USING DEC-POMDP ALGORITHMS TO SOLVE MULTI-AGENT DECISION PROBLEMS IN ROBOT SOCCER

by

Okan Aşık

B.S., Computer Education and Educational Technology, Boğaziçi University, 2009

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering
Boğaziçi University

2012

USING DEC-POMDP ALGORITHMS TO SOLVE MULTI-AGENT DECISION PROBLEMS IN ROBOT SOCCER

APPROVED BY:

Prof. H. Levent Akın (Thesis Supervisor)	. . . . .
Prof. Ümit Bilge	. . . . .
Assist. Prof. Albert Ali Salah	. . . . .

DATE OF APPROVAL: 11.06.2012

ACKNOWLEDGEMENTS

First of all, I would like to thank to my supervisor Prof. H. Levent Akın for his genuine support and patience. He always helped me whenever I was lost.

I am grateful to Prof. Ümit Bilge and Assist. Prof. Albert Ali Salah for kindly accepting to be a member of my thesis jury.

I want to express my deepest gratefulness to my family. They always supported me through this journey.

I also thank to the great people studying in the Robotics lab. Their support and sharing is the true source of a great work.

This thesis has been supported by the Scientific and Technological Research Council of Turkey (TUBİTAK) 2210 National Graduate Scholarship Program and Boğaziçi University Research Fund project 09M105.

ABSTRACT

USING DEC-POMDP ALGORITHMS TO SOLVE MULTI-AGENT DECISION PROBLEMS IN ROBOT SOCCER

Decentralized Partially Observable Markov Decision Process (Dec-POMDP) is a recent mathematical framework which has been used to model multi-agent coordination and decision making. However, its real life applications are limited. Robot soccer is one of the good testbeds to investigate the potential of Dec-POMDP algorithms. In this work, we use the Dec-POMDP algorithm developed by Eker and Akın [1]. The algorithm is a policy search algorithm. It searches the policy space with a genetic algorithm. The genetic algorithm uses a simulator to estimate the fitness of chromosomes. There are two policy representations. The finite state controller representation is used for discrete Dec-POMDP models. We extend Eker and Akın's algorithm by using a neural network representation for continuous Dec-POMDP problems. The experiments are carried out in the *RoboCup* 2D robot soccer simulator and *TeamBots* simulator. We show that the algorithm is capable of solving complex problems such as robot soccer. We have experimented with different fitness functions, and we have found that the game score is the best one. We also compare the performances of the two methods, namely Dec-POMDP algorithm and reinforcement learning. It is found that the Dec-POMDP algorithm with the finite state controller representation is better than the reinforcement learning method. We also show that, in the case of the Keepaway problem, the Dec-POMDP algorithm with the neural network representation is better than a hand-coded benchmark policy, and is also comparable to the reinforcement learning method.

ÖZET

ROBOT FUTBOLUNDAKİ ÇOKLU KARAR VERME PROBLEMLERİNİ ÇÖZMEK İÇİN MO-KGMKS ALGORİTMALARININ KULLANILMASI

Merkezi Olmayan Kısmen Gözlemlenebilir Markov Karar Süreci(MO-KGMKS) modeli, çoklu etmen koordinasyonu ve karar vermede kullanılan yeni bir matematiksel çerçevedir. Buna rağmen, gerçek hayat uygulamaları kısıtlıdır. Robot futbolu, MO-KGMKS algoritmalarının potensiyelini araştırmak için en iyi sınama ortamlarındandır. Bu çalışmada, Eker ve Akın [1] tarafından geliştirilen MO-KGMKS algoritması kullanılmaktadır. Algoritma strateji arama algoritmasıdır. Strateji uzayını genetik algoritmayla arar. Genetik algoritma kromozomların uygunluğunu değerlendirmek için bir benzetici kullanır. İki strateji gösterimi vardır. Sonlu durum denetleyicisi gösterimi ayrık MO-KGMKS modelleri için kullanılmaktadır. Eker ve Akın'ın algoritmasına ek olarak, yapay sinir ağları gösterimi sürekli MO-KGMKS modelleri için kullanılmaktadır. Deneyler, *Robocup* 2B robot futbolu benzeticisinde ve *TeamBots* benzeticisinde uygulanmıştır. Algoritmanın, robot futbolu gibi karmaşık bir problemi çözebildiği gösterilmektedir. Değişik uygunluk fonksiyonları denenmiş ve oyun skorunun en iyisi olduğu bulunmuştur. MO-KGMKS algoritması, destekli öğrenme algoritması ile karşılaştırılmıştır. Sonlu durum denetleyicili MO-KGMKS algoritmasının destekli öğrenme algoritmasından daha iyi olduğu bulunmuştur. Ayrıca, Keepaway probleminde yapay sinir ağlar temsilini kullanan MO-KGMKS algoritması elle kodlanmış kriter stratejisinden daha iyi olduğu ve destekli öğrenme metoduna yakın sonuçlar elde edildiği gösterilmiştir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

A	A set of actions
n	The number of agents
N_B	Number of simulations before evolution
N_D	Number of simulations during evolution
N_A	Number of simulations after evolution
Ω	A set of joint observations
Obs	Observation function
R	Reward function
S	A set of states
T	Transition function

LIST OF ACRONYMS/ABBREVIATIONS

2D	Two Dimensional
3D	Three Dimensional
Dec-POMDP	Decentralized Partially Observable Markov Decision Process
FSC	Finite State Controller
JGAP	Java Genetic Algorithms Package
MDP	Markov Decision Process
NN	Neural Network
POMDP	Partially Observable Markov Decision Process
SMDP	Semi Markov Decision Process

1. INTRODUCTION

Robots can be modeled as physical agents which interact with their environment via their sensors and actuators. Robots make decisions based on the data they acquire via their sensors, and their decisions are executed by the actuators of the robot. The main problem of a robot is finding a mapping of its perceptions to actuator outputs to achieve its designated goal. This can can be modeled as a decision making problem. There are many methods to solve such decision making problems. However, due to the ever increasing complexity of the problems encountered in real situations, approaches using machine learning methods have become more widely used compared to other methods recently.

Even though many tasks can be carried out by a single robot, there are some tasks which require the cooperation of multiple robots, such as robot soccer. Here, eventhough, all the robots act autonomously, they should be coordinated to achieve their task, such as scoring a goal. If we know the dynamics of the problem, we may develop a suitable algorithm. However, such an algorithm generally fails due to high uncertainty associated with noise in sensors, actuators and unmodeled dynamics in real life problems. Also, the performance of such an algorithm is highly dependent on the designer's understanding of the problem. Even if we may fully understand the problem, we may not be able to develop a robust algorithm which achieves the coordination of robots in a bottom up fashion. Because of all these reasons, using machine learning methods for multi-agent decision making becomes more favorable.

Decentralized Partially Observable Markov Decision Process (Dec-POMDP) model is one of the promising approaches to solve multi-agent decision making under uncertainty. However, it has been shown that the computational complexity of *Dec-POMDP* model is NEXP-Complete for the worst case [2]. *Dec-POMDP* algorithms can be categorized in two groups: optimal and approximate. Optimal algorithms have theoretic value and are not suitable for real life problems. Approximate algorithms are expected to generate satisfactory results for real life problems.

Most of the *Dec-POMDP* algorithms in the literature have been tested on rather simple benchmark problems such as Dec-Tiger, multi-access broadcast channel, meeting in a grid, box pushing, and fire fighting problems. Although benchmark problems are useful for comparing and contrasting different algorithms, they do not provide enough information about the real life applications of algorithms. Robot soccer is one of the good testbeds to experiment with multi-agent decision making under uncertainty. It provides the required complexity of real life in a well-defined game structure. It is noisy, requires the coordination of robots, and also its dynamics are not as obvious as benchmark problems.

In tge recent years, there is a focus on developing scalable approximate *Dec-POMDP* algorithms. The main motivation of this work is to explore the capabilities of *Dec-POMDP* algorithms in real life problems. To realize the true potential of *Dec-POMDP* algorithms, solving a real life problem has utmost importance. By solving robot soccer with Eker and Akın's *Dec-POMDP* algorithm [1], we provide a particular example of how to solve a real life problem with *Dec-POMDP* model.

The organization of the rest of the thesis is as follows. In Chapter 2, we give brief information about multi-agent decision making, and robot soccer. We also mention *MDP*, *POMDP*, and *Dec-POMDP*. We present *Dec-POMDP* algorithms, and related work.

The algorithm, developed by Eker and Akın [1], is presented in Chapter 3 together with our neural network representation. The robot soccer *Dec-POMDP* model is shown with particular implementation details for both the *Keepaway Framework* and *TeamBots*.

In Chapter 4, we give the implementation details of the experiments. The results of both continuous and discrete *Dec-POMDP* experiments are presented. Also, the comparison of finite state controller and neural network policy representations are provided.

Finally, we present the conclusion by summarizing the results of experiments in Chapter 5. We also mention possible future works.

2. BACKGROUND

2.1. The Decision Making Problem

Decision making is one of the most important desirable capabilities of any agent. An agent takes input from the environment, makes a decision, and carries out its decision. Decision making is choosing an action from the action set which is predefined by the designer. The performance of an agent is dependent on the action selection. A decision making algorithm defines a strategy to choose the optimal action according to the current state.

2.1.1. Markov Decision Processes

The Markov Decision Process (MDP) is a mathematical framework for decision making where actions are probabilistic. However, to use this powerful framework, we need to formalize the problem and define the following:

- A set of states, S.
- A set of actions, A.
- Transition function, T.
- Reward function, R.

At a time step, the process is in some state $s \in S$, and the agent chooses an action $a \in A$. At the next time step, the process moves into a new state $s' \in S$ with probability $T_a(s, s')$, and reward r, $r = R(s, s')$, is taken. A state is only dependent on the previous state and the action taken, and this is called the *Markov Property*. However, *MDP* allows probabilistic state transitions. In other words, the outcome of an action is probabilistic. It also assumes that the process knows its state [3].

2.1.2. Semi Markov Decision Process

Semi Markov Decision Process (SMDP) is a modification of *MDP* to include *options* instead of actions. Options are special actions which may take more than one steps to complete. Therefore, transition and reward functions are defined in terms of options [4]. Therefore, in a *SMDP*, the next state is defined by the option and the current state.

2.1.3. Partially Observable Markov Decision Process

A Partially Observable Markov Decision Process (POMDP) is a generalization of *MDP* to the case where an agent does not know its state exactly. The observation set, O, and the observation function, Ω, are added to the definition of *MDP*, also the reward function is modified as follows: $R(s, a)$ where $s \in S$ and $a \in A$.

2.1.4. Decentralized Partially Observable Markov Decision Process

The Decentralized Partially Observable Markov Decision Process (Dec-POMDP) is an extension of *POMDP* to the multi-agent case. When decentralized agents act to achieve some task, the problem can be modeled as a *Dec-POMDP*. At each time step, each agent carries out an action and receives a local observation and the joint immediate reward. A local policy for each agent is a mapping from its observation sequences to actions. In this study, we use the formalization defined by Bernstein *et al.* [2].

Definition 2.1. *The* Dec-POMDP model *is a 7-tuple* $(n, S, A, T, \Omega, Obs, R)$ *where:*

- n *is the number of agents.*
- S *is a finite set of states.*
- A *is the set of joint actions which is the Cartesian product of* A_i $(i = 1, 2..., n)$ *i.e. the set of actions available to* $agent_i$*.*
- T *is the state transition function which determines the probabilities of the possible next states given the current state* S *and the current joint action* a*.*
- Ω *is the set of joint observations which is the Cartesian product of* Ω_i $(i = 1, 2..., n)$

i.e. the set of observations available to $agent_i$*. At any time step the agents receive a joint observation* $o = (o_1, o_2, ..., o_n)$ *from the environment.*

- Obs *is the observation function which specifies the probability of receiving the joint observation* o *given the current state* s *and the current joint action* a*.*
- R *is the immediate reward function specifying the reward taken by the multi-agent team given the current state and the joint action.*

2.1.5. *Dec-POMDP* Solution Algorithms

We can categorize *Dec-POMDP* algorithms as optimal and approximate algorithms. These two categories of algorithms can also be classified into two groups: finite horizon and infinite horizon algorithms. *Horizon* defines the time step the algorithm considers. For example, for the finite horizon case, the algorithm tries to compute the best policy considering the given finite time steps. However, for the infinite horizon case, the algorithm tries to compute the best policy if the process takes an infinite number of time steps.

2.1.5.1. Optimal Finite-horizon Algorithms. A policy of an agent is a mapping of observation histories to actions. A policy can be represented as a tree. the nodes represent actions and the branches connecting the nodes represent observations. The horizon is given by the depth of the tree. By constructing all possible decision trees, we can find the optimal policy. However, the complexity of such an algorithm is double exponential in the horizon t and exponential in the number of agents [5]. Therefore, two improvements are developed over the exhaustive search method: dynamic programming and heuristic search.

- *Dynamic programming based Dec-POMDP algorithm* is based on the dynamic programming algorithm for POMDPs. At every step, all possible policies are constructed, then pruning is applied. To apply pruning the POMDP is converted to an MDP by defining a state which contains all possible beliefs about the current state. Pruning eliminates policy trees whose value vector is not greater than the value functions of other policy trees over all belief states. However, Dec-POMDP does not have the notion of belief state. We need belief states to compare value functions over belief states.

When we compare vector values of every agent against other agents over all possible beliefs, we can eliminate the policy whose value vector is not greater than any other policies' value vector. However, the degree of pruning is highly depended on particular problem, and it drastically affects the performance of the algorithm [5].

- *Heuristic search algorithm* is another method which tries to decrease the complexity over exhaustive enumeration method. It is based on the $A^\star$ algorithm and searches the joint policy space. Joint policies are constructed as in exhaustive enumeration. Search tree is constructed, but not all nodes are expanded. only the node whose heuristic estimation is the greatest is expanded. The level of the search tree corresponds to the horizon of the joint policies. It is important to note that the heuristic function should be admissible. However, the performance of the algorithm is highly depended on tightness of the heuristic function [5].

2.1.5.2. ϵ-Optimal Infinite-horizon Algorithms. There is no optimal algorithm for the infinite-horizon case due to undecidability. Bernstein *et al.* [6] proposes a policy iteration algorithm which represents the policies of agents as a finite state controller. With the finite state controller representation, there is no correlation between the agents. However, a correlation is needed since it drastically affects the performance [5]. The number of finite state controller nodes might be less than required because there may be an infinite sequences of observations. Therefore, we can increase the number of finite state controller nodes successively with *exhaustive backup*. However, we need to apply pruning to discard the unneccessary policies. This can be done by using value-preserving transformations. It decreases the number of finite state controller nodes without reducing its value with linear programming [5].

2.1.5.3. Approximate Finite-horizon Algorithms. Improved memory-bounded dynamic programming makes use of the ideas proposed in the optimal algorithm which uses dynamic programming [7]. The main problem of the dynamic programming *Dec-POMDP* algorithm is that, in spite of pruning, the algorithm exceeds the any available memory. However, pruning considers all possible belief states. Most of the belief states are not reachable. Therefore, the algorithm uses heuristics to identify reachable belief states and the most probable observations, then it uses dynamic programming to select the appropriate joint policies [5]. Since

it chooses a fixed number of belief state and observations, it is called memory bounded.

2.1.5.4. Approximate Infinite-horizon Algorithms. Since infinite-horizon policies cannot be represented as decision trees, most of the approximate infinite-horizon algorithms use the finite state controller representation. Bernstein *et al.* [8] improves Poupart and Boutilier's *bounded policy iteration* algorithm [9] for *Dec-POMDP*s. The algorithm represents policies as finite state controllers having fixed number of nodes. It starts with a randomly generated list of controllers. Then, those controllers are improved iteratively with linear programming. A controller of an agent is improved while the controllers of the other agents are fixed. Every iteration of the algorithm produces a new set of policies which are at least as good as the previous controllers. Since only one agent's controller is improved at a time, the algorithm can reach a local optima [5]. Therefore, to achieve better results many non-linear optimization techniques are proposed [5]. Although non-linear optimization methods do not guarantee a global optima, they perform better than the bounded policy iteration algorithm in practice.

Eker and Akın's algorithm [1] is also an approximate infinite-horizon algorithm. It is a policy search algorithm which uses genetic algorithms as a search method. The agents' policies are represented as finite state controllers, and those policies are executed on the simulator of the problem. The simulator calculates the fitness of the joint policies. The genetic algorithm evolves policies based on the fitnesses of the individuals in the population.

2.2. Genetic Algorithm

In genetic algorithms, a candidate solution is encoded as a chromosome and the set of all chromosomes is called a *population*. The fitness of a candidate solution determines how good the candidate is. Through the application of evolutionary operators such as selection, crossover, and mutation, a new population is created from the current population. When the convergence criteria are met, the algorithm terminates and the best candidate becomes the solution of the algorithm [10].

2.2.1. Selection Operator

Selection, also called natural selection, is the process of choosing chromosomes which will contribute to the creation of a new generation. The fitness metric used chooses chromosomes which will play a role in the creation of a new generation. Therefore, before the natural selection, fitnesses of individual chromosomes have to be determined. A good fitness function is expected to calculate how good a chromosome is compared to the other chromosomes in the population. However, it is not easy to develop a fitness function for each and every problem. An appropriate fitness function should help evolution while decreasing the need for prior information used in its calculation [11].

When the chromosomes are ranked according to their fitness values, the most intuitive idea is to choose the best chromosomes. However, statistical selection is found to be more successful. Roulette wheel selection is one of the most widely used selection method. Every chromosome has a chance to be chosen proportional to its fitness value. In the experiments, we use this selection method.

2.2.2. Mutation Operator

Once the chromosomes are selected, genetic operators are used to modify them to create a new generation. Mutation is one of the widely used genetic operators. It makes a random change in the chromosome to solve stagnation at a local minimum. There are different methods to apply a random change, but one of the most widely used one is flipping a random bit. With the given mutation probability, a bit of a new generation's chromosome is flipped. In other words, it is changed to 0 if it is 1, and vice versa.

2.2.3. Crossover Operator

Crossover is another widely used genetic operator. It combines two chromosomes into a chromosome. There are different methods to combine two chromosomes, but one point crossover is one of the most widely used ones. A point is selected, and chromosomes are divided into two parts from the selected point. Those parts are swapped, and two new

chromosomes are constructed. Crossover occurs with a probability determined by the user of the algorithm.

2.3. Robot Soccer

Robot soccer has been proposed as a complex research problem [12]. It is addressed in different levels. In the *RoboCup* competitions [13], robot soccer is used as a research problem in five different leagues with changing research foci.

- *The humanoid league* focuses on developing both hardware and software for robot soccer.
- *The middle size league* focuses on full autonomy and cooperation at plan and perception levels.
- *The small size league* focuses on the problem of intelligent multi-robot cooperation and control in a highly dynamic environment with a hybrid centralized system.
- *The standard platform league* focuses on intelligent software development.
- *The simulation league* focuses on artificial intelligence and team strategy.

2.3.1. Robot Soccer Metrics

Before teaching how to play soccer to robots, we need to know the dynamics of robot soccer. Meriçli [14] provides an informative list of robot soccer metrics that can be used in evaluating the learning process. These metrics are developed to provide immediate rewards for learning algorithms.

At every simulation step, the following metrics are calculated based on the position of teammates, opponents and the ball. These metrics are as follows:

2.3.1.1. Convex Hull Metrics.

Definition 2.2. Convex Hull *of a set of points is the smallest convex polygon in which all points in the set lies [14].*

If we use the player positions as a set of points, we can calculate the convex hull of a team. Informally, it is a measure of spread on the soccer field. We use two metrics based on the convex hull: our team's convex hull area and opponent team's convex hull area.

2.3.1.2. Vicinity Occupancy Metrics.

Definition 2.3. Vicinity Occupancy *is the ratio of the team players to the opponent players within the vicinity of the object of interest [14].*

It basically measures the dominance of the team in a particular area. Negative values indicate that the opponent is dominant in that area, and the value of dominance is between -1 and 1. We use vicinity occupancy of three areas: ball, own goal area and opponent goal area.

2.3.1.3. Pairwise Separation Metrics.

Definition 2.4. Pairwise Separation *measures the degree of separation of an object of interest with the opponent team [14].*

Informally, it measures whether the team is able to block the opponent team from the object of interest. For example, if there are two players close to the ball and the opponent can get the ball only through these players, it is said that the opponent is pairwise separated from the ball. We use six pairwise separation metrics as follows:

The pairwise separation of

- ball from opponent players by our players,
- ball from our players by opponent players,
- own goal from opponent players,
- own goal from our players,
- opponent goal from our players,
- opponent goal from opponent players.

2.4. Keepaway Soccer

Keepaway is a sub-problem of robot soccer where there are two teams. The *keepers* try to maintain the control of the ball as long as possible, and the *takers* try to take control of the ball from the keepers. If the takers take control of the ball from the keepers, the game ends. The *Keepaway Framework* has been developed by Stone and *et al.* [15] as a set of simulation tools to provide an easy to use learning benchmark framework. It is based on the *Robocup2D soccer simulator*. It provides an API to develop your own learning agents. In addition to the learning agents, there is also a hand coded policy, random policy and an always ball holding policy. Therefore, the researchers using this framework can compare the performance of their algorithms with both the benchmark policies and other studies.

An agent can take an action if it has control of the ball according to *Keepaway Framework*. Otherwise, it carries out the *get-open* action. The *get-open* behavior is implemented such that the agent tries to move to a position where it can take a pass sent by the agent having control of the ball. The agent having control of the ball can take k actions, where k denotes the number of keepers. It can either continue to keep the ball or pass it to one of the $k-1$ teammates.

The *Keepaway Framework* provides an *SMDP* model.

2.4.1. The Benchmark Policies

In the *Keepaway Framework*, there are three benchmark policies:

- *Always hold* policy always keeps the possession of the ball. This policy provides a benchmark to see how fast the takers can reach a keeper.
- *Random* policy chooses an action with uniform probability. This policy provides a benchmark so that any meaningful policy should outperform it.
- *Hand coded* policy implements a basic strategy. If the agent is not under pressure, it keeps the ball. If it is under pressure, it calculates an openness metric to pass to a more convenient teammate. This metric is based on the angle between the closest taker and a

teammate, and distance of the closest taker to a teammate. The detail of the algorithm can be seen in Figure 2.1.

Algorithm Hand-coded Policy

if no taker is within 4m **then**

$HoldBall()$

else

for $i \in [2, n]$ **do**

$V_i \Leftarrow Min(ang(K_i, K_1, T_1), ang(K_i, K_1, T_2)) + \alpha \times Min(dist(K_i, T_1), dist(K_i, T_2))$

end for

$I \Leftarrow argmax_i V_i$

if $V_I > \beta$ **then**

$PassBall(K_I)$

else

$HoldBall()$

end if

end if

Figure 2.1. The Hand-coded Algorithm [16].

2.5. Related Work

Wu and Chen solve the soccer problem modeled as a *Dec-POMDP* with *Correlation-MDP*s in the *Robocup* domain [17]. They base their work on the memory-bounded dynamic programming algorithm proposed by Bernstein *et al.* [8]. Their main contribution is proposing an approximate algorithm to calculate the correlation device. They used the algorithm to improve the coordination of soccer playing agents in the *Robocup* 2006 Soccer 2D Simulation Competitions and won all the matches except one. This study is important in terms of showing the capabilities of the *Dec-POMDP* framework in the robot soccer domain.

Keepaway soccer was put forth as a testbed for machine learning [15], and there is a wide variety of reinforcement algorithms which are tested with keepaway soccer [16, 18, 19]. Di Pietro *et al.* used evolutionary algorithms to learn a policy which results in coordinated

behavior [20]. They formulate the problem so that the agent decisions are based on parameters such as the distance to the recipient. The evolutionary algorithm searches for the optimal parameters to keep the ball as long as possible which is the ultimate goal of keepaway soccer. This work is close to our work in terms of using an evolutionary algorithm and trying to solve the soccer problem, but they provide a limited solution to a sub-problem of robot soccer.

3. METHODOLOGY

There are two components of the proposed approach in this study: encoding of the *Dec-POMDP* policies and the genetic algorithm. Two encoding methods are used in this study: finite state controller, and neural network. Finite state controller is used to encode policies of discrete *Dec-POMDP* problems, and neural network encoding is used to represent policies of continuous *Dec-POMDP* problems.

3.1. The Genetic Algorithm

In addition to the basic operations of genetic algorithms outlined in Chapter 2, Eker and Akın [1] developed a three phase genetic algorithm. The main difference between the phases is the precision of the fitness calculation. Similar results might be achieved, if we use a simple genetic algorithm with a very high precision fitness calculation. However, the algorithm's running time becomes infeasible. With such a three phase genetic algorithm, we calculate the fitness of the candidate solution in a sufficiently precise manner to achieve the expected functionality.

The phases are as follows:

- *Pre-Evolution Phase:* Initially k best chromosomes are calculated and put into the best chromosomes list. In this phase a higher precision fitness calculation is used because the quality of the genetic algorithms is dependent on the initial population. If there are good solutions at the beginning, the next generations will be reproduced from them. For some problems, very good solutions may exist in the population, but their fitnesses may be calculated to be very low. By having a high precision calculation, we guarantee not to miss such a solution [1]. Therefore, at the beginning of the evolution and at the end of every evolution cycle, good chromosomes' fitnesses are recomputed. If they are still good to be in the list of best chromosomes, they are added to the best chromosomes list.
- *Evolution Phase:* After each generation, k best chromosomes of the current *population*

are compared with the chromosomes which are in the best chromosomes list. If one of the chromosomes is good enough to be in the best chromosomes list, its fitness is calculated with higher precision. If it is still good enough to be in the best chromosomes list, it is added to the list. The algorithm is detailed in Figure 3.1.This phase has the least precise fitness calculation because the fitness of the chromosomes will be used only to rank the chromosomes. Once the chromosomes are ranked, basic genetic operators are applied. In order to converge to a good solution, it is enough to identify bad solutions, good solutions, and the intermediate ones.

- *Post Evolution Phase:* This phase has the highest precision calculation because it is the final phase of the algorithm. The evolution ends if one of two conditions holds, either generation count reaches a prespecified maximum number or the fitness of the best chromosome does not change for some generations. When the termination criteria are satisfied the evolution, is terminated. The best solution of the best solution list. The fitness of the solution should be calculated as precisely as possible so that the algorithm does not choose a moderate chromosome as the final solution while there is a better solution in the list.

3.2. Finite State Controller Representation

A finite state controller (FSC) is a special finite state machine. A finite state machine has states, and transitions. It can only be in a state at a time. The machine may change its state when a transition occurs. A finite state controller operates exactly like a finite state machine except in a finite state controller the state corresponds to an action so that at every time step the action of the state is executed. Also, there is no relation between *Dec-POMDP* states and finite state controller states. Therefore, we call FSC states as FSC nodes. An observation results in a node transition.

The solution of a *Dec-POMDP* model is a mapping of observation histories to actions for each and every agent. A solution of a *Dec-POMDP* model can be represented as a finite state controller. This finite state controller does the mapping of observation histories to the actions. By encoding *Dec-POMDP* problem as a finite state controller, we have a new search space. The problem is transformed to that of finding the optimal finite state controller.

Algorithm $GeneticAlgorithm$

Form random population.
Compute fitness of the population with N_B.
Form $kBestChromosomes$ with k best chromosomes of the population.
for $generation \Leftarrow 0$ to $maxGeneration$ **do**
  Compute fitness of the population with N_D.
  Evolve population.
  Compute $kGenBestChromsomes$ of the current population.
  for $i \Leftarrow 0$ to k **do**
    if $kGenBestChromsomes[i] > MinFitness(kBestChromosomes)$ **then**
      Compute fitness of $kGenBestChromsomes[i]$ with N_B.
      if $kGenChromsomes[i] > MinFitness(kBestChromosomes)$ **then**
        Remove $MinFitness(kBestChromosomes)$ from the list.
        Add $kGenBestChromsomes[i]$ to the $kBestChromosomes$ list.
      end if
    end if
  end for
  Compute $bestFitness$.
  if $bestFitness = previousBestFitness$ **then**
    Increase $noChangeCounter$
  else
    $noChangeCounter = 0$
  end if
  if $noChangeCounter = maxNoChange$ **then**
    Break
  end if
  $previousBestFitness \Leftarrow bestFitness$
end for
Compute fitness of $kBestChromosomes$ with N_A.
Return $max(kBestChromosomes)$

Figure 3.1. The Genetic Algorithm [1].

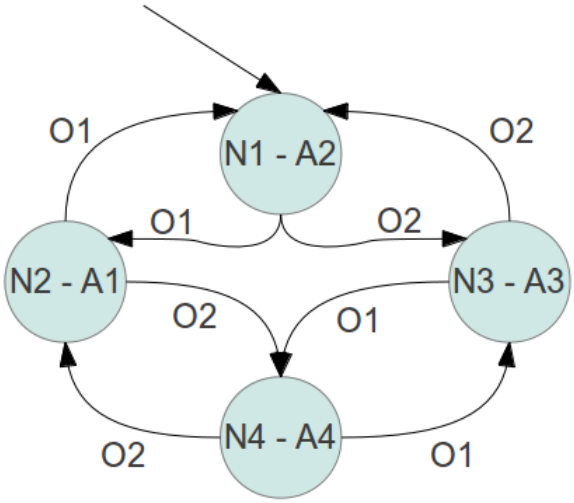

Figure 3.2. An Example Finite State Controller: N represents finite state controller nodes, A represents actions, O represents observations, and incoming arrows represents transitions.

The finite state controller representation enables a compact representation of *Dec-POMDP* policies compared to the other methods, such as tree representation. It is also flexible in the sense that the number of FSC nodes can be changed.

An example finite state controller can be seen in Figure 3.2. This finite state controller is designed for a problem having two observations and four actions. There is always a starting node. For this particular finite state controller it is $N1$. At first, since the initial FSC node is $N1$, the action associated with the node, $A2$, is executed. If the agent gets an observation $O2$, it updates its current node to $N2$ and executes the action $A1$. Action execution and state update continues until the agent completes its task or the episode ends. This finite state controller represents the policy of a single agent. Since, every agent has its own finite state controller; they may have different observation and action models.

3.2.1. Finite State Controller Encoding

In our study, we encode the finite state controller as an integer chromosome. The first n genes represent a node-action mapping. In other words, they determine which action will be executed in a FSC node. Their values are in the range of 1 to the number of actions. Then,

A2	A1	A3	A4	N2	N3	N1	N4	N4	N1	N3	N2
Best Action for N1	Best Action for N3	Best Action for N2	Best Action for N4	O1 is taken at N1	O2 is taken at N1	O1 is taken at N2	O2 is taken at N2	O1 is taken at N3	O2 is taken at N3	O1 is taken at N4	O2 is taken at N4

Figure 3.3. An Example Finite State Controller Encoding.

for each node, there is an observation-node mapping which denotes the transition when an observation is taken. The encoding of the Figure 3.2 can be seen in Figure 3.3.

3.3. Neural Network Representation

A neural network is a network of interconnected artificial neurons. In this study we use the multi-layer perceptron model [21]. Every artificial neuron has a threshold value and a weight value for every incoming connection. A neuron activates the outgoing connected neurons if the weighted sum of incoming connected neuron values are greater than the threshold values. There are three types of layers of neurons:

- The **input** layer consists of neurons which are activated by system inputs.
- A **hidden layer** is the layer between input and output layers. There can be more than zero or more hidden layers.
- The **output** layer consists of neurons which provide the solution of the neural network.

A fully connected neural network example can be seen in Figure 3.4.

As a *Dec-POMDP* model, a neural network can take observations as inputs and return outputs as actions. However, thresholding in the artificial neurons is omitted to reduce the number of parameters. When we use the neural network representation with a *Dec-POMDP* having discrete action set, we need to discretize real-valued outputs. There are as many as output neurons as the number of actions. Every output neuron corresponds to an action. The

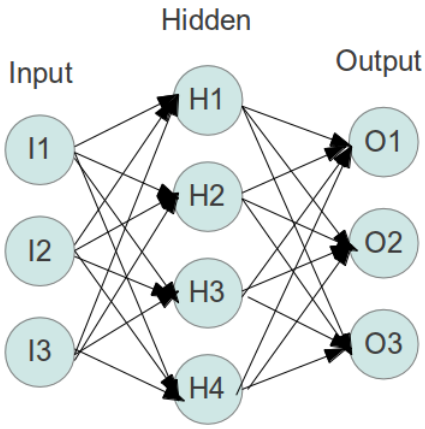

Figure 3.4. An Example Neural Network.

algorithm chooses the action whose output value is the greatest. With these extensions, if the architecture of the neural network is fixed, the problem turns into the optimization of the parameters of a neural network. In other words, the problem changes to the optimization of the weights of the connections between artificial neurons.

One of the most significant advantages of the neural network representation is that there is no need for the discretization of observations and actions. The neural network can take real numbers as inputs and provide outputs real numbers. This is a very powerful feature because many real life problems by nature are continuous and there is a loss when they are discretized.

The process of determining the architecture of neural network is tricky. The performance of architecture is highly problem specific. Also, the complexity of the representation is highly dependent on the complexity of the architecture. The number of connections between the artificial neurons determines the length of the chromosome. The length of the chromosome is very important for genetic algorithms because a linear increase in the length of chromosome results in an exponential increase in the search space. In this study to decrease the length of the chromosomes, a two layered architecture is suggested. Only the input and the output layer are considered. The number of neurons in the input layer is determined by the dimensions of the observation data. The number of neurons in the output layer is determined by the number of actions.

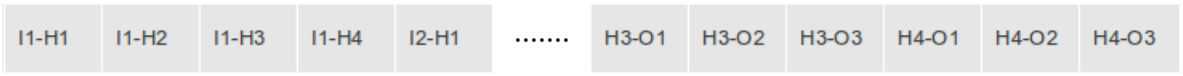

Figure 3.5. An Example Neural Network Encoding.

3.3.1. Neural Network Encoding

In this sudy, the architecture of the neural network is fixed. In other words, the number of neurons in the input layer, hidden layer, and output layer are determined before the optimization process. In addition to the number of neurons, the connections between neurons are also fixed. Therefore, it is sufficient to encode only the weights of the connections. An example encoding can be seen in Figure 3.5. Every value corresponds to the weight of the connection between two neurons, which are identified in Figure 3.4.

4. EXPERIMENTS AND RESULTS

All the experiments are carried out with two different 2D robot soccer simulation environments: *TeamBots* and *Keepaway Framework*.

- **TeamBots** is one of the early robot soccer simulation environments developed by Balch [22]. Although it can be used to simulate different environments, its main aim is to simulate 2D robot soccer. In this work, we use *TeamBots* since it has been previously used in reinforcement learning studies [23–25]. It also has four hand-coded benchmark teams: *BrianTeam*, *Kechze*, *SibHeteroG*, and *AIKHomoG*. Those teams have different strength. In addition to these teams, we also developed a stationary team, *NullTeam*. It has 5 stationary players. It is used to train very basic behaviors such as dribbling ball to the opponent goal.

  We simulate 5 vs. 5 robot soccer games with *TeamBots*. The *TeamBots* environment supplies noisy position information of teammates, opponents and ball to every agent via their sensors.
- **Keepaway Framework** is a set of simulator tools developed by Stone *et al.* to test different learning algorithms. The framework is based on *Robocup* 2D robot soccer simulator [26], and it presents a sub-problem of robot soccer.

The main aim of the experiments is to prove that robot soccer modeled as a *Dec-POMDP* problem can be solved and satisfactory results can be obtained. *TeamBots* is used to show that discrete robot soccer *Dec-POMDP* problem can be solved with finite state controller representation, and *Keepaway Framework* [15] is used to show that continuous robot soccer *Dec-POMDP* problem can be solved with neural network representation. The performances of neural network and finite state controller are compared via *TeamBots*.

The genetic algorithm is implemented with the Java programming language using *JGAP* genetic algorithms package [27]. The communication between the simulation environments and the genetic algorithm is established using both simple file system and network programming. For example, the encoded policy is written to a file and then the agent running

on the simulation reads that file, and acts according to the policy.

All experiments are run on a cluster computer having 8 Intel Xeon E5420 2.50 GHz CPU, 18715 MB Main Memory and 133 GB Hard Disk. The computer runs Ubuntu 10.10 64-bit version.

4.1. Discrete Robot Soccer Dec-POMDP Problem

4.1.1. Robot Soccer Dec-POMDP Model

To model robot soccer as *Dec-POMDP*, we need to define the number of agents, the finite set of states, the set of actions, the set of observations, the transition function, and finally the reward function. However, the transition and reward functions are replaced by the simulation. Since the *FSC* representation needs only the set of observations, actions, and the number of agents, the algorithm tries to optimize the policy without knowing other properties of the model.

In this study, we use the base team developed by Tekin and Çetin Meriçli [23–25]. The number of agents is five, one of them is a hand-coded goal keeper. Therefore, the algorithm learns the coordination of four agents.

The finite set of actions is as follows:

- *Go to the ball.*
- *Go to support position.*
- *Go to defense position.*
- *Pass to the closest teammate.*
- *Pass to the teammate closest to the opponent goal.*

Therefore, there are five different actions which an agent can choose at a particular time. These actions are implemented as potential fields in the *TeamBots* simulation. *Potential fields* is a motion planning approach which produces an action vector for a robot. This action

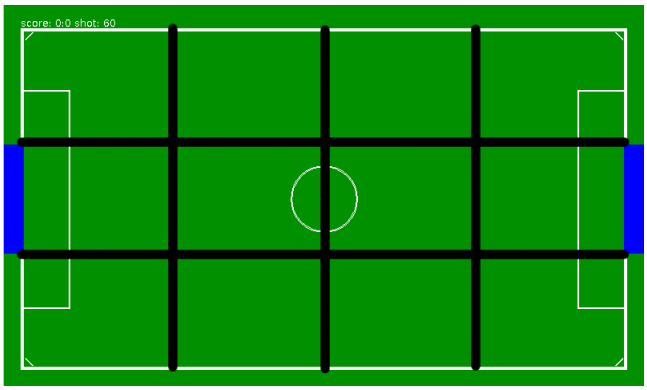

Figure 4.1. *TeamBots* Field.

vector defines the direction of motion and its speed. We can think the motion of a robot as the motion of charged particle in magnetic field. As magnetic field directly determines the motion of the particle, potential field determines the motion of a robot. Therefore, motion planning turns into the design of potential field. The construction of potential field is carried out by defining repelling forces and attractive forces. For example, *going for ball* behavior is developed by defining ball as the source of attractive forces and opponent players as the source of repelling forces. Behaviors of an agent are developed on top of this motion module.

The finite set of observations is as follows: The *TeamBots* field is divided with two equally spaced lines from the narrow edge and three equally spaced lines from the wide edge. In total, there are 12 grid cells as seen in Figure 4.1. The *Location* information is based on this grid.

We define two observation metrics in those grid cells. The first observation metric called *Dominance* has three possible values based on the number of players in the cell the ball resides:

- *Equal number of players*,
- *The opponent team has more players*, and
- *Our team has more players*.

The other observation metric is called *Closeness*. It also has three possible values

which are based on which player is the closest to the ball:

- *An opponent player is the closest*,
- *A teammate is the closest*, and
- *The robot itself is the closest*.

Therefore, the observation set includes three critical pieces of information about the environment: The location of the ball in the grid, the player which is the closest to the ball, and the team which is the dominant in the cell where the ball resides. There are 108 different observations. Those observation metrics have been developed by Meriçli [14].

$$Observation = \textit{Location} \times \textit{Closeness} \times \textit{Dominance}$$

4.1.2. Number of FSC Nodes

According to Eker and Akın [1] there is no significant difference between different number of *FSC* nodes for the problems they have tested. However, for the robot soccer problem, this may not be the case. Using an action for a state imposes a more strict policy. There might be a situation where two sequences of observations lead to the same action, but after this point although we want to choose two different actions, they must choose the same one since they take same observations. According to one action-one *FSC* node encoding, in spite of the fact that it is beneficial to choose different actions for the same observation, the same observation results in the same action. However, if we use more *FSC* nodes than the number of actions, there could be a strategy which can choose the same action and continue differently according its observations. This is possible because the strategy can take two different paths for the same action sequence.

To understand the effect of the number of *FSC* node to the performance of the algorithm, we trained a policy having 10 *FSC* nodes which is twice as many as the number of actions. Training is carried out only against the *SibHeteroG* team with the genetic algorithm parameters given in Table 4.2. However, as it can be seen in Table 4.1, increasing the number

Table 4.1. The Score Difference of a Policy Having Different Number of FSC Nodes.

Opponent	**5 Nodes**	**10 Nodes**	**20 Nodes**
NullTeam	11.39	3.90	9.07
BrianTeam	9.19	6.49	11.376
Kechze	2.44	2.69	3.648
SibHeteroG	1.01	0.98	1.268
AIKHomoG	0.98	1.41	1.436

Table 4.2. Parameters of the Genetic Algorithm.

Parameter	**Value**
Population Size	50
Mutation Rate	0.1
Crossover Rate	0.5
N_B : Number of Simulations Before Evolution	100
N_D : Number of Simulations During Evolution	50
N_A : Number of Simulations After Evolution	500
Fitness Function	Score
Maximum Number of Generations	50
Convergence Limit	20

of *FSC* nodes to 10 does not have a significant effect on the performance of the algorithm, but increasing it to 20 does have an effect on easy teams. However, it is important to note that increasing the number of *FSC* nodes, increases the length of the chromosome. Changing the number of *FSC* nodes have different effects on different teams. For example, 10 *FSC* nodes has the lowest score against *NullTeam*.

4.1.3. The Genetic Algorithm

When we define our problem as a *Dec-POMDP* and use a genetic algorithm as a search method, the quality of the solution is highly dependent on the parameters of the genetic algorithm. We determined the genetic algorithm parameters shown in Table 4.2 empirically.

The evolution cycle for training the *Dec-POMDP* team against a selected standard

team is as follows. The first population is initialized randomly. Then, the best chromosomes of evolution are calculated by running N_B simulations. In each generation, the fitness of the chromosomes in the population are calculated by running N_D simulations. At the end of every generation, the top 10 chromosomes of the population are taken and their fitness are recalculated by running N_B simulations. If any one of them is still good enough to be in the best chromosomes list, it is added to the list and the evolution continues. If the algorithm evolves more generations than the maximum number of generations or the best chromosome does not change for a number of generations, evolution ends. When the evolution ends, we calculate the best solution from the best chromosomes list by running N_A simulations.

4.1.3.1. Iterative Genetic Algorithm. When the policies are randomly initialized, none of the policies in the population scores goal against the good teams so that they all have the same fitness. We know that some policies are more successful at playing soccer, but they may not be able score at some particular games. Those chromosomes should be selected for the later generations. Therefore, to solve this problem, we train policies iteratively starting with the weaker teams and continuing with the stronger teams.

In all experiments, we use the standard TeamBots teams. There are four standard *TeamBots* teams with different strengths. We also add *NullTeam*, whose players are stationary on the field, and do nothing. Although we have general knowledge about the strength of teams, we wanted to know their relative strengths. All standard teams played 2000 evaluation games against each other. According to the experiment, the strength of teams is as follows starting from the most strong: *AIKHomoG*, *SibHeteroG*, *Kechze*, *BrianTeam*, and *NullTeam* as seen in Figure 4.3. In the Table 4.3, the result of *SibHeteroG* and *Kechze* against *NullTeam* are interesting. They are both 0.0 in spite of the fact that *NullTeam* has stationary players. When we study games, we find that due to their fixed strategies, the players of *SibHeteroG* and *Kechze* stick to the opponent players, and they are not be able to move away from them. Even the trained teams have this problem. This is due to inefficient basic actions.

Table 4.3. Average score differences of standard *TeamBots* teams.

Teams	AIKHomoG	SibHeteroG	Kechze	BrianTeam	NullTeam	Total
AIKHomoG	-	1.24	4.03	9.79	25.68	40.75
SibHeteroG	-1.24	-	0.51	9.79	0.0	9.10
Kechze	-4.03	-0.51	-	6.52	0.0	1.98
BrianTeam	-9.79	-9.79	-6.52	-	11.23	-14.87
NullTeam	-25.68	0.0	0.0	-11.23	-	-36.91

Table 4.4. The Performance of Iterative Training with Score Difference Fitness Method for FSC Representation.

Opponent	**Evaluation Runs**	**The End of Evolution**	**Best Score Difference**	**Win**	**Draw**	**Loss**
NullTeam	8.42	43.96	19	499	1	0
BrianTeam	7.04	22.9	13	500	0	0
Kechze	3.68	4.97	9	493	7	0
SibHeteroG	1.31	1.74	4	399	90	11
AIKHomoG	2.48	3.77	7	460	37	3
Meriçli *et al.* Team [23]	1.74	N.A.	6	421	78	1

Training is carried out in stages. We first train against the *NullTeam*, then against the other standard *TeamBots* teams, in the order of increasing difficulty. The population of a previous team is used for the next team except the *NullTeam* whose population is randomly initialized. The performance of the method can be seen in Table 4.4.

The difference between the average scores at the end of evolution and the average scores of 500 evaluation runs is high for weak teams such as *NullTeam*, and *BrianTeam*. Since the policies trained against those teams easily converge to successful policies which are a series of simple actions, the score of the evaluation run is lower than the score at the end of evolution for that team. This is an expected result since the final best policy is highly adapted to the last teams, and forgets some behaviors it learned against the initial (weaker) teams.

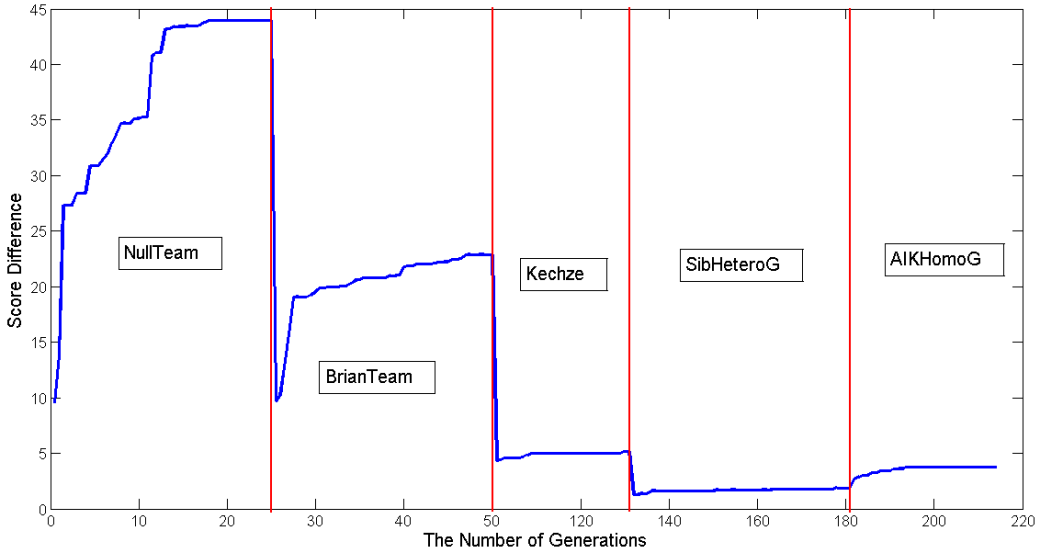

Figure 4.2. Evolution of Iterative Training.

One of the most important performance measures for the algorithm is the number of wins and losses. As it is seen in Table 4.4, the trained policy never loses against *NullTeam*, *BrianTeam*, *Kechze*, and loses only 11 games against *SibHeteroG*, 3 games against *AIKHomoG* out of 500 games. Although, the average score difference against *SibHeteroG*, and *AIKHomoG* is not very high, the number of wins are quite satisfactory.

In addition to the standard *TeamBots* teams, we also report the average scores against the team trained by Meriçli *et al.* [23]. Even though our team was trained only against the *TeamBots* teams we have a positive average score against the Meriçli *et al.* team. We win most of the games as seen in Table 4.4.

The evolution of the algorithm can be seen in Figure 4.2. When we look at the evolutions team by team, we can see that there is a sharp increase in first generations. This increase slows down very quickly especially for strong teams such as *Kechze*, *SibHeteroG*, and *AIKHomoG*.

4.1.4. Fitness Function

The main problem about the fitness calculation is that we try to estimate the fitness of a policy by taking many simulation runs. Therefore, we need to find the number of simulation runs which is sufficient to rank the chromosomes so that the genetic algorithm can converge. In Figure 4.3, we show the change in the rank of 50 chromosomes over the number of simulations. The change in rank is calculated by summing the change of all chromosomes between two consecutive runs. It is found that 50 simulation runs are sufficient for distinguishing a good solution candidate. After 50 simulations the rank of chromosomes does not oscillate much. However, we need to determine two additional numbers for simulation runs to achieve higher precision when deciding whether the policy is good enough to be kept as one of the best solutions, N_B, and when deciding what is the best of all best candidates, N_A. By considering running time limitations, we choose 100 simulation runs to decide whether a policy is good enough to be in the best chromosome list, and we choose 500 simulation runs to decide which is the best solution of the best chromosomes list.

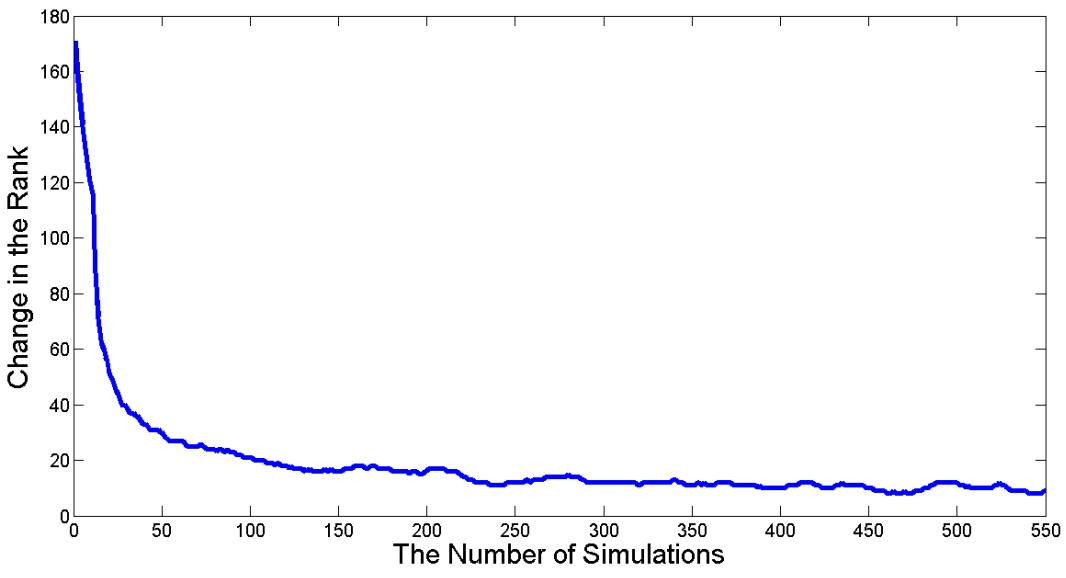

Figure 4.3. The Change in the Rank of Chromosomes by the Number of Simulations.

Another problem of fitness function specification is selectiveness. In robot soccer, the fitness of a policy can be calculated in different ways. One of the possible fitness calculation methods is the score difference. However, score difference may not be a good method since

Table 4.5. Metric Weights proposed by Meriçli [23].

Metric	**Weight**
Own Goal Vicinity Occupancy	3.745
Opponent Goal Vicinity Occupancy	1.266
Ball Vicinity Occupancy	1.124
Own Goal Separation Own	0.336
Own Goal Separation Opponent	−0.748
Opponent Goal Separation Own	−0.199
Opponent Goal Separation Opponent	−1.604
Ball Separation Own	2.116
Ball Separation Opponent	0.039
Own Area of Convex Hull	0.910
Opponent Area of Convex Hull	−0.425
Ball Possession	0.634
Score	1000

it may not be selective enough to differentiate a good soccer player policy from a bad one when their score is the same. Therefore, we experiment with different fitness methods for robot soccer.

4.1.4.1. Metric-Based Fitness Function. In a soccer game, the most important metric is the score difference. However, it is a delayed-reward. In genetic algorithms, delayed-reward fitness functions perform badly in terms of convergence since it may not provide the required selectiveness. To solve this problem, we propose to use a metric based fitness function. Therefore, we have information about fitness of a chromosome even if it cannot score. A series of metrics are calculated according to the positions of the players at every simulation step and the summation of these values is used as a fitness value. We use the metric weights proposed by Meriçli [23]. The details of metrics are given in Section 2.3.1.

In this experiment, we use an iterative approach with a weighted metric calculation. We train a policy starting with the easy opponents to hard opponents. The first opponent

Table 4.6. The Evaluation of the Performance of Weighted Metric Fitness Function.

Opponent	**Average Score Difference**	**Best Score Difference**	**Win**	**Draw**	**Loss**
NullTeam	8.88	20	449	51	0
BrianTeam	3.14	9	456	30	14
Kechze	-0.062	4	169	147	184
SibHeteroG	0.922	5	323	139	38
AIKHomoG	2.352	7	449	38	13

is *NullTeam*. *NullTeam* helps to learn how to score without interfering with any opponent. Then, we continue with *BrianTeam*, *Kechze*, *SibHeteroG* and *AIKHomoG*. We also include score difference in the weight calculation with 1000 weight value as seen in Table 4.5. We wanted the score difference to be the dominant factor in the fitness calculation. The score difference is multiplied by its weight at the end of the game and added to fitness value. Other metric based factors help the fitness function when the score difference is not selective enough. The metric weights are taken from the work of Meriçli [23]. To see the performance of the algorithm, we run 500 simulations against standard TeamBots teams with the trained policy, and we get the results seen in the Table 4.6. It is interesting to see that we are very good against two strong teams, namely *SibHeteroG* and *AIKHomoG*, but we lost most of the games against *Kechze* which is the third strongest team in the pool. It might be attributed to the policy which is lastly trained against *AIKHomoG* and, before that, trained against SibHeteroG. It might be well adapted to the specific behaviors of these two teams, but not to that of *Kechze*.

Although we have satisfactory results with this approach, the score difference fitness function is still the best method. This is due to the fact that score difference is the metric we decide whether training is successful or not. If it is able to rank chromosomes properly, it is expected to be the best fitness function. When we use iterative training, we solve the selectiveness problem and achieve satisfactory results.

4.1.4.2. Competitive Fitness Function. Competitive fitness function method is widely used in problems where there is an adversary. Also competitiveness is in the nature of a soccer

Table 4.7. The Evaluation of the Performance of Competitive Fitness Function.

Opponent	**Average Score Difference**	**Best Score Difference**	**Win**	**Draw**	**Loss**
NullTeam	13.97	26	498	2	0
BrianTeam	5.40	12	491	6	3
Kechze	0.45	6	232	116	152
SibHeteroG	0.87	5	308	130	62
AIKHomoG	1.49	6	374	76	50

game. The application of competitive fitness function is generally based on the competition of two separate populations, but we use *in population* competition. "The arms race" is the main driving force of the competitive evolution. At every evolution cycle, some individuals are selected based on their relative fitnesses. At the later cycles, new individuals are created according to the used genetic algorithm, and the fitness of individuals are again calculated according to this new population. In competitive fitness functions, we cannot impose any global fitness value since the best fitness value is highly dependent on the current population. Although the fitness of the population does not increase when we consider only the fitness values, at every evolution cycle the new individuals compete with the best of the current population, and if they are better, new individuals are reproduced from the best of the population. This process moves whole population to a higher global fitness value.

4.1.5. The Comparison of Dec-POMDP Algorithm with Reinforcement Learning Algorithm

Although there is no benchmark for the *TeamBots* simulation environment, in order to assess the performance of the method, the average scores are compared with the scores reported in [23]. Although the focus of the work reported in [23] is different from this work, both studies use the same *MDP* model and the simulation environment, i.e., the same basic actions, observations and simulator. They use the reinforcement learning approach with soccer metrics developed by Meriçli *et al.* [14]. In Table 4.8, we compare our results with the scores reported in [23]. Although, our average scores are lower, we achieve positive average scores against all teams and win most of the games against *SibHeteroG* whereas the

Table 4.8. The Comparison of Average Scores.

Opponent Team	**Average Scores of Dec-POMDP Based Approach**	**Average Scores of Reinforcement Learning Based Approach [23]**
NullTeam	8.42	28.25
BrianTeam	7.04	17.80
Kechze	3.68	12.67
SibHeteroG	1.31	-4.90
AIKHomoG	2.48	N.A.

reinforcement learning based team has a negative average score against *SibHeteroG*.

The *Dec-POMDP* algorithm performs better against the stronger teams such as SibHeteroG because *FSC* can represent more complicated coordinated behaviors compared to that of reinforcement learning.

4.2. Continuous Robot Soccer Dec-POMDP Problem

4.2.1. Robot Soccer Dec-POMDP Model

As already pointed out in Section 4.1.1, we do not need to define a complete model. It is enough to define a set of actions and set of observations. In this section, we model *TeamBots* robot soccer as a continuous robot soccer model which is previously modeled as a discrete robot soccer problem. We use the same discrete actions defined in Section 4.1.1. However, we change the observation model completely. We omit all observation metrics and directly define the position information of the teammates, opponents and ball as observation.

The number of agents is 5, one of them is a hand-coded goal keeper. Therefore, the algorithm learns the coordination of 4 agents.

Table 4.9. Parameters of the Genetic Algorithm.

Parameter	**Value**
Population Size	30
Mutation Rate	0.1
Crossover Rate	0.5
N_B : Number of Simulations Before Evolution	50
N_D : Number of Simulations During Evolution	10
N_A : Number of Simulations After Evolution	100
Fitness Function	Score
Maximum Number of Generations	50
Convergence Limit	10

The observation set is as follows:

- Position of teammates,
- Position of opponents,
- Position of the ball.

Every position information has *x* and *y* coordinate values. Therefore, for a 5 vs. 5 game there will be *22* real valued observations.

4.2.2. The Genetic Algorithm

The parameters of genetic algorithm in this case are changed. We determined the genetic algorithm parameters shown in Table 4.9 empirically. When we compare these parameters with that of Table 4.2, population size, number of simulations and converge limit is decreased to decrease the running time of the algorithm.

We use the iterative genetic algorithm detailed in Section 4.1.3.1. Training starts against the easy teams. Once training is completed against one team, a new training is started against a harder team.

Table 4.10. The Performance of Iterative Training with Score Difference Fitness Method for NN Representation.

Opponent	**Evaluation Runs**	**The End of Evolution**	**Best Score Difference**	**Win**	**Draw**	**Loss**
NullTeam	5.54	48.06	13	378	122	0
BrianTeam	4.70	15.6	11	493	6	1
Kechze	1.99	3.40	8	408	64	28
SibHeteroG	1.14	1.64	5	361	128	11
AIKHomoG	1.73	3.54	8	454	41	5

In this experiment, the neural network representation is used. Since we use 22 real-valued position information as the observation, we cannot use the *FSC* representation. We use a neural network having 22 input neurons and 5 output neurons, i.e. no hidden layers. We use a fully connected neural network architecture so that there are $22 \times 5 = 110$ weight values for an agent. A chromosome is a concatenation of policies of every agent. The length of chromosome for this experiment becomes $110 \times 4 = 440$.

In Table 4.10, we can see the performance of the algorithm with the neural network representation. The results are comparable to that of the algorithm with the *FSC* representation. However, this algorithm uses less expert knowledge than the *FSC* representation since it uses only the position information.

The evolution of the algorithm can be seen in Figure 4.4. In the initial generations, the score difference increases sharply especially against the easy teams. When evolution continues with the strong teams, it continues with small increases. When we compare Figure 4.2 and 4.4, we can see that evolution curves are similar.

4.3. Continuous Keepaway Soccer Dec-POMDP Problem

As a continuous robot soccer problem, we experiment with *Keepaway* problem which is a sub-problem of robot soccer. Here there are two kinds of players: keepers and takers. The keepers try to keep the ball as long as possible, whereas the takers try to take the control

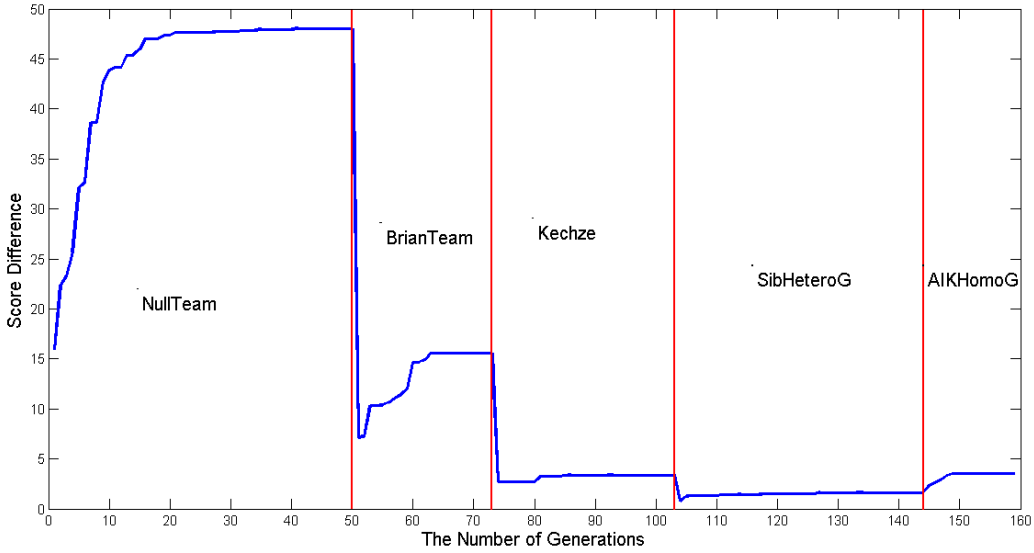

Figure 4.4. Evolution of Iterative Training.

of the ball as soon as possible. The *Keepaway Framework* provides a *SMDP* model with the required basic actions and observations. However, the only limitation is that an agent takes an action only if it possesses the ball. The other agents take the same action, namely *receive* action, while they do not have the ball [15]. Because the actions may take more than one time step, the model is called *SMDP*.

There are several published benchmark results for the *Keepaway Framework*, so that, we can compare and contrast our algorithm with the other algorithms. However, at the time of our experiments, keepaway agents can run with older version of *Robocup* 2D robot soccer simulator which cannot be compiled with the current compilers and libraries. Therefore, we used a modified version of the *Keepaway Framework* with the *Robocup* 2D robot soccer simulator 15.0.1.

4.3.1. Keepaway Soccer Dec-POMDP Model

Although the *Dec-POMDP* model requires many properties, most of them are not needed for the algorithm. The missing properties such as the transition function and the set of states are directly delegated to the simulation. Only the set of actions and the set of

observations are needed for the algorithm. In contrast to the discrete Dec-POMDP model, this algorithm uses observations without discretization.

The set of actions are as follows:

- *Keep the ball.*
- *Pass the ball to the closest teammate.*
- *Pass the ball to the* 2^{nd} *closest teammate.*
- *Pass the ball to the* $(n-1)^{th}$ *closest teammate.*

Therefore, there are n actions, same as the number of keepers.

The set of observations are as follows [15]:

- *Distances from players to center of region.*
- *Distances from other players to the keeper.*
- *Distances from teammates to their closest opponent.*
- *For each* K_i $(i = 2, .., n)$*, the minimal angle with the vertex at* K_1 *between* K_i *and an opponent, where* K_1 *is the keeper having the ball.*

Therefore, there are $4n + 2t - 3$ observation values, where n denotes the number of keepers, t denotes the number of takers. For example, for a problem having 3 keepers and 2 takers, there will be 13 observation values. A screenshot of *keepaway* can be seen in Figure 4.5. The blue robots are the takers and the yellow robots are the keepers. The white box denotes the border of the field. In this figure, there are 3 keepers and 2 takers in a 20×20 soccer field.

4.3.2. The Genetic Algorithm

Although most of the parameters are the same as Table 4.2, some parameters are changed for the *keepaway* problem as seen in Table 4.11. Especially N_B, N_D, and N_A parameters are decreased because of the running time restrictions. Also, the fitness function

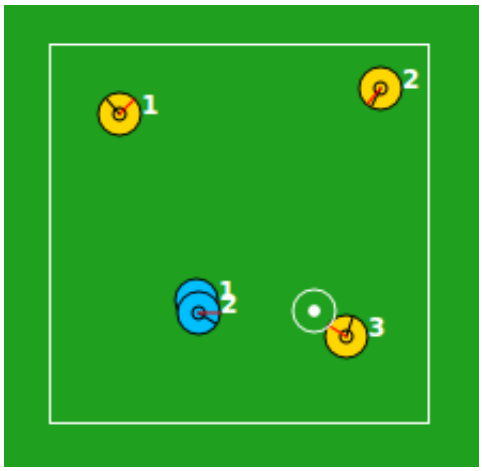

Figure 4.5. *Keepaway.*

Table 4.11. Parameters of the Genetic Algorithm.

Parameter	**Value**
Population Size	50
Mutation Rate	0.1
Crossover Rate	0.5
N_B : Number of Simulations Before Evolution	50
N_D : Number of Simulations During Evolution	10
N_A : Number of Simulations After Evolution	100
Fitness Function	Duration of the episode
Maximum Number of Generations	200
Convergence Limit	50

is changed as the duration of an episode.

4.3.2.1. Fitness Function. Since our algorithm estimates the fitness of a policy by running many simulations, there is a trade-off between the running time and accuracy. However, for the genetic algorithm to operate successfully, the chromosomes' rank should be stabilized. The stabilization of the rank of chromosomes require many runs, due to the nature of the *Keepaway* problem. For the hand-coded agent, the average duration is $13.3s$ and standard deviation is $8.3s$ [16]. The change in the rank of 50 chromosomes by the number of run can be seen in Figure 4.6. Similar to Figure 4.3, the change in the rank of chromosomes

sharply decreases as the number of simulations increases. By considering the running time constraints and the nature of the problem, we choose $N_B = 50$, $N_D = 10$, and $N_A = 100$.

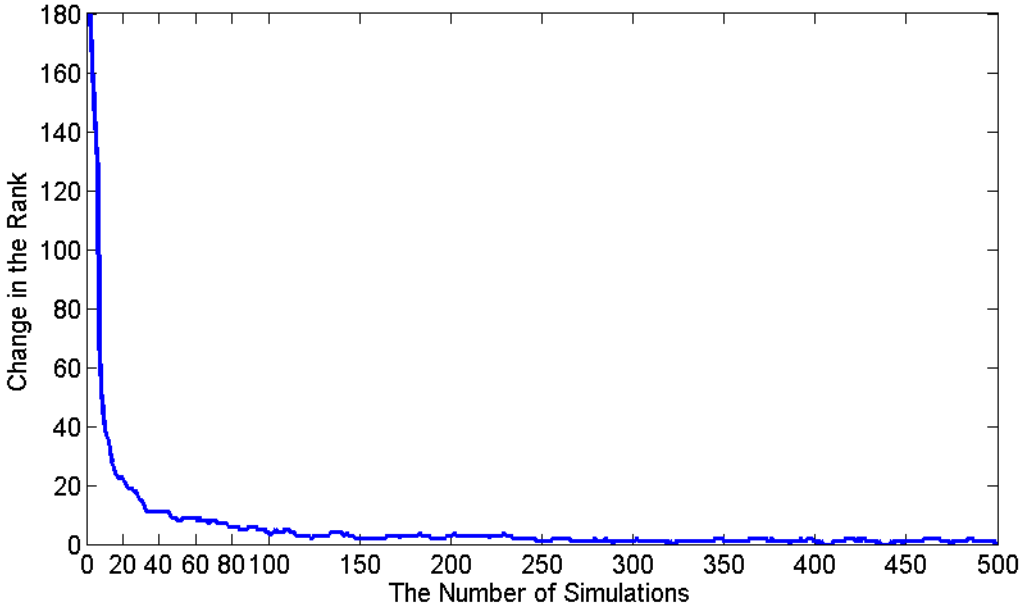

Figure 4.6. The Change in the Rank of Chromosomes by the Number of Simulations.

4.3.3. The Comparison of Dec-POMDP Algorithm with Reinforcement Learning Algorithm

After the *Keepaway Framework* was proposed as a learning benchmark, performances of a reinforcement learning algorithm was published [16]. Stone *et al.* use the Sarsa(λ) algorithm [28] with *CMAC* function approximation. We solved Keepaway with *Dec-POMDP* algorithm using a neural network representation. Since their results are published for the Robocup 2D robot soccer simulator version 11, and we test the algorithm on the version 15.0.1, we also publish the performances of benchmark policies in Table 4.12. As it can be seen in Table 4.12, we have comparable results. *Dec-POMDP* algorithm performs better than *hand coded* policy. When we compare the performance of Sarsa(λ) algorithm with *Dec-POMDP* algorithm, we can see that their performances are close.

When we investigate the *Hand-coded* algorithm detailed in Figure 2.1, we see that the algorithm calculates a metric based on the position information of the team mates and the opponents for every possible action. Then, it ranks those action values, and chooses

Table 4.12. The comparison of average possession times (in seconds) of hand-coded, random, always hold, and learning policies.

Policy	**Sarsa(λ) [16] Algorithm**	***Dec-POMDP* Algorithm**
Always Hold	2.9	3.0
Random	5.3	5.5
Hand Coded	13.3	14.1
Learning	15.0	16.0

the action having the highest value. It is expected that any neural network can learn such a policy. Also, the results in Table 4.12 verify our conjecture.

4.4. The Comparison of *FSC* and Neural Network Policy Representations

A *Dec-POMDP* algorithm is expected to map observation histories to actions. Since a *Dec-POMDP* agent does not know its state, its histories of observations have utmost importance. A series of observations may provide the required state information. However, the neural network representation does not have a mechanism to handle a series of observations, but *FSC* representation provides a mechanism to encode histories of observation. This is the most important feature of *FSC* representation since it can achieve this in a compact way compared to decision trees. Although Eker and Akın propose a method [29] to use observation histories for neural networks, their solution is not feasible for problems whose dimension of observation is big since they define previous observations also as an input to the neural network.

Another difference between the *FSC* and neural network policy representations is that *FSC* can be used to represent discrete *Dec-POMDP* models. On the other hand, neural network representation can be used to represent continuous *Dec-POMDP* models. There is a trade-off between using the neural network representation and discretization. For problems whose dynamics are studied extensively, it is easy to discretize and simplify the model, but this discretization will play a central role in the success of the solution. In our experiments, we see that *FSC* is better than the neural network representation. The results in Table 4.4 and results in Table 4.10 are close, but we should also note that *FSC* is considerably better than

Table 4.13. Game scores of 500 evaluation runs where *FSC* vs. neural network.

	***FSC* Team**	**Neural Network Team**
Average Score	0.002	-000.2
Win	32	30
Draw	438	438
Loss	30	32
Max Score	1	2
Min Score	0	0

neural network policy representation.

In Table 4.13, we show that neither *FSC* nor the neural network method outperforms each other. Most of the games end with a draw. This is an expected result since the tested policies are trained against the same teams. The trained teams learned the weakness of the same teams and developed strategies to make use of them. Therefore, when they play against each other, they cannot score since both policies try to do the very similar behaviors.

4.5. Discussion on the Learned Behaviors

4.5.1. *TeamBots* Simulator

When we examine the learned behaviors on the *TeamBots* simulator, we see that the player which is closest to the ball moves to grasp the ball. The other players get behind the player which controls the ball. We can call those players as *supporters*. If the player drops the ball, one of other players takes the control of the ball and becomes the *attacker*. This behavior is very advantageous in the *TeamBots* domain since the team does not lose its advantage when the opponent players tackle the attacker. This behavior can be seen in Figure 4.7.

Another learned behavior is predicting when the position of the ball is to be reset. This is one of the most useful behaviors since there are many ball resettings in a game. Ball resetting occurs when the ball's position does not change for 20 steps. The position of the

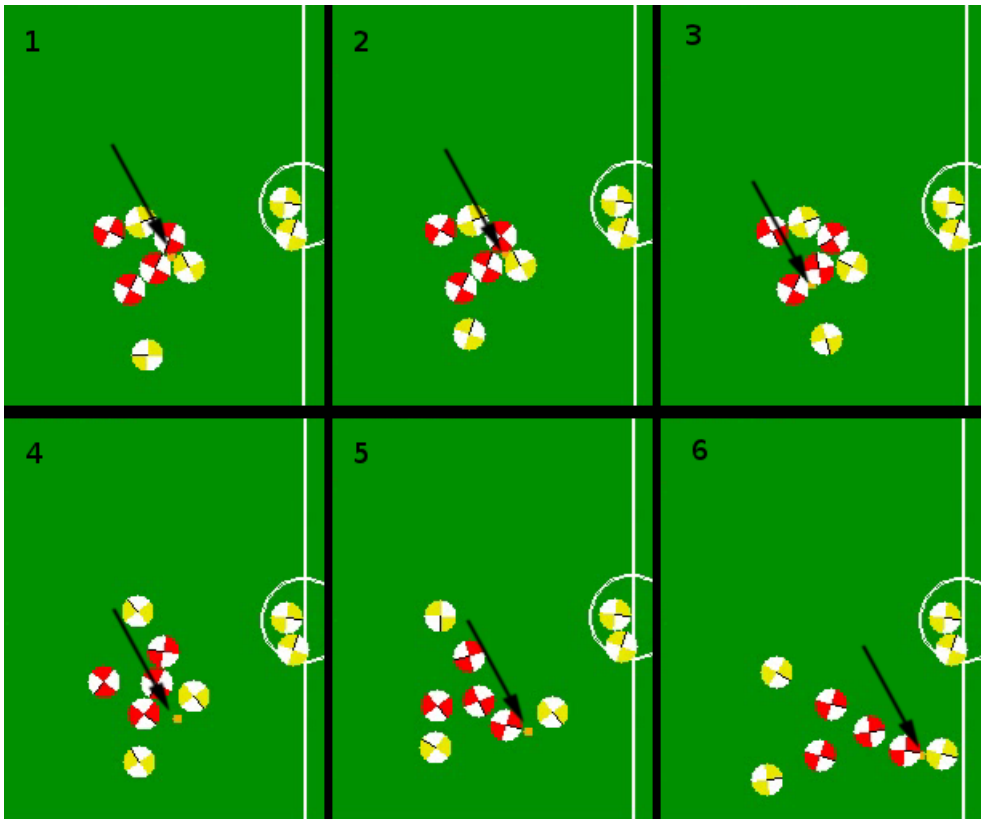

Figure 4.7. A Series of Screen shots Showing the Backup Behavior.

ball is changed according to its stuck position. Therefore, understanding that the game is stuck and predicting possible positions where the ball is to be moved has utmost importance. This way players gain control of the ball before the opponents as seen in Figure 4.8.

When we investigate both the neural network and *FSC* teams, we see that both policy representation methods learn similar behaviors. However, there is a crucial difference between the two methods. One of the players of *FSC* team takes a defensive role. As it can be seen in Figure 4.10, there is a dynamic role assignment. The neural network team does not have a defensive player, but one of the players take a supporter role dynamically as seen in Figure 4.9. A player goes to the opponent's goal to score a goal when the player which has the ball passes to the supporter.

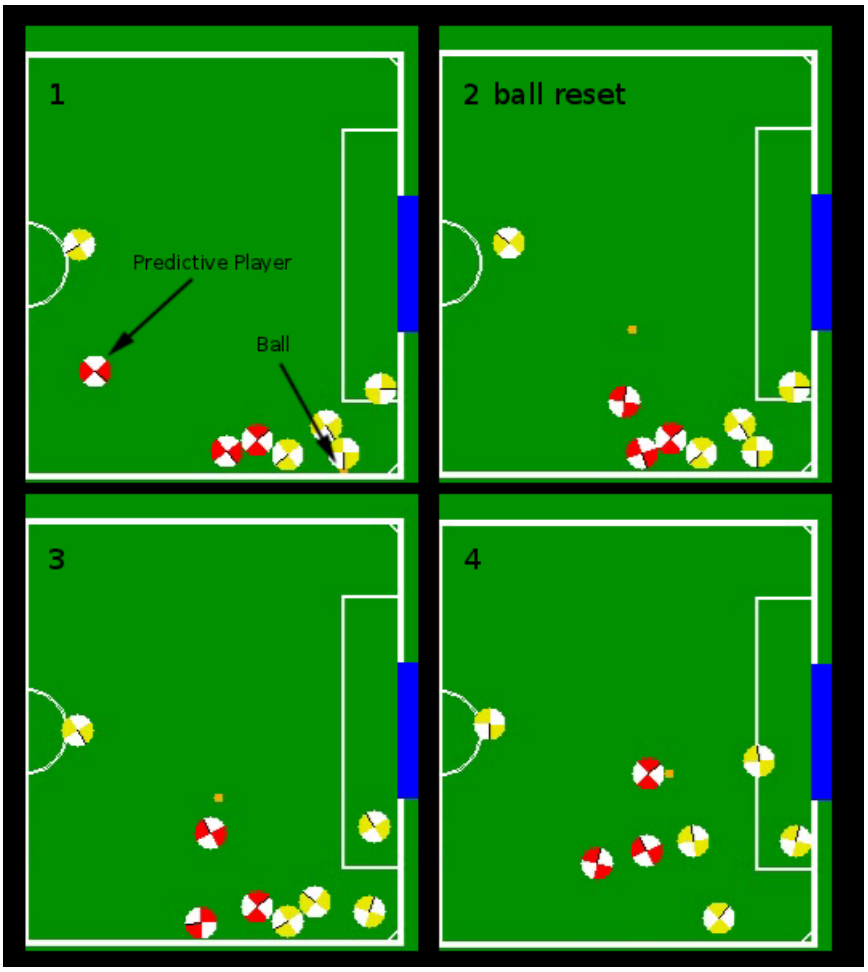

Figure 4.8. A Series of Screen shots Showing the Predictive Player Behavior.

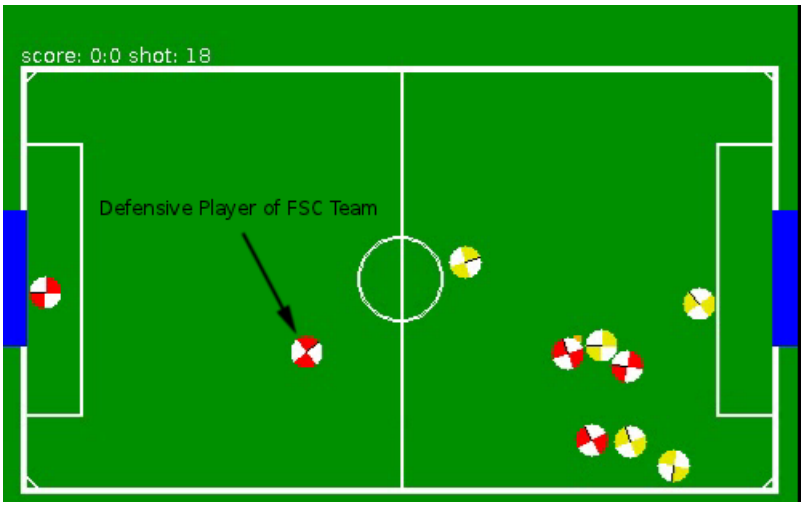

Figure 4.9. A Screen shot from a Game Showing Defensive *FSC* Team Player.

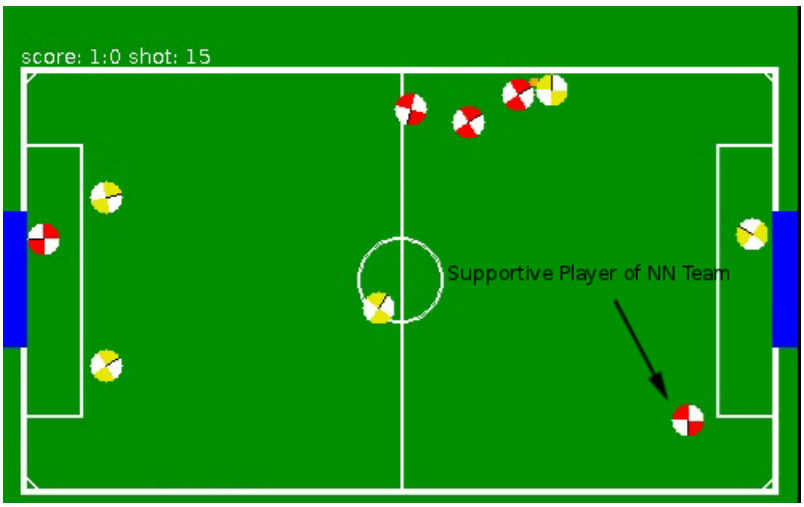

Figure 4.10. A Screen shot from a Game Showing Supportive Neural Network Team Player.

5. CONCLUSION

We implemented a scalable *Dec-POMDP* algorithm [1] to solve the multi-agent decision making problem in robot soccer. We used *TeamBots* and *Robocup* 2D robot soccer simulators. In the *TeamBots* domain, we solved decision making in 5 vs. 5 robot soccer games, and in *Robocup* simulator, we solved the keepaway problem. We show that the *Dec-POMDP* algorithm developed by Eker and Akın [1] gets better results than the reinforcement learning methods. We also compare performances of *FSC* and neural network policy representation methods.

We can group our experiments according to the simulators. In *TeamBots*, we first tune the parameters of the algorithm for robot soccer and show that the *Dec-POMDP* algorithm outperforms the reinforcement learning method. We propose an iterative genetic algorithm which starts learning from the easiest team to the hardest. The *Dec-POMDP* team wins most of the games against all standard *TeamBots* teams and the reinforcement learning team. However, since we used a genetic algorithm, our training time is longer than that of reinforcement learning. When we investigate the policies developed by the algorithms, we see that the *Dec-POMDP* algorithm provides better coordination. the players dominate the active play field by backing up the player which moves the ball forward.

One of the most important contribution of this work is the comparison of *FSC* policy representation and neural network policy representation. We show that they have similar results against standard *TeamBots* teams. Although we expect *FSC* to be better than the neural network representation, similarity of performance may be attributed the simulation environment. In other words, there might be two successful policies one is simple, and another one is complex. They may also produce similar behaviors originating from different dynamics.

There is a trade-off between neural network representation and *FSC* representation. If we use *FSC* representation, we gain a compact policy representation which can handle observation histories to some extent. Since *FSC* is encoded with integer numbers, it is

more appropriate for genetic algorithms. However, it needs a well-designed discrete *Dec-POMDP* model. The neural network representation does not need a well defined discrete *Dec-POMDP* model, but it cannot handle observation histories cheaply. Also, it is harder to optimize real-valued weights compared to integer-valued ones.

In *Robocup* simulator, we solve the keepaway problem. The *Keepaway Framework* is important to compare the performance of different learning algorithms. We solve keepaway soccer with the *Dec-POMP* algorithm having neural network representation, and have satisfactory results. The algorithm outperforms all benchmark policies and also reinforcement learning method.

5.1. Future Work

As a future work, to prove scalability of the algorithm, we plan to run the algorithm on real robots. Therefore, training will be done on a 3D simulator and the trained policy will be run on real robots.

We did not thoroughly experiment the effect of the number of *FSC* nodes to the performance of the algorithm. It is a future work to develop an algorithm which optimizes both the number of *FSC* nodes and *FSC* itself.

We intend to use alternative architectures like recurrent neural networks to further study the difference between *FSC* representation and neural network representation.

REFERENCES

1. Eker, B. and H. L. Akın, "Solving Decentralized POMDP Problems Using Genetic Algorithms", *Journal of Autonomous Agents and Multi-Agent Systems*, 2012.

2. Bernstein, D. S., R. Givan, N. Immerman and S. Zilberstein, "The Complexity of Decentralized Control of Markov Decision Processes", *Mathematics of Operations Research*, Vol. 27, No. 4, pp. 819–840, Nov. 2002.

3. LaValle, S. M., *Planning Algorithms*, Cambridge University Press, Cambridge, U.K., 2006.

4. Sutton, R., D. Precup and S. Singh, "Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning", *Artificial Intelligence*, Vol. 112, pp. 181–211, 1999.

5. Seuken, S. and S. Zilberstein, "Formal models and algorithms for decentralized decision making under uncertainty", *Autonomous Agents and Multi-Agent Systems*, Vol. 17, No. 2, pp. 190–250, Oct. 2008.

6. Bernstein, D. S., E. A. Hansen and S. Zilberstein, "Bounded Policy Iteration for Decentralized POMDPs", *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pp. 1287–1292, Edinburgh, Scotland, 2005.

7. Seuken, S. and S. Zilberstein, "Improved Memory-Bounded Dynamic Programming for Decentralized POMDPs", *Proceedings of the Twenty-Third Conference on Uncertainty in Artificial Intelligence*, pp. 344–351, Vancouver, British Columbia, 2007.

8. Bernstein, D. S., E. A. Hansen and S. Zilberstein, "Bounded Policy Iteration for Decentralized POMDPs", *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pp. 1287–1292, Edinburgh, Scotland, 2005.

9. Poupart, P. and C. Boutilier, "Bounded Finite State Controllers", S. Thrun, L. Saul and B. Schölkopf (Editors), *Advances in Neural Information Processing Systems 16*, MIT Press, Cambridge, MA, 2004.

10. Akın, H. L., "Evolutionary Computation: A Natural Answer to Artificial Questions", *Proceedings of ANNAL: Hints from Life to Artificial Intelligence*, pp. 41–52, METU, Ankara, 1994.

11. Nelson, A. L., G. J. Barlow and L. Doitsidis, "Fitness functions in evolutionary robotics: A survey and analysis", *Robot. Auton. Syst.*, Vol. 57, No. 4, pp. 345–370, Apr. 2009.

12. Mackworth, A. K., "On Seeing Robots", *Computer Vision: Systems, Theory, and Applications*, pp. 1–13, World Scientific Press, 1992.

13. Robocup Federation, "Robocup Federation", `http://www.robocup.org/`, accessed at June 2012.

14. Meriçli, Ç. and H. L. Akın, "A Layered Metric Definition and Evaluation Framework for Multirobot Systems", *RoboCup 2008: Robot Soccer World Cup XII*, pp. 568–579, Springer, 2009.

15. Stone, P., G. Kuhlmann, M. E. Taylor and Y. Liu, "Keepaway Soccer: From Machine Learning Testbed to Benchmark", A. Bredenfeld, A. Jacoff, I. Noda and Y. Takahashi (Editors), *RoboCup*, Vol. 4020 of *Lecture Notes in Computer Science*, pp. 93–105, Springer, 2005.

16. Stone, P., R. S. Sutton and G. Kuhlmann, "Reinforcement Learning for RoboCup Soccer Keepaway", *Adaptive Behavior*, Vol. 13, No. 3, pp. 165–188, 2005.

17. Wu, F. and X. Chen, "Solving Large-Scale and Sparse-Reward DEC-POMDPs with Correlation-MDPs", U. Visser, F. Ribeiro, T. Ohashi and F. Dellaert (Editors), *RoboCup*, Vol. 5001 of *Lecture Notes in Computer Science*, pp. 208–219, Springer, 2007.

18. Stone, P., R. S. Sutton and S. P. Singh, "Reinforcement Learning for 3 vs. 2 Keepaway", *RoboCup 2000: Robot Soccer World Cup IV*, pp. 249–258, Springer-Verlag, London, UK, UK, 2001.

19. Whiteson, S., N. Kohl, R. Miikkulainen and P. Stone, "Evolving Soccer Keepaway Players Through Task Decomposition", *Machine Learning*, Vol. 59, pp. 5–30, 2005.

20. Pietro, A. D., L. While and L. Barone, "Learning In RoboCup Keepaway Using Evolutionary Algorithms", W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke and N. Jonoska (Editors), *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1065–1072, Morgan Kaufmann Publishers, New York, 9-13 Jul. 2002.

21. Haykin, S., *Neural Networks: A Comprehensive Foundation*, Pearson Education, NY, Second edn., 1998.

22. Balch, T., "TeamBots Mobile Robot Simulator", `http://www.teambots.org`, accessed at June 2012.

23. Meriçli, Ç., T. Meriçli and H. L. Akın, "A Reward Function Generation Method Using Genetic Algorithms: A Robot Soccer Case Study", *9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, 2010.

24. Meriçli, T., *Braitenberg Soccer: Learning How to Play Soccer with Primitive Behaviors*, Master's Thesis, Boğaziçi University, 2008.

25. Meriçli, Ç., *Developing A Novel Robust Multi-Agent Task Allocation Algorithm for Four-Legged Robot Soccer Domain*, Master's Thesis, Boğaziçi University, 2005.

26. Robocup Federation, "The Robocup Soccer Simulator", `http://sourceforge.net/apps/mediawiki/sserver/`, accessed at June 2012.

27. Meffert, K. and N. Rotstan, “Java Genetic Algorithm Package”, `http://jgap.sourceforge.net/`, accessed at June 2011.

28. Sutton, R. and A. G. Barto, *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA, 1998.

29. Eker, B. and H. L. Akın, “Using evolution strategies to solve DEC-POMDP problems”, *Soft Computing-A Fusion of Foundations, Methodologies and Applications*, Vol. 14, No. 1, pp. 35–47, 2010.