Genetic Programming of Multi-agent System in the RoboCup Domain

by

Jonatan Aronsson

LUND INSTITUTE OF TECHNOLOGY
Lund University

A thesis presented to the Lund Institute of Technology for the degree of

Master of Science in Engineering Physics

Lund, Sweden, 2003

Supervisors

Jacek Malec
Department of Computer Science,
Lund University

Robert Gorbet
Department of Electrical and Computer Engineering, University of Waterloo

Abstract

Simulated robotic soccer is a frequently used as a test method for contemporary artificial intelligence research. It provides a real-time environment with complex dynamics and sensor information that is both noisy and limited. Team coordination between the robots is essential for success.

Genetic programming enables machines to learn skills and it is developed from the principle of survival of the fittest. A population of computer programs is generated and each program is tested against a fitness function. The best programs according to the fitness function are cloned, mutated, and recombined to create a new generation of programs. This process continues until the evolved programs satisfy a user defined criterion.

In this study, genetic programming is used to teach software robots to play soccer. The robots quickly learn to chase and kick the ball towards the goal. With time, a number of players in each team develop defensive abilities and recognize that team coordination is necessary for further development.

The fitness evaluation was extremely demanding and therefore, several compromises were made to limit the duration of the 'evolution'. Each run was reduced to two weeks and this compromise consequently resulted in weaker robots.

To develop better performing players, additional work should be carried out for the fitness evaluation, the set of terminals and functions should be extended, and more computational resources are necessary.

Acknowledgements

This study is a six month long master's graduation project which was performed during an exchange to the University of Waterloo and at my home university, the Lund Institute of Technology.

I would like to thank my supervisors, Jacek Malec for his guidance and advice, and Robert Gorbet for his supervision and care.

I would like to express thanks to Singh Sanjay, UNIX expert at the University of Waterloo, Fredrik Heintz, the developer of RoboSoc, and Tom Howard, the co-developer of the RoboCup Soccer Server, for their help and support with various software problems under Sun Solaris.

Thanks to Alissa Boroditsky for reading and pointing out errors in my draft of the thesis.

Also, a final thank you should be expressed to my parents, grandparents, and brother who have always been there for me during my exchange.

Table of Contents

Chapter 1

Introduction

Genetic programming is a method to automatically generate programs and algorithms through simulated evolution. These programs are constructed from a pre-defined set of functions and terminals. A simulation starts by creating an initial population of random programs. Like biological evolution, the process continues for generations by letting the best "genes" from every generation survive. These genes (parts of programs) are mutated and crossed in every evolving step. Genetic programming has proven to be useful in a wide range of applications, including multi-agent systems.

Multi-agent system is the sub field of artificial intelligence, which studies systems involving multiple agents and their coordination. An agent can be viewed as anything that is able to perceive information about the environment and perform actions upon it.

RoboCup is an annual competition between soccer playing robots for the purposes of research and education. It is designed to be an environment for the development of agents and it is an uncertain and dynamically changing domain. One of the many leagues in RoboCup is the simulated league in which the real world is simulated by a very complex system. This relieves researches from handling with hardware related problems, as well as today's limitations in robotics.

This study was initiated by the University of Waterloo's future plans to enter the RoboCup Simulation League. Therefore, the initial stage is focused on creating an environment for the development of simulated RoboCup players. The goal of this study, which was shaped during the initial stage, is to investigate how genetic programming can be used to teach robots to play soccer. A secondary goal is to explore their ability to incorporate team coordination.

This thesis is written for a wide range of readers and the required background is equivalent to the education of an upper year engineering and/or computer science student. This thesis is organized as follows:

Chapter 2 introduces the most essential background. It includes an introduction to RoboCup, multi-agent systems, genetic algorithms, genetic programming, and a survey of related work previously made in this domain.

Chapter 3 describes the approach and implementation of the experiments.

Chapter 4 presents the results of the most important performed experiments.

Chapter 5 summarizes and discusses the results.

Chapter 6 presents suggestions for future work.

Chapter 2

Background

2.1 RoboCup

RoboCup is a competition held between artificial intelligence researchers, which allow new ideas and developments to be tested against one another on an even basis. There are several separate leagues in the RoboCup competition - the robot leagues and the simulator league. Competitions in the robot leagues are played with real-world robots, moving a ball around a small soccer field. The simulation league allows software robots to compete within a computer simulated soccer environment. This relieves the researchers from handling robot problems such as object recognition, communications, hardware issues and today's limitations of robotics. The ultimate goal of RoboCup soccer as Noda states is "by the mid-21st century, a team of autonomous humanoid robots shall beat the human World Cup champion team under the official regulations of FIFA[1]" [Noda *et al.*, 1999]. This thesis deals only with the simulator league and all future references to RoboCup therefore refer to this simulated version.

2.1.1 Simulated League

A match is carried out in a client/server style with a server that provides a virtual field and simulates all movements of a ball and players. Each client (or player) in the simulation is its own process with communication between players limited to messages passed only through the server. Communication between the server and each client is done via UDP/IP sockets. A brief description of the RoboCup server follows while a full description can be found in [SS Manual, 2002].

[1] Fédération Internationale de Football Association (FIFA) defines the rules of soccer.

Command string	Description
`(turn 90)`	This command will turn the player's body 90 degrees relative to the current body direction.
`(kick 70 30)`	This command will kick the ball with a power of 70 in a direction of 30 degrees relative to the current body direction (if the player is close enough to the ball).
`(score)`	This command requests the server to send score information.

Table 1: Examples of player control commands. A full list of commands available to players can be found in [SS Manual, 2002, section 6.1].

Server message string	Description
`(see 400 ((b) 15 50))`	The *see* message contains information about objects that can be seen from the player's view. This message informs the player that at cycle 400, the ball is 15 meters away in a relative direction of 50 degrees.
`(hear 3000 referee half_time)`	The *hear* message returns the messages that can be heard through the field. This message informs the player that the referee announced half time at cycle 3000.
`(sense_body 500 (view_mode high narrow) (stamina 3000 2) ....)`	The *sense_body* message returns the states of the player as well as information to keep track of lost or delayed messages.

Table 2: Examples of server messages. A full list of available server messages can be found in [SS Manual, 2002, section 6.1].

The communication between players and the server is made with messages represented as strings. The messages sent by players inform the server of actions they wish to execute. The messages from the server inform players of the position of the ball, other players, lines, flags and goals which a player sees on the field. These soccer games are typically two halves each made up of 5 minutes and a half-time break. After a game has begun the server sends updated percept information to each player. This occurs once every cycle or once every

one and a half cycles. However, players are expected to send actions to the server once every cycle. A cycle is typically 100 milliseconds.

2.2 Multi-agent Systems

What are agents? There is not a general accepted definition of an agent but an adaptation from [Russel and Norvig, 1995] is:

Anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors.

Figure 1 illustrates agents in their environment. The agent is autonomous if it is capable of interacting independently with the environment and without interference by humans or other systems [Woolridge, 2002]. In most domains, agents do not have complete control over their environment and therefore, they cannot predict the outcome of an action. Each time an action is performed it could have a totally different effect on the environment. Also, an action may fail to have its supposed effect. Russell and Norvig suggest the following classification of an environment's properties:

- **Accessible vs. inaccessible**

  In an accessible environment agents can receive complete, accurate and up-to-date information about the environment's state. Most environments of reasonable complexity are inaccessible.

- **Deterministic vs. non-deterministic**

  If the environment is deterministic every action has one single guaranteed effect. A chess board would be a deterministic environment and the real world would be non-deterministic.

- **Episodic vs. non-episodic**

  In an episodic environment, an agent's experience is divided into episodes, where the quality of an action does not depend on previous episodes.

- **Static vs. dynamic**

  A static environment is only changed by the agent's actions, whereas a dynamic environment is affected by other processes beyond the agent's control.

- **Discrete vs. continuous**

  An environment is discrete if there are a limited number of clearly defined actions and precepts.

RoboCup's environment is inaccessible, non-deterministic, non-episodic, dynamic and continuous. Therefore, it is categorized as the most complex class of environments.

Multi-agent systems (MAS) focus on systems in which many autonomous agents interact with each other. The agents can share a common goal and be cooperative or their interactions can be selfish.

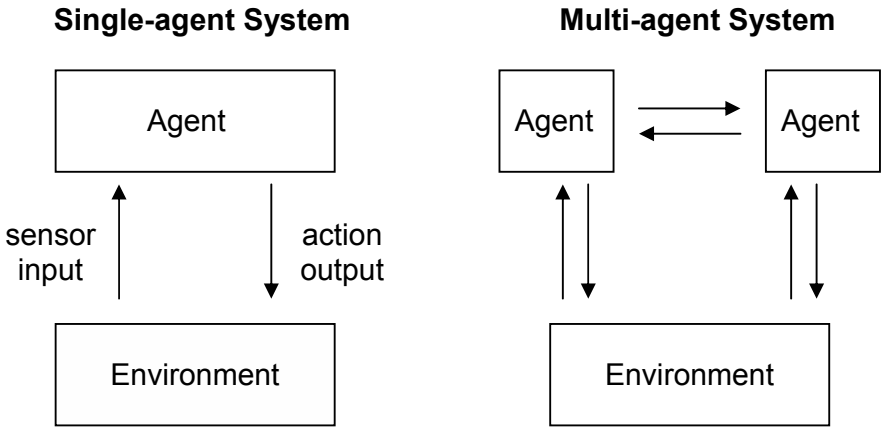

Figure 1: General frameworks for a single-agent system and a multi-agent system. Agents receive information about the environment through their sensors and perform actions on the environment. If other agents exist in a single-agent system, they are considered as a part of the environment. In multi-agent systems agents may interact directly as indicated by the arrows between the agents.

2.3 Genetic Algorithms

Genetic Algorithms were invented by John Holland and developed by his students and colleagues in the 1970s [Holland, 1975]. The method is inspired by the 'survival of the fittest' principle or Darwin's theory of evolution. A solution is represented as a genome, which in the basic case, is a string of binary values, letters or numbers. The genetic algorithm creates a population of genomes. It then applies crossover and mutation operators to those in the population, to create the individuals in the next generation. Different criteria are used to select the best individuals for the operators. Each individual is assigned a value, which determines the fitness of an individual.

The genetic algorithm is simple and its basics involve nothing more than merging and swapping strings. However, numerous modifications to the basic algorithm can be made and there are several parameters to tweak. But in most cases, modifications only result in minor improvements. Of more importance is

the objective function that determines an individual's fitness. A good objective function sets a value that is proportional to an individual's real fitness.

2.3.1 An example

A genome should contain the solution it represents. The most common encoding is binary strings. The genomes would appear as follows:

1 1 0 0 1 0 1 1 1 0 1 0 1 0 1 1

1 0 1 0 0 0 1 1 0 1 0 0 1 1 0 1

Each bit could possibly represent some characteristics in the solution or the whole string could represent a binary number. There are several ways of encoding and the choice is primarily arbitrary and depends on the problem.

The most important parts of the algorithm are the crossover and mutation operators along with the objective function. The design of the two operators is problem specific but a simple crossover operator is illustrated here:

Genome 1:	1 1 0 0 1 0	1 1 1 0 1 0 1 0 1 1
Genome 2:	1 0 1 0 0 0	1 1 0 1 0 0 1 1 0 1
Offspring 1 :	1 1 0 0 1 0	1 1 0 1 0 0 1 1 0 1
Offspring 2 :	1 0 1 0 0 0	1 1 1 0 1 0 1 0 1 1

The space illustrates the crossover point and is picked randomly. The crossover operator simply crosses two genomes to create two offspring and copy them into the next generation.

To prevent the solutions from getting stuck at a local optimum, some genomes in each generation are normally mutated. A mutation operator randomly changes some characteristics in a solution. In this example the mutation operator picks a few bits on random basis and inverts them:

Genome:	1 1 0 0 1 0 1 1 1 0 1 0 1 0 1 1
Mutated genome:	0 1 0 1 1 0 1 1 1 0 0 0 1 0 0 1

2.4 Genetic Programming

Genetic Programming is an extension of the genetic algorithm in which the genomes are computer programs. The idea of combining genetic algorithms and computer programs originated in the 1970s but it was John Koza [1992] who successfully applied genetic algorithms to the programming language LISP. Koza has showed in his books that this method can be applied to a wide range of problems [Koza, 1992, 1994 and 1999]. Examples of solutions to these problems include the automated synthesis of analog electrical circuits, the automatic discovery of detectors for letter recognition, the obstacle-avoiding

robot, the minesweeper problem and multi-agent programming. Even though genetic programming seems to be a general method, there are types of problems for which it has not yet demonstrated success.

2.4.1 Introduction

Genetic programming is a method that gives computers the capability to automatically learn problem-solving abilities without explicitly being programmed. It uses the genetic algorithm to evolve programs. The evolved programs are made up of functions and terminals, which are combined into a tree-like hierarchal structure. Functions form the internal nodes and terminals form the leaf nodes. When a program is executed, the tree is traversed by evaluating the root node first, which in turn evaluates its arguments and so forth. This continues on until the leaf nodes are traversed.

Genomes are evolved in every generation. Between generations, each genome is assigned a fitness value, which determines its probability to become a part of following generations. The fitness value should reflect the performance of a genome. Therefore, the best performing programs will be more likely represented in following generations. This is the fundamental concept underlying genetic algorithms and is an imitation of Darwin's theories on biological evolution. Figure 2 illustrates the flow sequence of the genetic algorithm.

2.4.2 Preparatory Steps

Before genetic programming can be applied to a problem a few preparatory steps must be taken. These steps include:

- Defining the set of terminals and functions.
- Determining the fitness measure.
- Specifying control parameters for the run.
- Defining the termination criterion.

Terminals and Functions

The terminals correspond to the inputs of a program. These inputs can be constants, random values, variables, instructions, etc. The functions may be logical expressions, mathematical functions, operations or problem specific functions and operators.

A genome or program is made up of terminals and functions that express a trial solution. These two sets must be defined so that a program is capable of expressing the solution. Successfully designing the sets of functions and terminals may not be trivial.

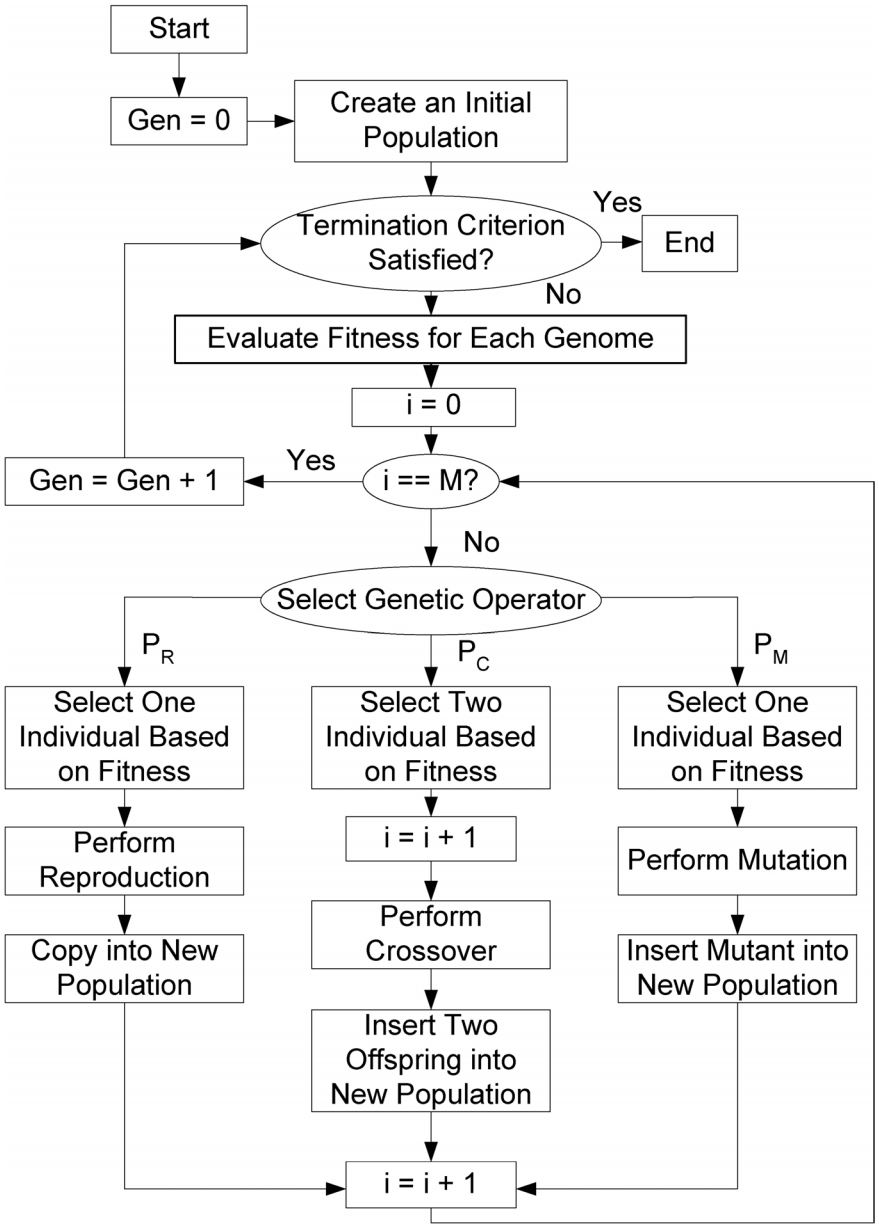

Figure 2: The flow sequence of the genetic algorithm. The flowchart illustrates the standard implementation of the genetic algorithm. An equivalent illustration for genetic programming would be identical as only the design of the genomes and operators differ. Gen refers to the current generation number M is the population size and i is the current genome index in the generation. The genetic operators are selected with the different probabilities P_R, P_C and P_M, whose sum is one.

Fitness Assignment

The fitness assignment is what drives the population towards the solution. Its purpose is to reward genomes depending on how close they are to the solution. An optimal solution should receive a high reward and reasonably good solutions should also be rewarded.

A genome's fitness value is based on how well it performs a certain task. This value can be based on the error in relation to a known optimal solution or a desired goal. The value may also be acquired by letting the genomes in a population compete against each other. This is often referred to as a competitive fitness assignment.

An error driven fitness measurement works fine for problems with a known desired solution. This is often the case for simple problems. However, in more complex domains, the desired goal is often too weakly defined for an error driven measurement. A more common alternative is that a measurement for comparing genomes against each other is available. Such a measurement will assign a genome's fitness in comparison to the rest of the population. One strategy for the use of a competitive fitness assignment is to hold a tournament where every genome plays against all other genomes. This strategy requires $\frac{M(M-1)}{2}$ fitness evaluations for a population of size M, which in most cases are too expensive. Several alternative strategies have been investigated to overcome this problem.

Angeline and Pollack examined a strategy that they called 'tournament fitness assignment' on the game of Tic Tac Toe [Angeline and Pollack, 1993]. Genomes in a population are paired up randomly to make a match and the winner will advance to the next round. If a match ends in a draw, one of the two genomes is selected at random to continue to the next round. This persists until there is only one winner. The genomes are then ranked based on the round they were eliminated in the tournament. This method has an advantage because it only requires *M-1* fitness evaluations. Angeline and Pollack compared their method with other strategies and found that 'tournament fitness assignment' was significantly better for the game of Tic Tac Toe.

Craig Reynolds explores another tournament design, where each genome plays a small number of fixed matches [Reynolds, 1994]. He applied this strategy to the game of Tag and each genome in the population played seven games against different genomes. The fitness was then set to the average of the seven values. If each genome played N other genomes, the number of fitness evaluations is $\frac{MN}{2}$.

Control Parameters

There are several parameters that control a run. The primary parameter is the population size, but there are a number of quantitative and qualitative

control parameters that must be specified. Koza [1995] described 19 secondary parameters. These include probabilities for selecting the different genetic operators during a run (shown in Figure 2) as well as the initial and the maximum permitted program size.

Termination Criterion

The usual termination criterion to stop the run is when the best solution passes a fitness test. The maximum number of generations to be run is normally specified along with the fitness test.

2.4.3 Execution Steps

When the preparatory steps are completed the run may begin. The execution steps define the actual run and are illustrated in Figure 2. The basic algorithm can be summarized in three steps:

- Randomly create an initial population.
- Iteratively execute the following sub-step until the termination criterion is satisfied.
    - Assign each genome a fitness value.
    - Create the next generation by repeatedly selecting the following genetic operators based on the operator's probability.
        - Reproduction
        - Crossover
        - Mutation
- The run is stopped and a solution may be presented.

2.4.4 Genetic Operators

There are mainly three operators as previously mentioned.

Reproduction Operator

A genome is selected based on its fitness and copied into the next generation.

Crossover Operator

Crossover operates on two parental genomes that are selected based on fitness. Swapping sub-trees between the parental genomes creates two offspring. The operator is demonstrated in Figure 3.

Mutation Operator

A genome is selected based on its fitness. A copy of the selected genome is mutated and inserted into the next generation. Koza tried to claim in his two first books [1992 and 1994] that genetic programming does not perform a random search, thus mutation is not necessary. A wide range of problems is solved without the use of mutation. However, the use of mutation in genetic programming is increasing and is used particularly for small population sizes. There are several ways of mutating a genome. The method used in this study is called sub-tree mutation and is demonstrated in Figure 4.

2.4.5 Premature Convergence

Premature convergence occurs when the fitness of a population converges to a sub-optimal solution. One indicator that a population converges is a decreasing diversity in a population. This is a common problem in genetic programming but according to Koza, it should be viewed as a part of the nature of genetic algorithms.

Koza believes that the best way to prevent premature convergence is to restart the whole run when it occurs. This may be optimal for problems with a cheap computational fitness evaluation but not for problems with complex fitness measures such as RoboCup. For example, in the RoboCup domain a run is computationally very expensive and a re-run might therefore not be the best solution. Alternative methods are to increase the population size or the mutation rate thereby improving the diversity of a population.

2.5 Related Work

Several attempts to apply genetic programming to the RoboCup domain have been made since the first international RoboCup competition, 1997.

Luke *et al.* [1997] developed a few RoboCup teams with the use of genetic programming. They entered the first international RoboCup competition with two of these teams and qualified to the third round. One team was ‘homogenous’ and the other was ‘pseudo-homogenous’. The homogenous team consisted of players with identical programs and the other team was made up of squads. Each squad was composed of three to four identical programs. A program consisted two sub-programs, a kick-tree and a move-tree. The kick-tree was executed when the ball was kickable whereas move-tree was executed otherwise.

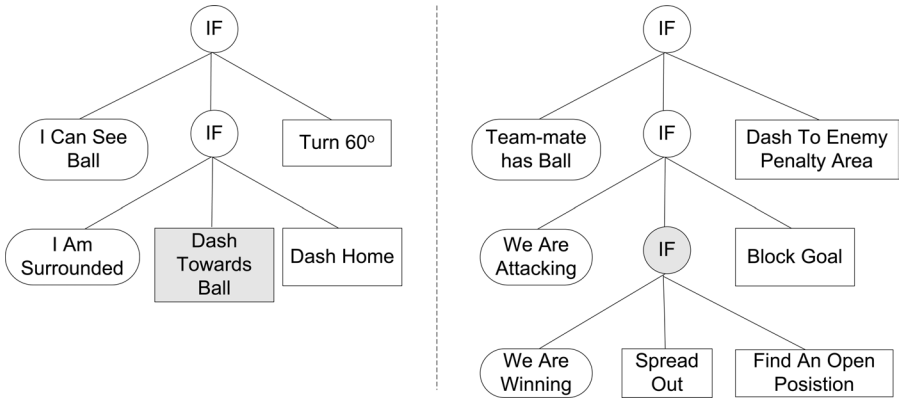

Figure 3a: Crossover operator - two parental programs. Crossover points are selected randomly over all nodes and are highlighted in the figure.

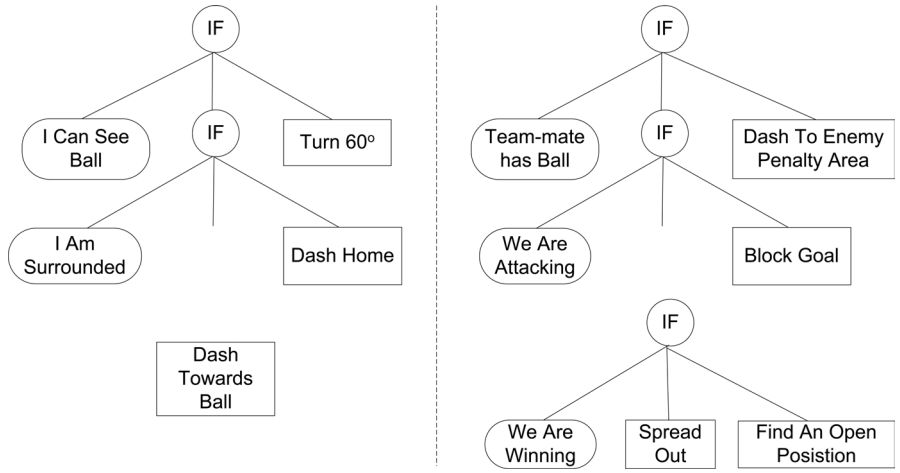

Figure 3b: Crossover operator - two sub-trees and two remainders. The sub-trees are selected with the crossover points as root nodes.

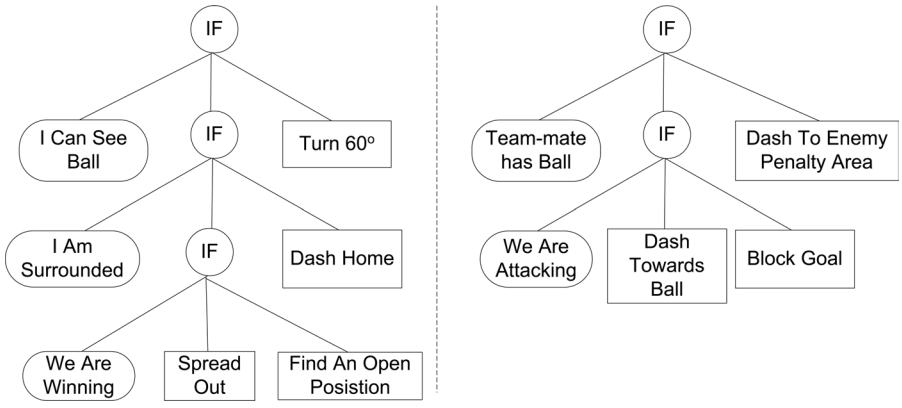

Figure 3c: Crossover operator - two offspring. The sub-trees are swapped and inserted back into the remainders.

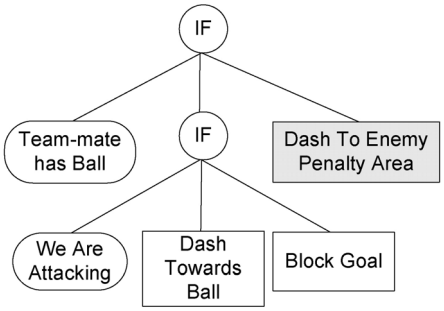

Figure 4a: Sub-tree mutation - initial program. A mutation point is selected randomly among all nodes (highlighted in the figure).

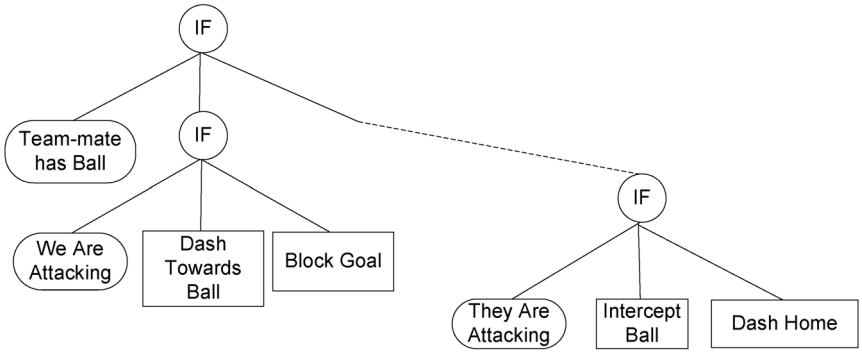

Figure 4b: Sub-tree mutation. The sub-tree with the mutation point as root node is deleted and a new random tree is created. The random tree is then inserted into the tree as showed in the figure.

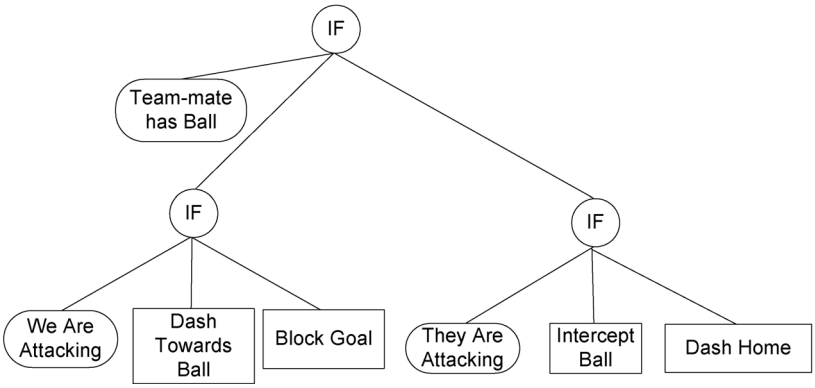

Figure 4c: Sub-tree mutation - mutated tree.

Luke's players learned to run after the ball and kick it towards the opponent's goal. They also learned some basic defensive abilities. It was the less complex homogenous team that performed the best. However, the researchers believed that the pseudo-homogenous team would outperform the homogenous team if it was given additional time to 'practice'.

The fitness function was only based on the number of goals scored by the team. To prevent premature convergence on the relatively small population size (128 genomes), a high mutation rate of 30 % was used.

Andre and Teller [1999] explored a fitness measurement, which was founded on human coaching principles in soccer. They developed a fitness function based on the observed hierarchal behaviour of human soccer players. This fitness function rewarded players by taking into consideration their position, distance to ball, number of goals scored and number of kicks.

Gustafson and Hsu [2001] explored an alternative to the basic genetic programming method, which applied layered learning techniques. In layered learning, several runs are performed in a sequence. Consequently, the initial population for one run is the final population from the previous run. This design facilitates the use of different control parameters and fitness functions for each run.

This method was applied to keep-away soccer, which is a sub-problem in the RoboCup domain and required multi-agent cooperation. The presented results showed that layered learning in genetic programming outperformed the standard method.

Ciesielski, Mawhinney and Wilson [2003] presented three different approaches to create RoboCup players using genetic programming. In the first experiment, the only actions available to the programs were those provided by the soccer server. The second experiment employed higher-level actions such as 'kicking the ball towards the goal' or 'passing to the closest team-mate'. These two experiments used a tournament fitness assignment [Angeline and Pollack, 1993] while the third experiment was a slight modification of the first.

The teams created by the first and third approaches performed poorly. The players from the second experiment were able to follow the ball and kick it around. Ciesielski *et al.* concluded that the use of genetic programming enabled teams to perform well. However, a significant amount of work is still needed for the development of higher-level functions and the fitness measure.

Two previous students, Niklas Persson [2001] and Christian Rahm [2001], did their master's graduation projects at the Lund University in the RoboCup domain.

Persson implemented a RoboCup team and explored the design of decision trees for different player roles (attacker, midfielder, and defender). He

concluded that the design of the players' low-level behaviour is essential for success.

Rahm explored the use of neural networks to improve RoboCup players' kicking behaviour. The completed experiments demonstrated that his learning approach was not successful for this problem. Rahm also discussed the problem of synchronisation between the players and the soccer server.

Chapter 3

Implementation

3.1 Agent Architecture

A RoboCup agent receives a large amount of unformatted data from the soccer server every cycle. The agent must first convert and interpret the server messages into a suitable data representation that fits its world model. In this project, the RoboCup Client Parser [RCC Manual, 2003] and the RoboSoc framework [RS Manual, 2002] are used to parse and update the agent's world model. The RoboCup Client Parser handles the interactions with the Soccer Server and parses the server messages to C++ objects. The RoboSoc framework processes these objects and updates the world model. Strategies define how the world model is updated based on new information and past world models. The information from the world model is extracted by Views in the RoboSoc framework. The Views present the information in an accessible way to the decision-making procedures. An overview of the architecture is presented in Figure 5.

The controller (Figure 5) directs the decision-making. Depending on the current state of the game and the agent, it can either use the evolved algorithm or a pre-defined behaviour to decide an action. Predicates are used by the evolved algorithm to test the current state of the world.

The evolved algorithm is executed by the controller when *play_on* [SS Manual 2002, section 4.7] is the play mode. The decision making for other play modes is pre-defined in the controller. The standard behaviour for kick offs, kick ins, free kicks and corner kicks is that the player closest to the ball goes towards it and passes it to a free team-mate. If no team-mate is free, the player passes the closest team-mate.

If the agent is a goalkeeper, the evolved algorithm does not affect its behaviour.

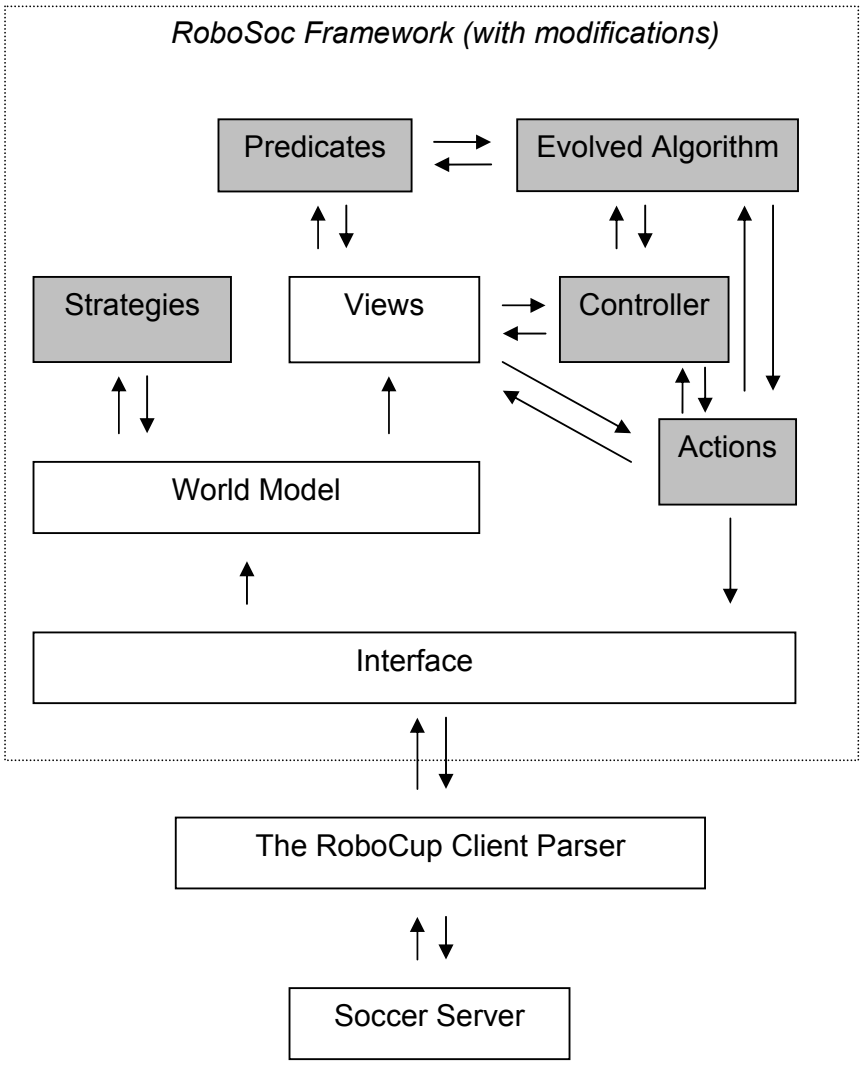

Figure 5: The agent's architecture. Coloured boxes indicate where modifications were made to the RoboSoc framework [RS Manual, 2002, page 6].

3.2 Genetic Representation

3.2.1 Implementation

An evolved algorithm is constructed as a decision tree with predicates as terminals and actions as functions. A list of all predicates and actions is presented in Appendix A. The algorithm is as follows:

```
If the ball's position is unknown
      Turn to look for ball
Else if the player can kick the ball
      If the player has a good chance to score
            Shoot to score
      Else
            Call the player's Kick Tree
Else if a team-mate passes the ball to the player
      Intercept ball
Else
      Call the player's Move Tree
```

The only parts of the algorithm that actually are evolved through genetic programming are the *Move Tree* and the *Kick Tree*. These trees will output an action each time they are executed. The move tree is executed when the position of the ball is known but the ball is too far away for a kick. The other tree is only executed if the ball is kickable.

The evolved decision trees are built of nodes and each node is either a predicate or an action as presented in Figure 6. A predicate tests if the world is in a certain state and returns either true or false. An action sends a command to the soccer server.

The population consists of several individuals and each individual has one kick tree and one move tree. The initial trees are created randomly under a number of restrains. For example, the initial height is pre-defined to an interval.

If a genetic operator is applied to an individual, the same kind of operator is applied to both its trees. A crossover is not allowed between a move tree and a kick tree. For example, if a crossover operator is applied to two individuals, the two move trees and the two kick trees are crossed separately. If the crossover results in a larger tree than maximally permitted tree size, the sub-tree added by the operator is shrunk to one node. This one node is randomly picked among the sub-tree's leaf nodes.

The mutation operator will first pick a random node in the decision tree. All nodes have an equal probability of being picked. The sub-tree with the picked node as a root node is deleted and replaced by a new tree. This new tree is constructed in the same way as the initial individuals. An upper limit for the tree's size is established so that the whole tree will not exceed the maximum size allowed. The reproduction operator merely copies an individual to the next generation without restrictions.

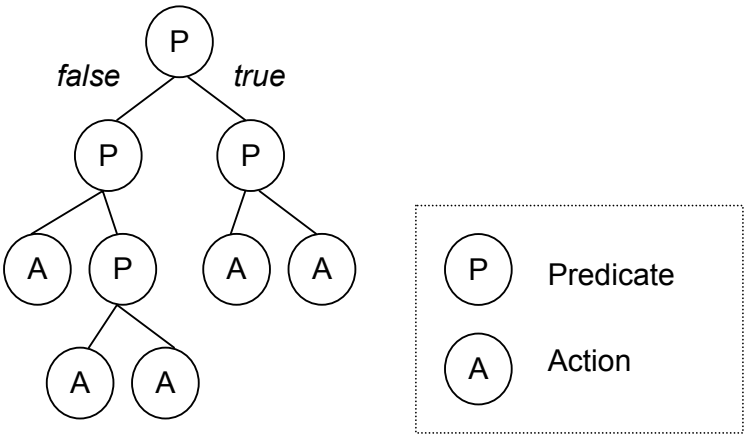

Figure 6: The structure of a decision tree. A leaf node is always an action and all other nodes are predicates.

3.2.2 Motivation

The approach utilized in the evolution of the two different trees (a move tree and a kick tree) has previously been tested by Luke *et al.* [1997]. Another obvious approach is the use of only one tree and allowing the agent to learn when it can and cannot kick the ball. With the two trees approach, a player will always try to kick the ball whenever it has the opportunity. This may not be the best alternative for all situations. For example, some situations may be enhanced if the agent turned around to look for open team-mates before kicking the ball. However, this would result in a more complex problem because the agent needs to learn when the ball is kickable. The RoboCup environment is already very complex and therefore, it may be beneficial to limit the search space of algorithms. The second approach has been applied by David Andre and Astro Teller [1999] and by Vic Ciesielski, Dylan Mawhinney, and Peter Wilson [2003]. My motivation for using the first approach is to limit the search space without drastically weakening the quality of possible solutions.

Those players that believe they have a good chance to score, will attempt to score. This behaviour overrides the evolved algorithm only if a player has an obvious chance of scoring. An expected result is that the agents will concentrate more on team coordination and positioning than on kicking the ball towards the goal.

The number of possible predicates limits the agents' input. The design of the set of predicates is not obvious. If a small set is used, the implementation of each predicate will be important and a gap between the predicates and desired solutions must not be present. A large set would minimize this problem even though it is still important that an algorithm is capable of expressing the desired solution. However, a large set results in a large search space.

3.3 Fitness

Each player is assigned a value based on its performance during a game. If a player plays several games the value is the average of all assigned values. The performance is calculated as weighted sum of the assessments in Table 3.

After the fitness assignment the genomes are ranked based on their fitness. The genomes are then assigned a probability value, which correspond to the probability to be picked by an operator. Figure 7 illustrates how the probability depends on order of fitness.

Assessment	Value
Won	1 if the player's team won 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 position 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 the player spent on the opposite half of the field, measured in the percentage of total time.

Table 3: Fitness assessments.

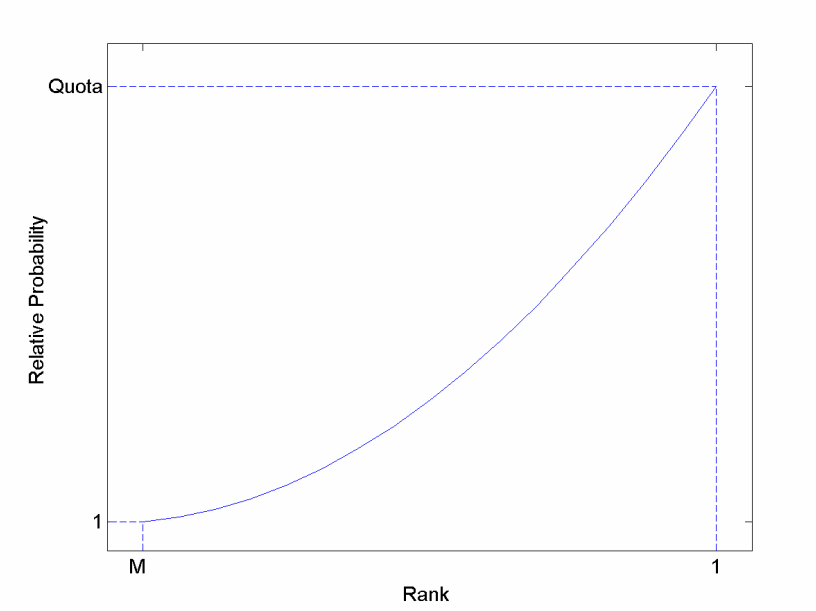

Figure 7: Selection probability depending on rank. M is the population size and one is the highest ranking. Quota is a parameter that modifies the quadratic probability function.

Chapter 4

Experiments

Several experiments have been performed throughout this research. Two selected experiments are presented in this chapter. These experiments are representative of what has been done and provide support for later conclusions.

All experiments were performed under Linux RedHat 9 on a Toshiba laptop with a Celeron 1.33 GHz processor. This configuration was not sufficient for the use of a full size team for the fitness evaluation. Therefore, a team of 9 members, including a goalkeeper was used instead of the 11 players.

4.1 Experiment 1

This experiment may be perceived to be a first investigation into the kind of behaviour that the evolved soccer robots can develop. In this investigation each player in a team is based on a separate algorithm. Players with different algorithms must cooperate in order to achieve team coordination.

4.1.1 Approach

Each player in the population is regarded to be an individual player. Matches are carried out between teams that are made up of players randomly picked from the population so that each player plays a fixed number of games. The fitness is then calculated in this manner (refer to Table 3 for more details):

$$
\begin{aligned}
& 200 * \textit{Won} \\
& 200 * \textit{Team Score} - \textit{Opponent Score} \\
& 250 * \textit{Score} \\
& 200 * \textit{Attempts} \\
& 30 * \textit{Kicks} \\
& 100 * \textit{Passes} \\
+\ & 200 * \textit{Active}
\end{aligned}
$$

Parameters used in this experiment are presented below:

Parameter	Value
Population size	128
Number of generations	52
Probability for crossover	60 %
Probability for reproduction	20 %
Probability for mutation	20 %
Number of games per player	4
Quota (Figure 7)	8

4.1.2 Result

Most players from the early generations are drifting around the field with what appears to be unsystematic movements. The players that run after and kick the ball increase in number rapidly until most players are chasing the ball (after approximately fifteen generations). Typically, most players will run after the ball and kick it towards the goal or eventually pass to a team-mate. This strategy is often referred to as “kiddie-soccer”. The average fitness reaches a maximum after approximately twenty generations and fluctuates around this value within the remaining generations. This is demonstrated in Figure 8. The game statistics are presented in Figure 9. A screenshot from a match between an early team and a ‘developed’ team is shown in Figure 10.

By manually observing games, players are perceived to improve slightly in the later generation. Particularly, a number of players develop defensive abilities that reduce the efficiency of the early strategy. In later generations, more players use the dribble skill rather than just kicking the ball towards the goal. Kicking the ball towards the goal was early a very successful tactic to quickly position the ball close to the goal. However, a defensive player would intercept a ball without difficulty, which headed in the direction of the goal. The dribble skill allows a player to dispense the ball around opponent players and thereby avoid the defence. However, the dribble skill also results in a slower advancement, as the ball is not going in a straight line. Figure 11 illustrates how two defensive players are positioned to defend their home goal.

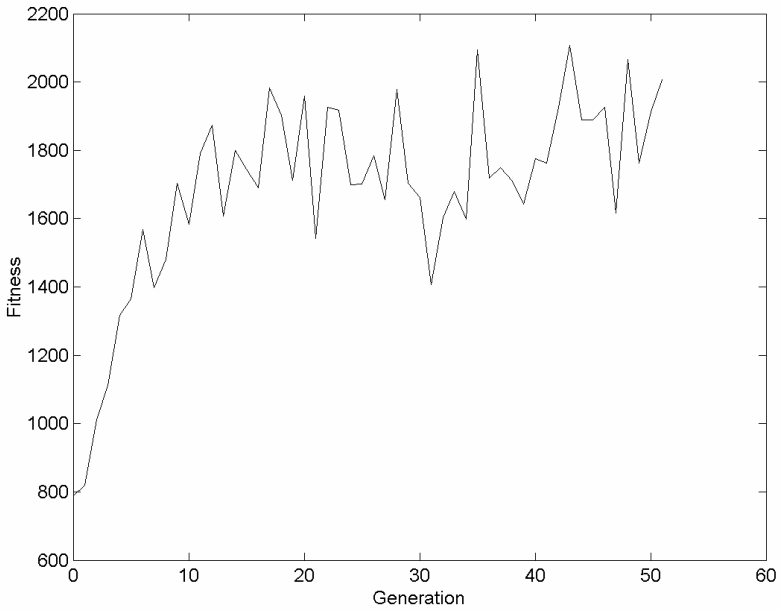

Figure 8: Average fitness in experiment one.

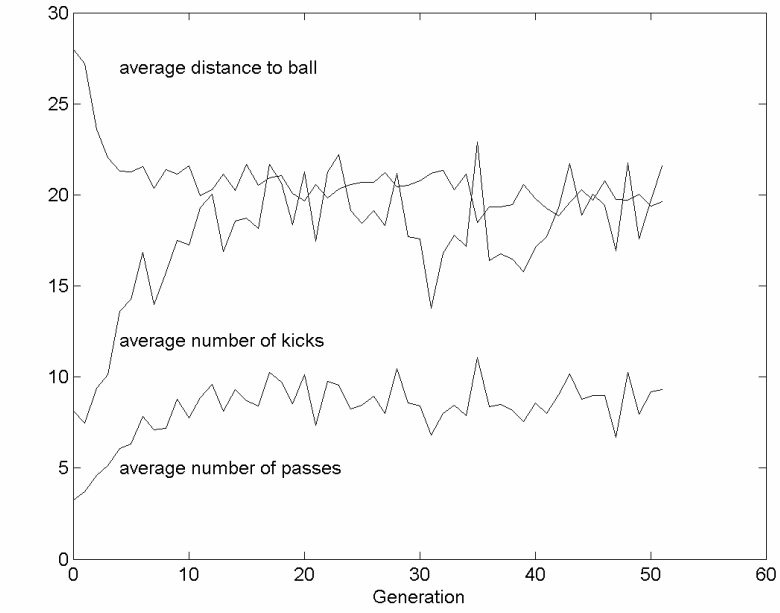

Figure 9: Game statistics from experiment one. The values are the average per player and match.

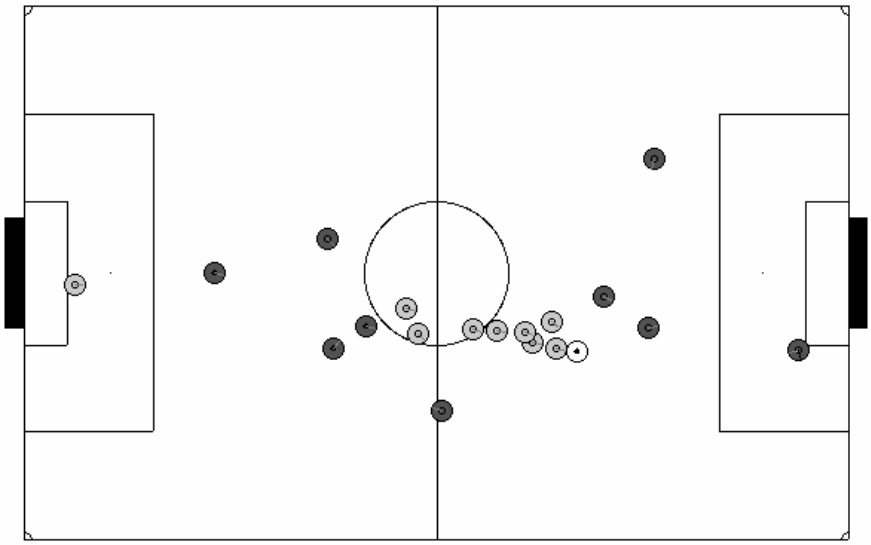

Figure 10: A screenshot from a match between an early team (on the right) and a team from generation 32.

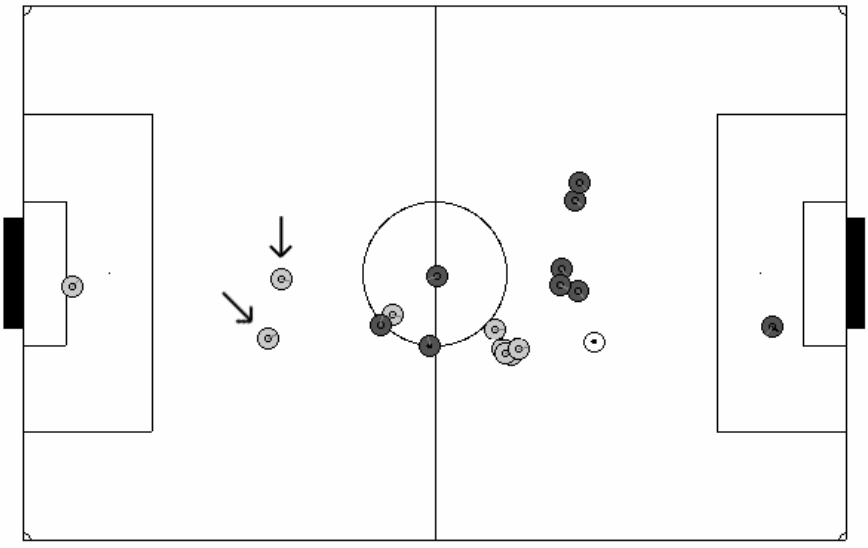

Figure 11: A screenshot from a match between two later teams. The arrows indicate two players that are using the BlockGoal action to defend the home goal, while other players are chasing the ball.

4.1.3 Conclusion

The software robots clearly improve their playing techniques throughout the generations. Initially, most players are wandering around the field illogically. The players rapidly develop a strategy that basically, is to run after the ball and kick it towards the goal. This strategy is very efficient against opponents that do

not have a defence besides the goalkeeper. Finally, the population learns defensive abilities and dribble skills.

The average fitness for the populations converges after approximately fifteen generations. However, the fitness is measured amongst players within the same generation and not in comparison to earlier teams. This is most likely the reason why the development of defence did not appear on the fitness scale (Figure 8).

4.2 Experiment 2

In previously performed work, a homogenous team configuration has been used. Luke *et al.* [1997] claimed that the evolution of a team with separate algorithms is a more complex problem than evolution of a homogenous team. This is because all players use the same algorithm. Luke also explored a pseudo-homogenous approach, where the players are put into different squads. This experiment investigates the pseudo-homogenous team configuration.

4.2.1 Approach

Instead of allowing each player to have its own algorithm, a team is built of three squads (each consisting of identical players). In order to increase the team coordination, each squad is given a separate role. The assigned roles are: attacker, midfielder, and defender. A team consists of 3 attackers, 3 midfielders, and 2 defenders.

The squads are co-evolved so that each squad is evolved from a separate population with a separate fitness function. The following are the formulas for the fitness functions:

Attacker

$$
\begin{array}{rrl}
 & 500 * & \textit{Score} \\
 & 400 * & \textit{Attempts} \\
 & 50 * & \textit{Kicks} \\
 & 100 * & \textit{Passes} \\
 & 200 * & \textit{Active} \\
 & 150 * & \textit{Ball Close} \\
 & 3 * & \textit{Time Free} \\
+ & 6 * & \textit{Time Offensive}
\end{array}
$$

Midfielder

	100 *	*Won*
	200 *	*Team Score – Opponent Score*
	100 *	*Score*
	100 *	*Attempts*
	50 *	*Kicks*
	300 *	*Passes*
	200 *	*Active*
	6 *	*Time Free*
+	2 *	*Time Offensive*

Defender

	200 *	*Won*
	200 *	*Team Score – Opponent Score*
	50 *	*Kicks*
	100 *	*Passes*
	200 *	*Active*
+	5 *	*Time Free*

Control parameters used in this experiment are presented below:

Parameter	Value
Population size	3 * 32
Number of generations	55
Probability for crossover	60 %
Probability for reproduction	20 %
Probability for mutation	20 %
Number of separate games per player	2
Quota (Figure 7)	8

4.2.2 Result

The early random teams acted uncoordinated and approximately one third of the initial players chased the ball like the initial teams from experiment one. After five to ten generations, the populations reached sub-optima that they essentially sustained for the remaining generations. This is demonstrated in Figure 12. These optima refer to kiddie-soccer players but they did not kick the ball towards the goal in the same amount as players from the first experiment.

An improved scoring strategy for the fitness function was then established – the players surrounded the ball and passed it around. This strategy generated many kicks and passes, which is also apparent in Figure 13 and 14. The number of performed passes per game is approximately two times the number completed in the first experiment.

However, the players were occasionally spread out and were able make long passes. This situation permitted a faster game because the ball quickly travelled all over the field. After a few long passes, one player usually failed to intercept the ball and cause all players to chase the ball. The players had problems intercepting passes especially the long passes because the ball had a higher speed.

The average distance between players and the ball decreased rapidly during the very first generation and then maintained a reasonably constant level when most players developed the kiddie-soccer strategy. For the defenders, this average distance was significantly larger. This is due to the fact that their initial positions were the furthest away from the ball's initial position. Figure 15 illustrates how the average distances are dependent on the generation number.

It appears that the different squads did not develop separate behaviours. The attackers did not attempt to score more frequently than the midfielders. For example, in generation 50 – 55, the attackers made 39 % of all scored goals, the midfielders 47 %, and the defenders 13 %.

Nearly all of the top ranked individuals from the populations of attackers and defenders executed the PassBeckham action when introduced with a chance to kick the ball and the DashToBall action otherwise. The midfielders showed greater variability and the winning algorithms changed frequently throughout the generations.

In order to compare players with those in the previous experiment, thirty-two matches between the two populations were carried out. The teams were selected randomly for each match. The teams from the previous population won eighteen games, the pseudo-homogenous teams from this experiment won nine, and five matches ended in a draw.

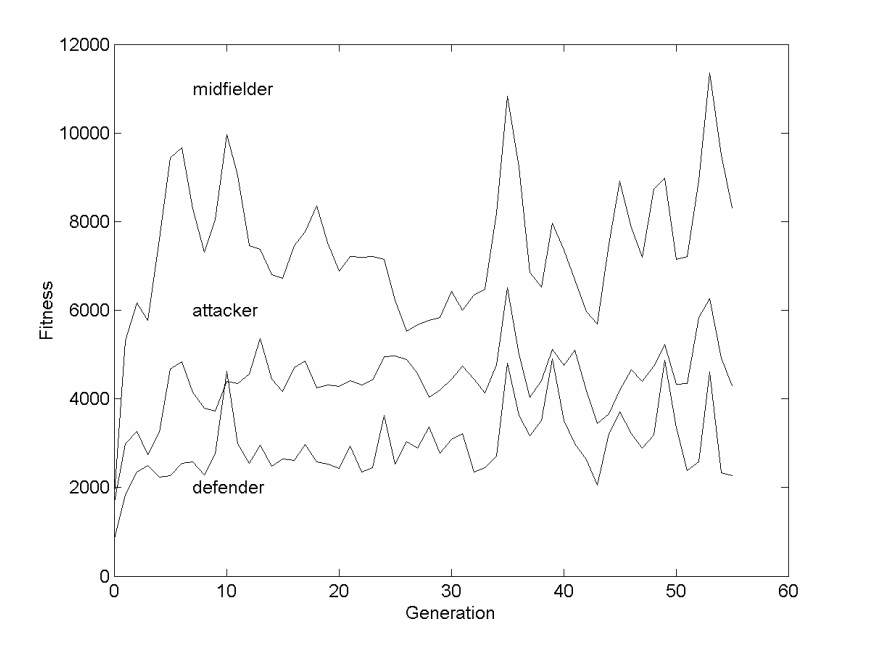

Figure 12: Average fitness for experiment two.

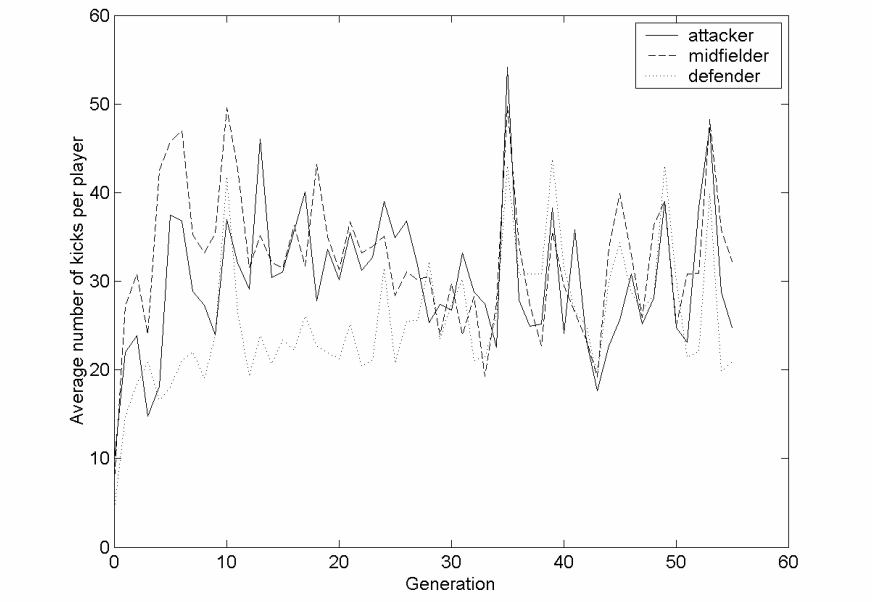

Figure 13: Average number of kicks performed during a match.

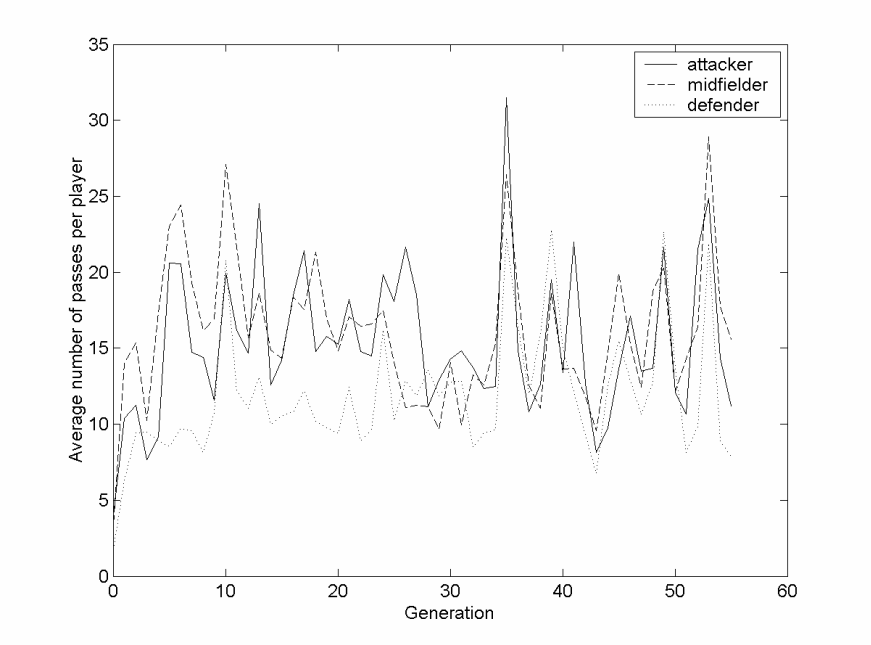

Figure 14: Average number of passes performed during a match.

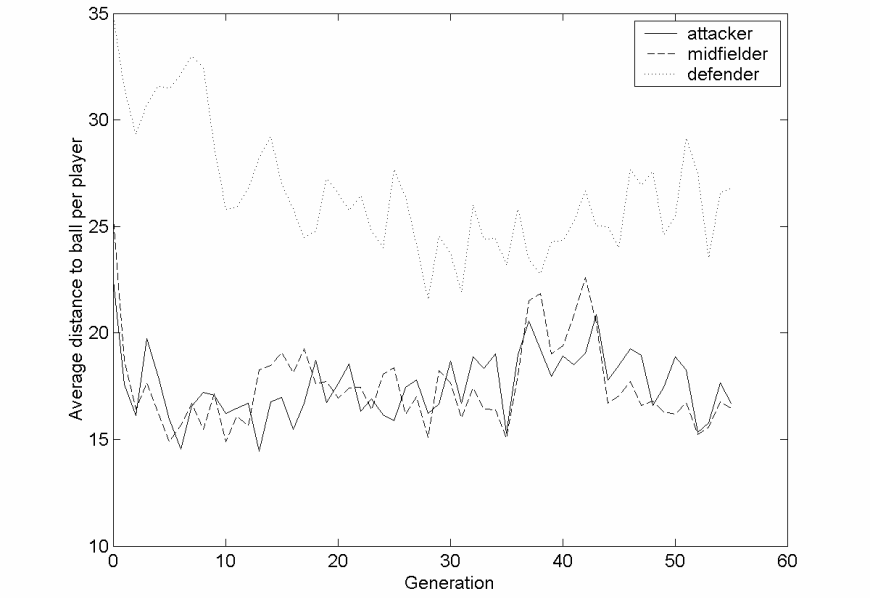

Figure 15: Average distance between the player and the ball.

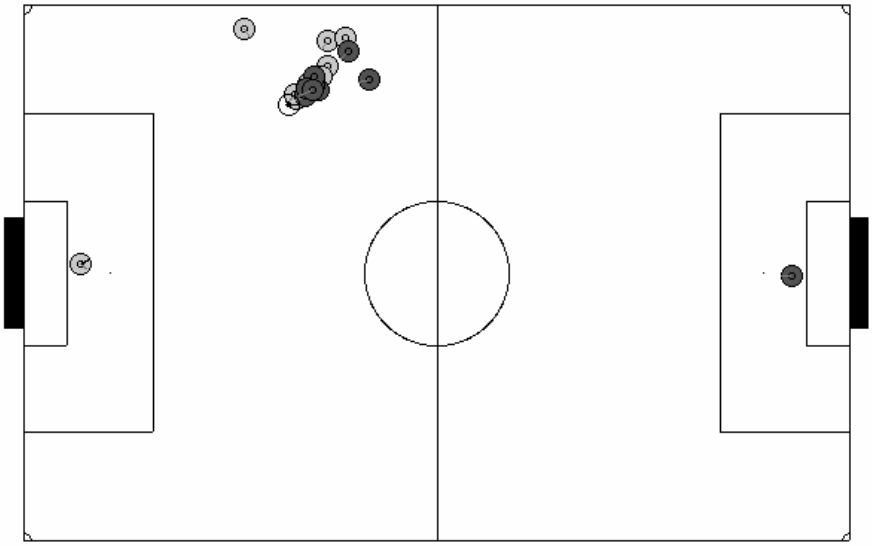

Figure 16: Screenshot of a typically situation in experiment two. All players are chasing the ball.

4.2.3 Conclusion

The populations converged after five to ten generations, which is twice as fast as in the previous experiment. This rapid convergence is referred to premature convergence and a lower complexity compared to the previous experiment (three separate algorithms compared to eight separate algorithms). However, the players did not seem to improve more after the ten generations.

Premature convergence clearly occurred amongst the populations of attackers and defenders. The high rate of identical genomes strongly reduced the efficiency of the genetic operators and the population was trapped in what appeared to be a troublesome suboptimum. The obvious approach to deal with this suboptimum is to restart the population at generation 0 or drastically increase the mutation rate. Preliminary testing showed that premature convergence occurs even if the population is restarted. This is likely due to the very small population size consisting of thirty-two individuals.

The resulting players from the first experiment did better than the players from this experiment. It can be concluded that this is due to the lack of defensive abilities in the second teams.

Chapter 5

Discussion

The goals of this study were to use genetic programming to teach software robots play simulated soccer, and secondarily to investigate the development of multi-agent strategies.

The two experiments only differ in the team set-up. In the first experiment, each team consisted of eight randomly picked players (excluding the goalkeeper) from a population of one hundred and twenty-eight individuals. The teams in the second experiment were made up of eight players consisting of three different genomes and they are evolved in separate populations of thirty-two individuals each.

Initially, the players from both experiments learned to chase and kick the ball towards the goal or pass it to a team-mate. The players in the second experiment converged and did not develop further. However, the players from the first experiment continued to develop slowly and recognized that team coordination was essential for further development.

The following are a number of reflections and possible explanations about why the robots did not develop further.

- **Premature convergence**

  The population size was very small for a problem of this complexity. In order to prevent premature convergence and increase diversity amongst the evolved players, a larger population size should have been used. Koza [1994] suggests a population size of 4000 for a typical problem.

- **Limited search space**

  The sets of predicates and actions provide a limited search space. A significantly improved solution may not be possible to express with the current sets.

- **Functions**

  Ultimately, it is the action functions that control the software robot. The players' performances were directly related to the quality of the functions. The players generated in this study had for example, troubles with the interception of passes.

- **Credit assignment**

  The fitness function may not correspond to the actual desired behaviour. The primary difficulty was to determine which individuals to credit for the whole team's success.

- **Overfitness**

  Each algorithm is only tested for a limited number of situations or in this case, number of matches. The fitness assignment used in the experiment is inaccurate. If a larger number of evaluations were used, it would minimize the fitness deviation and give a more accurate measurement.

- **Computational resources**

  The fitness evaluation is computationally expensive. The presented experiments were completed in roughly one month (computer time). Numerous compromises were made in this study to reduce the evaluation time and this eventually resulted in weaker players.

This study demonstrates that software robots are able to learn to play the game of simulated soccer despite its very complex dynamics. The strategies that the robots developed are most likely to be inferior to human coded algorithms, but are significantly better than the initial random strategies.

Chapter 6

Future Work

The approach used in this study is just one of several possible approaches. This chapter lists some suggestions and ideas about future investigations into this subject.

- The population size used in these experiments is simply too small. Various methods for accelerating the fitness evaluation should be investigated.
- The decision tree approach with fixed predicates and actions certainly limits the search space over possible solutions. To modify this, allow the predicates and actions to be dependent on some parameters that are initially set in a random manner. The genetic operators could then alter these parameters. Another modification is to allow some actions and predicates to evolve separately. Koza [1994] introduces this as automatically defined functions.
- More information about the game could be available to the players by expanding the set of predicates or introducing game states. A game state can be based on past information and contain data such as coach messages, previous playing styles, when the opponents attacked last time, etc.
- Introduce an online coach that broadcasts values to some predicates. These values concern the whole team and are difficult for an individual player to calculate with its limited perceptions. Examples of such predicates could be: “Is any team attacking?” or “Do they have good defence?”.
- Instead of using a fixed fitness function for all generations, one that changes as the players improve could be used. For developing basic skills, a simple fitness function may be sufficient or even better than a complex one. In order to teach the robots advanced behaviours, it may be necessary to customize the fitness function to a particular skill. One approach could be the development of a system similar to the way humans learn to play soccer with the help of trainers. This system could

analyse the players throughout the generations and output statistics and data to a human operator regulating the fitness function.

Bibliography

[Andre and Teller, 1999] David Andre and Astro Teller, *Evolving team Darwin United*, RoboCup II: Proceedings of the second annual conference, Springer-Verlag, 1999

[Angeline and Pollack, 1993] Peter J. Angeline and Jordan B. Pollack, *Competitive Environments Evolve Better Solutions for Complex Tasks*, In Proceedings of the Fifth International Conference on Genetic Algorithms, pp. 264–270, Morgan Kaufmann Publishers, 1993

[Ciesielski *et al.*, 2003] Vic Ciesielski, Dylan Mawhinney, and Peter Wilson, *Genetic Programming for Robot Soccer*, 2003

[Gustafson and Hsu, 2001] Steven M. Gustafson and William H. Hsu, *Layered Learning in Genetic Programming for a Co-operative Robot Soccer Problem*, In Proceedings of the Euro GP 2001, pp. 291-301, Springer-Verlag, 2001

[Holland, 1975] John H. Holland, *Adaptation in Natural & Artificial Systems*, University of Michigan Press, 1975

[Koza, 1992] John R. Koza, *Genetic Programming: On the Programming of Computers by Natural Selection*, MIT Press, Cambridge MA, 1992

[Koza, 1994] John R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press, Cambridge MA, 1994

[Koza, 1999] John R. Koza, Forrest H. Bennett III, David Andre, Martin A. Keane, *Genetic Programming III: Darwinian Invention and Problem Solving*, Morgan Kaufmann Publishers, 1999

[Luke *et al.*, 1997] Sean Luke, Charles Hohn, Jonathan Farris, Gray Jackson and James Hendler, *Co-evolving Soccer Softbot Team Coordination with Genetic Programming*, RoboCup I: Proceedings of the first annual conference, Springer-Verlag, 1997

[Montana, 1995] D.J. Montana, *Strongly Typed Genetic Programming*, In Evolutionary Computation, The MIT Press, Cambridge MA, 1995

[Noda *et al.*, 1999] Minoru Asada, Hiroaki Kitano, Itsuki Noda, and Manuela Veleso. *RoboCup: Today and tomorrow - what we have learned*, Artificial Intelligence, 110:193-214, 1999

[Persson, 2001] Niklas Persson, *Samarbetande autonoma agenter i RoboCup-miljö*, Master's Thesis, Lund University, 2001

[Rahm, 2001] Christian Rahm, *Att skapa mjukvaruagenter för RoboCup*, Master's Thesis, Lund University, 2001

[RCC Manual, 2003] Tom Howard, *The RoboCup Client Parser Reference Manual 1.2.2*, 2003

[Reynolds, 1994] Craig Reynolds, Competition, *Coevolution and the game of tag*, In Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems, pp. 59-69, MIT Press, 1994.

[RS Manual, 2002] Fredrik Heintz and David Rosèn, *The RoboSoc Manual 2.8 Version 0.1*, 2002, http://www.ida.liu.se/~frehe/ RoboCup/RoboSoc/

[Russell and Norvig, 1995] Stuart Russell and Peter Norvig, *Artificial Intelligence: A modern Approach*, Prentice-Hall, 1995

[SS Manual, 2002] Mao Chen, Ehsan Foroughi, Fredrik Heintz, ZhanXiang Huangy, Spiros Kapetanakis, Kostas Kostiadis, Johan Kummeneje, Itsuki Noda, Oliver Obst, Pat Riley, Timo Ste_ens, Yi Wangy and Xiang Yin, *Soccer Server Manual Version 7*, 2002

[Stone, 1998] Peter Stone, *Layered Learning in Multi-Agent System*, PhD thesis, Carnegie Mellon University, 1998

[Woolridge, 2002] Michael Woolridge, *An Introduction to Multi-agent Systems*, Wiley, 2002

Appendix A

Predicates and Actions

A.1 Predicates

IAmClosestToTheBall is true if and only if:

- The position of the ball relative to the agent is known.
- The agent is closest to the ball amongst the players it can see.

IAmClosestToTheBallOnOurTeam is true if and only if:

- The position of the ball relative to the agent is known.
- The agent is closest to the ball amongst the agent's team-mates that it can see.

WeAreClosestToBall is true if and only if:

- The position of the ball relative to the agent is known.
- The agent or a team-mate that the agent can see is the closest to the ball amongst the players it can see and distance to the ball is less than 10.

BallIsClose is true if and only if:

- The position of the ball relative to the agent is known.
- The distance between the agent and the ball is equal to or less than 5.

BallIsFarAway is true if and only if:

- The position of the ball relative to the agent is known.
- The distance between the agent and the ball is greater than 35.

BallIsNearGoal is true if and only if:

- The position of the ball is known.
- The distance between the ball and centre of the opponent's goal is less than 32.

BallIsNearOurGoal is true if and only if:

- The position of the ball is known.
- The distance between the ball and the centre of home goal is less than 32.

OpponentIsClose is true if and only if:

- The agent can see at least one opponent.
- An opponent in the agent's field of vision is closer than a distance of 5.

IAmAlone is true if and only if no player in the agent's field of vision is closer than a distance of 15.

WeAreWinning is true if and only if the agent's team is leading with at least 2 points.

IAmNearGoal is true if and only if:

- The agent's position is known.
- The distance between the agent and the centre of opponent's goal is less than 25.

IAmNearOurGoal is true if and only if:

- The agent's position is known.
- The distance between the agent and the centre of home goal is less than 25.

WeAreAttacking is true if and only if:

- The agent's position is known.
- The agent is located on the fifth of the field furthest away from home.

Or

- The positions of at least two team-mates (including the agent) are known.
- At least two team-mates (including the agent) are located on the 30 % of the field furthest away from home.

OpponentsAreAttacking is true if and only if:

- The positions of at least two opponents are known.
- At least two opponents are located on the 30 % of the field which is closest to home.

TeammatesAreFree is true if and only if:

- The agent cannot see any opponents.

Or

- The position of at least one team-mate is known.
- A team-mate is further than a distance of 10 away from all opponents visible to the agent.

WeAreSpreadOut is true if and only if:

- There is only one team-mate visible to agent.
- The team-mate is at least a distance of 15 away from the agent.

Or

- More than one team-mate is in the agent's field of view.
- V({x})+V({y}) > 200, where V is the variance function. {x} and {y} are the coordinates for visible team-mates, including the agent.

A.2 Actions

A.2.1 Move Actions

Move Actions are only executed in the move tree.

DashToBall

If the agent's body direction is not facing the ball (±8º), the agent will turn towards the ball otherwise a dash command will be executed.

DashToGoal

A target point is set as one of the corners of the penalty area at the opponent's goal (e.g. if the agent is playing on the left side team, the target point will be either (32, 20) or (32, -20)). If the agent's body direction is not facing the point (±5º), the agent will turn towards the ball otherwise a dash command will be executed.

BlockGoal

A virtual vector is set between the ball and the centre of the home goal. A target point is then set to be on this vector. If the agent is close to the ball (<7), the target point will set to the ball's position instead. If the agent's body direction is not facing the point (±8º), the agent will turn towards the ball otherwise a dash command will be executed.

InterceptBall

This action is included in RoboSoc and will determine an intersection point between the agent and the ball. The agent will then execute a dash or turn command to get to this point in the same time or before the ball does.

SpreadTeam

A list with normalized vectors will be constructed for each team-mate that the agent sees. A vector is determined as an average of all of these vectors. The agent will then move in the opposite direction from this vector.

A.2.2 Kick Actions

Kick actions are only executed in the kick tree ensuring that the ball will be kickable. If the action is a pass, the agent will execute a say command to inform the targeted player.

PassPlayer1

If the agent sees one or more team-mates, it will pass the one closest to the agent. If no team-mate is in the agent's field of vision, no command will be executed.

PassPlayer2

If the agent sees more than one team-mate, it will pass the second closest team-mate. If only one team-mate is in the agent's field of view, the agent will pass it. If no team-mate is in the agent's field of vision, no command will be executed.

PassPlayerFarAway

If the agent sees one or more team-mates, it will pass the ball to the one closest to the opponent's goal. If no team-mate is in the agent's field of vision, no command will be executed.

PassBeckham

This action is included in RoboSoc and enables the agent to pass to the team-mate that is most suitable for a pass (according to a number of criterions). If no suitable team-mate is found, no command will be executed.

DribbleToGoal

This action is included in RoboSoc and the agent will dribble the ball towards the opponent's goal while avoiding its opponents.

KickGoal

The agent will kick the ball towards the centre of the opponent's goal.

Appendix B

Evolved Algorithms

This appendix presents a few examples of evolved algorithms from the experiments presented in chapter 4. Throughout the experiments, more than 10000 different algorithms were evolved and evaluated. The following examples are intended to give the reader an idea of the algorithms' composition. A number of statistical data follows each example.

B.1 Experiment 1

The top ranked algorithm from generation 3

Kick Tree

```
IF WeAreSpreadOut
  IF WeAreAttacking
    IF IAmNearGoal
      IF OpponentsAreAttacking
        DribbleToGoal
      ELSE
        IF TeammatesAreFree
          PassPlayerFarAway
        ELSE
          PassPlayer2
    ELSE
      KickGoal
  ELSE
    PassPlayer1
ELSE
  PassBeckham
```

Move Tree

```
IF WeAreWining
  IF IAmAlone
    IF IAmNearOurGoal
      SpreadTeam
    ELSE
      IF IAmNearGoal
        InterceptBall
      ELSE
        DashToBall
  ELSE
    BlockGoal
ELSE
  DashToBall
```

	Match average	Population match average
Number of kicks	24	20
Number of passes	22	9
Distance to ball	17	22
Made goals	0	0.3

The top ranked algorithm from generation 20

Kick Tree

```
IF IAmNearGoal
  PassPlayer2
ELSE
  PassBeckham
```

Move Tree

```
IF WeAreWining
  IF OpponentIsClose
    DashToBall
  ELSE
    IF IAmNearGoal
      IF BallIsClose
        InterceptBall
      ELSE
        BlockGoal
    ELSE
      SpreadTeam
ELSE
  DashToBall
```

	Match average	Population match average
Number of kicks	59	28
Number of passes	32	14
Distance to ball	14	20
Made goals	0.5	0.3

The top ranked algorithm from generation 51

Kick Tree

```
IF OpponentIsClose
   PassBeckham
ELSE
   IF IAmNearGoal
      IF WeAreAttacking
         PassPlayer2
      ELSE
         IF TeammatesAreFree
            PassBeckham
         ELSE
            PassPlayer2
   ELSE
      IF WeAreAttacking
         PassBeckham
      ELSE
         PassPlayer1
```

Move Tree

```
IF WeAreWining
   IF BallIsNearOurGoal
      IF OpponentsAreAttacking
         DashToGoal
      ELSE
         DashToBall
   ELSE
      DashToBall
ELSE
   DashToBall
```

	Match average	Population match average
Number of kicks	83	22
Number of passes	28	9
Distance to ball	18	20
Made goals	0.5	0.3

The second ranked algorithm from generation 51

Kick Tree	Move Tree

IF IAmNearGoal
  IF WeAreAttacking
  KickGoal
  ELSE
  IF TeammatesAreFree
  IF OpponentsAreAttacking
  DribbleToGoal
  ELSE
  PassBeckham
  ELSE
  PassPlayerFarAway
ELSE
  PassBeckham
	IF WeAreWining
	  DashToBall
	ELSE
	  IF OpponentsAreAttacking
	  IF IAmNearOurGoal
	  BlockGoal
	  ELSE
	  SpreadTeam
	  ELSE
	  DashToBall

	Match average	Population match average
Number of kicks	56	22
Number of passes	30	9
Distance to ball	14	20
Made goals	1.5	0.3

The 64th ranked algorithm from generation 51

Kick Tree	Move Tree

IF OpponentsAreAttacking
  IF IAmNearGoal
  PassPlayer2
  ELSE
  PassPlayerFarAway
ELSE
  PassBeckham
	IF WeAreWining
	  IF BallIsFarAway
	  DashToBall
	  ELSE
	  IF OpponentIsClose
	  IF OpponentsAreAttacking
	  IF IAmClosestToBallOnOurTeam
	  BlockGoal
	  ELSE
	  DashToGoal
	  ELSE
	  InterceptBall
	  ELSE
	  InterceptBall
	ELSE
	  IF BallIsNearOurGoal
	  BlockGoal
	  ELSE
	  DashToBall

	Match average	Population match average
Number of kicks	46	22
Number of passes	7	9
Distance to ball	18	20
Made goals	0	0.3

B.2 Experiment 2

The top ranked attacker algorithm from generation 5

Kick Tree	Move Tree

`PassBeckham`	`IF BallIsNearOurGoal`
	`  BlockGoal`
	`ELSE`
	`  DashToBall`

	Match average	Population match average
Number of kicks	211	37
Number of passes	149	21
Distance to ball	5	16
Made goals	0.3	0.3

The top ranked midfielder algorithm from generation 5

Kick Tree	Move Tree

`IF OpponentsAreAttacking`
`  KickGoal`
`ELSE`
`  PassBeckham`
	`IF OpponentsAreAttacking`
	`  IF OpponentIsClose`
	`  InterceptBall`
	`  ELSE`
	`  DashToBall`
	`ELSE`
	`  DashBall`

	Match average	Population match average
Number of kicks	122	46
Number of passes	88	23
Distance to ball	7	15
Made goals	0	0.4

The top ranked defender algorithm from generation 5

Kick Tree	Move Tree

IF WeAreSpreadOut
  PassPlayer1
ELSE
  IF WeAreAttacking
  PassPlayer2
  ELSE
  PassPlayerFarAway
	IF BallIsFarAway
	  InterceptBall
	ELSE
	  IF OpponentsAreAttacking
	  DashToBall
	  ELSE
	  DashToGoal

	Match average	Population match average
Number of kicks	43	18
Number of passes	48	9
Distance to ball	26	32
Made goals	0	0.2

The top ranked attacker algorithm from generation 55

Kick Tree	Move Tree
PassBeckham	DashToBall

	Match average	Population match average
Number of kicks	103	25
Number of passes	52	11
Distance to ball	15	17
Made goals	0	0.3

The top ranked midfielder algorithm from generation 55

Kick Tree	Move Tree

IF OpponentsAreAttacking
  IF IAmAlone
  PassPlayer1
  ELSE
  PassBeckham
ELSE
  IF WeAreSpreadOut
  PassBeckham
  ELSE
  IF IAmAlone
  PassPlayer1
  ELSE
  PassBeckham
	IF BallIsNearOurGoal
	  IF IAmAlone
	  IF OpponentsAreAttacking
	  DashToBall
	  ELSE
	  BlockGoal
	  ELSE
	  DashToBall
	ELSE
	  DashToBall

	Match average	Population match average
Number of kicks	98	32
Number of passes	75	16
Distance to ball	26	16
Made goals	0	0.2

The top ranked defender algorithm from generation 55

Kick Tree	Move Tree
`PassBeckham`	`DashToBall`

	Match average	Population match average
Number of kicks	40	21
Number of passes	18	8
Distance to ball	21	27
Made goals	1	0.2