A Hybrid Genetic Programming Based Decision Making System for Multi-Agent Systems of RoboCup Soccer Simulation

by

© *Amir Tavafi*

A thesis submitted to the

School of Graduate Studies

in partial fulfilment of the

requirements for the degree of

Master of Science

Department of Computer Science

Memorial University of Newfoundland

April 2017

St. John's Newfoundland

Abstract

In this thesis, a hybrid genetic programming approach is proposed for decision making system in the complex multi-agent domain of RoboCup Soccer Simulation. In the past, genetic programming was rarely used to evolve agents in this domain due to the difficulties and restrictions of the soccer simulation domain. The proposed approach consists of two phases, each of which tries to cover the other's restrictions and limitations. The first phase will produce some evolved individuals based on a GP algorithm with an off-game evaluation system and the second phase will use the best individuals of the first phase as input to run another GP algorithm to evolve players in the simulated game environment where evaluations are done during real-time runs of the simulator. It is observed that the individuals evolved after the second phase are able to outperform the same team with a decision making system which is not evolved.

Acknowledgements

I would first like to thank my supervisor Prof. Wolfgang Banzhaf of the Computer Science Department at Memorial University of Newfoundland. The door to Prof. Banzhaf office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this paper to be my own work, but steered me in the right the direction whenever he thought I needed it.

I thank all my friends for their continuous support through all the challenges that I faced throughout my academic life at Memorial University of Newfoundland.

Finally, I must express my very profound gratitude to my family and my girlfriend for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A multi-agent system is a system in which there are multiple interacting intelligent agents within an environment [10]. These systems can be used in different domains for solving problems where a single agent is unable to achieve the defined goal or reaching it might be time consuming and inefficient. The problem of decision making in multi-agent systems is a complex one. Each agent in a multi-agent system needs to have an independent decision making system in order to be able to interact with other agents and also the environment. This communication between agents is limited in each environment and the amount of information that they can give or receive from each other has a limit per each cycle depending on the environment type.

Each agent has different functions and skills that it can execute in each cycle or a period of time. In order to execute the best action for reaching the goal of the system, there are two main problems. First, the functions and skills of an agent need to be optimized to reach the best possible outcome. Second, when all functions and skills are available, a good decision making system is required so that the skills and

functions are used at the right time so the goal can be reached in an efficient manner. The problem of generating a good decision making system when we have the functions and skills available is the problem that is considered for this thesis. For this purpose, a hybrid genetic programming based approach is proposed for improving the outcome of an agent's actions.

Genetic Programming (GP) is one of the research areas within the field of Evolutionary Computation (EC) [5], which generates programs and algorithms through simulated evolution. In Genetic Programming, computer programs are evolved during the generations that the algorithm is running. Each computer program in GP is commonly represented as a tree which is defined using a set of terminals and functions depending on the problem definition. Crossover and mutation operators are used for evolving individuals through generations. Evaluation of trees in GP is done by recursively traversing the corresponding tree in the evaluation method that is defined depending on the problem statement.

Soccer 2D Simulation is one of the main leagues of the official RoboCup competitions that many researchers from around the world participate in. It is also one of the oldest leagues of the competitions, running since 1997, the first year of the official RoboCup competitions. The Soccer 2D Simulation Server (RCSSServer) [22] is the software in the Soccer 2D Simulation league that provides a complex multi-agent system allowing groups to develop their own soccer teams of 11 individual agents and play against another team. Intelligent soccer teams of agents developed by different groups of researchers can play against each other using the RCSSServer in order to test the intelligence of their agents and their strategies. This domain offers an integrated research task covering broad areas of AI and robotics such as: real-time

sensor fusion, multi-agent systems, strategy acquisition, machine learning, real-time planning, pattern recognition, vision, strategic decision-making, motor control and intelligent robot control.

RCSSServer is the domain used for testing our proposed method of decision making. This thesis and the implementations done for testing the proposed method are based on the code of the award winning team named *MarliK* [29]. I was a member of this team from when it was founded in 2005 and I was acting as leader of the group from 2008 [29]. *MarliK* was placed 3rd in the world two times in official events of RoboCup 2011 in Istanbul, Turkey [9] and RoboCup 2012 in Mexico City, Mexico [8]. *MarliK* was also placed 1st in many international RoboCup Open competitions in the Netherlands and Iran during the years 2009-2014. The motivation for the proposed method for decision making is that *MarliK* has really good quality high-level skills for its agents such as shoot, pass, dribble, block and mark, but lacks a stable decision making system, which was always felt a deficit during the past years. With the use of an optimized static decision making system, we won the award in the Drop-in Player Challenge of RoboCup 2013 in Eindhoven, Netherlands [20]. In the Drop-in Player Challenge, a random number of agents from each team were placed in a "super team" which consisted of different players from different teams in the competitions. They played against other randomly chosen players and the challenge was for the agents of each team to be able to co-operate well with their new unknown teammates. Our agents managed to get the highest scores and placed first in this challenge. A detailed analysis of the results and performance of agents from each team is available in [20].

My teammates and I were always struggling to find an efficient way of getting the best out of the available skills in our team's code in order to improve the gameplay

stability and to be able to score more goals and also suffer less goals from opponent teams. During several competitions and through discussions with other teams it was found that this is one of the main problems of many teams participating in these competitions. I am proposing this new method which uses genetic programming and I am hoping to be able to improve agent behavior in the soccer simulator domain using evolution. Because of the similarity of the domain of soccer simulation with other real or simulated environments and the flexibility of the proposed method, it is expected that it can be modified to be used in other domains as well.

In this thesis, background information about the RoboCup Simulation environment and Genetic Programming is provided in Chapter 2. Chapter 3 provides information about related work using Evolutionary Algorithms in the soccer simulation framework. Chapter 4 explains the hybrid GP method for decision making in our simulated soccer agents. Chapter 5 details implementation of the proposed method. Chapter 6 shows the results of this research and compares results in different scenarios. Finally, conclusions and future suggested work are discussed in Chapter 7.

Chapter 2

Background

This chapter will provide background information on the problem domain and the proposed approach in the next chapters. RoboCup and the Soccer Simulation Server are detailed first, then an introduction to genetic programming and its configurations and operators are discussed.

2.1 RoboCup Soccer Simulation

RoboCup (Robot World Cup) is held every year by the official RoboCup federation, with the objective of promoting robotics and research in the area of Artificial Intelligence (AI). The mission of this competition, set in the first year of the event in 1997, is to build a team of robots that will be able to play and win against the human soccer champion team of the Soccer World Cup by year 2050 [15]. In order to achieve this mission, researchers are focusing on different parts of building a robot, in different leagues of the RoboCup competitions, which is held each year since 1997. RoboCup also expanded into other relevant application domains in order to help ful-

fil the needs of our society. The main divisions of RoboCup competitions, each of which including different leagues, are: RoboCup Soccer, RoboCup Rescue, RoboCup @Home, RoboCup Junior.

RoboCup soccer is divided into two main subdivisions which are (i) physical leagues and (ii) simulation leagues. In the physical leagues of RoboCup soccer, individuals or teams of robots play against each other in a real environment under the rules of the league. As of 2016, the physical leagues of RoboCup soccer include Standard Platform League (formerly known as Four Legged League), Small Size League, Middle Size League, Humanoid League (including kid size, teen size, and adult size robots). Soccer Simulation leagues are divided into 2 sub-leagues, known as 2D and 3D. Soccer 2D Simulation is one of the oldest leagues of the competition in place from the first year of the event in 1997. In this league, two teams of eleven autonomous software programs known as agents play soccer against each other in a two-dimensional virtual environment called Soccer Server. The framework used in this thesis for implementing and testing the proposed approach is RoboCup Soccer Simulation Server (RCSSServer) from Soccer 2D Simulation league that is detailed in the following sections.

The RoboCup Soccer Simulator consists of three main parts which are:

- Soccer Server
- Soccer Monitor
- Logplayer

Soccer Server is the main software which is designed to model the simulated soccer environment. The project is open source and it is available for download on

Figure 2.1: RoboCup Soccer Monitor

SourceForge.net[1]. All of the information related to sensors, actors, noise-generating algorithms, and the patterns used for communication between agents are created, processed and sent to agents by soccer server. The soccer simulation system includes two types of monitors that communicate with the soccer server and allow to visualize the information received from the server offline or online. Online communication with the server is done by the soccer monitor which visualizes the soccer games while they are being played on the server. An example of a game running on the soccer server and shown on the soccer monitor is presented in Figure 2.1. Everything that happens in the simulated environment caused by server and agents can be visualized and shown to users on the soccer monitor at the same time. Logplayer is used for analyzing a

[1]https://sourceforge.net/projects/sserver/files/

game in order to study behavior of agents after the game is over. Users will have the option of watching the game and applying controls such as pausing, reversing, and speeding up the game, in order to extract needed information from Logplayer. Figure 2.2 shows a sample screen of Logplayer. Here, details of the soccer server will be discussed, as well as rules of the simulated environment and the simulated field's properties.

Figure 2.2: RoboCup Soccer Logplayer

Soccer Simulation Server is a software that allows 11 agents of two different teams to play against each other in a simulated soccer field. The framework creates a rich multi-agent environments that can be used for many different areas of research, such as Artificial Intelligence (AI) or Evolutionary Algorithms (EA). Each agent connects

to the server as a separate process and interacts with it as a client. The server provides the soccer field and simulates movement of ball and agents in the field. Connection between each agent and server is established via UDP/IP protocol and any programming language that supports UDP/IP can be used to talk to server from client side. Each team can use up to 11 player clients connected to the server. Each of these 11 processes is able to act as the "brain" of one agent and can only talk to other agents through the server. This makes the soccer simulator a standard multi-agent environment.

The server uses some parameters to simulate the environment so that it is more similar to the real world. Getting to know some of these parameters will help to understand the system better. A list of important parameters of the server is listed in Table 2.1.

2.1.1 Rules

Rules in Soccer Simulation fall into two main categories. The first category are rules that can be automatically recognized by calculations of the server. The second type are the rules that the system cannot recognize which are controlled by a human referee (members of technical committee of each competition).

2.1.1.1 Server Rules

The rules which can be controlled through mathematical or logical calculations are controlled by the server. Mathematically recognizable rules include: offside, keeping 9.15m distance from opposing players when a team is taking a free kick, and controlling the timing of the game, such as handling half times and extra time. Logical

Table 2.1: Some important parameters of soccer simulation server

Parameter Name	Value in current version of server	Description
version	15.3.0	Current version of server
goal_width	14.02 (m)	Width of each goal in the field
ball_size	0.085 (m)	Size of the ball
ball_decay	0.94	Decrease rate in speed of ball after each cycle
ball_speed_max	3.0 (m/cycle)	Maximum speed of ball
ball_accel_max	2.7 (m/cycle)	Maximum acceleration of ball
player_size	0.3 (m)	Size of each player
player_decay	0.4	Player speed decrease rate for each cycle
player_speed_max	1.05 (m/cycle)	Maximum speed of each player
player_accel_max	1.0 (m/cycle)	Maximum acceleration of each player
stamina_max	8000.0	Maximum amount of stamina for each player
stamina_inc_max	45.0	Maximum increase of stamina per each cycle
max_dash_power	100.0	Maximum speed of dash for players
max_dash_angle	180 (degrees)	Maximum angle of dash for players
maxneckang	90 (degrees)	Maximum angle that player's neck can turn
visible_angle	90 (degrees)	Maximum angle player can see each cycle
port	6000	Port number that players should connect to
say_msg_size	10 (bytes)	Size of string player can say
simulator_step	100 (msec)	time step of each cycle
half_time	300 (sec)	Length of each half time of game

rules include different game modes such as: goal kicks, kick ins, corner kicks, and play on. The server can be considered a deterministic finite automaton that handles recognizable possible faults in the game.

2.1.1.2 Human Rules

These types of rules do not interfere with the first type of rules and are not recognizable by the server. Often, they are against the fair play rules and will be handled by a human referee. These rules include: surrounding the ball with lots of players, lining up lots of players in front of the goal, blocking opponent players intentionally when they don't have the ball, bombarding the network with more commands than allowed, and some more.

2.1.2 Features of the Soccer Field

The simulated soccer field is a two dimensional flat area with dimensions of 68×105 meters. The width of each goal is 14.02 meters, almost twice the size of a real goal. Agents and ball are modeled as circles in the simulation and all of the relative distances and angles are calculated using the center of these circles. The center of the field is considered as the coordinate origin. The X-axis has a range of -52.5 to 52.5 and the Y-axis from -34 to 34. There are 59 flags placed throughout and around the field with pre-defined fixed locations that help agents find their position in the field. All of the flags are shown in the Figure 2.3.

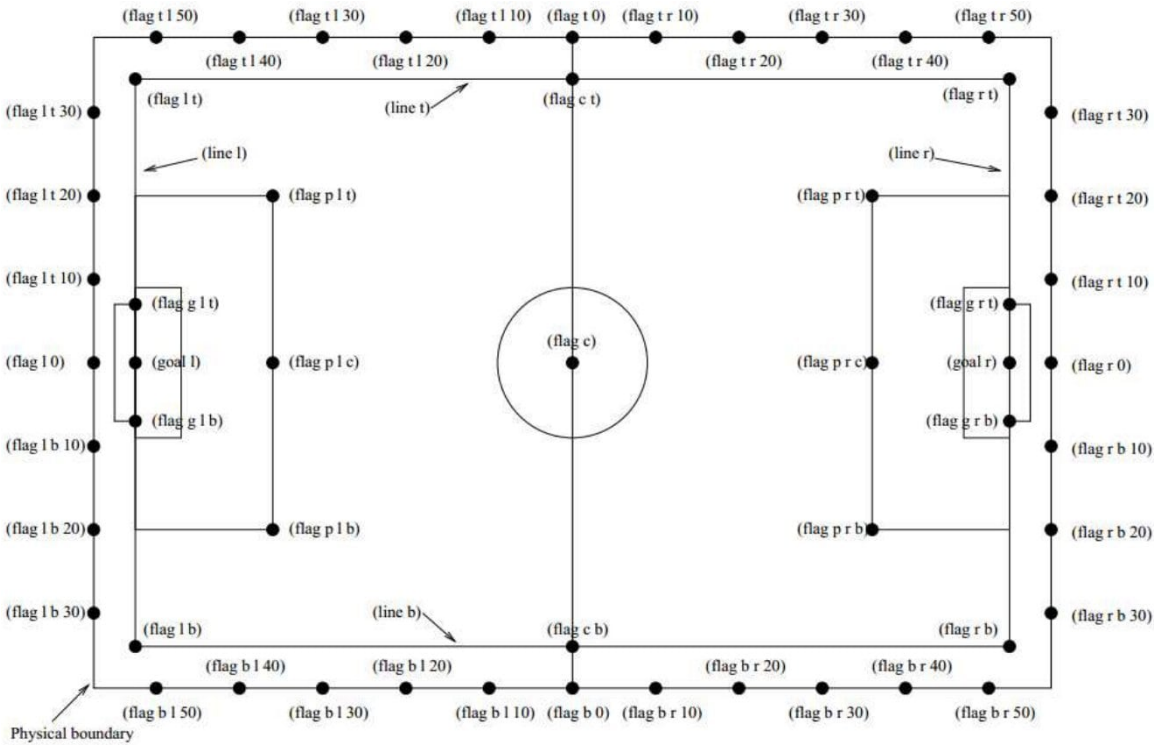

Figure 2.3: Location of all of the flags of the soccer simulation server [7]

2.1.3 Sensor, Movement, and Action Models

An agent in the simulated framework needs to have a two-way communication with the server so that it can act similar as in the real world. In other words, an agent should be able to receive useful information that can be a motivator for making decisions to act in the field, as a response to the events of the environment. Creating this two-way interaction between client and server should be supported by the server. In the following sections, some of the models for creating this interaction are reviewed.

2.1.3.1 Sensor Models in Server

The soccer simulator uses three sensor models in order to transfer information from the environment to an agent so that it will be aware of everything happening around it during the game. These three models cover visual sensors, aural sensors, and body sensors. In this section, each of these sensor models are briefly discussed.

- **Visual Sensors:** Information visible on the field is transmitted to a player through visual sensors. This information refers to all of the moving and fixed objects visible in the field, including other players of the same team and of the opponent team, the ball, and all of the flags. This information is received relative to the current position of an agent so that it will not receive its exact location in the field. In order to calculate an accurate approximation of its location, the player must use the flags in the field.

  Each player has a limited visual field and can see within an angle and a distance according to its current neck angle and view mode. A player is able to sense objects which are out of its sight in a small radius as well. Visual information is sent from the server in a time step (between 1 and 3 cycles) depending on the view mode of the agent. An agent has three different view mode options that it can select depending on the situation of the game. There are three parameters that change depending on the view mode the agent is using: view quality, time to receive visual information, and view angle. The three view modes are called narrow, normal, and wide, and their view angles are 90, 120, and 180 degrees. The noise level of the information received from server will increase as the view angle gets wider. The furthest object that a player can see is 60 meters away.

Figure 2.4 shows visual model of agents in more detail. As depicted in Figure 2.4, agents *A, E,* and *G* are using the narrow view mode in this specific cycle of the game. Agents *C, D, F, H,* and *I* are using the normal view mode and agents *B* and *J* are using the wide view mode. More details of the view model are shown for agent *A* demonstrating that in this cycle it can only receive visual information of the ball object and agents *C, D*, and *E* because these are the only objects within its view angle. The visible angle of agent *A* v and d is the small distance around the agent within which it can perceive all objects regardless of its visible angle.

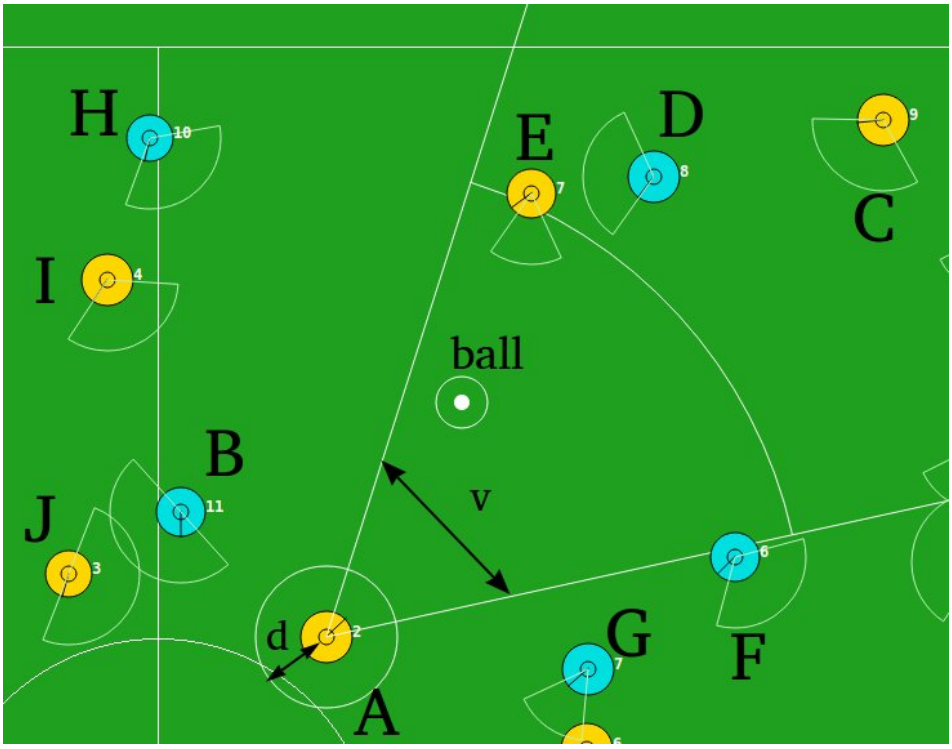

Figure 2.4: Visual model of a soccer agent

- **Aural Sensors:** The soccer server simulates a busy environment with a low

bandwidth where all agents are able to communicate directly with each other only through one way of communication [7]. In this environment, each agent is able to receive and process the information shared through the server from all other agents. This information is collected by the server and then broadcast with no delay to all agents not further away than *audio_cut_dist*. A message sent by an agent is in the form of a string not longer than a pre-defined length named *say_msg_size*.

- **Body Sensors:** The body sensor model of the soccer server includes all physical information of agents such as speed, stamina, body angle, and neck angle. Body information is received from the server in time steps of *sense_body_step*. The body sensor report is sent by the server in the following format:

  (sense_body Time

  (view_mode ViewQuality ViewWidth)

  (stamina Stamina Effort)

  (speed AmountOfSpeed DirectionOfSpeed)

  (neck_angle NeckDirection)

  (kick KickCount)

  (dash DashCount)

  (turn TurnCount)

  (say SayCount)

  (turn_neck TurnNeckCunt)

(catch CatchCount)

(move MoveCount)

(change_view ChangeViewCount)

where:

Time = The number of cycle that this message was transmitted.

ViewQuality = {high, low}

ViewWidth = {narrow, normal, wide}

Stamina = A positive integer between 0 and *stamina_max*.

Effort = An integer between *effort_min* and *effort_max*.

AmountOfSpeed = An approximation of speed of the agent.

DirectionOfSpeed = An approximation of the angle of agent's speed.

NeckDirection = Direction of agent's neck relative to its body.

Count variables = All the variables that have a Count show the total number of that action that was executed by server for this agent so far.

2.1.3.2 Movement Model

For each simulation cycle, each agent's position as well as the ball position are calculated using their last cycle's position and the body action that was sent by each client to the server. In the following calculation, the position of an object in cycle t is written as (p_x^t, p_y^t) and the velocity of an object in cycle t is written as (v_x^t, v_y^t). Acceleration of objects is expressed with (a_x^t, a_y^t), the move vector of objects for each

cycle is written as (u_x^t, u_y^t), and *decay* is either equal to *player_decay* or *ball_decay* depending on the type of object. The parameter *decay* is applied for calculation of velocity of the objects. For example, if velocity of the ball is 3.0 in one cycle and *ball_decay* is 0.94, velocity of the ball will be 2.82 in the next cycle if it is not kicked by an agent.

$$(u_x^{t+1}, u_y^{t+1}) = (v_x^t, v_y^t) + (a_x^t, a_y^t) : accelerate \tag{2.1}$$

$$(p_x^{t+1}, p_y^{t+1}) = (p_x^t, p_y^t) + (u_x^{t+1}, u_y^{t+1}) : move \tag{2.2}$$

$$(v_x^{t+1}, v_y^{t+1}) = decay \times (u_x^{t+1}, u_y^{t+1}) : decay\ speed \tag{2.3}$$

$$(a_x^{t+1}, a_y^{t+1}) = (0, 0) : reset\ acceleration \tag{2.4}$$

Acceleration of an object is updated with the actions from the previous cycle. Acceleration is created and changed by the *dash* action for agents and by the *kick* action for the ball object. If two objects collide, they will be pushed back in the direction that they approached each other and their speed will be reduced to 10% of their previous speed. A list of server parameters used for the movement model is provided in Table 2.2.

2.1.3.3 Action Model

The action model includes all of the actions that an agent is allowed to perform in the simulated soccer environment. In each cycle, an agent is allowed to choose from one of the main actions *catch, dash, kick, move,* or *tackle*, as well as one optional command for each action *say, turn, turn_neck*, and *change_view*. A few of the important actions are detailed below:

Table 2.2: Movement parameters of soccer server and their values

Parameter Name	Value	Description
ball_decay	0.94	Decay of the ball object
player_decay	0.4	Decay of the player object
ball_weight	0.2	Weight of the ball object
player_weight	60.0	Weight of the player object

1. **catch:** This action is only allowed for the goalie agent of each team and is executed only when the game is in *play_on* mode. The catchable area of the goalie is slightly bigger than the kickable area as shown in Figure 2.5. After each time that a goalie performs a *catch* action, there is a time period of *catch_ban_cycle* during which the server will not allow the goalie to execute a *catch* action again. This parameter is to simulate the actual catch action of goalies in a real game: After performing a dive, the goalie needs some time to get up and try to perform another one. In the current version of the simulator *catch_ban_cycle* is set to 5 and *catchable_area* is 1.2 meters.

2. **dash:** This is the action that allows players to accelerate and move in a certain direction in the soccer field. In earlier versions of the soccer simulator, an agent was only allowed to perform *dash* and accelerate in its body direction to the front or back. Since 2009, the *dash* model has changed and one parameter was added allowing players to *dash* in different angles relative to their bodies. *dash* is the only action that reduces an agent's *stamina*. An agent's *stamina* recovers in each cycle that the agent does not perform a powerful *dash*. The format of a

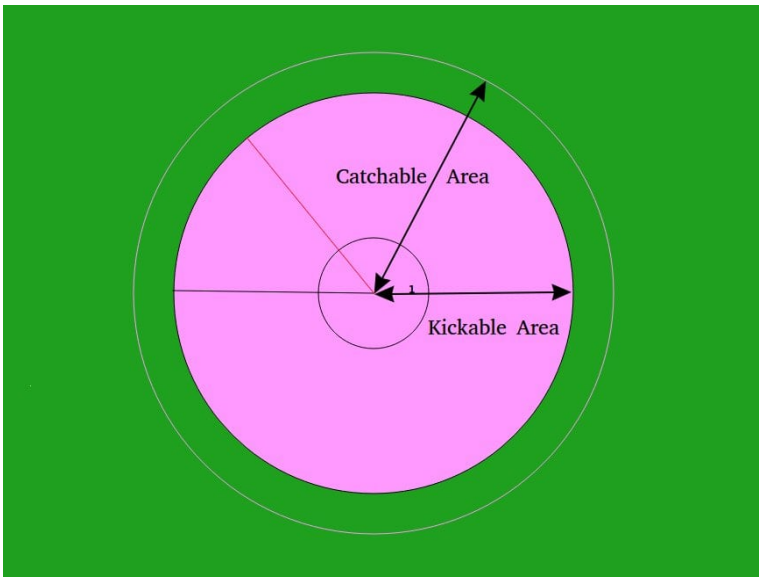

Figure 2.5: Catch area of goalie agent in soccer simulator

dash command is as follows:

$$(\textbf{\textit{dash}}\ dash\_power\ \ dash\_angle) \tag{2.5}$$

dash_power specifies the power of the desired *dash* action. In the current version of the simulator, minimum and maximum power of a *dash* command are -100 and 100.

dash_angle specifies the angle of desired *dash* action. It is a number between -180 and 180 degrees in the current version.

3. **kick:** The action that allows agents to send the ball in the desired direction with the desired speed. This command is sent to the simulator in the following format:

$$(\textbf{\textit{kick}}\ kick\_power\ \ kick\_dir) \tag{2.6}$$

kick_power is the power of the shot that the agent is performing and is between *minpower* and *maxpower* which for current server version have values of -100 and 100.

kick_dir is the direction that the agent is willing to send the ball and should be a number between *minmoment* and *maxmoment* which are -180 and 180 degrees for the current simulator version.

Each agent has to be close enough to the ball object so that the *kick* action can be performed. This distance is called kickable area and it has a slightly different value around 1 meter depending on the player type of the agent that is set either by the server or the coach of that team before the match begins.

4. **move:** This action has a very restricted use and moves an agent directly to a specific desired position in the soccer field. It is only available for use in certain play modes such as *before_kick_off*, *goal_l* and *goal_r* to enable players of both teams move to a certain position to form the team and start the game either at the beginning or after a goal is scored. The goalie agent of each team is also allowed to use *move* after a *catch* of the ball, in order to be able to move to a certain position in the team's penalty area to pass the ball to a desired teammate. Since there is no third dimension to allow goalie to send the ball with a low risk to the center of the field, this is a feasible approach.

5. **tackle:** This command was added to the soccer simulator in 2008 and allows players to tackle the ball even if it is not in their kickable area. It can be used both for defense or attack purposes. The player who wants to perform a *tackle* can calculate the chance of tackle being successful and the probability of

committing a foul (with some noise), prior to performing the action using the information provided by the server so that, depending on the situation of the game, it can decide if the risk of performing a *tackle* is worth it. The tackle area of an agent is a rectangle in front of an agent's body. If the ball is nearer to the center of an agent's body, the chance of a successful *tackle* also increases. Figure 2.6 shows the tackle area in front of an agent's body from the soccer logplayer software. The success probability of a *tackle* for the case shown in Figure 2.6 is 0.6.

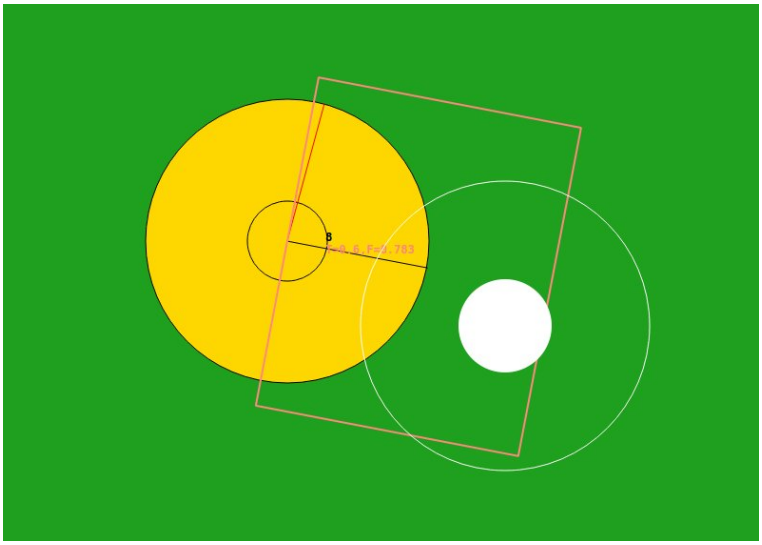

Figure 2.6: Tackle area of an agent in soccer simulator

6. **say:** Allows an agent to broadcast a message to other players in the field in the form of a string with a size of no more than *say_msg_size*. This message is sent by the simulator to players of both teams with a distance less than *audio_cut_dist* with no delay.

7. **turn:** This command allows an agent to *turn* its body towards a desired angle. As *kick* and *dash* commands, the server applies a noise model when executing it in the field, so the actual turn angle might be slightly different from the angle that was asked for. The noise applied considers current acceleration speed and angle of the agent to align the simulation with the real world.

8. **turn_neck:** This command acts almost as the same as the *turn* command and changes the neck direction of the agent instead of its body direction. Unlike *turn* that can not be executed at the same time with *dash*, *kick*, *tackle* or *move*, one *turn_neck* command is allowed to be sent to the server in each cycle so the player can see different directions of the field while moving.

9. **change_view:** Allows agents to change their view width and quality depending on their need. It has to be sent to the server in the following format:

$$(\textbf{\textit{change\_view}}\ \textit{view\_width}\ \textit{view\_quality}) \tag{2.7}$$

   view_width can be narrow, normal or wide.

   view_quality can be either high or low.

   Depending on how the agent chooses the parameters of this command, frequency and quality of the received visual information will be affected. For example, if it changes the quality from low to high, the frequency gets halved and the time between two see sensor readings is doubled [7].

2.2 Genetic Programming

Genetic Programming (GP) is a type of Evolutionary Algorithm that is inspired by biological evolution for handling complex problems. The most common type of GP is tree-based GP which is used in this thesis. Each computer program (individual) in a tree-based GP system is represented as a tree and each tree consists of a set of functions and terminals. There is a population of individuals that reproduce with each other in each generation. During the evolution process in GP, a predefined fitness function evaluates how well individuals perform toward a user defined goal [6]. The algorithm uses multiple generations of individuals in order to generate fitter solutions.

Each computer program in GP is represented as a tree and consists of some nodes chosen from a set of functions or a set of terminals. Each function has one or more children, depending on its type, and the child nodes of the function are the arguments of that function. While functions form the internal nodes of the tree, terminal nodes are the leaf nodes and have no children and usually are inputs to the program. Each terminal may be a variable or a constant with a value either preset or randomly generated [24]. Figure 2.7 shows the representation of the $\frac{(m+n)}{p}$ function as a genome of a tree-based GP where m, n, and p are terminals, and the functions are $+$ and $/$.

The GP algorithm starts by generating an initial population formed by a number of individuals. This initial population is generated randomly using the predefined function and terminal set of the problem. In the next step, each of these individuals are evaluated with the fitness function which specifies quality and performance of each genome. Genomes that perform better are more likely to survive into the next generation. The selection method for this step of the algorithm which is repeated for

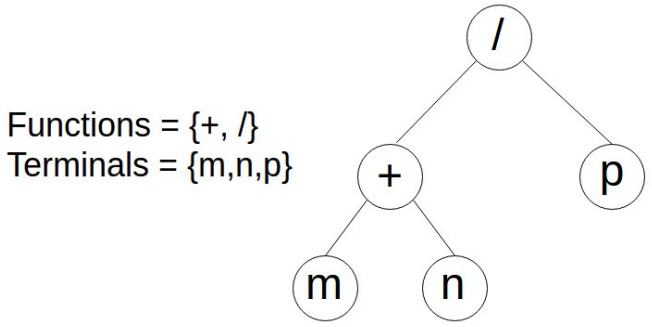

Figure 2.7: Representation of the function $\frac{(m+n)}{p}$ in a tree-based GP system

all of the generations, varies depending on the parameters of that specific run.

Before implementing a GP system for a problem, there are some things that need to be considered such as determining the best function and terminal set for the problem, choosing a selection method, defining a termination criterion, specifying GP parameters, and the most important one, designing a fitness function.

2.2.1 Preparing a Problem for GP

The first step in preparing a problem is to define the terminal and function sets that will be used for defining genomes in each generation.

Terminals are inputs to the algorithm and each terminal can be a variable, constant, random value, etc., and they often take the form of a named variable such as x or y. A function with no argument also resides in the terminal set (e.g., *rand()*, *dist_from_goal()*, *dribble_to_goal()*). Functions vary from arithmetic functions such as *PLUS*, *DIVIDE*, etc. to logical expressions such as *AND*, *OR*, etc. Depending on the type of the problem, we may also have some problem-specific functions and operators as well [3].

A termination criterion is necessary so that GP knows when to stop since in most complex problems it is time-consuming and very hard, if not impossible to find the desired best fitness value. After the termination criterion is met, the best fit individual is considered as the solution to the problem.

Finding and defining the fitness function is one of the most important parts of every GP system and varies from problem to problem. This function is the key factor of the GP that drives genomes toward the solution that we are looking for. Each genome has a fitness value that shows how well this genome performs toward the desired objective. Fitness can be measured in many ways. For example, with an error-based fitness function, a genome's fitness value is calculated as the sum of the absolute value of the differences between actual output of the program and the output given by the training set (the error) [6]. So the larger the fitness value of a genome in GP with an error-based fitness function, the less probable it is that this genome will survive to the next generation.

The final step of preparing a problem for GP is to define the GP parameters. There are about 20 parameters that need to be specified before running a GP system. The most important one is population size. Other parameters include maximum depth of trees, selection method, initial population creation method, probability of performing the genetic operators, etc. Setting these parameters heavily depends on the application and it is almost impossible to know which option is the best one, but some options are more common. For example, it is common to use the ramped half-and-half method for an creating initial population. This method creates half of the initial trees using the grow method and the other half using the full method with a depth of no more than 10.

2.2.2 GP Operators

After the first population is created with the use of random genetic operators, genomes will have the chance to evolve and become more fit in order to survive more generations. The main genetic operators are Crossover and Mutation.

2.2.2.1 Crossover

The crossover operator in tree-based GP swaps two subtrees from two genomes that are selected as parents. Combining genetic material of two genomes creates two new genomes that each have some parts of each of their parents. The most common type of crossover operation in tree-based GP is one-point crossover which works by selecting a crossover point (node) from parent trees and then swaps the corresponding subtrees [24].

2.2.2.2 Mutation

There are a few types of mutation used for tree-based GP. In this research, the subtree mutation method is used. In subtree mutation, if a genome is selected for mutation, one randomly-chosen point (node) of the tree is chosen and the subtree below that node will be replaced by a new subtree created using the same method as creation of initial population [6].

Chapter 3

Related Work

Genetic programming has been applied to multi-agent coordination before. Andre in [1] evolved communication between agents with different skills. In his research, agents were multi-part computer programs that communicated through a shared memory. Both the programs and the representation scheme were evolved using genetic programming. An illustrative problem of 'gold' collection was used to demonstrate the approach in which one part of a program made a map of the world and stored it in memory, and the other part used this map to find the gold. The results of his research indicated that the approach can evolve programs that store simple representations of their environments and use these representations to produce simple plans [1].

Qureshi in [25] evolved agent-based communication in a cooperative avoidance domain. They showed that genetic programming can be used to automatically program agents which communicate and interact to solve problems. The programs evolved simultaneously to define when and what to communicate, and how to use the communicated information to solve the given problem. Raik and Durnota in [26] used

GP to evolve cooperative sporting strategies. Luke and Spector in [19], Haynes et al. in [12] used GP to develop cooperation in predator-prey environments. Iba in [14] applied a similar approach to cooperative behavior in the TileWorld domain. These were some of the early related work done before Sean Luke applied Genetic Programming to a very difficult problem domain, RoboCup Soccer 2D Simulation.

The RoboCup soccer server is said to be not a good match for GP. The soccer server domain is very complex and there are many options and controls with lots of special cases and boundary conditions that are very important for each decision or action to make it hard for GP to get integrated. Another difficulty is the time factor. The soccer server runs in real-time and all players are connected separately via UDP sockets to the server. Each game takes ten minutes to play and there is an enforced 10ms delay between world model updates which makes the whole game about 10 minutes (equal to 6000 of 10ms cycles) [18].

In 1997, Sean Luke proposed using Genetic Programming for producing a team of competitive agents for the RoboCup97 official competition [17, 21]. The soccer simulation environment is very difficult, real-time, noisy, and highly dynamic. For many different reasons which will be addressed later in this research, the agents in the soccer simulation environment are very difficult to evolve. The objective that Luke and his teammates set for their team was fairly modest. Their goal at first was to produce a team of agents able to play the whole game [17, 18]. They managed to produce agents that were able to decide how to disperse throughout the field, pass, kick to the goal, defend the goal, and coordinate with and defer to other teammates. By the time their team participated in RoboCup97, all other teams were hand-crafted human code algorithms and their team was the only team with intelligent players

using an AI approach. They finally managed to win their first two games in the competition but lost others. They won the RoboCup97 scientific challenge award for their valuable research [17].

One of the reasons which makes it challenging to evolve a computer program to work successfully in this domain is that it would likely require a very large number of evaluations. In this case it would mean that each evaluation is equal to a run of a game in the simulator. In one of their previous results in a simpler domain, Luke and Spector found that GP will probably require about 100,000 evaluations in order to find a reasonable solution [19]. In a more complex domain like soccer server it is likely that more evaluations were necessary to find a reasonable solution. If we consider each evaluation in the soccer simulation server run to take 5 minutes, it would take up to a full year to do 100,000 evaluations. The challenge of cutting down this time from years to a few weeks or months while still being able to produce a relatively good-playing soccer team from only a small number of evolutionary runs was one of the main problems they faced. They managed to take on this problem in several ways [17]:

- They used brute force in order to speed up the process, running 32 parallel games and also cutting down time for each evaluation from a full game of 10 minutes to limited periods of games of between 20 seconds and one minute duration.

- They cut down population size and number of generations.

- They developed an additional layer of software to simplify the domain in order to eliminate many boundary conditions the GP programs would have to account

for. They also spent much time designing a function set and evaluation criteria to promote better evolution in the soccer simulation domain.

- They did parallel runs with different genome structures to have more options when they were close to the competition.

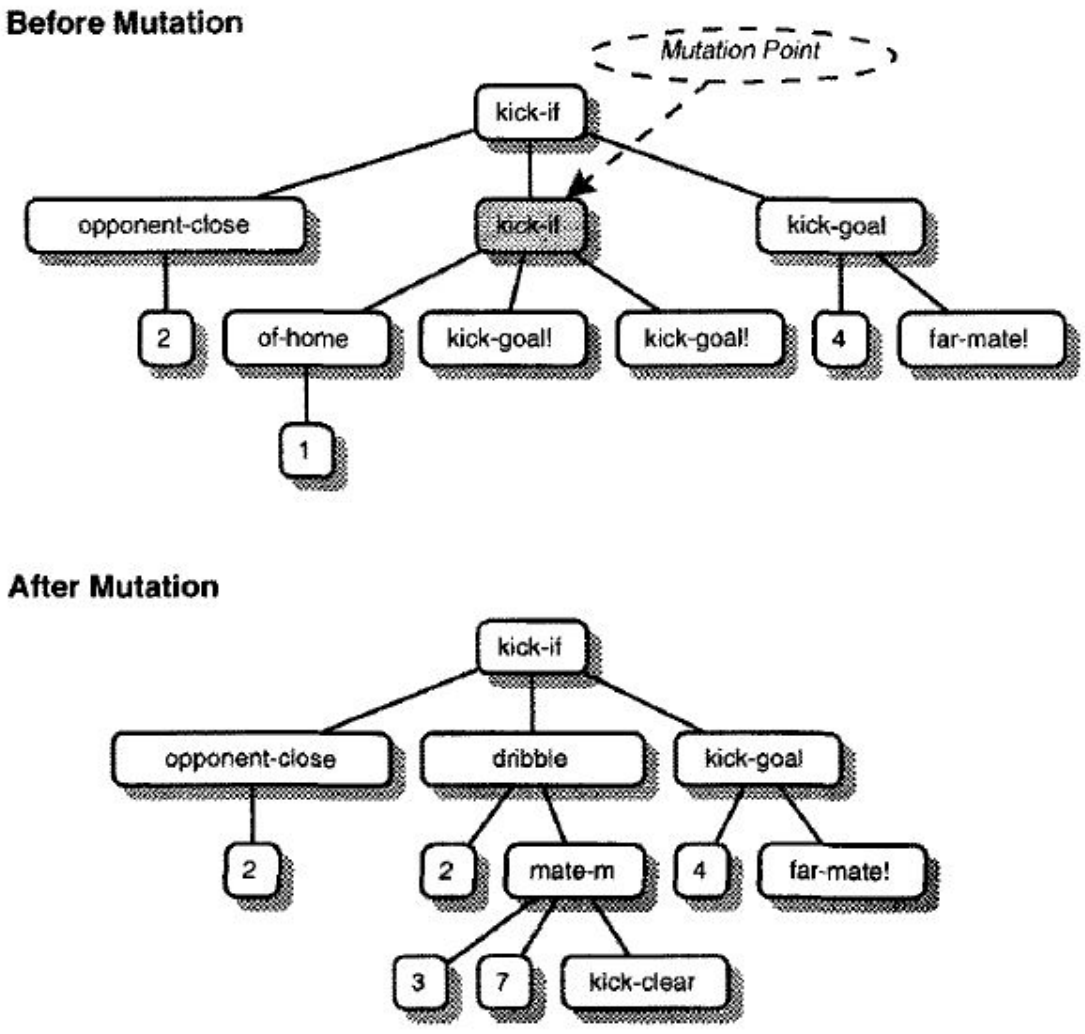

Figure 3.1: Point mutation operator in Luke's research [18]

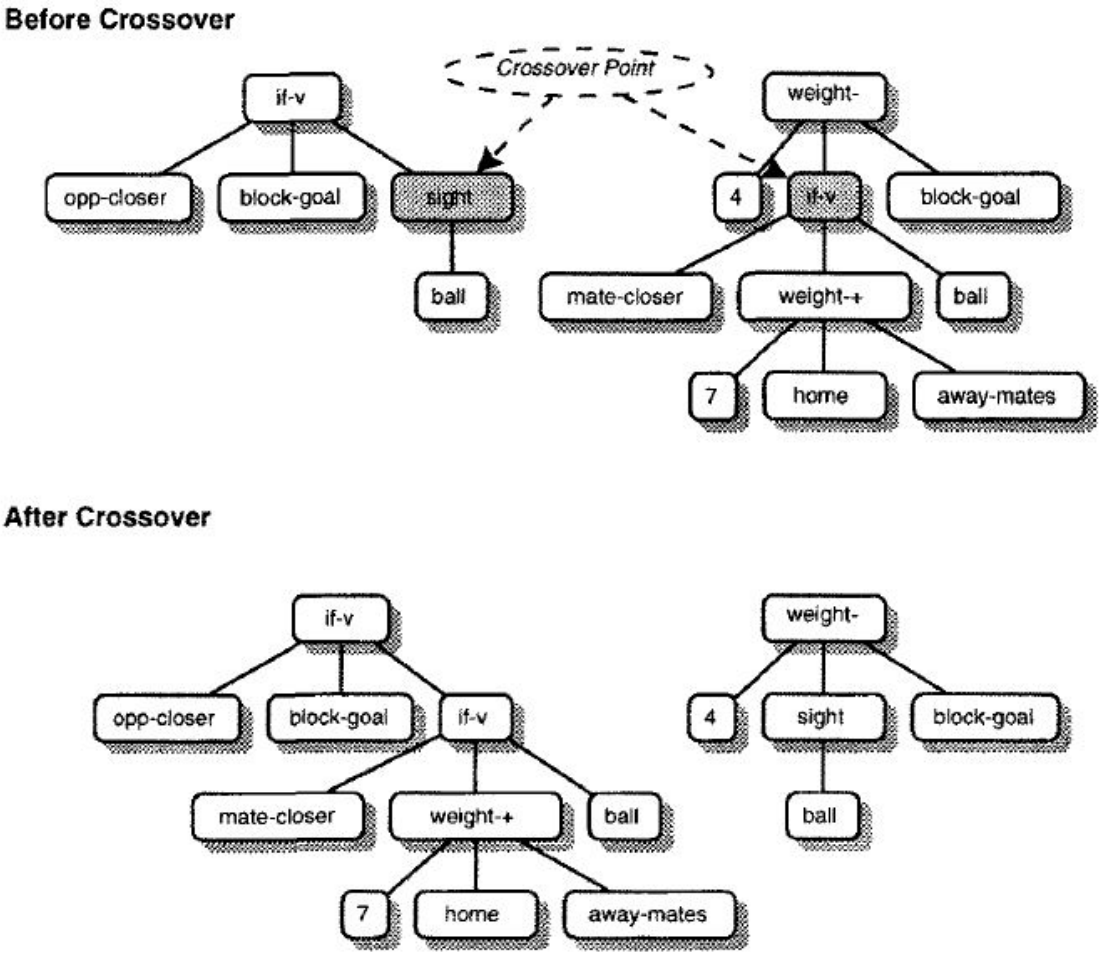

Figure 3.2: Subtree crossover operator in Luke's research [18]

In Luke's research, they predefined the low level skills of agents, such as dashing, kicking, and moving. They added evolvability to the high level skills of agents, such as pass, dribble, and shoot. They made some changes to the traditional GP genome, and instead of having one tree for each agent, they had two trees. One was the with-ball decision tree and the other one was the without-ball decision tree (one for moving the player and one for kicking the ball when ball is owned). A basic state-rule was set to determine which tree to call in each cycle of the game. If a player was close enough

to kick the ball, the kick tree would be called and if a player could see the ball but it was not reachable, the move tree would be called. Also, if the player wasn't able to see the ball, a simple turn body command would run until the player was able to see the ball [17]. Two examples of evolving trees from their research are shown in Figure 3.1 and Figure 3.2 which demonstrate how the point mutation and crossover operators act during evolution.

The fitness function in this work was based only on the number of goals that a team scored in a game. In order to prevent premature convergence (a problem because of their small population size) they used a high mutation rate of 30% [17, 3].

They also implemented both homogeneous and pseudo-heterogeneous approaches for their teams but because of the limited time and excessive size of pseudo-heterogeneous genomes, their pseudo-heterogeneous teams could not outperform their homogeneous team before the competitions.

After Luke's team pioneered GP in the soccer simulation domain in RoboCup97, a team named *Darwin United* used GP to evolve their agents in RoboCup98 [2]. Darwin United used a different method than Luke's team for evolving their players and they employed an optional coach agent -which receives noiseless data from the server but has very limited communication with agents- for storing data and coordinating the decision for rewarding players after each command execution [23].

After Luke and Darwin United, J. Aronsson in [3] also used GP to teach software robots to play soccer. He focused more on designing a better fitness function. Aronsson also made several compromises to limit the duration of the evolution process because of the excessive run times needed for evaluating each population due to complex nature of the soccer simulation framework.

Aronsson used a similar approach as Luke for implementing kick and move trees in his GP system. The population consists of several individuals, each of which having one move and one kick tree. Figure 3.3 demonstrates the structure of a decision tree in Aronsson's research. Each leaf node is always an action and all other nodes are predicates in his experiments.

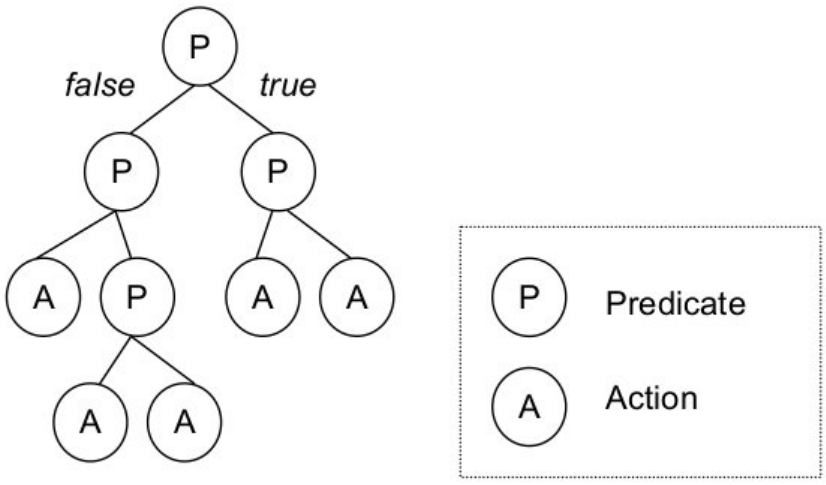

Figure 3.3: The structure of a decision tree in Aronsson's research [3]

One main difference in the decision making of Aronsson's agents compared to previous approaches was that players that believed to have an obvious chance of scoring a goal would attempt to score without considering the evolved kick tree's decision. This "action overriding" was expected to make agents concentrate more on team coordination and positioning and improving the quality of their decision making. It was concluded in his result that it worked as expected.

The fitness measure used in Aronsson's research was based on each player's performance during a game, or on the average of its performance if that player played

multiple games during the evaluation process. The fitness value was calculated as weighted sum of the parameters that are shown in Table 3.1

Table 3.1: Fitness assessments of Aronsson's GP implementation

Assessment	Value
Won	1 if the player's team own the game and 0 otherwise.
Team score	The number of goals made by the team.
Opponent score	The number of goals made by the opponent team.
Score	The number of goals made by the player.
Attempts	The number of shoots on goal made by the player.
Kicks	The number of times the player kicked the ball.
Passes	The number of passes made by the player.
Active	1 if the player kicked the ball during a game and 0 otherwise
Ball close	2 if the average distance to ball is less than 15, otherwise 1 if this distance is less than 20 or 0 if this distance is greater than 20.
Average y	The player's average y during a game.
Time free	The time the player was free during a game, measured in the percentage of total time. A player is free when no other player is closer than a distance of 10.
Time offensive	The time player spent on the opposite half of the field, measured in the percentage of total time.

Aronsson made two experiments, different only in the team set-up. In the first

experiment, the players learned to chase and kick the ball towards the goal or pass it to a teammate, but in the second one players converged and did not develop further than the first experiment. Also it was concluded that players from the first experiment were developing towards team coordination slowly. He mentioned some possible reasons for the agents not developing further [3]:

- **Premature convergence:** A larger population size was needed for a problem of this complexity in order to prevent premature convergence. Due to the nature of soccer simulation framework, evaluating individuals in the simulator is very time consuming and each game is limited to a small number of agents that forced them to use a small population size.

- **Limited search space:** The predicate and action sets that were defined would provide a limited search space for the experiments that would result in being unable to find significantly improved solutions.

- **Functions:** Action functions are controlling agents in the field. Skill functions that were used in his study did not have a good quality. For example, their agents had problems intercepting a pass.

- **Credit assignment:** Defining a fitness function that can express the actual desired behavior is one of the main complexities of this problem. Their main challenge was to determine which individuals to credit for a team's success.

- **Overfitness:** Each individual was only tested for a limited number of situations due to evaluations being very time-consuming. If more evaluations were done, it would minimize the fitness deviation and give a more accurate measurement.

- **Computational resources:** Each evaluation is computationally expensive. Their experiments were done in about one month and there were lots of compromises made in order to reduce evaluation time resulting in weaker players eventually.

Aronsson's conclusion was that software robots are able to learn to play simulated soccer but he believed that the strategies that the robots developed were most likely inferior to human-coded algorithms, though better than initially random strategies [3].

Stone and Veloso in [27] presented layered learning, a hierarchical machine learning paradigm. Layered learning applies to tasks for which learning a direct mapping from inputs to outputs is intractable with existing learning algorithms. Given a hierarchical task decomposition into subtasks, layered learning seamlessly integrates separate learning at each subtask layer. The learning of each subtask directly facilitates the learning of the next higher subtask layer by determining at least one of three of its components: (i) the set of training examples; (ii) the input representation; and/or (iii) the output representation. They introduced layered learning in its domain-independent general form and then presented full implementation in a complex domain, namely simulated robotic soccer [27].

Hsu and Gustafson in [13] presented an adaptation of the standard genetic program (GP) to hierarchically decomposable, multi-agent learning problems. To break down a problem that requires cooperation of multiple agents, they used the team objective function to derive a simpler, intermediate objective function for pairs of cooperating agents. They applied GP to optimize first for the intermediate, then for the team objective function, using the final population from the earlier GP as the

initial seed population for the next. Their layered learning approach facilitated the discovery of primitive behaviors that can be reused and adapted towards complex objectives based on a shared team goal [13].

After these works, GP was rarely used in the domain of soccer simulation. Lichocki et al. in [16] evolved team compositions by agent swapping. Aşık and Akın in [4] also used genetic algorithms for solving multi-agent decision problems. Sullivan and Luke in [28] presented a novel hierarchical learning from demonstration system which can be used to train both single-agent and scalable cooperative multiagent behaviors. The methodology applies manual task decomposition to break the complex training problem into simpler parts, then solves the problem by iteratively training each part. They discussed application of their method to multiagent problems in the humanoid RoboCup competition, and applied the technique to the keepaway soccer problem in the RoboCup Soccer Simulator [28].

Chapter 4

Method

In this chapter, we will discuss our proposed approach to improve decision making for a multi-agent system of the RoboCup soccer simulation. First, we will summarize the problems and challenges faced in previous works in this area. Then, we will explain the proposed method and justify how this method addresses the mentioned challenges.

4.1 Challenges

Some works in the area of decision making system for multi-robot systems were mentioned in the last chapter. Luke, Aronsson, and the Darwin United team used GP for evolving soccer agents in the soccer 2D simulation framework. Some important aspects and challenges of research done by Luke's team are as follows;

1. In his research, all agents were evolved only during the actual run of the soccer simulator and there is no learning for agents except inside the matches.

2. For evaluating individuals during a game, each agent can execute a decision tree so that it will be evaluated by its decisions. Due to the limited number of players (11 per team) in each game and the long run times for each game, having a large population size is almost impossible.

3. Since it was the first year of the competition, skills developed for agents (such as pass, dribble, shoot, etc.) were very simple and not as mature as today.

4. Opponents that the GP evolved agents were tested against in that year's competitions were all teams crafted by hand by experts. Today we have the option to test GP evolved agents against very powerful teams which are developed using many different AI methods.

Aronsson also used GP in the soccer simulator framework. Here are some problems and challenges he faced during his research:

1. All agents were evaluated during actual runs of the simulator. Evaluating individuals during running a game was very time-consuming so he had to limit the evaluation times and that resulted in weaker agents.

2. The agents that were evolved in his research had difficulties with basic skills such as ball interception. Even when GP found a very good individual, these basic skills might have resulted in bad performance and receiving a low fitness value ultimately leading to the elimination of that individual.

3. Due to the nature of the soccer simulator, he was forced to use a small population size that limited the search space and had a substantial impact on the final result.

4. His team wasn't competitive enough to play against teams that were participating in that year's RoboCup competition.

4.2 The Hybrid GP method

In this thesis, the goal is to use GP with a newly proposed evaluation method to generate decision trees for soccer agents, followed by another GP algorithm with a different setup and evaluation method which uses the best individuals of the first GP as input of the algorithm instead of a random initial population.

In the first phase of the approach, a GP algorithm with a random initial population and a large population size will create decision trees for agents. For evaluating individuals of each generation, some pre-defined situations from real games with a set of desired outputs will be used, each of which with a pre-defined score for different actions performed. The data used for evaluating individuals in order to train our agents will be generated using actions of agents from top teams of the world from the latest RoboCup competition. Individuals will be scored by the decisions they made for the specific situations using a fitness function that evaluates their decision. The selection method will be tournament selection for this phase of the method, crossover and mutation will be used for evolving trees with the sub-tree replacement method. Function and terminal sets are player skills (such as pass, dribble, etc.) and predicates from the environment needed for making decisions (such as inOppField, oppIsClose, etc.).

In the second phase of the proposed method, the best individuals resulting from the first phase will be used as material to seed the initial population in the second

phase. This phase is also using a genetic programming method for evolving decision tree of agents. Differences between the GP method used in second phase and first phase of the approach are:

(A) In the first phase, first generation's individuals are initialized randomly using the ramped half-and-half method and there will be lots of random trees at the beginning. In the second phase, the best individuals of the first phase will be used as first generation individuals. So in the second phase, those fit individuals that showed good behavior in the pre-defined scenarios of real games, will be evolved by taking part in real games and getting feedback from their actions in the real run of a soccer game against intelligent opponents with different strategies.

(B) Individuals of the first GP will be evaluated using some pre-defined situations and also a pre-defined scoring system as the fitness function. In the second phase, evaluating each individual will be done in a real run of a simulated soccer game, mostly by the feedback for each action of agents. For instance, if an agent decides to pass a ball to a teammate and that action results in losing the ball in a short period of time, or it decides to dribble with the ball when it is not safe to do so, it will be considered a bad decision and will negatively affect its fitness value compared to other individuals. The total result of the game also affects all team members' fitness values.

(C) In the second phase, the population size is much smaller due to the long run time of each game needed for evaluating an individual (about $5 \sim 10$ minutes per run) as well as the limited number of agents that are able to be tested in

the field (one individual per agent, so no more than 10 evaluations in each game excluding the goalkeeper).

An overview of the proposed hybrid method and the main characteristics of both GP systems used in two phases of the method is presented in Figure 4.1.

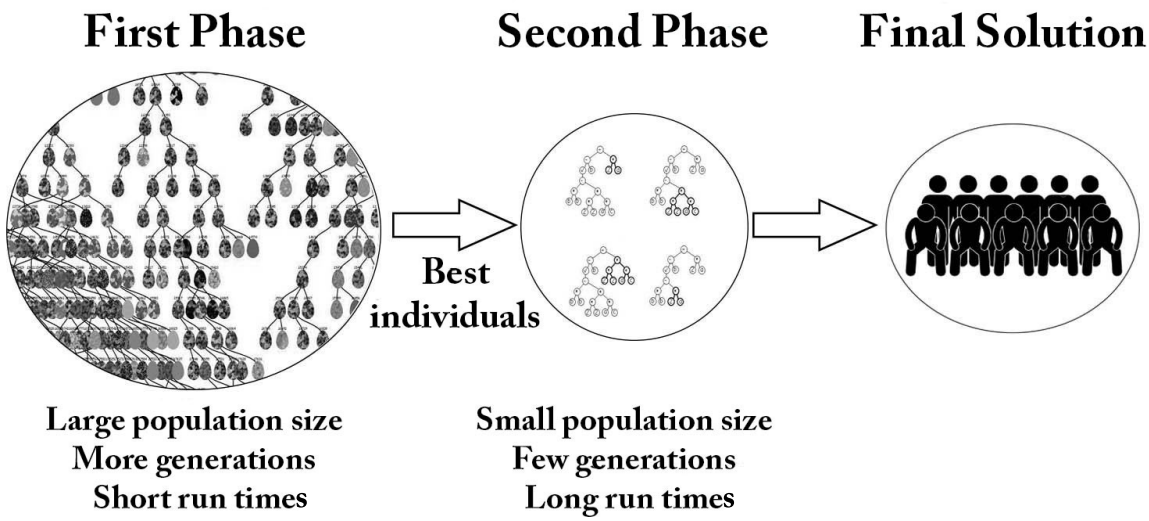

Figure 4.1: An overview of the proposed hybrid method

4.2.1 Addressing Challenges

This hybrid GP method will address some of the main challenges that were faced in previous works using GP in the soccer simulation environment such as:

4.2.1.1 Small Population Size

Because each evaluation was done previously during a partial run of a game in the simulator, the limit of evaluating 10 individuals per game prevented researchers from

having a large population size for each generation of the GP run. Using the first GP with a fitness function that can evaluate individuals without the need for running a game makes it possible to execute millions of evaluations in a much shorter period of time.

4.2.1.2 Time-consuming Evaluations

Each run of a full game in the soccer simulator takes about 10 minutes. If an individual is to be evaluated during one game, it takes 10 minutes to evaluate. The most number of evaluations that can be done in a game is 10 as we have 10 players excluding the goalkeeper agent. In order to achieve a desired number of evaluations we would need months and years of evaluations. The first GP in this hybrid method helps finding some intelligent agents so that we don't have to start with a random population when doing the time-consuming evaluations in the second GP and there will be more chances of early improvements in behavior of agents during the run of second GP.

4.2.1.3 High Level Skills of Agents

This research is being done using source code and skills of the *MarliK* team that is an internationally recognized team in the RoboCup soccer 2D simulation league and was placed third in the world in 2011 and 2012, and also first in some international competitions such as DutchOpen and IranOpen from 2009 to 2013. In the related works mentioned in this thesis, the agents that were used to evolve had very basic skill functions such as intercepting, passing, and shooting skills but in this research, the high level skills of the *MarliK* team are used which will lower the chance of eliminating good individuals during the evolution process due to errors that were caused by basic

skill functions, as it happened in previous research.

Chapter 5

Implementation

In this thesis, I have implemented both GP algorithms using the GPC++ library that will be introduced in the following section [11]. For the first GP, the GPC++ library was modified in order to be able to match our problem domain. Different functions and methods were implemented that will be discussed further. For the second GP, another modified version of GPC++ was used which was very similar to the first GP with the main difference being the fitness function and initial population. C++ is used for all of the implementations done in this thesis.

The proposed method is implemented here for evolving the kick tree of agents to help them decide what to do when they own the ball. In *MarliK*, the kick tree of agents has the responsibility of choosing between the pass, dribble, or clear skills to execute in each cycle of the game. Like most of the RoboCup teams, the shoot skill (kicking the ball towards the goal) is always checked first before executing the kick tree when an agent owns the ball. If the shoot skill predicts that a shoot is available for scoring the goal taking into account opponent players and their goalie, it will

execute this action, otherwise the kick tree will be executed to decide what to do.

5.1 The GPC++ Library

The GPC++ library (GP kernel), is a C++ class library that can be used for applying genetic programming techniques to all kinds of problems. It was developed mainly between 1993 and 1997 by Adam Fraser and then Thomas Weinbrenner. The software package comes as a library and defines several classes with a certain hierarchy [11]. Here are some features of this library:

- Automatically defined functions (ADFs)
- Tournament and fitness proportionate selection
- Demetic grouping, independent of the selection type
- Optional steady state Genetic Programming kernel
- Subtree crossover
- Swap and shrink mutation
- Possibility of multiple populations
- Changeable system parameters without the need of recompilation
- Loading and saving of a population

This software makes use of the object oriented programming scheme and the class hierarchy of the system can be viewed in Figure 5.1.

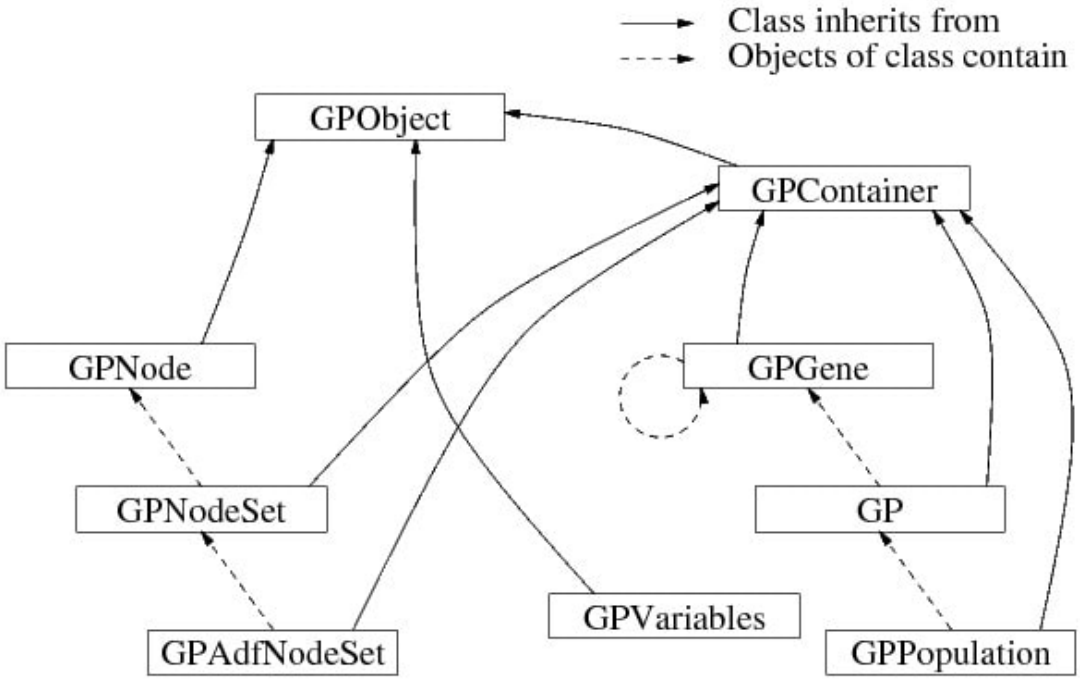

Figure 5.1: Class hierarchy of the GPC++ library [11].

5.1.1 Using the GP Kernel

For using the GP kernel and to solve our problem, we have to inherit and implement three main classes: *GP*, *GPGene*, and *GPPopulation*. During the process of creating populations, the kernel functions use some virtual functions to create genes and genetic programs, enabling us to overwrite those functions and create objects that belong to the inherited classes [11]. These are the main functions that have to be defined for this purpose:

- *GPGene::createChild()*
- *GP::createGene()*
- *GPPopulation::createGP()*

The above mentioned functions allocate an object of the class and return it.

The most difficult part of this implementation is to define a fitness function for our individuals. Function *GP::evaluate()* must be defined in order to implement the fitness evaluation method of the GP algorithm. As output it has to store the evaluated fitness value in the class variable *stdFitness*. Function *GPGene::evaluate()* shall be defined too, in order to parse the tree thus evaluating the fitness.

Function *printOn()* is also rewritten to redefine the way a population, gene, or an individual within the population is printed as output. In order to access a genetic program within a population, a tree within the genetic program, or a child within the gene class, functions such as *MyGene::NthMyGene()* can be defined for executing type conversion from type *GPObject* returned by *GPContainer::Nth()* to the type of the inherited class [11].

For loading or saving the whole population or an individual, the functions *isA()*, *load()*, *save()*, and *createObject()* as well as a parameterless constructor have to be defined in the class *MyGene*.

There are a few samples available in the library that show how to use the GP kernel and how to customize it for problems such as "Santa Fe Trail" and "Lawn Mower" problems. There is also a file named *skeleton.cc* available in the library that is a complete skeleton for all three classes that need to be inherited and their functions to be redefined.

5.2 First Phase

5.2.1 Terminals and Functions

The structure of a tree in our tree-based GP is very similar to what Aronsson used in his research, shown in Figure 3.3. Each of the trees consist of some leaf nodes that represent skills of agents to be executed in the soccer field, while other (inner) nodes of the tree are predicates which contain information about the environment around the agent for deciding what to do.

Recall that the terminal set consists of skill functions from the *MarliK* team, such as pass, dribble, and clear. When called, each of these skill functions decides how to execute that action and then generates the corresponding command that needs to be parsed. Finally it sends that command to the soccer server so that it can be executed in the field. For example, according to *MarliK*'s code structure, if the pass skill is chosen to be executed, a function named *execute()* from the class *Bhv_MarliKPass* will be called in order to decide the receiver agent and the type of the pass to execute. It might end up sending a direct pass, through pass, leading pass, or a cross due to the nature of the implemented pass skill.

Function nodes of our trees are chosen from the function set which includes predicates that reflect important parameters of the simulated soccer field. All of these predicates are Boolean variables and their value is either *true* or *false*. Typically, if the value of a predicate is *true*, then the first child node will be evaluated. If its value is *false*, then the second child node will be evaluated.

5.2.1.1 Function Set

The following list shows members of function set in our GP:

- **nearGoal** is true only if the agent's distance to the opponent's goal is less than 25m.
- **nearOwnGoal** returns true only if the agent's distance to their own goal is less than 25m.
- **inOppField** is true only if the agent's x coordinate is greater than 0, which means the agent is currently in the opponent's field.
- **oppIsFar** is true only if both of the following conditions are met:
  - The agent has seen at least one opponent within the last 5 cycles.
  - Distance of nearest opponent from agent is more than 20m.
- **oppIsClose** is true only if both of the following conditions are met:
  - The agent has seen at least one opponent within the last 5 cycles.
  - Distance of nearest opponent from agent is less than 10m.
- **oppIsVeryClose** is true only if both of the following conditions are met:
  - The agent has seen at least one opponent within the last 5 cycles.
  - Distance of nearest opponent from agent is less than 5m.
- **weAreWinning** is true only if the agent's team scored at least two goals more than the opponent team ($ourGoals$ - $OppGoals > 1$).

- **weAreDefending** is true only if all of the following conditions are met:
  - The agent has seen at least 5 teammates within the last 5 cycles.
  - At least 5 teammates' x coordinate is less than -25m.
  - The agent's x coordinate is less than -25m.
- **weAreAttacking** is true only if all of the following conditions are met:
  - The agent has seen at least 5 teammates within the last 5 cycles.
  - At least 5 teammates' x coordinate is greater than 10m.
  - Agent's x coordinate is greater than 10m.
- **tmmAvailable** is true only if all of the following conditions are met:
  - The agent has seen at least one teammate within the last 5 cycles.
  - A teammate's distance from the agent is greater than 5m and less than 30m.
  - There is no opponent with less than 10m distance from the same teammate.
- **pathClear** is true only if there is no opponent with a x coordinate greater than the agent and distance is less than 15m from the agent.
- **ballInDangerArea** is true only if ball is in the agent team's penalty area.
- **alone** is true only if there is no teammate or opponent with a distance of less than 25 from the agent.

5.2.1.2 Terminal Set

The terminal set of our GP includes:

- **PASS**: Execute a direct, through, leading, or cross pass to the best teammate depending on the situation of the game.
- **DRIBBLE**: Dribble with the ball in order to get closer to the opponent's goal.
- **CLEAR**: Clear the ball to a safe point which can be either outside of the soccer field (causing a corner or a kick in for opponent team), or a point that is far from opponents.

5.2.2 Implementation of C++ Classes

The implementation consists of various C++ classes that control our GP system.

5.2.2.1 GPContainer Class

This class holds objects of type *GPObject* or of an inherited class. It helps handling objects and works as a base class for almost all other classes because they are all containers. Each container manages the objects it owns by allocating an array of pointers that point to these objects [11]. This array has a fixed length. The *GPContainer* class was implemented in the GP kernel and is used in the classes that we inherited.

5.2.2.2 MyGene Class

This class inherits from *GPGene*. *MyGene* class represents the tree structure and serves as our base class. These are the functions that we needed to implement in this class so that it can create and evaluate our desired GP algorithm:

- **MyGene** constructor function.
- **duplicate** function to make copies of members whenever needed. It is a virtual function from the *GPContainer* class that *GPGene* class is derived from.
- **evaluate** function for tree evaluation. This function evaluates the fitness of a genetic tree and works in cooperation with the *evaluate* function from *MyGP* class. It returns the desired result to *MyGP::evaluate()* to put the final fitness value in the class member *stdFitness*.
- **NthMyChild** function for accessing children.

5.2.2.3 MyGP Class

Class *MyGP* inherits from *GP* class. Class *GP* also inherits from the *GPContainer* class and contains the root gene of each tree. After defining *MyGP* class, some functions had to be implemented as they were needed by the *GP* class:

- **MyGP** constructor function.
- **duplicate** function which is a virtual function of class *GPContainer* that has to be defined for every inherited class for making copies of members.
- **createGene** function for creation of own class objects.
- **evaluate** function for tree evaluation. This function evaluates the fitness of a genetic program and saves it into the class variable *stdFitness*.
- **NthMyGene** function for accessing trees.

5.2.2.4 MyPopulation Class

This class is inherited from the *GPPopulation* class. *GPPopulation* is a container that contains all of the genetic programs of a population. After defining *MyPopulation* class and inheriting it from *GPPopulation* class, some of its functions had to be redefined with the desired arguments matching our problem such as:

- **MyPopulation** constructor that gets GP parameters as input so that our GP is created as desired.
- **duplicate** function which duplicates members of population which is needed for creating next generations.
- **createGP** function for creation of own class objects.
- **NthMyGP** function for accessing different genetic programs within a population.

5.2.3 Fitness

Each individual in our tree-based GP is evaluated by the actions it takes in some predefined simulated situations that were derived from real games. These simulated situations are extracted using the decisions that are made by agents of the strongest teams from the RoboCup 2016 competition. Each of these situations is called a "snapshot" and it contains information about the field, such as position of teammates and opponents in that specific cycle of the game. Each snapshot is associated with a reward system for each action that could be made in that situation. A decision in

a snapshot is classified as being one of the following options and there is a number associated with each of them that increases as the action gets worse;

- **perfect:** If an action is the best possible action in that snapshot and it is undoubtedly the best that could be executed in that cycle. An example is choosing the dribble action when an agent is in a one-on-one situation in front of the opponent's goal and there is no teammate available to pass the ball to. Choosing dribble in this situation is classified as *perfect* because the agent should get as close as possible to the goal so that the opportunity to shoot and score a goal is created.

- **good:** If an action is good to be executed and there is not much risk included in the action in that situation of the game.

- **bad:** If an action is not a good choice and might result in losing possession of the ball or an opportunity, and there is at least one better option to choose from.

- **veryBad:** If an action is a very bad choice and it is very risky for that cycle of the game.

- **worst:** If an action is an obvious bad choice and will immediately result in losing a great opportunity or losing possession of the ball. For example, choosing to clear the ball in an attacking one-on-one situation against the opponent's goal is classified as *worst*.

Figure 5.2 shows a snapshot of the RoboCup 2016 final game between the Helios2016 and Glider2016 teams that was used in our fitness function. In this situation,

player number 9 of the attacking team has a great opportunity to score a goal and decided to dribble with the ball towards the goal to create more chance for scoring. The associated values with this snapshot of the game is presented in Table 5.1.

Figure 5.2: A snapshot of the RoboCup 2016 final game

Figure 5.3 shows another snapshot from the RoboCup 2016 third place game between the Ri-one and CSU_Yunlu teams. Player number 6 of the defending team is in a dangerous position inside their own penalty area and there is an opponent close to him trying to get the ball. In this situation, the player decided to pass the ball to teammate number 7. Values of parameters from this snapshot of the game are shown

in Table 5.1.

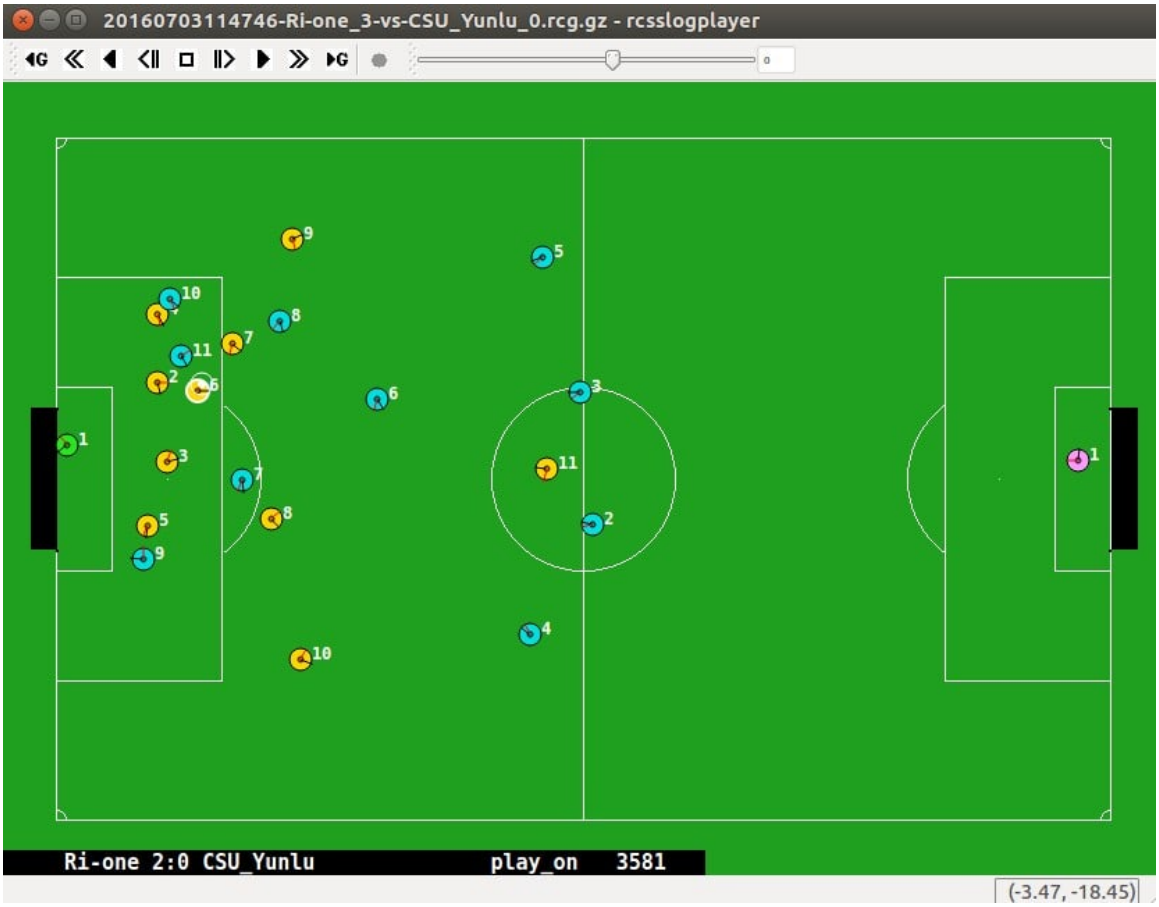

Figure 5.3: A snapshot of the RoboCup 2016 third place game

Each individual from a population that is being evaluated will be tested against 100 snapshots of real games and the average performance of the corresponding tree will be the fitness output.

Table 5.1: Values associated with each snapshot from Figure 5.2 and Figure 5.3

Parameter Name	Value in Figure 5.2	Value in Figure 5.3
nearGoal	true	false
nearOwnGoal	false	true
inOppField	true	false
oppIsFar	false	false
oppIsClose	true	true
oppIsVeryClose	false	true
weAreWinning	false	true
weAreDefending	false	true
weAreAttacking	true	false
tmmAvailable	false	true
pathClear	false	false
ballInDangerArea	false	true
alone	false	false
pass	bad	perfect
dribble	perfect	veryBad
clear	worst	good

5.2.4 GP Parameters

The GP parameter configuration we used for this phase of the method is presented in Table 5.2.

Table 5.2: GP parameters of the first phase of our method

Parameter Name	Value
Population Size	1000
Number of Generations	100
Creation Type	Ramped half-and-half
Crossover Probability	98%
Mutation Probability	2%
Maximum Depth for Crossover	17
Maximum Depth for Creation	6
Selection Type	Tournament selection
Tournament Size	10
Demetic Grouping	false
Add Best To New Population	true
Steady State	true

5.3 Second Phase

After running the implemented GP of the first phase, the top 10 fittest individuals of the latest population are used as initial population for the GP that is implemented

in the second phase of the proposed method. Evaluation of individuals in this phase of the algorithm will be done using the actual runs of the soccer server against the *Agent2D* team which is the open source base code for most of the RoboCup teams.

5.3.1 Terminals and Functions

For this phase of the method, the same set of functions and terminals as the first phase is used:

- **Functions** = {nearGoal, nearOwnGoal, inOppField, oppIsFar, oppIsClose, oppIsVeryClose, weAreWinning, weAreDefending, weAreAttacking, tmmAvailable, pathClear, ballInDangerArea}
- **Terminals** = {PASS, DRIBBLE, CLEAR}

5.3.2 Fitness

Due to the complexity and nature of the soccer simulator, evaluation of individuals in this phase of the algorithm is very time consuming compared to the previous phase. Each run of a full game in the soccer simulator takes about 10 minutes (6000 cycles) and a maximum of 10 individuals can be evaluated in each game since the goalie agent is completely different from the other 10 players.

The fitness function for this phase of the algorithm is reward-based and each individual will get a reward based on the action that it took in each cycle of the game. Evaluation of these actions is achieved using a combination of some parameters from consequences of the actions taken by agents and some basic rules that are mostly

common sense, such as not clearing the ball when you are in an attack situation in front of the opponent's goal.

Each population is evaluated during one full game run in the soccer simulator. We have followed a homogeneous approach for this phase of the algorithm. We evolved and used the same decision tree for all agents of a team, without considering their specific role. An individual is randomly assigned to a player before beginning of each match and the fitness value of that individual will be the average reward that it received from all the actions that it took during each game.

A list of events that affect fitness values of individuals of each population:

- If ball possession is lost to the opponent team in the following 20 cycles after the agent executed an action.
- If a goal is scored in the next 100 cycles.
- If ball object's x coordinate is increased or decreased (an increase means ball is moved toward the opponent's goal).
- If the player is not a defender and cleared the ball while in opponent's field.
- If opponent team scored a goal in the next 50 cycles.
- If the player decided to dribble with the ball while teammate goalie agent is very close (chance of giving a back pass fault to opponent).

Each of the above mentioned events will positively or negatively affect the fitness value of individuals in the population, depending on the effectiveness or severity of the event on the whole team's gameplay. Each of these events have a value range of -40 to 40. As the action gets worse, the number added to fitness value will be higher.

5.3.3 GP Parameters

The GP parameter configuration that we used for this phase of the method is shown in Table 5.3.

Table 5.3: GP parameters of the second phase of our method

Parameter Name	Value
Population Size	10
Number of Generations	20
Creation Type	Top individuals of previous phase
Crossover Probability	98%
Mutation Probability	2%
Maximum Depth for Crossover	17
Maximum Depth for Creation	6
Selection Type	Tournament selection
Tournament Size	10
Demetic Grouping	false
Add Best To New Population	true
Steady State	true

Chapter 6

Results

As mentioned in Chapter 5, we implemented the proposed method using C++ in Linux. First phase was implemented using the GPC++ library as a base code and second phase was implemented using the *MarliK* soccer 2D simulation team's code in combination with the code implemented from first phase and a script that connected these two parts. The hardware and software specifications used for the experiments are as follows:

- Intel Core 2 Quad CPU Q6700 @ 2.66GHz x 4
- 4 GB Memory
- Ubuntu 16.04 LTS 64-bits
- RoboCup Soccer Simulation Server (RCSSServer) Version 15.3.0

6.1 Runs of the First Phase

We performed 10 runs with different random seeds for the first phase of the method. The statistics of the average of these runs as well as the best run is presented and discussed in this section.

During the runs of the first phase of the method, as it was expected, initial individuals made good progress toward reaching the target fitness. Smaller fitness values for an individual means that it performed better actions in more snapshots of the game. For example if an individual makes the *perfect* decision in 75 out of a total of 100 snapshots, and makes 15 *good*, 6 *bad*, 2 *veryBad*, and 2 *worst* decisions, it will end up with a fitness value of 510. The scores associated with each of these parameters are shown in Table 6.1.

Table 6.1: Scoring system of individuals

Parameter	Value
perfect	0
good	10
bad	20
veryBad	40
worst	80

The best individual after 100 generations had a fitness value of 180 which means it had a very good performance in the simulated snapshots of the game and it chose the *perfect* decision in at least 82 snapshots out of 100 snapshots.

Figure 6.1 shows the development of the fitness of the best and worst individual of each population as well as the average fitness of individuals in each generation during evolution of the first phase. The data shown in Figure 6.1 is the average of 10 runs that were performed during our experiment. Smaller fitness value represents a better performance.

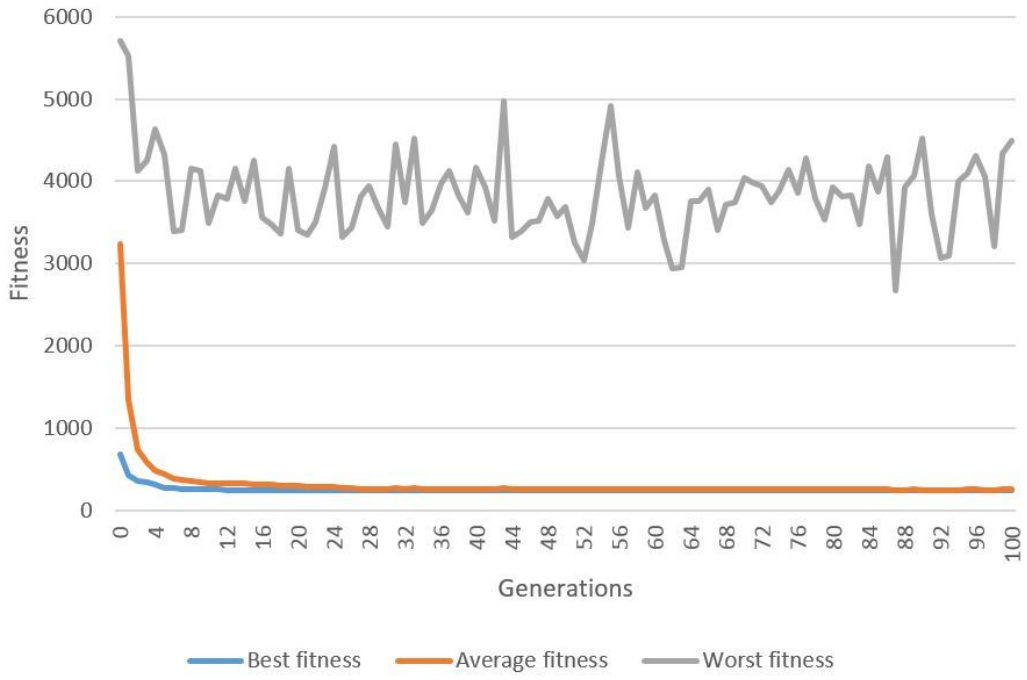

Figure 6.1: Evolution of best, average, and worst fitness in runs of the first phase

Figure 6.2 shows error bars on the average fitness of individuals over generations in the runs of the first phase using standard deviation of individuals in each generation indicating where majority of each population are.

The fitness of the best, worst, and average of population from the best run of the first phase is also demonstrated in Figure 6.3. It can be observed that in all runs

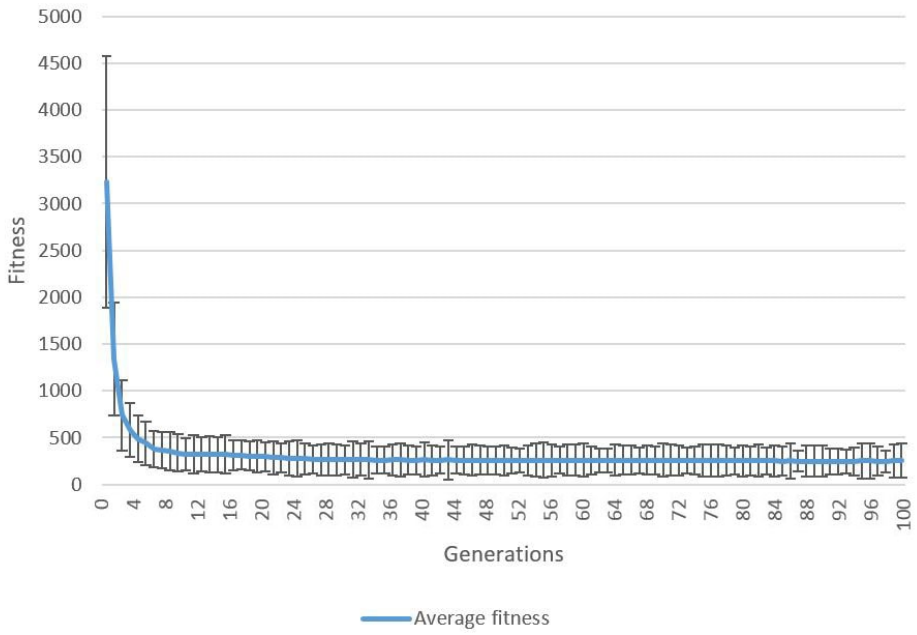

Figure 6.2: Error bars indicating (one) standard deviation of individuals in runs of the first phase

of the first phase, major progress of individuals is made in the first 20 generations and no improvement was observed in the best fitness after generation 83. Destructive mutation and crossover operators are the main reasons that cause significant changes in the fitness of worst individual over generations.

Figure 6.4 illustrates standard deviation of individuals in the best run of the first phase using error bars on the average fitness graph over generations. It can be observed that the population seems to converge repeatedly in certain generation, just to later break out again. This seems to happen when the worst individual of population has a better fitness and it seems to break when a significant increase in worst population happens due to a destructive mutation or crossover as it happened in generations 36 and 52.

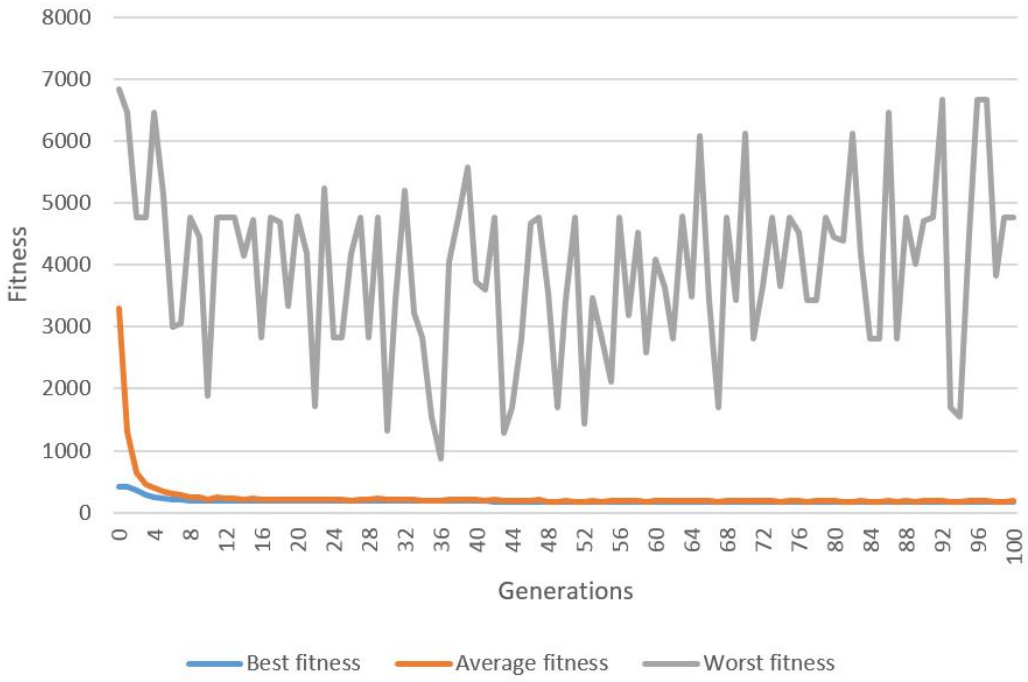

Figure 6.3: Evolution of best, average, and worst fitness in the best run of the first phase over generations

6.2 Second Phase

The top 10 individuals of the first phase of running GP were used as the initial population of the GP in our second phase in order to be evolved during real runs of the soccer simulator. A significant improvement in best fitness was not expected since the initial population was not randomly generated and individuals were already evolved during the first phase. This also demonstrates that the off-game evolution using snapshots was useful as guidance in real-time games of this phase.

The individuals that were evolved during the first phase of the algorithm were only tested against 100 specific snapshots from real games. The fitness function of this phase is designed in a way that the individuals can be tested even more and they will

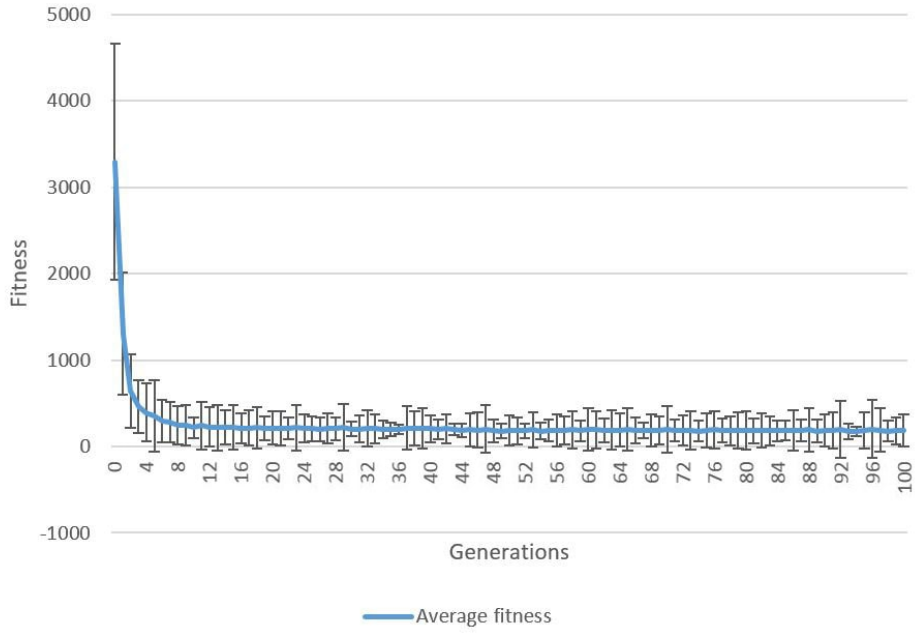

Figure 6.4: Error bars indicating standard deviation of individuals in the best run of the first phase

be judged by the consequences of what they choose to do during real games, against a real opponent. For example, an individual could decide to *dribble* in one of the snapshots during the first phase, and this could result in getting a *good* fitness value for that snapshot, but this decision might actually result in loosing the ball possession after a short time of making this decision in a real game. The fitness function of the second phase of the algorithm evolves the individuals during real games where they could get actual feedback of their behavior to ensure the decisions made from simulated snapshots actually work well during real games as well. As expected, a smaller improvement in fitness values and behavior of agents was observed during the runs of GP in this phase of the method for 25 generations.

Figure 6.5 illustrates evolution of the best, worst, and average fitness in each

generation for this phase of the algorithm. Elitist selection is used in both phases of the method and the best individual will always survive to the next generation. The reason for a worsening best fitness value in some consecutive generations is the fact that roles of individuals are assigned to players randomly in each generation. For example, the best individual of generation 4 was the one that was assigned randomly to a defender role during the evaluation run and it received a fitness value of 150. The same individual was assigned to an attacker role during the next generation and its fitness value worsened as it did not make good decisions as an attacker during the second time that it was being tested. As it was explained, this process makes sure that during the evolutionary process, the best individuals are the ones that are most likely to perform well in all different roles.

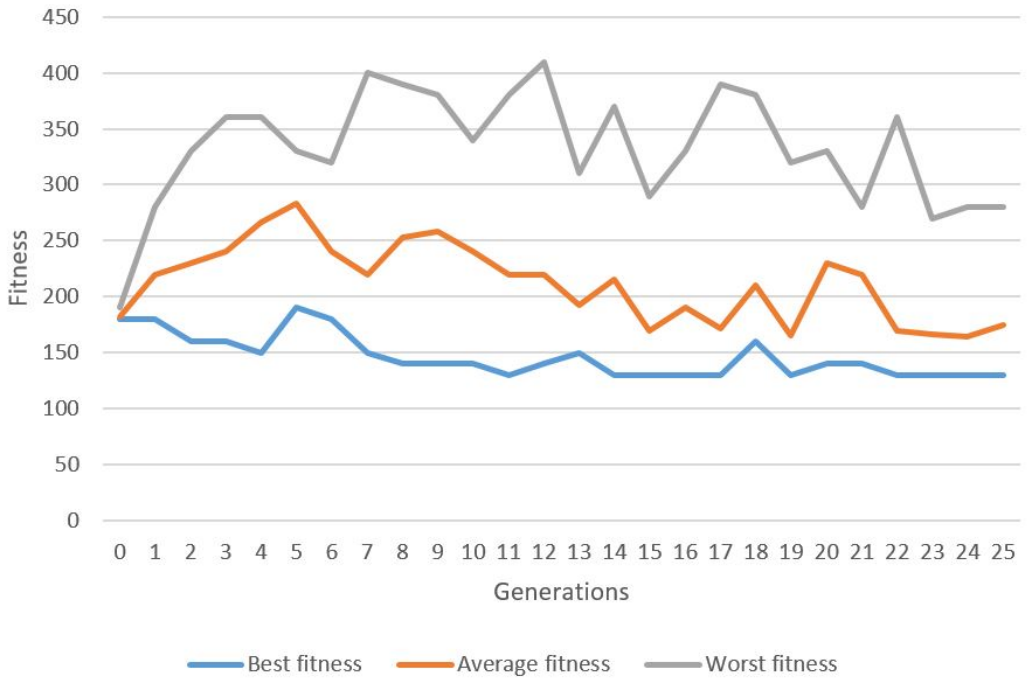

Figure 6.5: Fitness evolution in second phase over generations

Looking at the average fitness values in the run of this phase, we can see that in the first generation all individuals of the population had good fitness since they were the best ones from the previous phase of the method. However, in the course of 25 generations, some changes took place that made some individuals perform better and some worse than at the outset. The significant changes in average fitness of different generations compared to first phase can be explained by the small population size for this phase of the algorithm.

For the same reason, the worst fitness of the first generation is significantly lower than most of the next generations as shown in Figure 6.5. Destructive crossover and mutation operators are the main reasons for the significant changes of the worst individual's fitness in each generation.

Figure 6.6 shows the distribution of individuals in each generation using one standard deviation of the individuals in each population.

6.3 Performance Analysis

Multiple experiments have been performed after both phases of the proposed method have been completed. Results of these experiments using individuals the from final generation of the second phase of the method are discussed in this section.

6.3.1 Hybrid GP Method vs. MarliK's Old Decision Making System

The final version of *MarliK* without the new decision making system built by the Hybrid GP method was tested in 25 games against the *Agent2D*, *Helios*, and *Gliders*

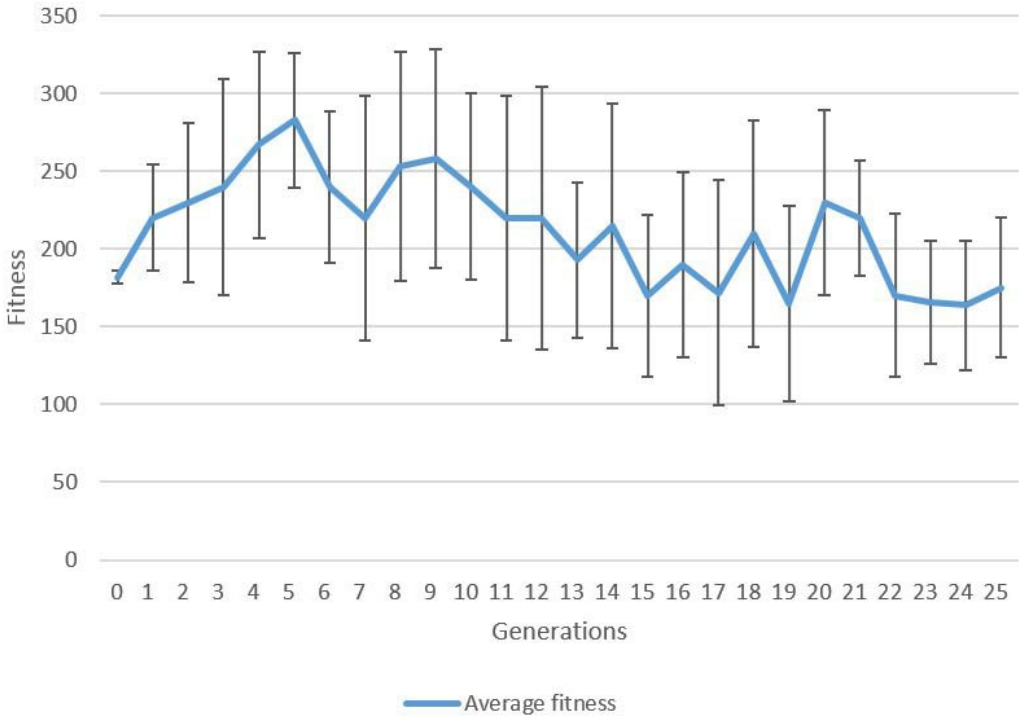

Figure 6.6: Error bars indicating standard deviation of individuals in the second phase

teams. Then, the new decision making system which consists of the best individual from the last generation of the second phase of the method was added to *MarliK* for all of the agents and tested against same teams.

Agent2D is an open-source base code that most of the teams in soccer 2D simulation league, as well as *MarliK*, are currently using, and *Gliders* and *Helios* are also the most powerful teams from latest (2016) RoboCup competition.

Table 6.2 shows the results of these experiments. Data from each row of table represents results of 25 games. Average number of goals scored and conceded per game as well as win percentages of each version of MarliK that was used is shown in the table.

Table 6.2: Performance of Hybrid GP method compared to *MarliK* with old decision making system

Opponent team	Tested team	Avg. goals scored (Standard deviation)	Avg. goals conceded (Standard deviation)	Percentage of wins
Agent2D	Hybrid method	6.12 (2.27)	0.40 (0.57)	100%
	Old MarliK	4.36 (1.81)	0.44 (0.50)	92%
Helios	Hybrid method	1.00 (0.89)	1.56 (1.20)	20%
	Old MarliK	0.72 (0.78)	1.96 (1.31)	8%
Gliders	Hybrid method	2.08 (1.81)	1.12 (0.77)	40%
	Old MarliK	1.84 (1.41)	1.12 (0.99)	40%

We observe that the Hybrid GP method of decision making outperformed *MarliK*'s previous static decision making system in most cases, leading to better overall results. Looking at the results of Table 6.2, it can be concluded that the attacking power of the team was more positively influenced than the defending power of the team. An explanation for this fact is that the Hybrid GP method is here only used for with-ball decision making of agents. Power of defense in a soccer team is more influenced by skills such as *block*, *mark*, and *intercept* which are executed in the without-ball decision making of players.

6.3.2 Phase 1 GP vs. Phase 2 GP vs. Hybrid GP Method

In order to make sure that the Hybrid GP method works better than either phase of the method separately, two other versions of decision making systems were examined as well.

First, the top individual from the last generation of the first phase was chosen as the decision tree of all players in *Team A*. Then, the second GP of the Hybrid method which had a small population size and used time-consuming runs of the soccer simulator for evaluating individuals was executed again, but instead of using the best individuals from the first GP as initial population, a randomly generated initial population was used and individuals were evolved for 25 generations (same as it was done in second phase of the Hybrid GP method). The best individual from the last generation was used as decision tree of all players in *Team B*.

Table 6.3 compares the performance of the Hybrid GP method with *Team A* and *Team B*. Each row of data shows the average performance of each team in 25 games. From the results of this experiment, we can observe that the Hybrid GP method outperforms both *Team A* and *Team B* in all aspects. The main reason for the poor performance of *Team B* compared to the other two teams is that the fitness function for the second GP was specifically designed to help the best individuals of the first GP evolve even further than they had already evolved.

6.3.3 Homogeneous vs. Heterogeneous Approach

In this research, we have followed the homogeneous approach meaning that we evolved/used the same decision tree for all agents of a team, without considering their specific role.

Table 6.3: Performance of Hybrid GP method compared to each phase of the method separately

Opponent team	Tested team	Avg. goals scored (Standard Deviation)	Avg. goals conceded (Standard Deviation)	Percentage of wins
Agent2D	Hybrid method	6.12 (2.27)	0.40 (0.57)	100%
	Team A	4.76 (1.39)	0.48 (0.50)	88%
	Team B	0.44 (0.50)	1.00 (1.02)	12%
Helios	Hybrid method	1.00 (0.89)	1.56 (1.20)	20%
	Team A	0.64 (0.79)	2.20 (1.10)	8%
	Team B	0.00 (0.00)	7.48 (2.17)	0%
Gliders	Hybrid method	2.08 (1.81)	1.12 (0.77)	40%
	Team A	2.12 (1.56)	1.20 (0.85)	32%
	Team B	0.00 (0.00)	5.52 (1.98)	0%

A heterogeneous approach is also possible to follow which means to evolve separate decision trees for each player, or for different groups of players with similar roles (defender, midfielder, and attacker).

Whether to have the same decision tree for all players or separate decision trees for different player roles (defenders, midfielders, and attackers) is one of the big challenges for RoboCup teams. Here we try to answer this question by comparing outcomes for these two approaches to our method.

For answering the question of whether a homogeneous or a heterogeneous approach works better for our method, we ran a parallel experiment with the second phase of the algorithm where agents were divided into the 3 main categories of *defenders*, *midfielders*, and *attackers*. Each of these categories had a separate (evolving) population and the experiment was done using the same GP configuration as the second phase of our method before. The best individuals from the latest population of the first phase GP were used as initial population for both teams.

Table 6.4 compares performance of these two approaches of our method. We observe that the heterogeneous team slightly outperforms the homogeneous against one opponent. From these results and the experiments from previously discussed research, we expect that over more generations, the heterogeneous approach has more chance of evolving better individuals compared to the homogeneous approach. Specializing in one behavioral role showed slightly better performance, in particular against the *Gliders* team.

Table 6.4: Performance of homogeneous approach compared to heterogeneous approach on our method

Opponent team	Tested team	Avg. goals scored (Standard Deviation)	Avg. goals conceded (Standard Deviation)	Percentage of wins
Agent2D	Homogeneous	6.12 (2.27)	0.40 (0.57)	100%
	Heterogeneous	5.72 (1.82)	0.36 (0.56)	100%
Helios	Homogeneous	1.00 (0.89)	1.56 (1.20)	20%
	Heterogeneous	0.92 (0.89)	1.32 (1.05)	16%
Gliders	Homogeneous	2.08 (1.81)	1.12 (0.77)	40%
	Heterogeneous	2.48 (1.70)	1.16 (1.08)	52%

6.3.4 Hybrid GP Method vs. Base Algorithms

In previous sections, the hybrid GP team was tested against 3 different teams and the results were compared to the results of each of the base algorithms against those teams. Testing against different teams with different strategies, as we did, is the normal procedure used by RoboCup teams in order to have a better understanding of their team's overall performance, instead of testing their team against a previous version of their own team.

Here we also tested the hybrid GP team in face-to-face matches against previous version of *MarliK* with the old decision making system, Team A (the team from top individuals from the first phase of the method), and Team B (the team from evolving random initial population in second phase of the method). Results of running 25

games between each of these teams against the hybrid GP team are shown in Table 6.5.

Table 6.5: Performance of Hybrid GP method against base algorithms

Opponent team	Tested team	Avg. goals scored (Standard Deviation)	Avg. goals conceded (Standard Deviation)	Percentage of wins
Team A	Hybrid method	1.64 (1.16)	1.44 (0.94)	64%
Team B	Hybrid method	5.36 (2.04)	0.00 (0.00)	100%
Old MarliK	Hybrid method	1.68 (1.43)	1.56 (1.44)	56%

The results from Table 6.5 support our previous experiments and ensure better performance of our method's final solution against each of the base algorithms.

Chapter 7

Conclusion and Future Work

In this research, we proposed and implemented a new hybrid GP method to improve the decision making system of soccer simulation teams. The proposed approach consists of two phases each of which tries to cover the other's restrictions and limitations. The first phase produces some evolved individuals based on a GP algorithm with an off-game evaluation system and the second phase uses the best individuals of the first phase as input to run another GP algorithm to evolve players in the real game environment where evaluations are done during real-time runs of the simulator.

To test the method's performance, we implemented it on the *MarliK* team which had a basic decision making system. Our hybrid GP method helped to improve the performance of agents as well as the whole team against 3 different teams with different strategies. Two of these teams are top teams from the latest (2016) RoboCup competition and the third team is the base code that is used by most of the RoboCup teams as a benchmark. Comparing to previous work that used an approach similar to our second GP, which had a small population size and a low number of generations,

it was observed that adding the first GP with a different type of fitness function that didn't require runs of the soccer simulator for testing each individual, improved overall performance of final solutions substantially.

Since evaluations in the second phase of our method are very time-consuming, we only implemented the method for the kick decision tree of agents, their decision tree for only when they have the ball. In future work, the move tree can also be implemented using the same method. It can then be examined if the hybrid GP method can also help soccer agents decide what to do when they don't have the ball.

Another important factor in this research that will affect the results is the opponent team which the individuals are being evaluated against in the second phase. Here we used Agent2D as opponent, which is the most widely used base code used in soccer 2D simulations. Depending on the behavior and power of the opponent team, the process of evolution for individuals might be different. This can also be investigated in future work to see how this factor might influence the final solutions.

Because of the nature of the pass skill function in *MarliK*, we only included one pass function in the terminal set of our GP and let *MarliK*'s pass function decide whether to perform a *direct pass*, a *leading pass*, a *through pass*, or a *cross pass*. Depending on the implementation type for pass skills, each type of these passes can also be used in the function set of both GPs. Moreover, other predicates might be considered for adding to the function set of both GPs that will directly affect the evolving decision trees.

Bibliography

[1] D. Andre. The automatic programming of agents that learn mental models and create simple plans of action. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'95, pages 741–747. Morgan Kaufmann Publishers Inc., San Francisco, 1995.

[2] D. Andre and A. Teller. Evolving team darwin united. In M. Asada and H. Kitano, editors, *RoboCup-98: Robot SoccerWorld Cup II*, pages 346–351. Springer Verlag, Berlin, 1998.

[3] J. Aronsson. Genetic programming of multi-agent system in the robocup domain. Master's thesis, Lund Institute of Technology, Sweden, 2003.

[4] O. Aşık and H. L. Akın. Solving multi-agent decision problems modeled as dec-pomdp: A robot soccer case study. In X. Chen, P. Stone, L. E. Sucar, and T. van der Zant, editors, *RoboCup 2012: Robot Soccer World Cup XVI*, pages 130–140. Springer Verlag, Berlin, 2013.

[5] W. Banzhaf. Evolutionary computation and genetic programming. In A. Lakhtakia and R. J. Martin-Palma, editors, *Engineered Biomimicry*, pages 429–447. Elsevier, Boston, 2013.

[6] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic programming: an introduction*. Morgan Kaufmann Publishers Inc., San Francisco, 1998.

[7] M. Chen, E. Foroughi, et al. Users manual: Robocup soccer server manual for soccer server version 7.07 and later, 2003.

[8] X. Chen, P. Stone, L. E. Sucar, and T. Zant. *RoboCup 2012: Robot Soccer World Cup XVI*. Springer Verlag, Berlin, 2013.

[9] E. Chown and J. Ruiz-del Solar. *RoboCup 2010: Robot Soccer World Cup XIV*. Springer Verlag, Berlin, 2011.

[10] J. Ferber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*, volume 1. Addison-Wesley Longman, Harlow, 1999.

[11] A. Fraser. Gpc++ genetic programming c++ class library, 1993. Available online at: http://www0.cs.ucl.ac.uk/staff/ucacbbl/ftp/weinbenner/gp.html.

[12] T. Haynes and S. Sen. Crossover operators for evolving a team. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 162–167. Morgan Kaufmann Publishers Inc., San Francisco, 1997.

[13] W. H. Hsu and S. M. Gustafson. Genetic programming and multi-agent layered learning by reinforcements. In W. B. Langdon et al., editors, *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 764–771. Morgan Kaufmann Publishers Inc., 2002.

[14] H. Iba. Emergent cooperation for multiple agents using genetic programming. In H. M. Voigt, W. Ebeling, I. Rechenberg, and H. P. Schwefel, editors, *International Conference on Parallel Problem Solving from Nature*, pages 32–41. Springer Verlag, Berlin, 1996.

[15] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. Robocup: The robot world cup initiative. In *Proceedings of the First International Conference on Autonomous Agents*, pages 340–347. ACM, New York, 1997.

[16] P. Lichocki, S. Wischmann, L. Keller, and D. Floreano. Evolving team compositions by agent swapping. *IEEE Transactions on Evolutionary Computation*, 17(2):282–298, 2013.

[17] S. Luke. Genetic programming produced competitive soccer softbot teams for robocup97. In J. R. Koza et al., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 214–222. Morgan Kaufmann Publishers Inc., San Francisco, 1998.

[18] S. Luke, C. Hohn, J. Farris, G. Jackson, and J. Hendler. Co-evolving soccer softbot team coordination with genetic programming. In H. Kitano et al., editors, *RoboCup-97: Robot Soccer World Cup I*, pages 398–411. Springer Verlag, Berlin, 1998.

[19] S. Luke and L. Spector. Evolving teamwork and coordination with genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Proceedings of the 1st Annual Conference on Genetic Programming*, pages 150–156. MIT Press, Cambridge, 1996.

[20] P. MacAlpine, K. Genter, S. Barrett, and P. Stone. The robocup 2013 drop-in player challenges: Experiments in ad hoc teamwork. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 382–387. IEEE, 2014.

[21] S. Nadarajah and K. Sundaraj. A survey on team strategies in robot soccer: team strategies and role description. *Artificial Intelligence Review*, 40(3):271–304, 2013.

[22] I. Noda. Soccer server: a simulator for robocup. In *JSAI AI-Symposium 95: Special Session on RoboCup*, pages 29–34. Japanese Society for Artificial Intelligence, Tokyo, 1995.

[23] L. Panait and S. Luke. Cooperative multi-agent learning: The state of the art. *Autonomous agents and multi-agent systems*, 11(3):387–434, 2005.

[24] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Lulu Press Inc., 2008.

[25] A. Qureshi. Evolving agents. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Proceedings of the 1st Annual Conference on Genetic Programming*, pages 369–374. MIT Press, Cambridge, 1996.

[26] S. Raik and B. Durnota. The evolution of sporting strategies. In R. Stonier and X. H. Yu, editors, *Complex Systems: Mechanism of Adaptation*, pages 85–92. IOS Press, Amsterdam, 1994.

[27] P. Stone and M. Veloso. Layered learning. In *European Conference on Machine Learning*, pages 369–381. Springer Verlag, Berlin, 2000.

[28] K. Sullivan and S. Luke. Real-time training of team soccer behaviors. In X. Chen, P. Stone, L. E. Sucar, and T. van der Zant, editors, *RoboCup 2012: Robot Soccer World Cup XVI*, pages 356–367. Morgan Kaufmann Publishers Inc., 2013.

[29] A. Tavafi, N. Nozari, R. Vatani, M. R. Yousefi, S. Rahmatinia, and P. Pirdir. Marlik 2012 soccer 2d simulation team description paper. In X. Chen, P. Stone, L. E. Sucar, and T. van der Zant, editors, *RoboCup 2012: Robot Soccer World Cup XVI*. Springer Verlag, Berlin, 2012.

Appendix A

Best Individuals of Populations

Best individuals from first and second phase of the method are shown below.

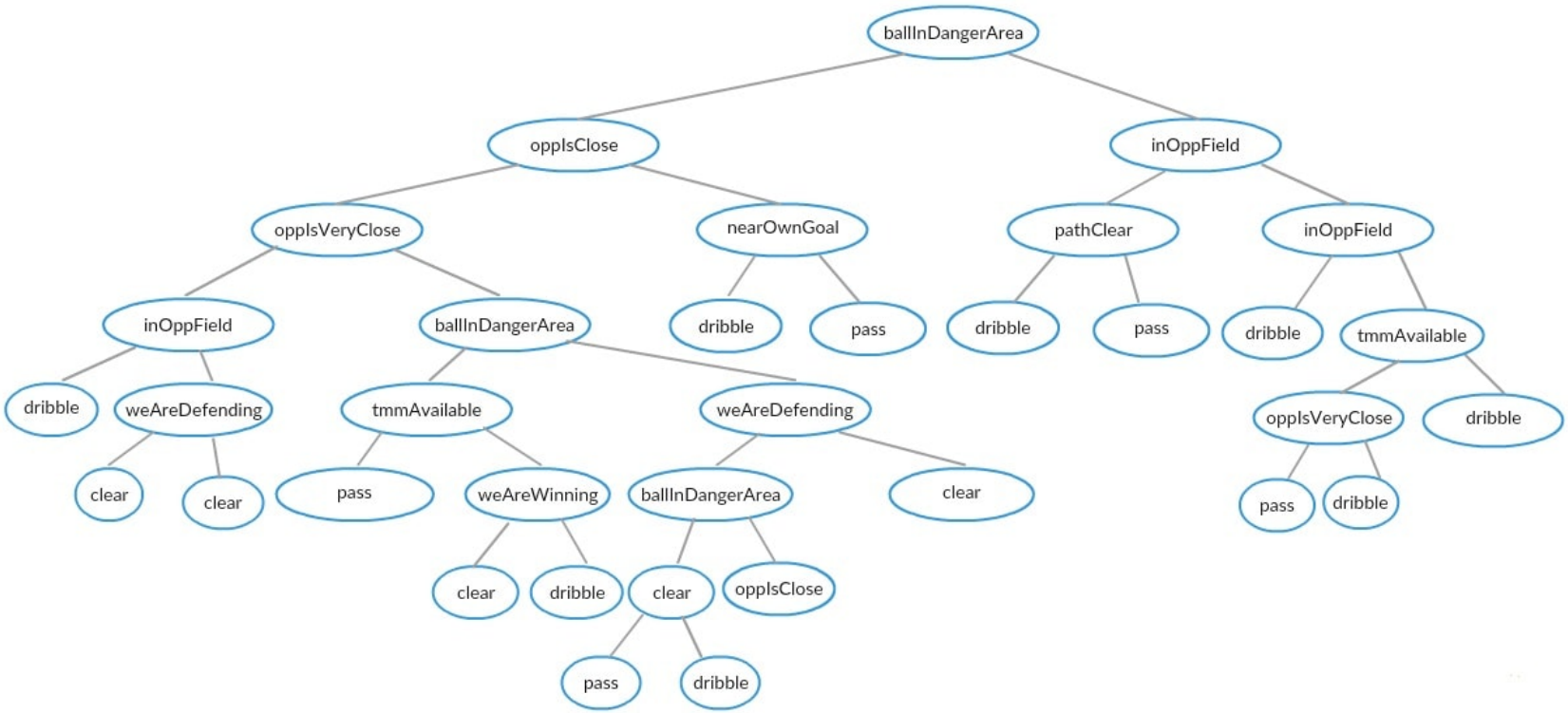

Figure A.1: Best individual from first phase of the algorithm.

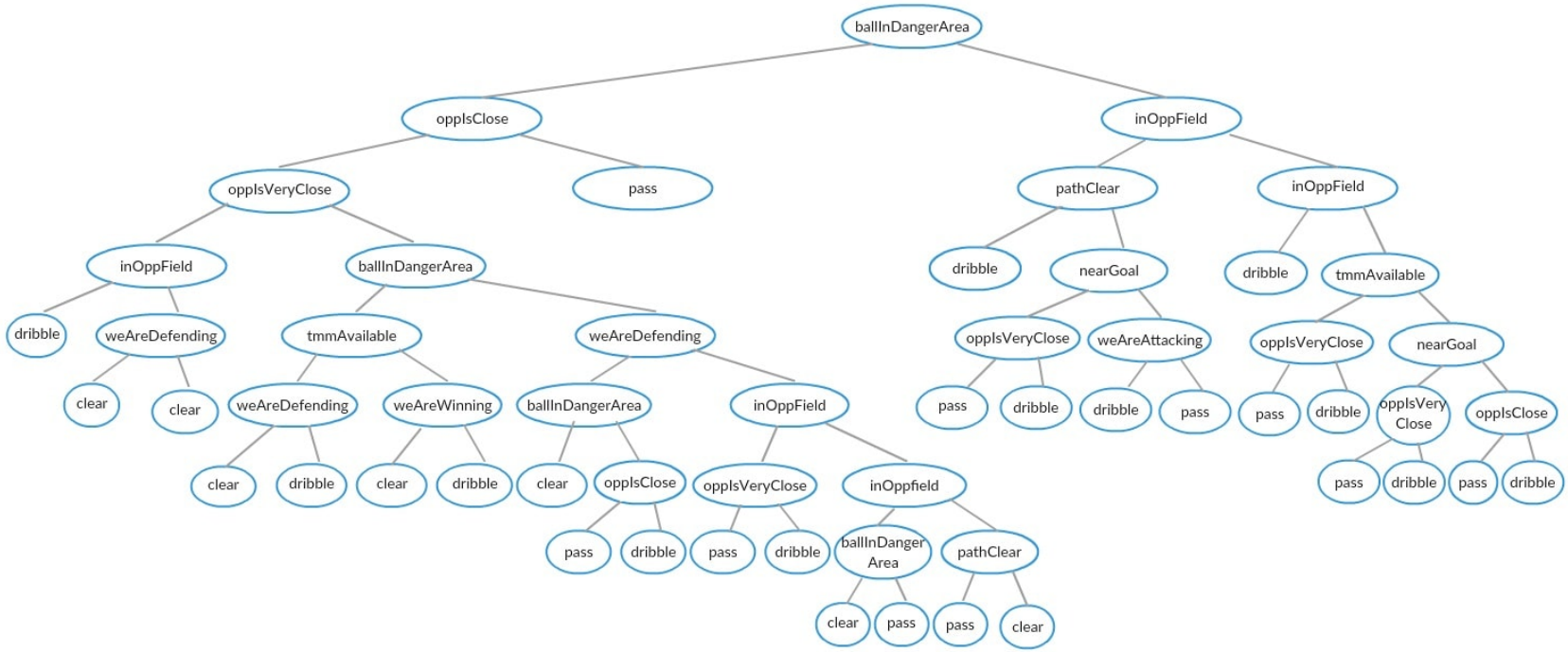

Figure A.2: Best individual of second phase of the algorithm.