divided into s_i number of subranges. Each such subrange j of parameter X_i will be represented by its midpoint mid^i_j. So each parameter X_i will be represented by a s_i-tuple $M^i = \{mid^i_1, mid^i_2, \ldots, mid^i_{s_i}\}$. Thus the parameters can be discretized to a desired level of granularity and this discretization will be used to map the function optimization problem onto the framework of decentralized and partially decentralized game of learning automata.

Using the discretized parameter space, each parameter X^i is represented by a learning automata A^i running pursuit algorithm. Assign M^i to be the action set of automaton A^i. All the initial action probabilities are set to $P^i_j = \frac{1}{s_i}$. Each automaton A^i selects an action by sampling its action probability vector P^i. The selected action $P^i_{i_s}$ corresponds to choosing a particular value $mid^i_{i_s}$ from the M^i vector. Then the entries of the game matrix are calculated as: $\mathbb{D}_{1_s 2_s \ldots N_s} = \mathbb{S}(f(mid^1_{1_s}, mid^2_{2_s}, \ldots, mid^N_{N_s}))$, where $\mathbb{S}$ is a Sigmoid function. By sampling from this distribution, the payoff β for all the automata in the team is determined. A suitable value can be used as the threshold for convergence. When one of the action probabilities in the action probability vector of an automaton reaches the threshold value, it is considered that the automaton has converged to that action. When all the automata in the team converge to a particular action, then the entire team converges to the corresponding action tuple.

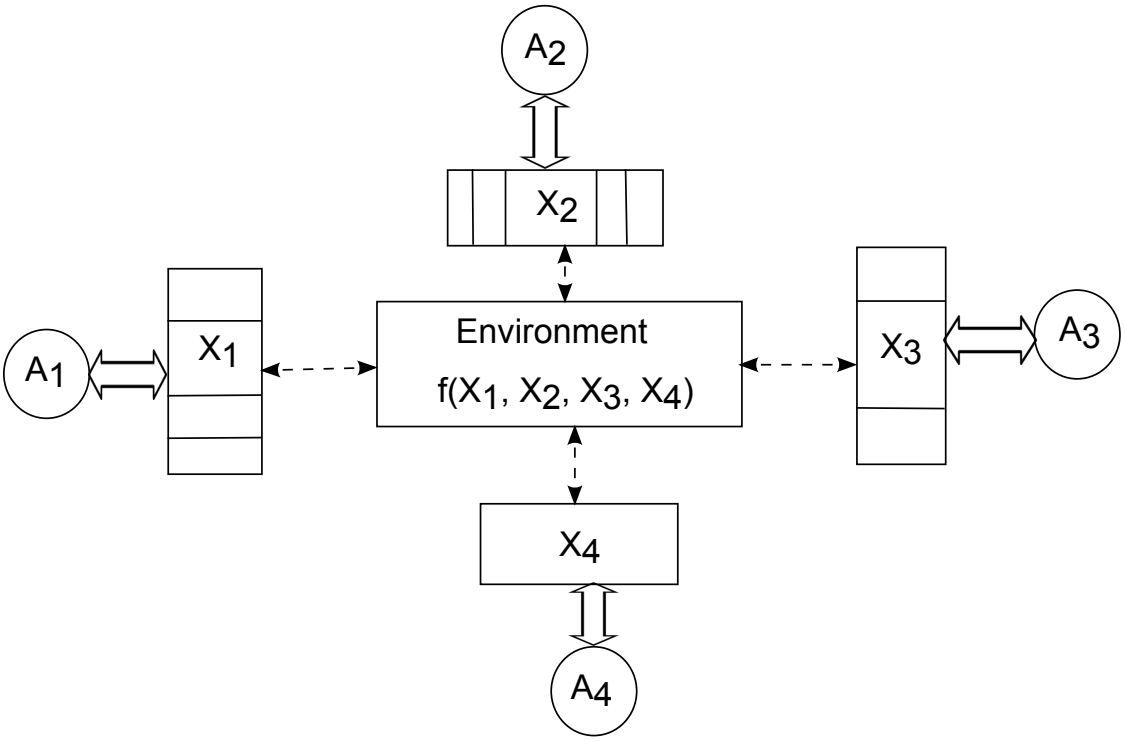


Figure 6.1. Function Optimization Using DPLA

Figure (6.1) describes the proposed arrangement. It describe a function $Y = f(X_1, X_2, X_3, X_4)$ which has four parameters. As shown in the figure, each parameter is assigned to one learning automata. Each LA runs DPLA and thus acts in a decentralized and autonomous manner. Each parameter is divided into different number of ranges (for instance, parameter X_1 is divided into four section while parameter X_4 is not divided into any subsections. Each action of the learning automata is associated with one particular subrange of the associated parameter. The automata team makes an action tuple selection in a stochastic manner. This forms a parameter vector which is then evaluated by the environment. The environment gives a feedback for the chosen parameter vector. This feedback is used by the automata team to update the action probabilities and learn the optimal parameter vector.

6.2 Optimal Sensor Subset Selection

The problem of selecting a subset of sensors in a distributed object tracking environment that optimizes an objective function consisting of a trade-off between data accuracy and energy consumption is known to be *NP-hard*. Many sensor selection algorithms proposed in the literature [50], [51], [52], [53] perform their analysis off-line and on a static and completely known sensor configuration. Coverage is considered as primary criteria and sensors are divided in subsets so that entire space is always covered by a suitable set of sensors. The problem is exacerbated because of the uncertainty and dynamic nature of either sensor characteristics or the environment or both. We propose, for the first time, a novel framework based on a reinforcement learning approach, to deal with the problems of computational complexity, dynamic nature and uncertainty for sensor subset selection [54]. Our proposed sensor subset selection approach is completely decentralized and sensors do not need to know even the presence of other sensors in the system. This makes our approach extremely scalable and easy to implement in a distributed system. To the best of our knowledge,

this is the first application of reinforcement learning to the domain of sensor subset selection.

6.2.1 Problem Description

Distributed object tracking in a multi-sensor environment is an important technological problem that is used for the design of various security, computer-aided surgery, and business traffic monitoring applications. In a distributed object tracking system, multiple, geographically dispersed sensors collaborate with each other to determine the position and other state variables of an object over time. However, the problem of integrating the data generated by a team of such sensors to reconstruct the state trajectory of an object is exacerbated by the presence of uncertainty concerning sensor characteristics as well as the dynamic nature of the state of the target object.

If the sensor infrastructure is static with known sensor characteristics, such a system can be hand-designed and the sensor subset selection problem can be solved a priori for each instance of the problem to minimize the objective function. However, in the real-world, there is usually a considerable amount of uncertainty concerning sensor characteristics such as errors and energy consumption and these properties are dynamic in nature. In such cases, the sensor selection problem needs to be solved online in a dynamic manner to account for uncertainties. We propose a framework which, for the first time, uses a reinforcement machine learning approach to decide which sensors are to be turned on or off for a given object state in a decentralized manner. The measurements generated by the selected subset of sensors, will subsequently be combined or fused to achieve object tracking. We will investigate and use different algorithms for selecting optimal subset of sensors and compare the performance of these algorithms.

We use a Federated Kalman Filter approach [55] for sensor data fusion where a separate Kalman Filter is assigned for each sensor and a Master Filter is then used to combine the observations of the multiple Kalman Filters. The advantage of the

federated Kalman Filter approach is that sensors can be modeled as autonomous local filters and need not know the presence of other sensors in the system. Once the Master Filter generates an overall combined error covariance, this error covariance, along with actual energy consumption can be used to generate a reinforcement signal based on the objective function for the sensor subset selection problem. These reinforcements can be used by individual reinforcement learners in sensors to update their strategies concerning whether to turn themselves on or off in a dynamic and adaptive manner. We assume that the our reinforcement learning code will be embedded in sensor software making each sensor intelligent and adaptive to the uncertainties and dynamic nature of the environment. This will help in the development of the next generation of distributed object tracking systems which will be able to deal with dynamic, and uncertain environments.

6.2.2 Techniques/Algorithms for Sensor Selection

The general sensor subset selection problem has been shown to be NP-Hard [56]. Many sensor selection algorithms proposed in literature perform their analysis off-line and on a static and completely known sensor configuration. Coverage is considered as primary criteria and sensors are divided in subsets so that entire space is always covered by a suitable set of sensors. In [50], the sensor nodes are divided into sets, such that each set is capable of providing complete coverage of the field and only one set is active at a time. This problem is formulated as a generalized maximum flow graph and an optimal solution is found through linear programming (LP). [51] solves the same problem but here sets are scheduled in a round-robin order and the focus is on the problem of finding the maximum number of disjoint sets. The problem is transformed into a max-flow problem, which is formulated as a Mixed Integer Programming (MIP) instance. The output of the MIP is used to compute the disjoint set covers in polynomial time. In [52], full coverage with minimal sensors is obtained by identifying the redundant sensors and turning them off. Identification

of redundant sensors is done using Voronoi diagrams [57]. In [53], the authors aim to provide k-coverage, which means that every point in the field is covered by at least k sensors. The sensors are turned on one by one in a greedy fashion, with the sensor with most contribution turning on first, then the next one, and so on. The contribution is computed based on the probability of detection of an event by that sensor within its sensing area. [58] provides a self-scheduling scheme, in which time of operation is the only parameter in the selection process. Here, the nodes dynamically schedule themselves while guaranteeing a certain degree of coverage. The sensors are time-synchronized, and each sensor generates a random reference time which is exchanged with its neighbors. Each sensor then establishes its sleep-awake cycle by observing the reference time of its neighbors. As mentioned earlier, these algorithms are centralized and are off-line in nature and require the knowledge of entire sensor configuration in order to make sensor selection. Our proposed reinforcement learning based algorithm (described next) on the other hand, is an on-line and decentralized algorithm.

There are some sensor selection approaches that use Entropy as criteria for sensor selection. Entropy refers to a measure of uncertainty. The lesser the entropy of some measurement, the more we can be certain of its accuracy. In [59], the authors use the mutual information about the future state and the current node measurement to determine the information gain of the different sensors. A greedy approach is used to solve the sensor selection problem for target localization and tracking. The goal here is to reach the required entropy level without using more sensors than necessary. In [60], given a prior probability distribution of the target location and the locations and sensing models of a set of sensors, an informative sensor is selected such that the collection of the selected sensor observation with the prior target location distribution results in the greatest reduction in uncertainty. The proposed heuristic adds one sensor at a time to reduce the entropy of the target location distribution. Both these approaches are centralized in nature, are computationally-intensive and consider only one criteria for sensor selection. Our reinforcement learning approach on the other

hand is decentralized in nature and we consider various trade-off while making sensor selection.

6.2.3 Distributed Tracking System Setup

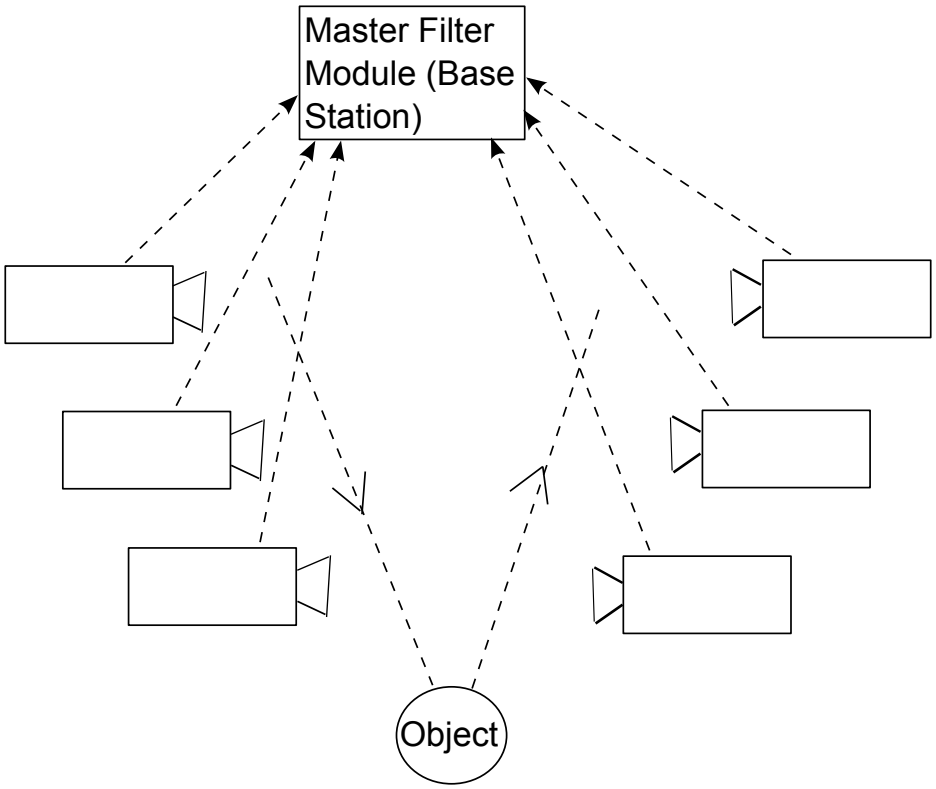


Figure 6.2. A Distributed Object Tracking System

The setup of a typical vision-based distributed object tracking system is shown in Figure (6.2). The object to be monitored roams around in a space which is covered by a group of cameras (representing the sensors in our system). We assume that the cameras are placed so that the entire space is covered by the cameras. So the object is continuously tracked by at least one camera while it moves around. The set of cameras tracking the object may change but at least one camera always tracks the object. The object has an attached bar-code like marker which helps the camera

to identify and calculate the position information of the object. Each camera is attached to a processing unit which can process the image generated by the camera to identify the marker and then use appropriate mathematical technique to generate the position information of the marker (and hence the object). Each camera (that can see the marker), then sends this information to a central module (also called the base station). The central module combines the reading received by all the sensors to generate the resultant position data of the object.

However, there are many errors that are introduced in the position data generated by the camera. They can be due to projection or image noise, to name a few. The error typically depends on the distance of the object from the camera. The farther away the object, greater is the error generated by the camera for its position. Hence we need a mechanism to model these errors. We use Kalman filter [61, 62]as the tool to model these errors in a Gaussian estimation framework and generate error covariance values using equations of the Kalman filter. Each sensor(camera) runs a local Kalman filter to produce object position data with the corresponding error covariance. This position data with corresponding error covariance is then sent to the base station for fusion. We use Federated Kalman Filter architecture [55] for data fusion. This architecture allows each sensor to operate in a completely decentralized manner. Since we do not know the set of cameras that can track the object in advance and since this set changes with time, it is not possible to design a centralized Kalman filter framework. The decentralized nature of Federated kalman filter allows us to deal with dynamicness of the system in an efficient manner. Figure (6.3) shows the sensor architecture utilizing the Federated Kalman filter framework.

Each camera runs its own local Kalman filter. The local Kalman filter runs in the Information mode [55]. This is an alternate form of Kalman filter which calculates value of information (inverse of error) associated with each sensor reading. The Master Filter combines these individual information matrices to produce resultant information matrix. The inverse of the resultant information matrix gives the re-

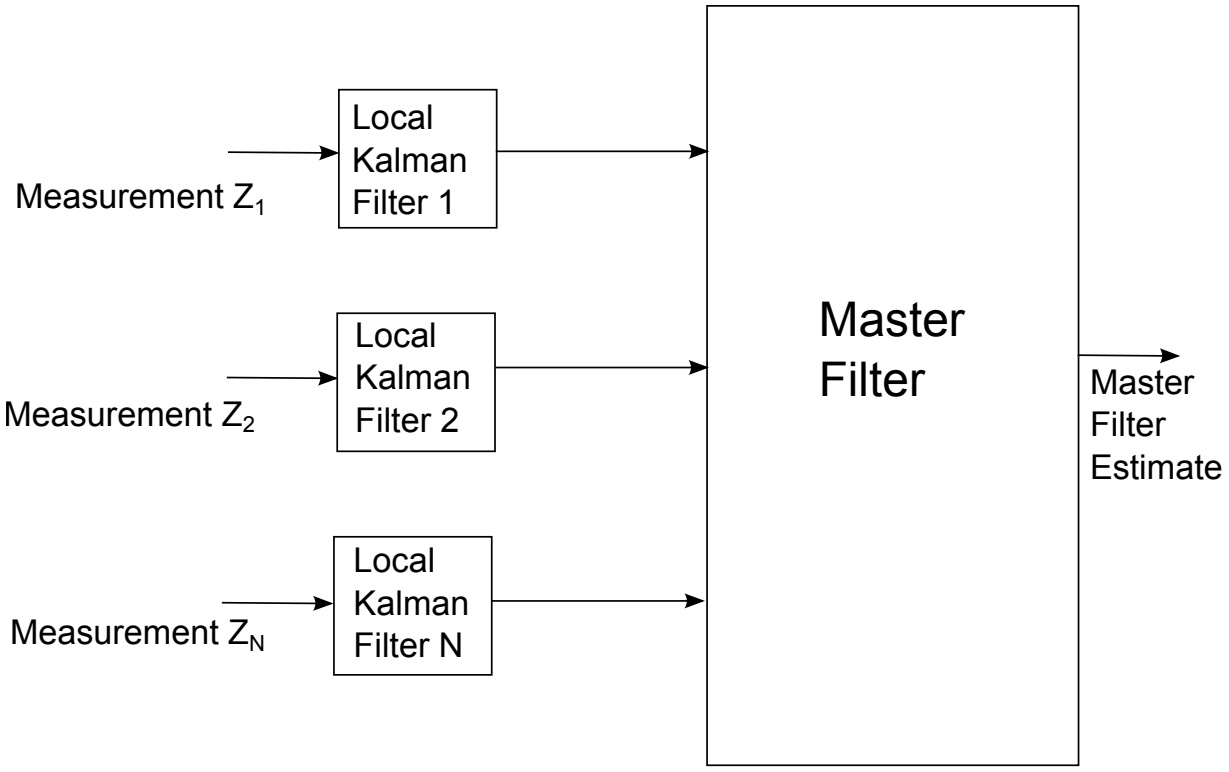


Figure 6.3. Federated Kalman Filter

sultant error covariance matrix P. We briefly describe the Kalman filter running in Information mode. We use following terms in the equations:

- $x_k = (n \times 1)$ process state vector at time t_k.
- $\varphi_k = (n \times n)$ matrix relating x_k to x_{k+1} i.e. state transition function.
- $w_k = (n \times 1)$ vector - process noise - assumed to be white Gaussian with known covariance Q.
- $z_k = (m \times 1)$ vector measurement at time t_k.
- $H_k = (m \times n)$ is the measurement matrix.
- $v_k = (m \times 1)$ vector - measurement noise - assumed to be white Gaussian with known covariance R.

- $P_k = (n \times n)$ Error covariance matrix for state variables.

The a priori estimates of a vector v are denoted as $\hat{v}_k^-$, where 'hat' denotes the estimate and the super-minus is a reminder that this is the best estimate prior to incorporating the knowledge at t_k.

The recursive form of Kalman filter in information mode is given by:

1. Enter loop with $(P_k^-)^{-1}$ and $\hat{x}_k^-$.
2. Compute P_k^{-1} from $P_k^{-1} = (P_k^-)^{-1} + H_k^T R_k^{-1} H_k$ and invert to get P_k.
3. Compute Kalman gain $K_k = P_k H_k^T R_k^{-1}$.
4. Update estimate: $\hat{x}_k = \hat{x}_k^- + K_k(z_k - H_k \hat{x}_k^-)$.
5. Project ahead: $\hat{x}_{k+1}^- = \varphi_k \hat{x}_k$ and $P_{k+1}^- = \varphi P_k \varphi_k^T + Q_k$ and invert to get P_{k+1}^-.
6. Go to step 2.

Now the Federated Kalman filter in informatiom mode operates as follows:

Local sensor filter 1:

$$P_1^{-1} = (P_1^-)^{-1} + H_1^T R_1^{-1} H_1$$

$$\hat{x}_1 = P_1((P_1^-)^{-1}\hat{x}_1^- + H_1^T R_1^{-1} z_1)$$

Local sensor filter 2:

$$P_2^{-1} = (P_2^-)^{-1} + H_2^T R_2^{-1} H_2$$

$$\hat{x}_2 = P_2((P_2^-)^{-1}\hat{x}_2^- + H_2^T R_2^{-1} z_2)$$

Local sensor filter N:

$$P_N^{-1} = (P_N^-)^{-1} + H_N^T R_N^{-1} H_N$$

$$\hat{x}_N = P_N((P_N^-)^{-1}\hat{x}_N^- + H_N^T R_N^{-1} z_N)$$

At the Master filter, the optimal global estimate and associated error covariance are calculated as follows:

$$P^{-1} = (P_1^{-1} - M_1^{-1}) + \ldots + (P_N^{-1} - M_N^{-1}) + (P^-)^{-1}$$

$$\hat{x} = P[(P_1^{-1}\hat{x}_1 - (P_1^-)^{-1}\hat{x}_1^-) + \ldots + (P_N^{-1}\hat{x}_N - (P_N^-)^{-1}\hat{x}_N^-) + (P^-)^{-1}\hat{x}^-]$$

This resultant error covariance will be used to calculate the total reinforcement given to the automata by the environment (explained later).

6.2.4 Proposed Solution

In a distributed object tracking system, cameras track an object moving in the environment and generate object position data. So for distributed object tracking system, we formulate the sensor selection problem as follows [54]: Given a set of sensors $C = \{C_1, C_2, \ldots, C_n\}$, determine the subset $C' \in C$ which optimizes an objective function consisting of a trade-off among accuracy and energy consumption. The objective function describes two conflicting goals: (1) to produce position data of high accuracy and (2) to conserve energy. This trade-off is usually modeled using the notions of utility and cost.

1. Utility: Accuracy of the information generated by each camera.

2. Energy Cost: the energy expended by the camera to send this information to the base station. Typically, this is directly proportional to the distance of the camera from the base station.

This sensor subset selection problem is NP-Hard [56]. This means that there is no solution that can run in polynomial time (in number of sensors). This is clearly not desirable for on-line implementation, especially in a network with large number of sensors. Hence, approximate and heuristic methods are necessary to solve this problem in tractable time. We propose a novel reinforcement learning approach (using

the decentralized pursuit learning game algorithm described earlier))to solve this problem.

The optimal sensor subset selection will be governed by an optimization function $\mathcal{F}^C(A)$ where C is the set of sensors and A is the set of action tuple selected by these sensors (explained later). The function $\mathcal{F}^C(A)$ is composed of two parts and is of the form:

$$\mathcal{F}^C(A) = W_{Error} * \mathcal{F}^C_{Error}(A) + W_{Energy} * \mathcal{F}^C_{Energy}(A) \tag{6.1}$$

$$\text{where } W_{Error} + W_{Energy} = 1$$

Here, W_{Error} specifies the importance associated with the error in object location and W_{Energy} specifies the importance associated with the energy used by the sensors to send this data to the central module for fusion. By specifying these weight values, one can specify the tradeoff associated. If $W_{Error} > W_{Energy}$, then accuracy of the data generated by the sensors will play a dominant role in selection whereas if $W_{Energy} > W_{Error}$, then sensors who spend less energy to send their data to the central module will be preferred. By tuning these weight parameters appropriately, the tradeoff between energy and error can be effectively expressed. The job of the reinforcement learning algorithm is then to learn the action tuple A_{opt} (i.e. to learn which sensors need to turn themselves on and which need to turn themselves off)so that the function $\mathcal{F}^C(A)$ attains its optimum value. We map this problem on the framework of game-playing learning automata so that the outcome of the game will represent the optimum configuration for the given value of weight parameters.

Each sensor (camera) in the system is represented by a learning automaton. A sensor can perform two actions and these actions represent the action set of the automaton. These actions are: *ON* and *OFF*. In our object tracking scenario, the sensors that can track the object at current instant form a team of automata and participate in the cooperative game to converge to a configuration which optimizes the

tradeoff function $\mathcal{F}^C$. During each trial, each automaton selects an action by sampling its action probability vector. The set of actions selected by all the automata forms an action tuple for current trial. Then based on the response from the environment, the automaton updates its action probability vector. We test the performance of L_{R-I}-Game Algorithm, Centralized Pursuit Game algorithm and Decentralized Pursuit Game algorithm in this scenario. When the automata selects the action *ON*, the corresponding sensor sends the object location data to the central module and hence pays the energy cost of sending the data. When the selected action is *OFF*, the corresponding sensor remains idle and doesn't send any data to the central module.

The central module is the Master filter in Federated Kalman filter configuration [55]. Each sensor runs its own local Kalman filter. The local Kalman filter runs in the in the Information mode [55, 61, 62]. This is an alternate form of Kalman filter which calculates value of information (inverse of error) associated with each sensor reading. The Master Filter combines these individual information matrices to produce resultant information matrix. The inverse of the resultant information matrix gives the resultant error covariance matrix P. Ideally, one would like to combine the data from all the sensors thus increasing the information and reducing the resultant error covariance matrix as much as possible. However, not all the data arrives at the Master filter with the same energy cost. The sensors closer to the filter will expend less energy in sending the object location data to the Master filter than one which is far away. Thus one may prefer a slightly less accurate data at the cost of saving energy or one may opt for more accuracy at the cost of spending more energy to collect this data at the Master filter. As mentioned earlier, this tradeoff is expressed by the values of weight parameters W_{Energy} and W_{Error}. As stated earlier, there will be a particular action tuple A_{opt} which will optimize the value of $\mathcal{F}^C$ and we use a reinforcement learning approach to deduce this configuration.

We calculate trace of the error covariance matrix ($trace(P)$) and use that value as the criteria for comparison. The less the $trace(P)$ value, less is the error covariance and thus better the object position data. As stated earlier, whenever a sensor sends

the object location data to the Master filter for fusion, it spends some energy for data transmission. Thus when the sensor is in *ON* mode and sends data to the Master filter, its energy expenditure will be tracked by environment. We use the model in [63] to model the energy expenditure of the sensor. According to this model, the energy expenditure is proportional to the square of the distance between the sensor and the central module. The energy expenditure will be combined along with the trace value of error covariance matrix to produce a common payoff to all the sensor automata. A Sigmoid function is used to map the energy expenditure and error covariance values to the $[0, 1]$ interval so that they can be used in the reinforcement learning framework.

Suppose $C(k) = \{C_1, C_2, \ldots, C_N\}$ be the set of sensors who can track the object at time instant k. Also assume that their distances from the base station at time instant k are $D = \{D_1, D_2, \ldots, D_N\}$. The corresponding location data produced by each sensor is given by $X(k) = \{X_1, X_2, \ldots, X_N\}$ and the associated information matrix set is $I(k) = \{I_1, I_2, \ldots, I_N\}$. The actions chosen by the sensors for the n^{th} trial of the learning algorithm are $A(k)(n) = \{A_1(n), A_2(n), \ldots, A_N(n)\}$. We associate a value of 1 with action *ON* and a value 0 with action *OFF*. Now we define $\mathcal{F}_{Energy}(A(k)(n))$ and $\mathcal{F}_{Error}(A(k)(n))$ as follows:

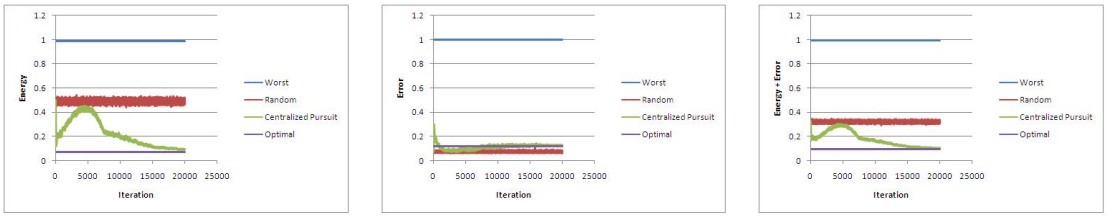
$$\mathcal{F}_{Energy}(A(k)(n)) = \mathfrak{S}(\sum_{i=1}^{N} A_i(n) \times D_i^2) \tag{6.2}$$

$$\mathcal{F}_{Error}(A(k)(n)) = \mathfrak{S}(Trace(Inverse(\sum_{i=1}^{N} A_i(n) \times I_i))) \tag{6.3}$$

where $\mathfrak{S}$ is the Sigmoid function and $Inverse(M)$ and $Trace(M)$ are functions that calculate the inverse and trace value of a matrix M respectively. Using the values of equations (6.2) and (6.3) in equation (6.1), we obtain the penalty probability associated with the action tuple $A(k)(n)$. Thus $(1-\mathcal{F}^C(A(k)(n)))$ gives us the reward associated with this action tuple. This is the entry we use in the game matrix which

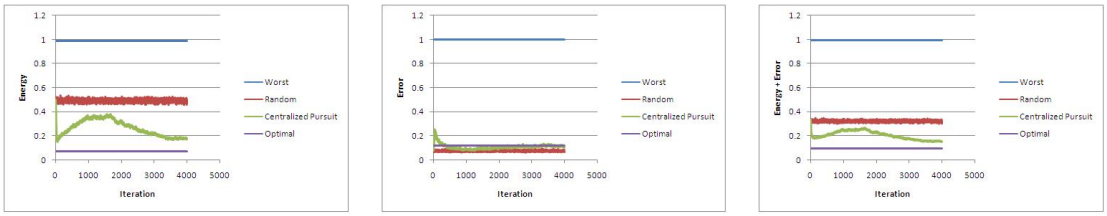
decides the probability of the reward returned to the sensors by the environment in the form of common payoff.

6.2.5 Results



(a) (b) (c)

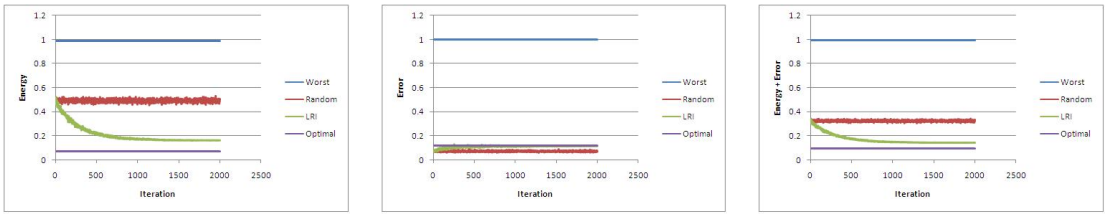
Figure 6.4. CPLA : Step Size = 0.05: (a) Energy (b) Error (c) Energy + Error



(a) (b) (c)

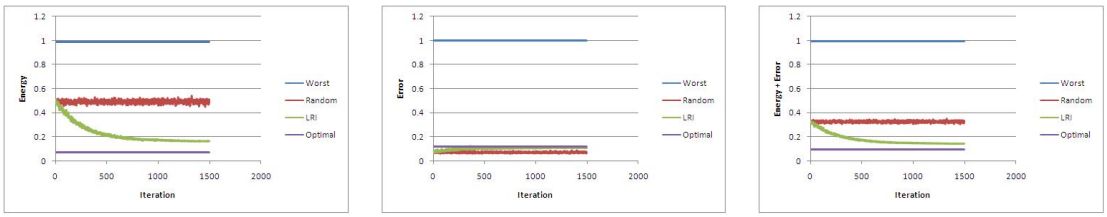
Figure 6.5. CPLA : Step Size = 0.09 (a) Energy (b) Error (c) Energy + Error

For analyzing the convergence properties of various learning algorithms in game scenario, we tested their performance for a simulated system consisting of 10 cameras tracking an object. Figures (6.4 - 6.9) show the results of our experiments. Each algorithm was tested against a random selection game algorithm in which each automata selects one action at random from its set of action and rest of the game algorithm remains same. This random selection game algorithm provides a benchmark to test other learning based game algorithms. In order to be deemed useful, these learning algorithms must perform significantly better than the random selection algorithm.



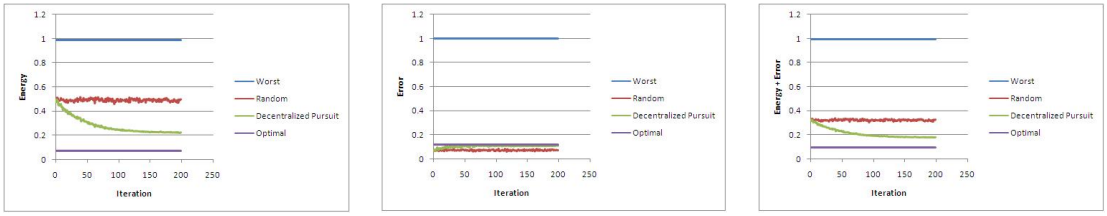
(a) (b) (c)

Figure 6.6. L_{R-I} Learning Game : Step Size = 0.05 (a) Energy (b) Error (c) Energy + Error



(a) (b) (c)

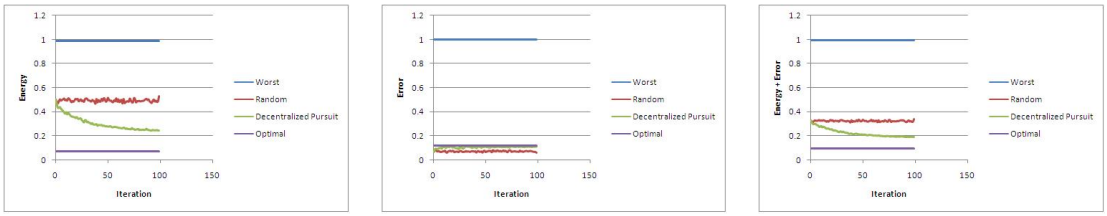
Figure 6.7. L_{R-I} Learning Game : Step Size = 0.09 (a) Energy (b) Error (c) Energy + Error



(a) (b) (c)

Figure 6.8. DPLA : Step Size = 0.05 (a) Energy (b) Error (c) Energy + Error

The figures show execution of the game algorithm until all the automata converge to one particular action from their action set. We use a value of 0.9 as the threshold for convergence. When one of the action probabilities in the action probability vector of an automaton reaches a value of 0.9, it is considered that automaton has converged



(a) (b) (c)

Figure 6.9. DPLA : Step Size = 0.09 (a) Energy (b) Error (c) Energy + Error

to that action. When all the automata in the team reach this state, then the entire team converges to the corresponding action tuple. Each algorithm was run 50 times and graphs show the average of the values obtained over these 50 runs. The graphs show the execution of various game algorithms till the team of 10 automata converges to the optimum value of the tradeoff function $\mathcal{F}^C(A)$.

As demonstrated by the graphs in Figures (6.4) and (6.9), the Centralized Pursuit algorithm causes the cameras to converge to the optimal action tuple but requires a large number of iterations. The number of iterations will increase with increase in number of cameras. Even at a modest number of 10 cameras, the centralized pursuit algorithm takes a long time for the automata team to converge which makes it unsuitable for an application like distributed object tracking where fast convergence is necessary.

Figures (6.6) and (6.7) show the performance of L_{R-I} game algorithm. The L_{R-I} game algorithm achieves much faster convergence than centralized pursuit algorithm but it doesn't always converge to the globally optimal action. Instead, it may converge to a Nash mode of the game matrix and we get a locally optimal solution as indicated by the graph. However, this solution is a reasonable approximation of globally optimal solution.

Figures (6.8) and (6.9) show the performance of the proposed Decentralized Pursuit Game algorithm. As the graphs indicate, by using Decentralized Pursuit Game algorithm, the automaton team converges to an optimal configuration much quicker

than the other two algorithms. The speed increase is almost ten times over L_{R-I} algorithm and almost hundred times over the Centralized Pursuit algorithm.

We also measured the total energy expenditure by the team of cameras to converge to optimal tuple value. This energy expenditure is represented by the area under the curve of function $\mathcal{F}_{Energy}(A(k)(n))$ ($\int \mathcal{F}_{Energy}(A(k)(n))\ d\mathcal{F}_{Energy}$). We calculated the average energy expenditure by automata team over a run of 50 experiments while using different game algorithms to reach convergence and different values for the step size (denoted by a). The algorithm which causes the least amount of energy to be used for convergence will be the preferred algorithm for resource constrained devices. In another set of experiments, we measured the average error incurred by the team of cameras to reach convergence. The algorithm which results in less average error has superior performance. As before, we calculated the average error of the automata team over a run of 50 experiments while using different game algorithms to reach convergence and different values for the step size. The results of these experiments are presented in the following table.

Table 6.1
Performance Comparison

	L_{R-I}		CPLA		DPLA	
	a=0.05	a=0.09	a=0.05	a=0.09	a=0.05	a=0.09
Total Energy	320	149	3772	1020	38	23
Average Error	0.1043	0.09783	0.1050	0.1011	0.0985	0.0986

As data in above table indicates, the DPLA consumes far less energy than other two algorithms. The DPLA algorithm shows an improvement by a factor of ten over L_{R-I} Game algorithm and an improvement by a factor of hundred over CPLA. Also, the average error in case of DPLA is least among all three algorithm. This shows that the DPLA outperforms L_{R-I} Game algorithm and CPLA under both total energy as well as average error criteria.

6.3 Designing a Distributed Wetland System in Watersheds

In this section, we will describe the application of identical-payoff games of learning automata as a framework to solve complex multi-criteria optimization problem of watershed management [64]. Multiple analytical criteria are used to assess design decisions for creating a distributed network of wetlands in the watershed. DPLA as well as a genetic algorithm based method are used for the analysis. Simulation studies are presented which compare the efficiency of the reinforcement learning approaches with a multi-objective genetic algorithm-based approach.

6.3.1 Problem Description

With the changing climate, impacts of flooding are expected to worsen in the coming years. For the United States, the latest climate change models have predicted a likely increase in duration of precipitation events during winter and spring months, increase in the intensity of precipitation, and greater evaporation during the summer, thereby, leading to periods of both floods and water deficits (Kundzewicz et al., 2007; Lettenmaier et al., 2008; Milly et al., 2008). To help mitigate the effects of increased flooding, the restoration of degraded upland and downstream storage capacities of watersheds has been proposed: storing the excess floodwater on the land during high precipitation events, rather than moving it rapidly off the land, could significantly reduce the amount of flood damages incurred further downstream, mitigate water quality impacts, and improve wildlife habitat. This work deals with the design of a system for upland (i.e. upstream regions of the watershed) storage, specifically by creating a network of wetlands for improving upland storage in the watershed. Design is complex because there are a large number of alternative inter-connected sites,thereby making it a complex combinatorial optimization problem. Additionally, there are multiple criteria for selection among alternatives.

Reinforcement learning provides a promising framework for solving complex combinatorial decision making problems. Previous researchers have applied many tech-

niques such as genetic algorithms and supervised neural networks to specific environmental problems. However, the use of reinforcement learning as an approach to solve complex watershed management problems has remained largely unexplored. We formulate the watershed management problem as an identical-payoff game of multiple reinforcement learning agents. Multiple criteria incorporating the effect of the decision variables on watershed hydrology and land use, e.g., reduction in peak stream flows and increase in baseflows (i.e. low flows), land area converted to wetlands etc. are suitably combined to generate a scalar binary-valued feedback for the learning agents. Experimental studies are conducted using different learning algorithms for the agents. These preliminary studies clearly indicate that the approach has the potential for determining high quality solutions. Further, though reinforcement learning is used for a deterministic case study in this work, it is naturally suited for stochastic environments. This increases its potential for applications such as watershed management where randomness arises naturally because of unpredictable nature of rainfall, hydrologic response of watersheds, anthropogenic drivers, etc.

6.3.2 Genetic Algorithms

Genetic Algorithms (GA) are heuristics-based optimization algorithms that emulate natural selection mechanism to perform the optimization task. GAs work with "strings" of decision variables mapped in binary space (also called "chromosomes"), and search from a population of possible designs ("individuals") using the information provided by the objective function ("fitness function"). GA uses three operators - reproduction, crossover, and mutation. These operators are used to evolve the population to solutions with higher fitness, until it converges to optimal or near-optimal solutions. Multiple types of genetic algorithms currently exist that optimize problems with one or multiple objectives. Commonly used multi-objective genetic algorithms (e.g., MOGA [65], NSGA II [66], NPGA [67], VEGA [68]) converge the population to a set of non-dominated solutions (i.e. a Pareto set). In this study, we compared the

reinforcement learning algorithms with the Non-dominated sorting genetic Algorithm (NSGA II), which has previously been widely tested for multiple surface water and ground water management problems [69–75].

6.3.3 Proposed Solution

The goal of this work is to develop a methodology for designing a large scale distributed wetland network system by identifying locations and sizes of wetlands in the watershed that could improve the overall storage of rainfall runoff and, thereby, decrease the intensity of peak flows and possible flooding. The design process was accomplished by posing the problem as a multi-objective, spatial optimization problem. The methodology was tested for Eagle Creek watershed located in Central Indiana, USA, about 10 miles northwest of downtown Indianapolis city (Figure (6.10)). Approximately 162 miles^2 of its drainage area drains into the Eagle Creek Reservoir, which is a major source of drinking water supply in Indianapolis, and is also used for flood control.

Using a Geographic Information Systems (GIS) based methodology, the 2008 land use-land cover, topography, and soil drainage characteristics were first analyzed to obtain all potential locations and scale of wetlands in the watershed. Figure (2) shows the multiple sub-basins in the watershed, and locations and sizes of potential wetland sites (see blue polygons in the zoomed section of the watershed). Based on this analysis, there are a total of 2953 wetland sites greater than 1000 m^2 wetland area, which could be potentially restored in the watershed. To assess the effect of these potential wetland sites on the watershed hydrology and stream flows, a distributed hydrologic model was also built for this watershed using the Soil and Water Assessment Tool (SWAT [76], [77]). Since SWAT models wetlands as water bodies within the sub-basins, and allows only one wetland per sub-basin, it was decided to aggregate the areas and volumes of all the 2953 potential wetlands into one large wetland per sub-basin. This resulted in 108 possible aggregated wetlands to choose, one in each of

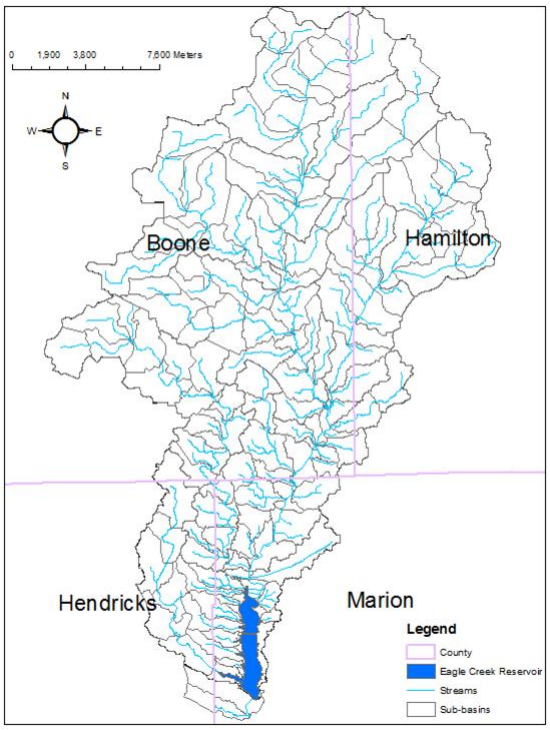


Figure 6.10. Eagle Creek Watershed and its counties, reservoir, streams and 130 sub-basins.

the 108 sub-basins that contained all the 2953 smaller wetlands. However, choosing an optimal subset of aggregated wetlands from 108 aggregated wetlands also poses computational hurdles because of the large design space. For example, if the problem was posed as a binary problem of restoring or not restoring aggregated wetlands in each of the 108 sub-basins, the design space would consist of 2^{108} alternatives.

Therefore, it was decided to divide the watershed basin into 8 regions (Figure (6.11)) and run multiple optimization experiments separately for each region. Such a division of regions assumes independence in overall performance of wetlands in any two regions, which is not entirely true if the regions are hydrologically connected. However, it provides a possible practical solution to working with large spatial optimization problems by constructing multiple smaller spatial optimization problems. The optimization problem was then converted into a binary decision problem of selecting or not selecting a sub-basin for restoring the corresponding aggregated wetlands.

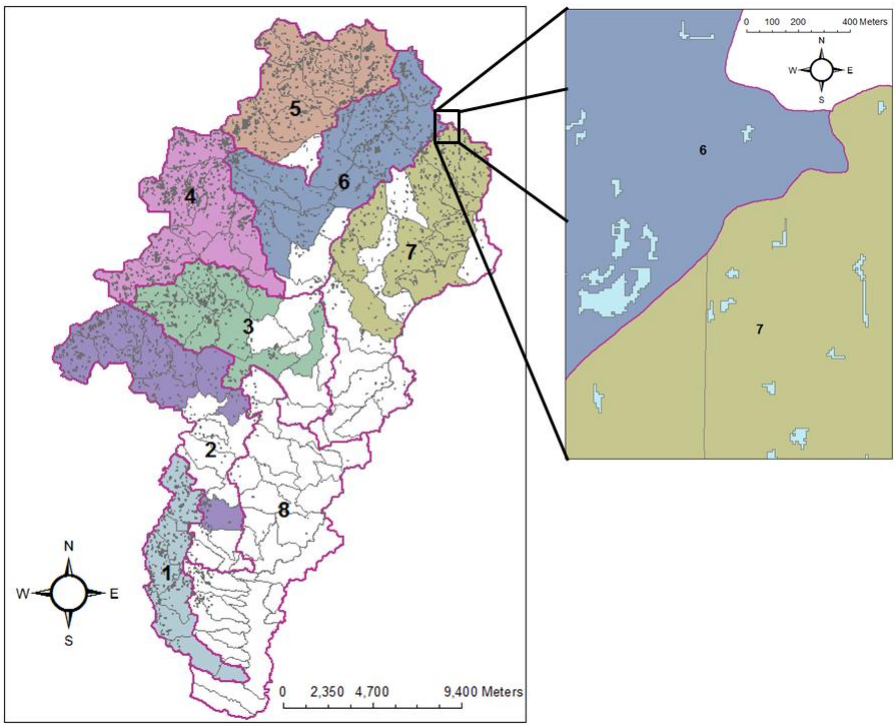


Figure 6.11. Left figure shows the 130 sub-basins and 2953 potential wetland polygons in the 8 regions (pink polygons) divided for optimization. Right figure shows the enlarged view of potential wetlands (blue polygons) in the watershed area surrounded by black box in left figure.

For each subbasin j in the region i, we associate a flag decision variable ($flag_j^i = 0/1$). If the $flag_j^i$ is 1 then the subbasin j in the region i has aggregated wetlands restored. Since some of the regions had many sub-basins with wetlands, it was decided to select the top 10 aggregated wetlands in every sub-basin in order to reduce computational complexity of the search process. This, therefore, reduced the number of decision variables for the binary decision problem to be equal to or lesser than 10 for each region, and made the search space lesser than 2^{10} alternatives for each region. The color-shaded sub-basins in the left figure of Figure (2) indicate the top 10 sub-basins with aggregated wetlands selected for optimization in every region. The top 10 sub-basins were selected based on a sensitivity analysis in which one by one we tested the effect of every aggregated wetland in a sub-basin on the overall reduction in flows in

all the streams in the region. The aggregated wetlands were then ranked from high to low based on their decreasing order of overall flow reductions estimated by the SWAT hydrologic model, and the corresponding sub-basins (i.e. the color-shaded sub-basins in Figure (2)) with aggregated wetlands were selected for optimization based on their ranks. The sensitivity analysis provided a useful heuristic to prioritizing which sub-basins with aggregated wetlands would be most useful for optimization. However, it also assumes the independence in wetland performance between high-ranking and low-ranking sub-basins; this is a reasonable assumption for the purpose of a large scale optimization.

The multiple, conflicting, quantitative objectives chosen for this problem were to minimize the total area used by wetlands for each of the regions and to minimize the root mean-square error between flows in streams when all the region's wetlands were restored/installed (i.e. the baseline conditions) and the flows estimated with only a particular subset of wetlands selected during the optimization process. It is clear that if we assign one learning automaton for each sub-basin in the region to decide whether it should have its aggregated wetland installed or not, the problem reduces to an identical-payoff game model of learning automata with multiple criteria corresponding to the multiple objectives. For each sub-basin flag_j^i, we associate an automaton $\mathcal{A}_j^i$ with two actions $\text{flag}_j^i = 0$ and $\text{flag}_j^i = 1$. During each iteration of the DPL algorithm, each automaton selects an action by sampling its action probability vector and based on the action selected, the flag_j^i decision variable is set to either 0 (potential aggregated wetland should not be installed) or 1 (potential aggregated wetland should be installed).

We also used two scaling parameters, namely S_{Area}^i and S_{Flow}^i, associated with a region i, to scale the values for the area and flow objectives, respectively, to lie between 0 and 1. S_{Area}^i is calculated by adding the areas of all the potential aggregated wetlands in a particular region (i.e. those in the top color-shaded sub-basins in Figure (6.11)).

$$S_{Area}^i = \sum_j \text{Area}_j^i \tag{6.4}$$

The S^i_{Flow} is calculated in two steps:

1. We first generate a baseline flow dataset (Baseline*) by installing all the top ranked wetlands in region i and running SWAT model for this configuration. This represents the best possible scenario offering the most reduction in the volume of water in the streams. Then we generate the output flow dataset (Output$^*_{\text{noWetlands}}$) by running the SWAT model for the configuration in which there are no wetlands in the region. This represents the worst possible scenario offering the least possible reduction in the volume of water.

$$S^i_{Flow} = \sum_{region} \ln(1 + [\text{Baseline}^* - \text{Output}^*_{\text{noWetlands}}\,]^2) \tag{6.5}$$

Using these scaling parameters we calculate the payoff values for area (P^i_{Area}) and flow (P^i_{Flow}) for a particular set of decision variables for the optimization algorithms (i.e., set of actions chosen by all the learning automata agents, or set of values of the genes in the genetic algorithm's choromosome) as follows:

$$P^i_{Flow} = 1 - \sum_{region} \frac{\log(1 + [\text{Baseline}^* - \text{Output}_{\text{subsetOfWetlands}}\,]^2)}{S^i_{Flow}} \tag{6.6}$$

$$P^i_{Area} = 1 - \frac{\sum_{region} (\text{flag}^i_j \times \text{Area}^i_j)}{S^i_{Area}} \tag{6.7}$$

where,

$$\text{flag}^i_j = \begin{cases} 1 & \text{if potential aggregated wetland } j \\ & \text{in region } i \text{ is installed} \\ 0 & \text{otherwise} \end{cases} \tag{6.8}$$

The total payoff is then calculated by combining both area and flow payoffs into one single criteria. This is done by weighting each payoff with a corresponding weight that have a real value between 0 and 1. Weights for all the criteria sum up to a total of 1.0.

$$P^i_{Total} = W_{Area} \times P^i_{Area} + W_{Flow} \times P^i_{Flow} \tag{6.9}$$

and

$$W_{Area} + W_{Flow} = 1 \tag{6.10}$$

The genetic algorithm, however, does not use any weights to combine objective functions into a single objective function. The multi-objective genetic algorithm (NSGA II) used in this study allows simultaneous exploration of multiple objective functions to directly create a non-dominated set of solutions.

6.3.4 Results

Using the above methodology, the Decentralized Pursuit Learning Algorithm (DPLA) and Non-dominated Sorting Genetic Algorithm (NSGA II) were implemented for each of the eight regions in the watershed. The Pareto-fronts generated by these algorithms were then compared with each other for performance measurement. Due to space constraints, we will discuss Pareto-fronts for only two regions. These regions were chosen based on how closely the Pareto-fronts of DPLA and NSGA II match with each other. As demonstrated by Figure (6.12), the DPLA and NSGA II Pareto-fronts of region 1 match closely each other. On the other hand, Figure (6.13) shows the mismatch in the Pareto-fronts for region 2.

Table (6.2) indicates the performance comparison of DPLA and NSGA II for region 1. It was observed that DPLA requires less iterations to obtain all the optimized alternatives/solutions. Also the converged action tuples (or, in other words, the decision variables) of DPLA and NSGA II are almost identical to each other. For example, as indicated in the last row of Table (6.2), the converged DPLA and NSGA II action tuples that were most similar to each other in objective space (measured via a Euclidean distance of objective function values) differed in decision space by just one aggregated wetland. DPLA tuple proposed installation of aggregated wetland in subbasin 127, while NSGA II proposed it to be uninstalled. However, as indicated

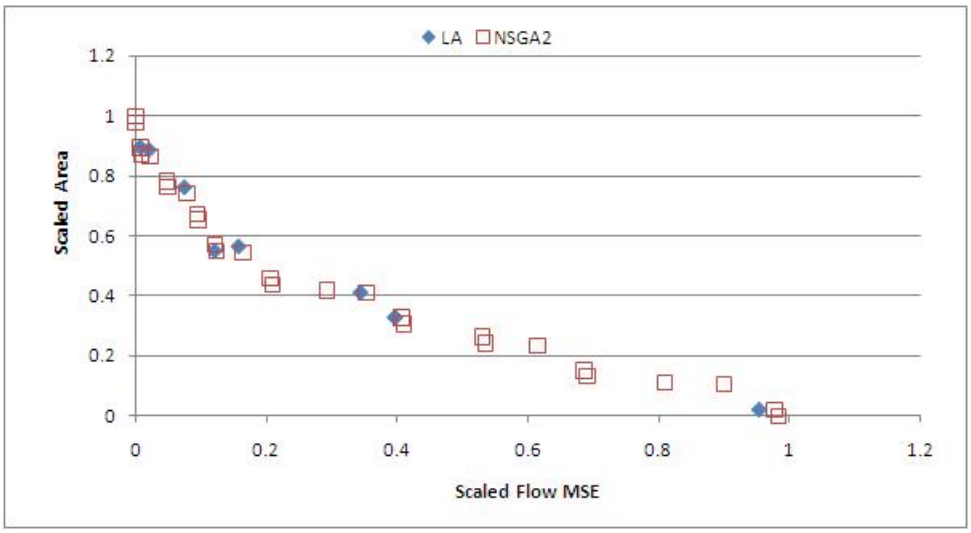


Figure 6.12. Region 1 Pareto-fronts

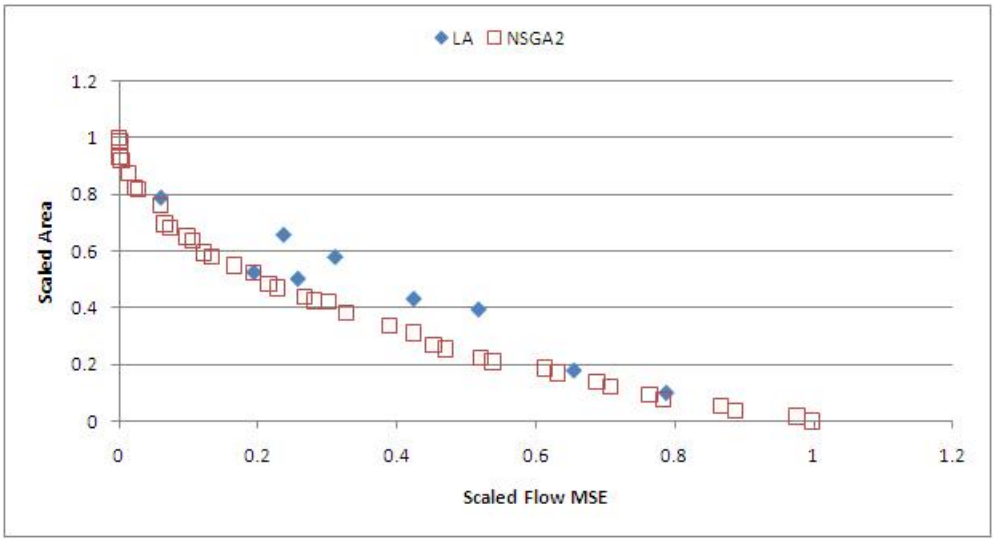


Figure 6.13. Region 2 Pareto-fronts

by the thick blue circle on the map in Figure (6.14), this aggregated wetland consists of only one very small wetland polygon with insignificant overall impact on flow. Thus, the solutions provided by DPLA and NSGA II are almost identical in the first approximation.

Table 6.2
Region 1

	DPLA	GA
Iterations	924	3600
Number of Converged Solutions	9	28
Number of Iterations per Solution	102	128
Most Similar Converged Tuples/Designs	1111011	1111010

The final set of solutions found for region 2 are not only dissimilar in objective space, but also decision space. The Pareto front created by converged tuples found by DPLA consisted mostly of inferior solutions compared to solutions found by the NSGA II. At first glance, this could be attributed to the fewer total number of iterations used by the DPLA (Table (6.3)). However, the NSGA II found a lot more distinct non-dominated solutions, thereby, having a smaller computational load (i.e. number of iterations) per solutions in the converged set. This indicates that the complexity in the design space of region 2 posed additional computational challenges to the algorithms' performance. For example, when two of the DPLA and NSGA II solutions with similar flow payoffs were compared, their tuples (Table (6.3) and Figures (6.15) and (6.16)) were dissimilar in 6 of the 10 sub-basins containing the aggregated wetlands. Whereas, when two of the converged DPLA and NSGA II solutions with similar area payoffs were compared (Table (6.3) and Figures (6.15) and (6.16)), the tuples were dissimilar in only 2 of the 10 sub-basins. This indicates that multiple spatial combinations of aggregated wetlands is possible in order to get similar flow benefits, however, fewer spatial alternatives with similar area are present in the decision space.

Additional experiments were also performed to compare the differences in converged tuples/solutions when the optimization was performed using all the high ranked aggregated wetlands in each of the regions identified in the previous section, without doing a separate optimization for individual regions as done in the previous section. This optimization resulted in a search space that had a total of 67 sub-basins

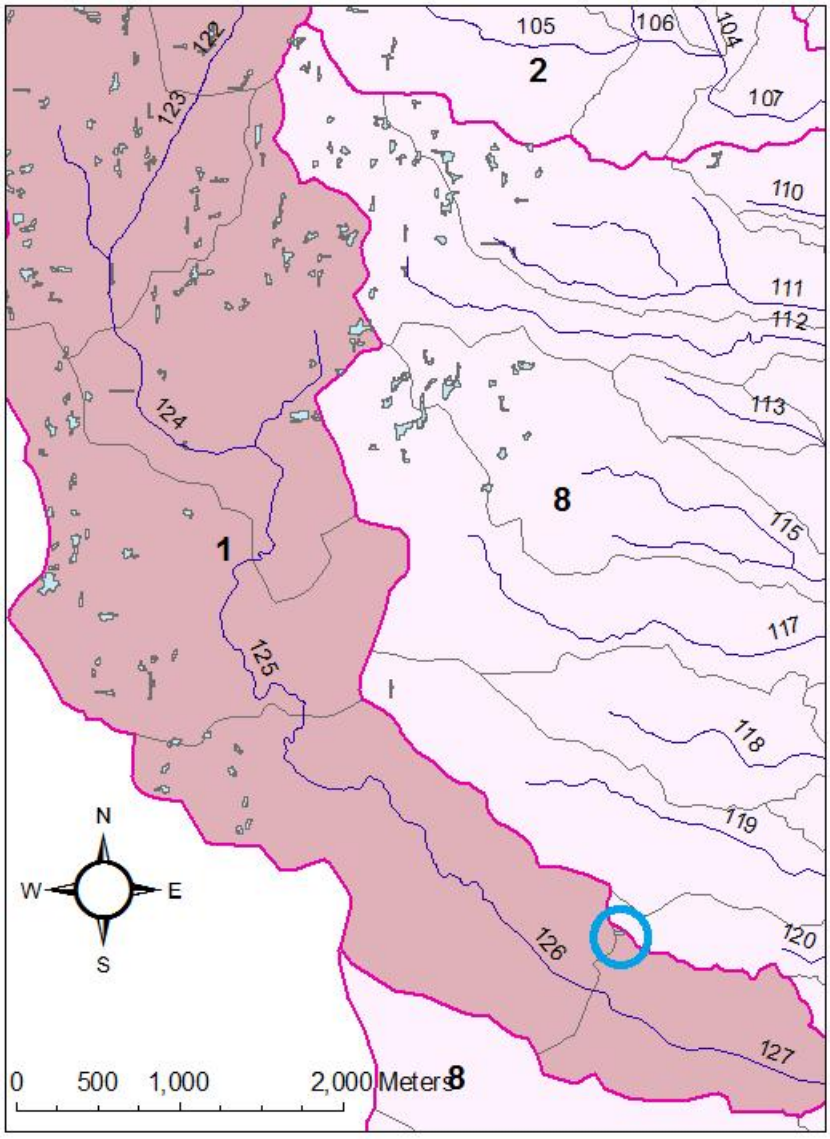


Figure 6.14. Region 1 Map

(i.e. all the colored sub-basins in Figure (2)) with aggregated wetlands to choose from. Hence, this experiment also allowed the exploration of dependencies between multiple regions on the overall optimization performance of all candidate aggregated wetlands in the watershed. Table (6.4) indicates the computational efficiency of these experiments using DPLA and NSGA II, which throw light on NSGA-II's lower computational load per solution.

Table 6.3
Region 2

	Learning Algorithm	Genetic Algorithm
Iterations	1092	3600
Number of Converged Solutions	9	38
Number of Iterations per Solution	121	94
Similar Converged Tuples/Designs (Similar Flow Payoff)	0011010100	0100111000
Similar Converged Tuples/Designs (Similar Area Payoff)	0100000100	0100010000

Table 6.4
All Regions

	Learning Algorithm	Genetic Algorithm
Iterations	2060	3600
Number of Converged Solutions	9	48
Number of Iterations per Solution	228	75

Converged solutions found by DPLA and NSGA II were then compared for all different sets of weights. Figure (6.17) indicates that in the objective function space, the solutions found by DPLA were inferior to the solutions found by the NSGA II. The solutions were then compared to each other in decision space, based on the specific set of weights. For example, Figures (6.18) and (6.19) show the spatial distribution of aggregated wetlands found by DPLA and NSGA II if a weight of 0.5 was chosen for both flow and area objective functions. DPLA found a more well-distributed set of aggregated wetlands over the entire watershed, whereas, the NSGA II found a solution with better overall flow and area payoff by clustering aggregated wetlands mostly in regions 5, 6, and 7. Though the overall impact of NSGA II solution on the flow and area objectives is better than the DPLA solution, but the DPLA solution provides options for utilizing land in other regions (e.g. regions 1, 2, and 3). The NSGA II solution, therefore, could add social constraints to the land manager and land owner's management plan if they have to convince more owners in a small region to convert their land area to wetlands. This would add an additional level of

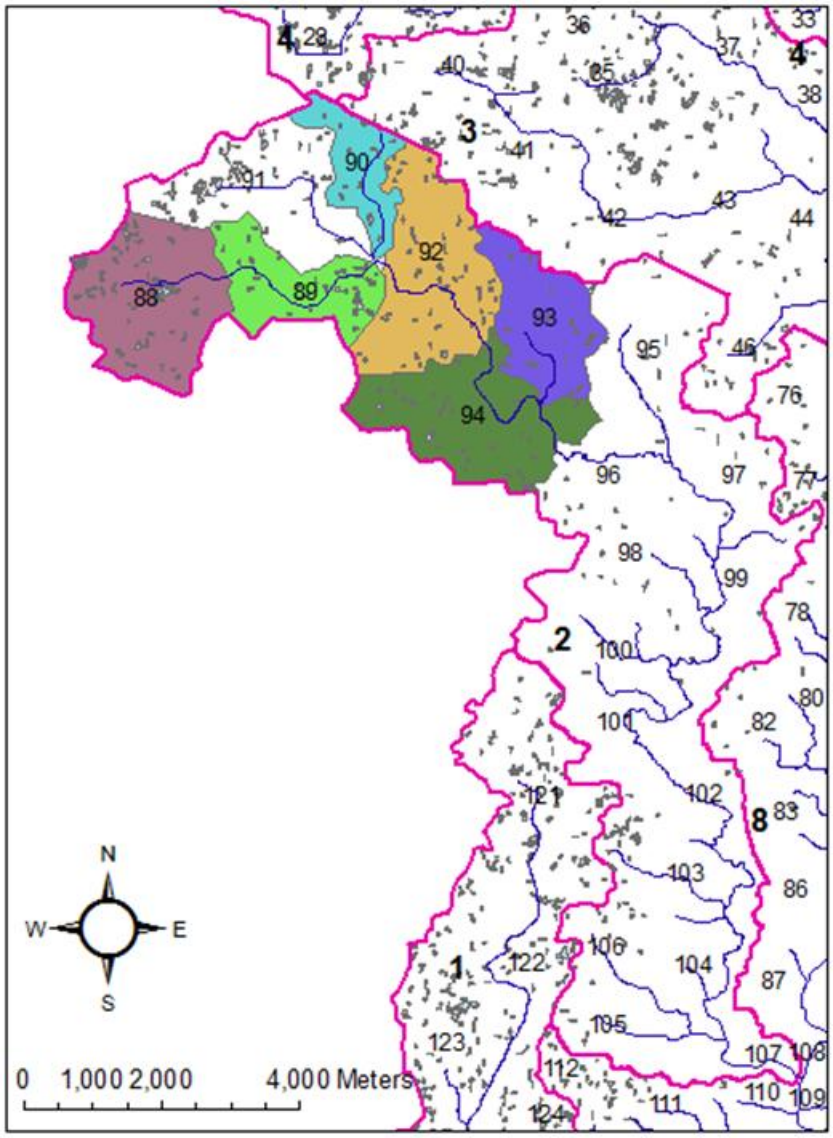


Figure 6.15. Solutions with similar flow payoffs found by DPLA and NSGA-II disagreed with each other on the aggregated wetlands in the colored sub-basins of region 2.

uncertainty and complexity to the optimization, since "human factors" would also need to be considered for assessing the overall quality of these optimized solutions.

The solution in Figures (6.18) and (6.19) were also compared to the spatial distribution of optimized aggregated wetlands found in the previous section for region-wise optimization, and if optimization was performed for the same values of weights (i.e.

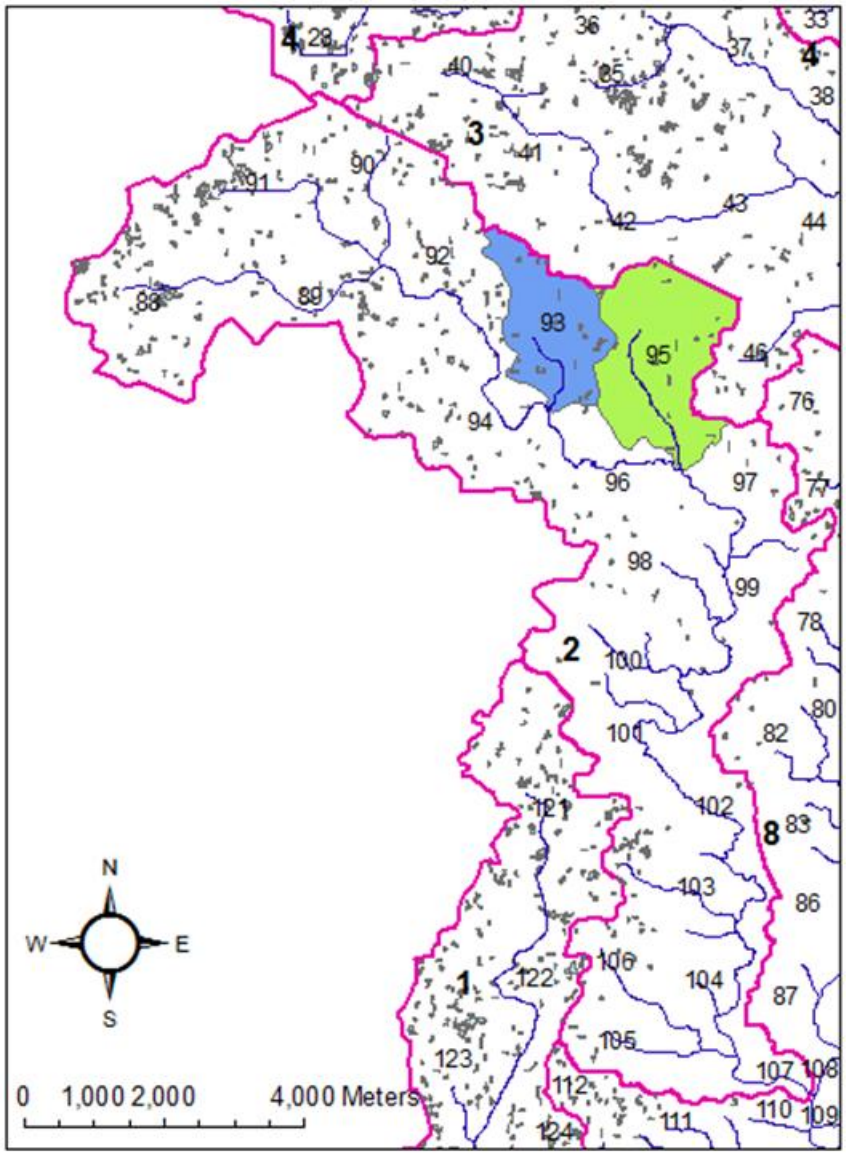


Figure 6.16. Solutions with similar area found by DPLA and NSGA-II disagreed with each other on the aggregated wetlands in the colored sub-basins of region 2.

0.5). Similarities and dissimilarities were observed in solutions found by the two approaches. For example, in region 1 the region-wise optimization solution found by DPLA differed from the solution found by optimizing all the 67 wetlands together solution in three aggregated wetlands in sub-basins with IDs 121, 123, 124. On the

other hand, for region 2, the region-wise optimization solution differed from the all 67 wetlands solution in sub-basins with IDs 97, 95, 94, 88, 89.

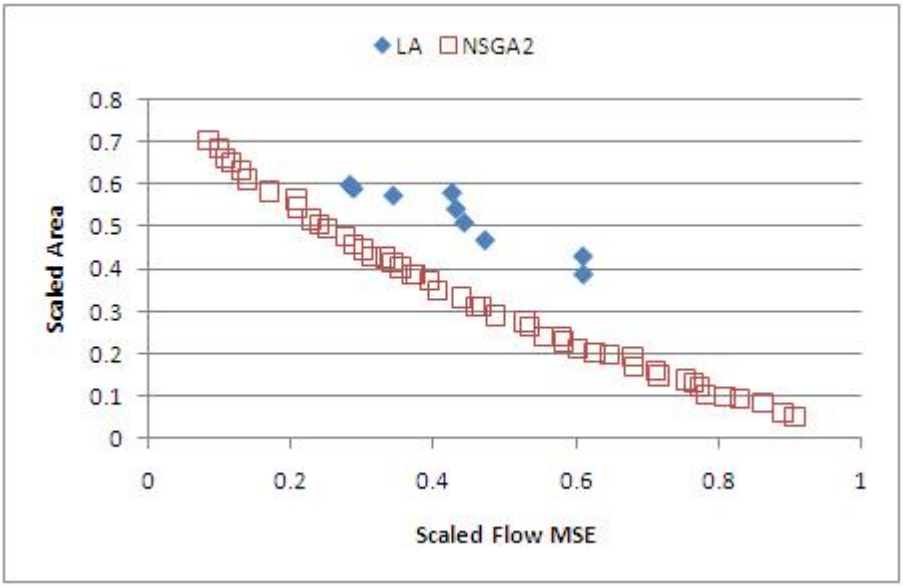


Figure 6.17. All Regions Pareto-fronts

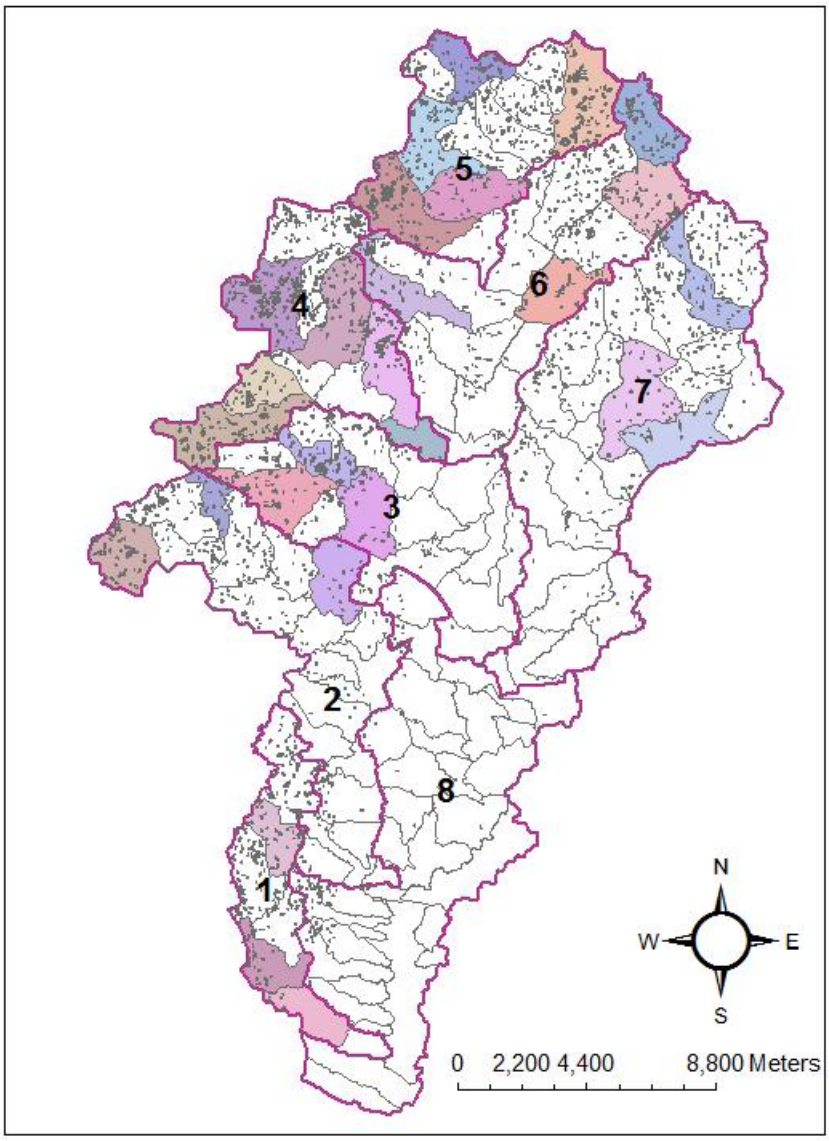


Figure 6.18. All Regions Map for DPLA Solution

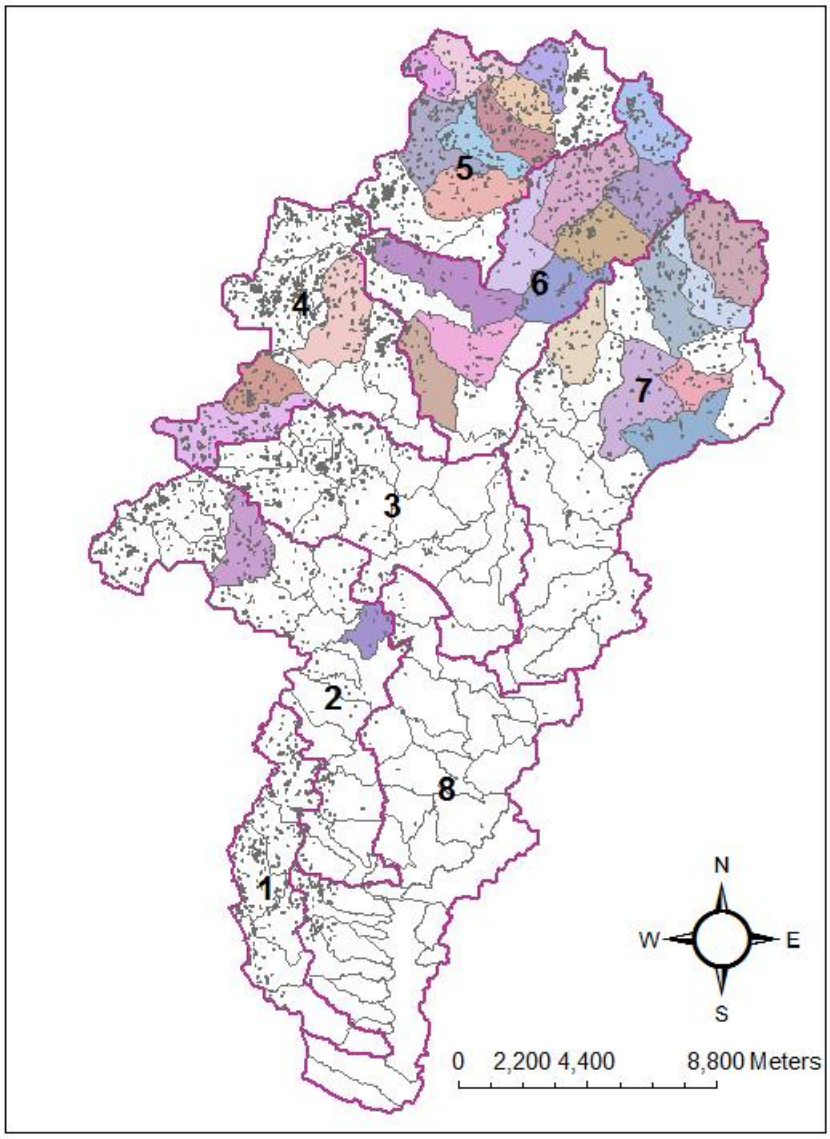


Figure 6.19. All Regions Map for NSGA II Solution

7 CONCLUSION AND FUTURE WORK

We will end this thesis by presenting the conclusions of this research and by pointing out some areas for future exploration.

7.1 Conclusions

1. MARL systems are ubiquitous. However, so far, the application of learning automata in the MARL context was limited because of the centralized nature of CPLA algorithm and slow convergence of L_{R-I} game algorithm. In this thesis, we proposed the DPLA algorithm which provides fast convergence in a decentralized manner. DPLA is an attractive candidate for applications in MARL systems and its performance is comparable or better than its counterparts.

2. PDGLA has the potential to provide a better payoff than the corresponding DPLA configuration. Slightly extra communication overhead incurred by the PDGLA can be often justified by the possibility of obtaining a better solution.

3. Various real-world combinatorial optimization problems can be modeled as the identical-payoff games of learning automata. DPLA promises to perform better than CPLA in such scenarios. The application studies presented in this chapter buttress this argument.

4. The HEGLA framework further improves the expressive power of the PDGLA by combining identical-payoff games and zero-sum games under one framework. This allows learning automata (or automaton) to participate in zero-sum as well as identical-payoff games. An automaton (or automata) can participate in both types of games at the same time. It is also possible automata can

form subgroups and each subgroup can be involved in one type of game while automata in the other group can be involved in other type of game.

7.2 Future Work

While the development of DPLA, PDGLA and HEGLA has made the application of learning automata for MARL systems feasible and affordable, there are still a number of interesting open problems to be solved in the area of the games of learning automata. Some possible future work in this area includes:

1. **Effects of Decentralization -** The CPLA converges to the globally optimal policy tuple in the game matrix. Even if the game matrix has multiple Nash equilibria, the centralization of the environment parameter estimates leads to the convergence to the Nash equilibrium point with the highest value (and thus the globally optimal action tuple). The DPLA, on the other hand converges to one of the Nash equilibria in the game matrix. Similarly, the decentralized L_{R-I} game algorithm converges to one of the Nash equilibria. This leads to an important question: What is the effect of decentralization/centralization on the behavior of the learning algorithms in the case of learning automata? One possible research direction is to create a formal framework for the interaction of learning automata in a game-like setting. This framework will be able to abstract the effects of different types of learning algorithms (model-free algorithms and model-based algorithms) and study the automata interaction in an algorithm-agnostic manner. It will be interesting to view the automata interaction from an information-theoretic point of view and explore the consequence of sharing partial information in the form of a distributed algorithm. One major contribution of such framework will be to prove that the decentralized configuration will always converge to one of the Nash equilibria of the underlying game matrix no matter the type of algorithm used for learning. Such theoretical framework will be a major step forward in the field of RL. So far, no

analytical framework studies different types of learning algorithm and different modes of communication (centralized vs. decentralized) in a unified manner. Indeed, even a negative result has not been proven yet. In particular, it has not been shown that a decentralized algorithm can never converge to the global maxima under any circumstances. Such proof will unify currently disparate fields of model-free and model-based algorithms and give a comprehensive and unified theory under which these algorithm can be studied.

2. **Rapidly Changing Environment -** It will be interesting to design and analyze algorithm for learning automata operating in rapidly changing environments. Such environments are characterized by rapidly or constantly changing reward values. DPLA analysis involves automata operating in an environment which is highly dynamic. This make the theoretical analysis of DPLA a very challenging task. New stochastic analysis tools are required to analyze the behavior of automata in such chaotic environments. Creation of new methodologies or application of existing techniques towards the analysis of such algorithms will open up a new area in the field of reinforcement learning using learning automata.

3. **Optimal Partial Decentralization -** PDGLA promises to alleviate the problem of complete centralization by allowing only a subset of learning automata to communicate with each other. Also, one can explore this design space to find partial communication configurations whose payoff is larger than that of the completely decentralized DPLA. The cost of slightly extra communication overhead can be justified by the better better quality of the solution. However, one needs to explore the entire design space to find out the PDGLA configurations that produce better outcomes that DPLA. It will be worthwhile to develop an algorithm which finds such better configurations. Such algorithm will also help in creating a comprehensive formal theoretical framework required to analyze the behavior of PDGLA for different configurations and a variety of

different learning algorithms. Another interesting option to consider is to allow partial communication within each individual state of the Markov chain. This will make the corresponding game even more decentralized. If all the automata within a state communicate with each other, then the corresponding game matrix has a unique equilibrium point. If we allow only some automata within a state to communicate with each other, then such formulation may also produce game matrix with a unique equilibrium point. As we described in the thesis, the control of finite, multi-agent Markov chains can be achieved by modeling it as a game of learning automata. However, translation in reverse direction gives us solution for the partial decentralization of learning automata games. Each multi-agent Markov chain problem generates a corresponding game matrix. Thus given a game matrix, we can translate it to the corresponding multi-agent Markov chain. Then if we allow the agents that reside in the same state to communicate with each other, the corresponding partially decentralized game formulation will converge to the globally optimal tuple. Based on autonomy, memory and communication constraints; this communication can be modeled as either a Superautomaton or a Master-Slave configuration.

LIST OF REFERENCES

LIST OF REFERENCES

[1] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[2] K. S. Narendra and M. A. L. Thathachar. *Learning Automata: An Introduction*. Prentice-Hall, 1989.

[3] M. A. L. Thathachar and P. S. Sastry. Varieties of learning automata: An overview. *IEEE SMC*, 32:711–722, 2002.

[4] M.A.L. Thathachar and P. Sastry. A new approach to the design of reinforcement schemes for learning automata. *IEEE Trans. on Systems, Man & Cybernetics, vol. 15, no. 1*, 1985.

[5] J. Crandall and M. Goodrich. Learning to compete, coordinate, and cooperate in repeated games using reinforcement learning. *Machine Learning*, 82:281–314, 2010.

[6] S. Talukdar, L. Baerentzen, A. Gove, and P. De Souza. Asynchronous teams: Cooperation schemes for autonomous agents. *Journal of Heuristics*, 4:295–321, 1998.

[7] M. Dorigo, M. Birattari, and T. Sttzle. Ant colony optimization– artificial ants as a computational intelligence technique. Technical report, IEEE Computational Intelligence Magazine, 2006.

[8] M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetic-sPart B*, 26:29 – 41, 1996.

[9] K. Verbeeck and A. Nowe. Colonies of learning automata. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 32:772 – 780, 2002.

[10] R. M. Wheeler and K. S. Narendra. Decentralized Learning in Finite Markov Chains. *IEEE Trans. on Automatic Control, vol. 31*, pages 519 – 526, 1986.

[11] M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. *Eleventh International Conference on Machine Learning*, pages 157–163, 1994.

[12] M. L. Littman. Value-function reinforcement learning in markov games. *Journal of Cognitive Systems Research*, 2:55 – 66, 2001.

[13] M. Lauer and M. Riedmiller. An algorithm for distributed reinforcement learning in cooperative multi-agent systems. *Proceedings of Seventeenth International Conference on Machine Learning (ICML-00)*, pages 535 – 542, 2000.

[14] C. Guestrin, M.G. Lagoudakis, and R. Parr. Coordinated reinforcement learning. *Proceedings Nineteenth International Conference on Machine Learning (ICML-02)*, pages 227 – 234, 2002.

[15] J. R. Kok and N. Vlassis. Using the max-plus algorithm for multiagent decision making in coordination graphs. *Robot Soccer World Cup IX (RoboCup 2005)*, 4020, 2005.

[16] Narendra, K. S., Wright, E. A., and Mason, L. G. . Application of Learning Automata to Telephone Traffic Routing and Control . *IEEE Trans. on Systems, Man & Cybernetics, vol. 7, no. 11*, pages 785–792, 1977.

[17] Lam, W. and Mukhopadhyay, S. A Two-Level Approach to Learning in Nonstationary Environments. In *Proceedings of the 11th Biennial Canadian Conference on AI*, pages 271–283, 1996.

[18] Barto, A. G. and Anandan, P. . Pattern recognizing stochastic learning automata. *IEEE Trans. on Systems, Man, and Cybernetis, vol. 15*, pages 360–375, 1985.

[19] Mukhopadhyay, S. and Thathachar, M. A. L. Associative Learning of Boolean Functions. *IEEE Trans. on Systems, Man, and Cybernetics, vol. 19*, pages 1008–1015, 1989.

[20] Mandayam A. L. Thathachar and P. S. Sastry. Learning optimal discriminant functions through a cooperative game of automata. *IEEE SMC*, 17(1):73–85, Jan 1987.

[21] W. Zhong, Y. Xu, and M. Tao. Precoding strategy selection for cognitive mimo multiple access channels using learning automata. *2010 IEEE International Conference on Communications (ICC)*, pages 23–27, 2010.

[22] S. Misra, V. Tiwari, and M. Obaidat. Lacas: learning automata-based congestion avoidance scheme for healthcare wireless sensor networks. *IEEE Journal on Selected Areas in Communications*, 27:466–479, 2009.

[23] T. Tuan, L. Tong, and A. Premkumar. An adaptive learning automata algorithm for channel selection in cognitive radio network. *2010 International Conference on Communications and Mobile Computing (CMC)*, 2:159–163, 2010.

[24] B. John Oommen and M. Khaled Hashem. Modeling a student's behavior in a tutorial-like system using learning automata. *Trans. Sys. Man Cyber. Part B*, 40:481–492, 2010.

[25] J. Torkestania and M. Meybodi. Clustering the wireless ad hoc networks: A distributed learning automata approach. *Journal of Parallel and Distributed Computing*, 70:394–405, 2010.

[26] M. Kashki, M. Abido, and Y. Abdel-Magid. Pole placement approach for robust optimum design of pss and tcsc-based stabilizers using reinforcement learning automata. *Electrical Engineering*, pages 383–394, 2010.

[27] J. Torkestani and M. Meybodi. An intelligent backbone formation algorithm for wireless ad hoc networks based on distributed learning automata. *Computer Networks*, 54:826–843, 2010.

[28] L. Lixia, H. Gang, X. Ming, and P. Yuxing. Learning automata based spectrum allocation in cognitive networks. *IEEE International Conference on Wireless Communications, Networking and Information Security (WCNIS)*, pages 503–508, 2010.

[29] B. J. Oommen and D. C. Y. Ma. Deterministic learning automata solutions to the equipartitioning problem. *IEEE JC*, 37(1):2–13, Jan 1988.

[30] B. J. Oommen, R. S. Valiveti, and J. R. Zgierski. An adaptive learning solution to the keyboard optimization problem. *IEEE SMC*, 21(6):1608–1618, Nov 1991.

[31] B. J. Oommen and E. V. de St. Croix. Graph partitioning using learning automata. *IEEE JC*, 45(2):195–208, Feb 1996.

[32] B. J. Oommen and T. D. Roberts. Continuous learning automata solutions to the capacity assignment problem. *IEEE JC*, 49(6):608–620, Jun 2000.

[33] O. Tilak, R. Martin, and S. Mukhopadhyay. A decentralized indirect method for learning automata games. *IEEE Systems, Man., and Cybernetics B*, Accepted and In Print., 2011.

[34] J. Von Neumann and O. Morgenstern. *Theory of Games and Economic Behaviour*. Princeton Univ. Press, 1944.

[35] K. Rajaraman and P. Sastry. Finite time analysis of the pursuit algorithm for learning automata. *IEEE SMC*, 26:590–598, 1996.

[36] David Freedman. Another note on the Borel-Cantelli lemma and the strong law, with the Poisson approximation as a by-product. *Ann. Probability*, 1:910–925, 1973.

[37] Leo Breiman. *Probability*. Addison-Wesley Publishing Company, 1968.

[38] J. L. Doob. *Stochastic Processes*. Wiley, 1973.

[39] Kazuoki Azuma. Weighted sums of certain dependent random variables. *Tôhoku Math. J. (2)*, 19:357–367, 1967.

[40] O. Tilak and S. Mukhopadhyay. Partially decentralized reinforcement learning in finite, multi-agent markov chains. *AI Communications (Accepted For Publication)*, 2011.

[41] J. M. Vidal and P. Buhler. A generic agent architecture for multiagent systems. Technical report, 2001.

[42] J. M. Vidal, P. Buhler, and H. Goradia. The past and future of multiagent systems. *AAMAS Workshop on Teaching Multi-Agent Systems*, 2004.

[43] http://www.cs.cmu.edu/ softagents/multi.html.

[44] R. A. Howard. *Dynamic Programming and Markov Processes*. M.I.T. Press, Cambridge, MA, 1960.

[45] P. S. Sastry, V. V. Phansalkar, and M. A. L. Thathachar. Decentralized learning of nash equilibria in multi-person stochastic games with incomplete information. *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS*, 24:769–777, 1994.

[46] O. Tilak and S. Mukhopadhyay. Multi agent reinforcement learning for dynamic zero-sum games. *Under Preperation*, 2011.

[47] L. S. Shapley. Stochastic games. *Proc Natl Acad Sci*, 39:1095–1100, 1953.

[48] S. Lakshmivarahan and K. S. Narendra. Learning algorithms for two-person zero-sum stochastic games with incomplete information: A unified approach. *Control and Optimization*, 20:541–552, 1982.

[49] O. Tilak and S. Mukhopadhyay. Decentralized and partially decentralized reinforcement learning for distributed combinatorial optimization problems. *Ninth International Conference on Machine Learning and Applications (ICMLA)*, pages 389 – 394, 2010.

[50] M. A. Perillo and W. B. Heinzelman. Optimal sensor management under energy and reliability constraints. In *Proc. IEEE Wireless Communications and Networking WCNC 2003*, volume 3, pages 1621–1626, March 20–20, 2003.

[51] Mihaela Cardei and Ding-Zhu Du. Improving wireless sensor network lifetime through power aware organization. *Wirel. Netw.*, 11(3):333–340, 2005.

[52] Kuei-Ping Shih, Yen-Da Chen, Chun-Wei Chiang, and Bo-Jun Liu. A distributed active sensor selection scheme for wireless sensor networks. In *Proc. 11th IEEE Symposium on Computers and Communications ISCC '06*, pages 923–928, June 26–29, 2006.

[53] Jun Lu, Lichun Bao, and Tatsuya Suda. Coverage-aware sensor engagement in dense sensor networks. *J. Embedded Comput.*, 3:3–18, 2009.

[54] O. Tilak, S. Mukhopadhyay, M. Tuceryan, and R. Raje. A novel reinforcement learning framework for sensor subset selection. In *IEEE ICNSC*, Chicago, IL, 2010.

[55] R. G. Brown and P. Y. C. Hwang. *Introduction to Random Signals and Applied Kalman Filtering, 3rd edition.* John Wiley & Sons, Inc, 1997.

[56] Isler, V., Khanna, S., Spletzer, J., and Taylor, C. . Target tracking with distributed sensors: the focus of attention problem. *Computer Vision and Image Understanding*, pages 225–247, 2005.

[57] Franz Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, 1991.

[58] Ting Yan, Tian He, and John A. Stankovic. Differentiated surveillance for sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 51–62, New York, NY, USA, 2003. ACM.

[59] Juan Liu, Maurice Chu, Jie Liu, Jim Reich, and Feng Zhao. Distributed state representation for tracking problems in sensor networks. In *IPSN '04: Proceedings of the 3rd international symposium on Information processing in sensor networks*, pages 234–242, New York, NY, USA, 2004. ACM.

[60] Hanbiao Wang, Kung Yao, Greg Pottie, and Deborah Estrin. Entropy-based sensor selection heuristic for target localization. In *IPSN '04: Proceedings of the 3rd international symposium on Information processing in sensor networks*, pages 36–45, New York, NY, USA, 2004. ACM.

[61] R. E. Kalman. A new approach to linear filtering and prediction problems. *Transaction of the ASME—Journal of Basic Engineering*, pages 35–45, 1960. Original paper by Kalman that invented the Kalman filters.

[62] Greg Welch and Gary Bishop. *An Introduction to the Kalman Filter*. ACM, Los Angeles, 2001.

[63] Wendi Rabiner Heinzelman, Anantha Ch, and Hari Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. 1999.

[64] O. J. Tilak, M. Babbar-Sebens, and S. Mukhopadhyay. Decentralized and partially decentralized reinforcement learning for designing a distributed wetland system in watersheds. *IEEE Int Conf on Systems, Man, and Cybernetics - Special Sessions*, To Appear 2011.

[65] C. M. Fonseca and P. J. Fleming. Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. *Proceedings of the Fifth International Conference on Genetic Algorithms*, page 416423, 1993.

[66] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Trans. Evol. Comput.*, 6:182–197, 2002.

[67] J. Horn, N. Nafpliotis, and D. E. Goldberg. A niched pareto genetic algorithm for multiobjective optimization. *IEEE World Congress on Computational Computation*, pages 82–87, 1994.

[68] J. D. Schaffer. *Multiple objective optimization with vector evaluated genetic algorithms.* PhD thesis, Vanderbilt University, 1984.

[69] E. Bekele. Multiobjective management of ecosystem services by integrative watershed modeling and evolutionary algorithms. *Water Resources Research*, 10, 2005.

[70] E. G. Bekele and J. W. Nicklow. Multi-objective optimal control model for watershed management using swat and nsga-ii. *ASCE Conf. Proc.*, 2007.

[71] J. L. Dorn and S. Ranjithan. Evolutionary multiobjective optimization in watershed water quality management. *Evolutionary Multi-Criteria Optimization, Lecture Notes in Computer Science (LNCS)*, 2632:692–706, 2003.

[72] C. Maringanti, I. Chaubey, and J. Popp. Development of a multiobjective optimization tool for the selection and placement of best management practices for nonpoint source pollution control. *Water Resour. Res.*, 45, 2009.

[73] M. Babbar-Sebens and B.S. Minsker. Case-based micro interactive genetic algorithm (cbmiga) for interactive learning: Methodology and application to groundwater monitoring design. *Environmental Modelling and Software*, 25:1176–1187, 2010.

[74] M. Babbar-Sebens and S. Mukhopadhyay. Reinforcement learning for human-machine collaborative optimization: Application in ground water monitoring. *Proceedings of the IEEE Systems, Man, and Cybernetics (SMC) Conference*, page 3563 3568, 2009.

[75] M. Babbar-Sebens and B.S. Minsker. Standard interactive genetic algorithm (siga): A comprehensive optimization framework for long-term ground water monitoring design. *J. of Water Resources Planning and Management*, pages 538–547, 2008.

[76] J. G. Arnold, R. Srinivasan, R. S. Muttiah, and J. R. Williams. Large area hydrologic modeling and assessment. part i: Model development. *J. Am. Water Resour. Assoc.*, 34(1):73 89, 1998.

[77] S. L. Neitsch, J. G. Arnold, J. R. Kiniry, and J. R. Williams. Soil and water assessment tool - theoretical documentation - version 2005. *Grassland, Soil and Water Research Laboratory, Agricultural Research Service and Blackland Research Center, Texas Agricultural Experiment Station, Temple, TX.*, 2005.

VITA

VITA

Omkar Jayant Tilak

Education

(1) B.E. in Computer Engineering, Mumbai University, Mumbai, India, 2004

(2) M.S. in Computer Science, Indiana University Purdue University Indianapolis, Indianapolis, IN, 2006

(3) Ph.D. in Computer Science, Purdue University, West Lafayette, IN, 2012

Relevant Publications

(1) **Tilak, O.**, Babbar-Sebens, M. and Mukhopadhyay, S., Decentralized and Partially Decentralized Reinforcement Learning for Designing a Distributed Wetland System in Watersheds, IEEE Int Conf on Systems, Man, and Cybernetics - Special Sessions, 2011.

(2) **Tilak, O.** and Mukhopadhyay, S., Partially Decentralized Reinforcement Learning in Finite, Multi-Agent Markov Chains, AI Communications (Accepted For Publication), 2011.

(3) **Tilak, O.**, Martin, R. and Mukhopadhyay, S., A decentralized indirect method for learning automata games, IEEE Systems, Man., and Cybernetics B (Accepted and In Print), 2011.

(4) **Tilak, O.** and Mukhopadhyay, S., Multi Agent Reinforcement Learning for Dynamic Zero-Sum Games, (Under Preparation), 2011.

(5) **Tilak, O.**, Mukhopadhyay, S., Tuceryan, M. and Raje, R., A Novel Reinforcement Learning Framework for Sensor Subset Selection, IEEE ICNSC, 2010.

(6) **Tilak, O.** and Mukhopadhyay, S., Decentralized and Partially Decentralized Reinforcement Learning for Distributed Combinatorial Optimization Problems, ICMLA, 2010.