LAYERED LEARNING IN GENETIC PROGRAMMING FOR A COOPERATIVE ROBOT SOCCER PROBELM

by

STEVEN MATT GUSTAFSON

B.S., Kansas State University, 1999

A THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Science
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2000

Approved by:

Major Professor
Dr. William H. Hsu

ABSTRACT

We present an alternative to standard genetic programming (GP) that applies *layered learning* techniques to decompose a problem. GP is applied to subproblems sequentially, where the population in the last generation of a subproblem is used as the initial population of the next subproblem. This method is applied to evolve agents to play keep-away soccer, a subproblem of robotic soccer that requires cooperation among multiple agents in a dynamic environment. The layered learning paradigm allows GP to evolve better solutions faster than standard GP. Results show that the layered learning GP outperforms standard GP by evolving a lower fitness faster and an overall better fitness. Results indicate a wide area of future research with layered learning in GP.

(discard this page)

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank my major professor, Dr. William Hsu, for his support and guidance during this research. Thank you to Dr. Mitchell Neilsen for serving on my committee. Also, I would like to thank my parents, Dr. and Mrs. David Gustafson, for their constant support and encouragement. Lastly, I would like to acknowledge and thank my wife Kristin, I appreciate her patience, support and encouragement.

Dedication

to my

MOM and DAD

with love

Chapter 1

Introduction

Genetic programming (GP) extends the genetic algorithm proposed by Holland [6] to evolve computer programs. In [11] Koza describes the transformation of the genetic algorithm to GP. In evolutionary methods [5], an individual in a genetic population represents a possible solution to a problem. Each individual is assigned a fitness that drives the breeding of new populations towards the best individuals.

For complex problems, such as robotic soccer [10][18], genetic programming (GP) may not be capable of finding a solution in its standard form. One reason is that the size of the GP search space grows so large that it effectively leads to an intractable problem [25]. GP was previously used for robotic soccer to evolve teams of agents, but modifications were usually made to simplify the problem. Luke used GP to evolve high-level team strategies in [15], and Andre and Teller developed a fitness function based on human coaching principles of soccer in [1]. In a multiagent system (MAS) such as soccer, there is a definite hierarchy of behaviors that can be observed from human soccer. GP produces hierarchical programs by evolving and using automatically defined functions (ADF).

Koza gives a good description of ADFs and hierarchical programs in [12]. Hierarchical programs are evolved using ADFs to allow for code structure and code reuse.

Rosca and Ballard, in [20], discuss ADFs and their importance in hierarchical programs. Although ADFs allow for the GP individuals to reuse evolved subprograms and develop solutions that have hierarchical code structure, they do not explicitly allow for hierarchical learning to take place. The distinction is that hierarchical learning describes a way in which behaviors are learned, not necessarily in how the code that represents them is structured. While code reuse and program structure may help to overcome the inherent complexity of MAS problems, we suggest an approach for learning cooperative behaviors in a team-based MAS that is based upon primitive team objectives.

In many cases, teamwork can be made more tractable to learning, both in efficiency and in robustness of the performance element through a logical decomposition of the main problem [26]. For example, in the robotic soccer domain Stone and Veloso in [21] produce a very effective team of agents playing soccer by learning the overall task in a hierarchical manner. Tasks such as passing and kicking were learned before overall team strategies were learned. This technique of learning in layers is called *layered learning* and is formally described in [22]. It was applied with reinforcement learning for robotic soccer and the results indicated that layered learning may be a good adaption for other machine learning methods such as GP.

To investigate layered learning in GP, the problem of learning keep-away soccer is chosen for its similarities to soccer and its properties of being a MAS problem. Also, keep-away soccer presents GP with a more reasonable search space than the full soccer game problem and should allow for standard GP to find a solution, so that comparisons can be made with the hybrid method of layered learning in GP.

Figure 1.1 is a screen capture of the visualization program used for the simulator built for keep-away soccer. The figure depicts the three offensive agents passing the ball in a counterclockwise motion (agent 3 passes the ball twice), with the trail of the ball denoted by the '-' character. The "*", "+", and ";" show the paths of other

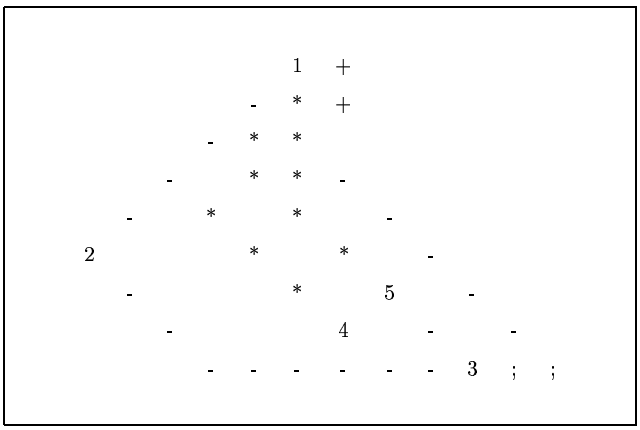


Figure 1.1: Screen capture of simulator. 1,2, and 3 are offensive agents. 4 is the defender, and 5 is the ball. Ball moves from 3 to 1, 1 to 2, 2 to 3, and back towards 1.

agents. The ball is passed from agent 3 to 1, 1 to 2, 2 to 3, and at the end of the simulation, the ball is being passed back from 3 to 1. The visualization was run for about 30 timesteps to collect the screen capture.

Several variations exist for MAS and keep-away soccer can be categorized as multiagent learning with homogenous, noncommunicating agents [24]. This type of MAS problem requires robust solutions and is an interesting problem for research. A natural way to reduce complex, MAS problems, such as keep-away soccer, could prove to be useful for other MAS problems. Now, the hypothesis is stated.

Hypothesis: The layered learning paradigm allows genetic programming to evolve more highly fit individuals in fewer generations than standard genetic programming by taking advantage of the natural decomposition of problems.

The keep-away soccer problem is described in Section 2, followed by a more detailed analysis of the application of layered learning to GP in Section 3. Section 4 describes experiments using an abstracted version of the TeamBots and SoccerServer

robotic soccer simulator. Results are in Section 5 and research findings, conclusions and future work follow.

Chapter 2

Background

2.1 Evolutionary Algorithms

As with all algorithms, evolutionary algorithms take an input and return the desired output. However, in the case of evolutionary algorithms, it is not known how good the output will be. A desired performance is specified, but may be too complex for the algorithm to find and output that produces that performance. Evolutionary algorithms can be represented by the general algorithm:

$$x[t+1] = s(v(x[t]))$$

A population of candidate solutions to the problem at some time t are denoted by $x[t]$. Random variations,v, and selection methods, s, are applied to the population at $x[t]$ to produce a new population at the next time step, $x[t+1]$. This algorithm is described in [5].

The idea behind evolutionary algorithms comes from the biological method of evolution, where selective pressures are applied to populations of organisms to evolve behaviors and features to allow for survival. Basically, if a difficult and complex search space exists with a solution somewhere inside it, that solution can be found

by properly specifying the survival criterion of individuals and allowing for the evolutionary algorithm to search the space. The survival criterion is generally referred to as the fitness of a candidate solution. Since each individual in a genetic population represents a possible solution, that solution can be evaluated and given a fitness describing how well it solves the problem. Then, the fitness of each candidate solution in a population is used to drive the creation of a new population. The search is based upon Darwin's principle of *survival of the fittest* [4].

Genetic algorithms and genetic programming apply the ideas of evolutionary algorithms to find the good solutions in complex search spaces, where the genetic algorithm solution is much different than the genetic programming solution. The two methods are described next.

2.2 Genetic Algorithms

Genetic algorithms were introduced by John Holland in 1975 [6]. They use the idea of evolutionary algorithms to solve problems where the problem usually consists of finding the optimal bit string that, when applied to some values, presents the solution to the problem. An example of a genetic algorithm is its application to the feature subset selection problem in the field of data mining.

In data mining, huge databases are used to derive relationships between the many different possible combinations of values of items in the database. These relationships can then be used for predictive purposes so that when the values of database attributes are of a particular combination, the classification previously given to that combination can be predicted for the new similar combination. The problem, however, exists when the database is so large that there are many irrelevant database attributes cluttering the search space that it becomes very difficult to find correct classifications of combinations of attribute values.

If we represent a combination of attribute values as an array of values, where the array can be seen as a horizontal slice of a database (each column in a database represents a different attribute), then we can assign all the different arrays of values a classification using other machine learning techniques like decision tree learning or neural networks. The problem consists of a training database that is used to derive classifications and a test database that is used to test the validity of those classifications. Now, as done in [7][8], we can apply a bit string mask to an array of values that allows for some values to be used, a 1 in the mask, and others to be ignored, a 0 in the mask. We can then repeat the problem with different bit string masks to determine if a better subset of features, or attributes, exists to find the best classifications, i.e. classifications with the fewest errors in the testing phase.

The problem of finding good bit masks can be solved by a genetic algorithm. A candidate solution represents a different bit mask, each solution is applied to the database, a machine learning technique is used to find classifications, those classifications are tested and the errors, along with other properties of the classification engine, are assigned to the candidate solution's fitness. That fitness is then used to drive the search for new and better candidate solutions.

Genetic programming is derived from the genetic algorithm, but the candidate solution in genetic programming is very different.

2.3 Genetic Programming

In the 1950s, Arthur Samuel asked how much longer will computers have to be explicitly programmed, i.e. how can we develop programs without writing line for line the program [12]. Genetic programming uses the ideas of evolutionary algorithms and the methods of genetic algorithms to answer the question posed by Samuel. Genetic programming has a simple idea behind it, but requires a fairly complex system

to implement it, similar to genetic algorithms and other evolutionary methods. An overview of the key components are given here, and a complete introduction to genetic programming can be found in John Koza's books [11] [12].

Genetic programming attempts to find solutions to problems by evolving computer programs that solve the problem. An individual in a genetic programming population is an actual computer program that, with a little overhead, can be run on a computer to attempt to solve a problem. Typically, the programs are represented in the LISP programming fashion by an S-expression. These S-expressions are easily represented by a tree structure and most often, genetic programming systems use the tree as the underlying data structure to represent individuals in a population. Appendix D shows two sample individuals, represented by S-expressions, that were developed in the research here. Individuals, or trees, are made up from a function set and a terminal set. Functions are functions from the programming world, where the '+', '-', '*' and '/' operators (addition, subtraction, multiplication and division) are arithmetic operations with arity 2, requiring two inputs. Terminals are inputs to the system that can be operated on by the functions. In the soccer domain, terminals may be the location, either globally or egocentrically, to the ball, other teammates, or opponents. Figure 2.1 is an example of a GP individual represented by a tree, where internal tree nodes are functions and leaf nodes are terminals. A tree can be then evaluated by applying the functions to terminals, their result to functions higher up in the tree, until the function at the root node evaluates and returns a value.

An issue arises as to whether or not every function can operate with every possible terminal. When specifying a problem for a GP system, it is necessary to address this and different GP systems handle this differently. Montana [19] describes Strongly Typed Genetic Programming, which has the property that every function can operate on every terminal, and the return value of every function is the same data type as every terminal. This greatly simplifies the GP system but not all GP systems use

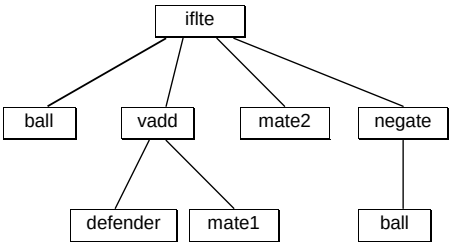


Figure 2.1: A GP individual represented by a tree of functions and terminals.

this and require the programmer to specify properties about functions and terminals. The system specified here, for keep-away soccer, is a strongly typed GP system that works on the *vector* data type.

A GP individual has two main characteristics, its genotype and its phenotype, meanings taken from biology. The genotype of an individual describes the structure of the program, i.e. characteristics about the tree representing the individual, how many nodes in the tree, how deep or broad is the tree. Different genotypic characteristics of individuals in a population can be used to measure genotypic fitness. Phenotypes describe the actual performance of an individual, how well did it find the desired solution defined as the problem's fitness function. Phenotypic fitness is typically used in genetic programming to drive the search for ideal individuals.

So far, the methods described in evolutionary algorithms, random variation and selection methods, have only been hinted to as a way to derive a new population of candidate solutions given a fitness for each solution. But those methods are the heart of evolutionary methods as they are what actually allow for good solutions to be found and do all the work in making bad solutions into good solutions.

2.4 Genetic Operations

The genetic operators in evolutionary algorithms come from ideas found in biological evolution. These operators, in evolutionary algorithms, are crossover, reproduction, and mutation, the most commonly used operators. Operators take one or more individuals as input and produce one or more individuals as output. Operators are applied in different proportions to produce a new population and these proportions vary per problem and are a topic of research. The standard proportions, used here, use reproduction to create 10 percent of the new population and crossover to produce the remaining 90 percent. Mutation is not used here, but usually is used in combination with crossover to produce approximately 90 percent of the new population. Again, as with deciding the functions and terminals of a GP system, there are many possibilities for crossover, reproduction, and mutation. Several parameters for each genetic operation can be modified to determine their effects.

2.4.1 Crossover

The operation of crossover typically takes two individuals, parents, as inputs and produces two new individuals, children, as output for the new population. To accomplish this, a crossover point is selected in each individual. Then, the crossover point node and its subtree are swapped between individuals, creating two new different trees. Remembering that because the system is strongly typed, the resulting individuals are guaranteed to be valid. Figure 2.2 shows two individuals where a crossover point has been selected in each. Figure 2.3 then shows the resulting children after the crossover operation is applied. These two children represent two new individuals in the new population. Figures 2.1, 2.2, 2.3 were adapted from figures found in [17].

Some issues dealing with crossover are: if roots can be selected as crossover points, how deep in the individual can a node be selected as a crossover point and how deep

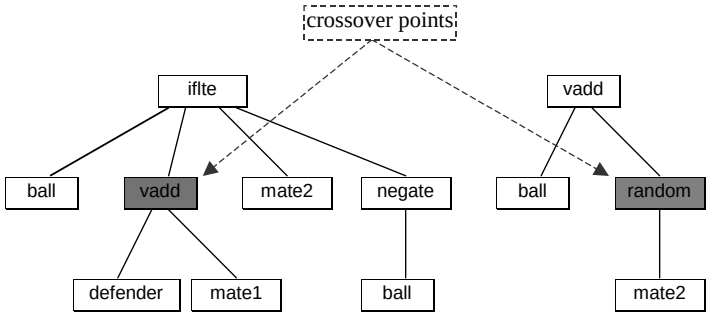


Figure 2.2: A crossover point is selected in each individual.

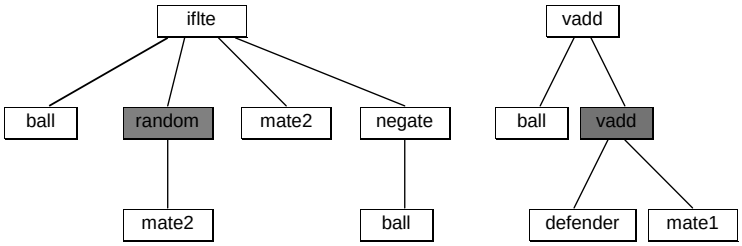


Figure 2.3: The new individuals resulting from the crossover operation.

the children produced can be.

2.4.2 Reproduction and Mutation

The reproduction operator simply copies an individual from the current population into the next population. Reproduced individuals are typically those individuals with the best fitness, the ones that came the closest to the ideal solution. The mutation operation is similar to crossover in selecting a mutation point in one individual. The selected mutation point, node, is then replaced, the node and subtree, with a different node selected usually at random from the available functions or terminals. Again, there are many parameters that determine exactly how reproduction and mutation perform. Still at issue is which individuals are selected to have the genetic operations applied to, resulting in offspring for the new population.

2.4.3 Selection

Once a fitness is assigned to each individual in a population, those individuals with the highest fitness typically represent the solutions closest to the ideal solution. However, as in other machine learning techniques, there is the problem of getting stuck in local, suboptimal solutions. The technique of simulated annealing is often used to allow global solutions to be found by jumping out of local solutions. If, for instance, only the best individuals in a population, those with the best fitness, are used as parents for the new population, and those individuals represent only a suboptimal solution, a local minimum, the whole GP system may then be doomed not to find the global solution. To prevent this, a selection method is typically used that selects a subpopulation randomly, then selects the most fit individuals in that subpopulation to be parents to produce offspring.

The above selection method is typically implemented using tournament selection,

which selects a subpopulation at random and then selects parents based on best fit individuals. Again, selection criterion is an area of research.

2.5 Hierarchical Genetic Programming

So far a genetic programming individual has been described as a tree with functions and inputs as the nodes and leaves, respectively. Trees represent programs that are possible solutions to problems. Ideal trees are found by evaluating a tree's performance and then using genetic operations to modify those trees to get new trees. Based on the survival of the fittest, from biological evolution, the resulting trees should be the closest, if not equal, to the ideal solution. Less obvious is that we leave the resulting structure, or genotype, of the tree to be completely determined by the GP system. However, an experienced programmer knows that programs often compute the same values several times throughout the evaluation of a program and programs that do this are much more easily created by reusing the code to computer these values.

The topic of object oriented programming addresses this and allows for code reuse in a particular program and in new programs. For a GP system to evolve an individual that has an ideal solution that computes the same values several times, it can be seen that this search space is very large and may even be intractable, as it becomes less and less likely that the same subtree will be evolved several times in one individual correctly. Intuitively, it would be ideal if a GP system could make use of the code reuse and program structure that programmers use so effectively.

Hierarchical GP attempts to allow code reuse and program structure to be evolved completely by the GP system, but with a smaller search space. Koza gives a good introduction and application description of hierarchical programming in [12], as does Rosca and Ballard in [20]. The basic premise of hierarchical genetic programming is the automatically defined function (ADF). In a program that reuses code, that code

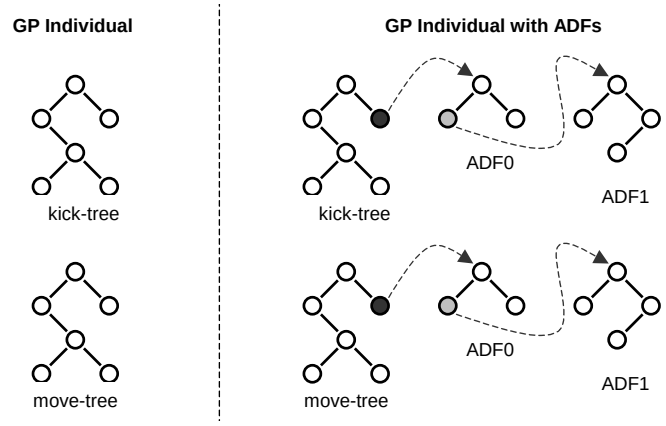


Figure 2.4: Representations of a GP individual and a GP individual with ADFs, where each individual is made up of two trees.

is generally defined in a function that can be called many times. GP individuals can call functions when evaluating their tree, but those functions come from the function set created by the programmer. ADFs allow for a GP system to evolve new functions to be used in an individual's tree. This gives a GP individual the power to reuse code without having to reinvent the same subtree several times, and gives the evolved program structure as usually used by computer programmers. Figure 2.4 shows a typical setup of ADFs and the one used in the research here. In the experiments here, each individual is represented by two trees, which are discussed later. When individuals have ADFs, the main tree is capable of calling the ADFs, which are also sometimes capable of calling other ADFs.

ADFs and hierarchical programming is another area of research in genetic programming and for the research done here, standard values and setups are used to explore the ability of ADFs to find a solution to compare with other methods. As shown in [12], ADFs typically evolve better solutions, and sometimes in less amount of time, than genetic programming without ADFs. Obviously, problems that reuse code would benefit from ADFs, and problems that do not would most likely show

no improvement with the use of ADFs. An issue with ADFs arose with the research here that showed hierarchical GP evolving two clusters of solutions, one much better than the other. A hypothesis is given for this occurrence, but it is really a topic for another research problem.

2.6 Layered Learning

Layered learning was introduced by Peter Stone and Manuela Veloso in [22]. The technique was used in the RoboCup competition to win simulation and robot leagues. The 1998 competition's winning team used layered learning as described in [21] and used the reinforcement learning method Team-Partitioned, Opaque-Transition Reinforcement Learning [25]. The basic idea behind layered learning is that some problems are so complex that they cannot be learned directly from the inputs provided by the system. But if the problem is decomposed into several layers of behaviors, each behavior can be learned and used in the learning of other behaviors. In this way, a complex problem can be solved. Four key principles of layered learning are described in Chapter 4.

The most interesting of these principles is the fact that layered learning relies on a bottom-up decomposition of the problem. There are some problems where this type of decomposition is natural, and obvious, and some where it is not. Problems that stem from human learning usually have a natural bottom-up decomposition since humans generally learn in this way. But just like any machine learning technique, different types of problems require different types of solutions. Having said this, it can then be assumed that layered learning may not be a good method for every type of problem.

After seeing the success of layered learning in the RoboCup competition with reinforcement learning, it is desirable to see if it could improve the performance

of genetic programming. Soccer, and the RoboCup competition, are very complex problems, and if a comparative study is to be done, the problem really needs to be simplified. For this reason, the keep-away soccer problem is selected, to allow layered learning in GP to be compared with other GP techniques.

2.7 Multiagent Systems

Multiagent systems (MAS) are an area of research in artificial intelligence (AI). An excellent survey of MAS is found in [24]. The general idea behind MAS is that a problem exists which can only be solved by, or the solution would be benefited from, the use of several independent agents working together. For instance, in the problem used in the research here, keep-away soccer presents a MAS problem where one agent acting alone cannot keep a ball away from another agent who is twice as fast as the agent. Other examples are found in the predator-prey problem [16] where agents (predators) try and cooperate to catch a much faster prey. MAS problems such as predator-prey are used to develop solutions for MAS that can be used in many other areas.

MAS problems provide research in the areas of efficient cooperation, adaptation, robustness, and real-time solutions, as described in [18]. Obviously, these areas are of interest to other areas of computer science, and successful MAS solutions would be beneficial to those independent research areas. In [26], a MAS solution is applied to three complex areas, robotic soccer, coordination of attack helicopters for military training, and the coordination of transportation helicopters and escort helicopters for successful transportation of goods in a military training. All three domains illustrate the importance of teamwork and coordination among agents and highlight the necessity of communication, knowledge of other agent's actions, robustness and adaptability in the solutions. It is seen here that while a domain like soccer may initially

not be seen as worthwhile, it is in fact a good testbed for other much more important domains, and solutions in robotic soccer have applicability in these domains.

2.8 Robot Soccer

The Robot World Cup Initiative (RoboCup) is proposed in [9] as a standard problem for research in the areas of AI and robotics, requiring the use of several technologies and research in a wide range of areas in AI and robotics. The competition has several leagues for different sizes of robots and a simulation league which uses the SoccerServer [2], allowing software agents to compete. Additionally, there are leagues to accommodate different number of players on the field. The soccer domain provides a very complex problem requiring robust solutions. In the real robot problems, issues of hardware and software arise as sensors and actuators must perform correctly and well to interact effectively with the software driving them, and that software must be designed to solve many problems such as cooperation in a dynamic environment. The simulation league focuses more on the design of intelligent strategies for agents and teams. Either league is an excellent testbed for AI.

The competition is made up of teams of agents competing against each other in a game similar to soccer. In the robot leagues, agents must be able to sense their opponents, teammates, location on the field, and the ball. They also have to be able to perform low-level tasks such as controlling the ball, passing the ball, and moving without the ball. Higher-level tasks such as coordination of movement between players to score a goal or stop a goal from being scored are also required. These agents must perform in a noisy, real-time environment where their information is only as good as their sensors and their ability to understand sensor information and react quickly enough to make the information useful. In the simulation league, the dynamics of the real-world robot league are lost, so the SoccerServer enforces its own

dynamics to simulate the real world. Players have stamina, a limited ability to communicate with other agents, and a limited ability to sense other agents. Additionally, when an agent senses something or attempts to communicate something, there is an inherent probability that the agent will be successful. Gravity dynamics are adding to the movement and motion of the ball and agents, and dynamics like wind and field conditions are also enforced. A screen shot of the SoccerServer is seen in figure 2.5.

Because of the complexity of the SoccerServer, which makes it a good domain for MAS, when attempting to apply machine learning methods to learn strategies for agents, it is often easier and necessary to simplify the domain. A simpler domain allows machine learning systems to focus on learning certain behaviors and ignore certain complexities like weather dynamics or low-level skills like ball control. For instance, a simulation similar to the SoccerServer was built to allow for robots to compete in the mid-size league to develop strategies by hand-coded or learning techniques. The simulator is in the TeamBots system [28]. TeamBots maintains several of the dynamics of the SoccerServer and allows for strategies to be easily built and tested. Additionally, once a strategy in TeamBots is ready for competition, the TeamBots system can transfer that strategy to real robots. Some low-level functions like sensing opponents and teammates are abstracted away and all agents have direct access to the location of other agents on the field, as well as the ball. This makes the soccer domain easier. Figure [28] gives a screen capture of the TeamBots environment.

In genetic programming, to test the ability of a GP individual for fitness assignment, it is necessary to run the individual in the simulator. In a simulator such as the SoccerServer, a single evaluation takes several seconds. When we are evaluating one thousand individuals in a population, and several generations of populations, it is very necessary to minimize the time required for simulation. This is another reason why alternative simulators are developed. It is possible to speed up the SoccerServer, but due to its complexity, a greater speedup is usually required. Domains such as

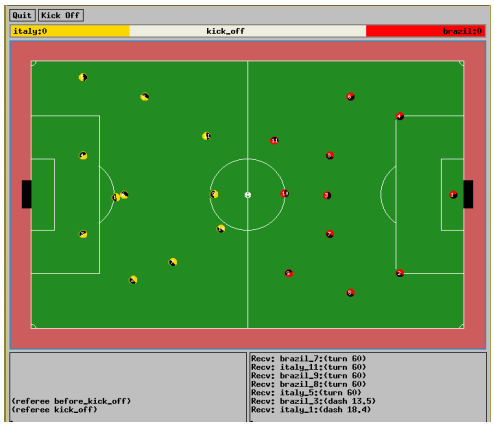


Figure 2.5: Screen capture of the SoccerServer simulator.

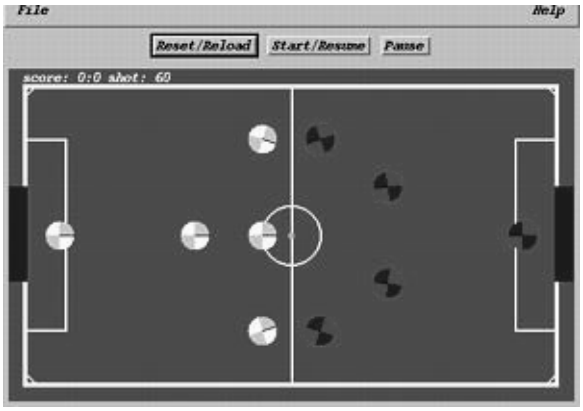


Figure 2.6: Screen capture of the TeamBots simulator.

TeamBots can be run without graphics and run in faster than real-time to greatly reduce the evaluation time. Also, GP systems, like ECJ used here, allow for the evaluation of individuals in a population to occur in parallel to again speed up evaluation time.

For the research done here using the keep-away soccer problem, while it is possible to learn keep-away soccer strategies in the TeamBots environment, which was done at the outset of the research, a simpler domain is used that allows for very quick evaluation and a simpler domain. But the properties of the simulator still would allow for good generalization if the TeamBots or SoccerServer were used. This is to allow for future research to incorporate strategies into teams of agents to compete in a competition like RoboCup. The keep-away soccer problem is described next.

Chapter 3

Keep-Away Soccer

The RoboCup competition is an excellent testbed for MAS and is of interest to a wide variety of MAS research areas [3] [27]. RoboCup competition occurs with real robots and in a simulation league, which presents several interesting challenges for researchers. Reinforcement learning, hierarchical sensing, neural networks, genetic programming, and a variety of hybrid combinations have been previously applied to the RoboCup simulation league [23] [26] and real robot leagues. However, hand-coded methods and hybrid learning still outperform purely learned-agent teams. This poses a continuing challenge to researchers.

3.1 The Problem

In keep-away soccer three offensive agents are located on a rectangular field with a ball and a defensive agent. The defensive agent moves twice as fast as the offensive agents, and the ball can move, when passed, twice the speed of the defensive agent. When passed, the ball can travel a maximum of ten units on the field, but the agent can kick the ball fewer if desired. This is similar to the predator-prey problem in [16] where more than one agent is required to solve the problem. The problem in

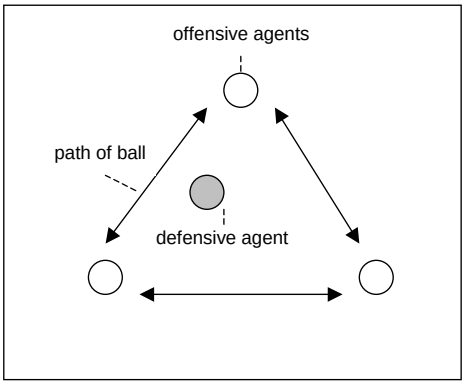


Figure 3.1: The triangle of players as corners and passing lanes as edges.

keep-away soccer is to minimize the number of times the ball is turned over to the defender. A turnover occurs every timestep that the defender is within one grid unit of the ball. Thus, the objective for the offensive agents is to continuously move and pass the ball to other offensive agents to keep the ball away from the defender and minimize turnovers. When humans learn to play keep-away soccer, they learn that to minimize the number of turnovers, they need to pass accurately, control the ball effectively, and move to receive the ball and make themselves open to receive the ball. In Figure 3.1 the triangle organization of players is depicted showing that effective keep-away is played by maintaining a triangle, where corners are players and edges are open passing lanes between the player with the ball and other players. The goal in learning keep-away soccer is for the agents to learn this high-level behavior of coordination.

3.2 Problem Decomposition

Soccer, whether analyzing it as a human game or robotic game, can be broken down into subproblems of optimizing skills like ball control, passing and moving. Keep-away

Table 3.1: The two layers of behaviors for keep-away soccer.

Layer 2 (high level)	Moving and passing to prevent turnovers
Layer 1 (low level)	Accurate passing without a defender

soccer can be decomposed in the same manner. For the experiments here, we think of keep-away soccer as two layers of behaviors: passing accurately to other offensive agents with no defender agent present, and moving and passing with a defender to minimize the number of turnovers that occur in a game. Table 3.1 shows the two layers and how they relate to each other.

The two layers of behaviors come from a human-like view of learning soccer, but are not heavily dependent on domain knowledge. Notice that these two types of behaviors are important to play good keep-away soccer, and are not just two ways to measure the effectiveness of a team of agents who have just played keep-away soccer, which would be useful for finding a fitness function.

3.3 Problem Generalization

Keep-away soccer is an interesting problem for many reasons, the most obvious is the way it generalizes to robotic soccer and other standard MAS problems like the predator-prey problem. If we view soccer as a multi-layered game, in that it requires learning to occur separately at different layers, then keep-away soccer could represent one of those layers. For instance, after learning keep-away soccer, the agents could learn to play soccer with one goal and one defender. Now they not only want to keep the ball away from the defender, as learned in keep-away soccer, they want to score as frequently as possible. For humans to learn soccer, many layers of these types of drills, or simple games, are used to develop the many types of skills needed to play

good soccer. It would then be likely that for robots, or software agents to learn to play the same game effectively, they too would require a similar hierarchy of learning.

Also, just as predator-prey is a good testbed for MAS, keep-away soccer would generalize well to other MAS systems. This generalization means that a solution that allows a system to develop good agents for keep-away soccer would allow good solutions for other MAS problems as in [26]. There are some issues with keep-away soccer that need to be addressed in the implementation phase.

In a soccer game, or in the predator-prey environment, when a goal is scored or the prey is caught, the simulation ends or is reset. But in keep-away soccer, it is unclear whether the simulation should reset, end or continue when a turnover occurs. Because the human game of keep-away soccer is played differently by different people there is not a standard way to handle this. In the research here, the game continues when a turnover occurs, forcing the agents to search out the ball and recover it from the defender. The defender cannot move the ball, so this is not such a difficult task. Some versions of keep-away soccer have the player who is responsible for losing the ball to a defender, or allowing a turnover to occur, becoming the new defender and the defender becomes an offensive player. This raises some interesting issues for learning. If learning was attempting in this type of simulation, it would require complex handling of fitness, as in a single simulation, an agent could score an offensive fitness, how well the ball was kept away from the defender, and a defensive fitness, how well the agent sought and caused a turnover. If this could be implemented in the system, it would allow for defender behaviors to be learned simultaneously, and would be beneficial to the agents ability to generalize quickly to the game of soccer. However, this increases the complexity of the system dramatically and is doubtful that it could be implemented in a clean, nice way, but this could be an area of future research.

In keep-away soccer, the strategy for offensive agents is learned and the defender's

strategy is hand-coded. The reason for this is simple: so long as the defender is faster than the offensive agents, the defender need only head for the ball to cause the most turnovers. If the defender is moving at the same speed as offensive agents, the defender would require more complex strategies to cause a turnover, like faking one way and going the other to trick the offensive agent into passing close to the defender. This kind of strategy would be very interesting to learn and very useful for soccer agents. The last issue that will be discussed is the defender's ability to avoid being blocked from the ball by other agents. In human soccer a player can shield the ball from a defender for a brief amount of time, any long-term attempts to block the defender will usually result in a foul, or turnover. It then needs to be handled in the simulator that agents cannot block the defender indefinitely from the ball. To accomplish this in the simulator, the defender agent is able to move around agents without the ball if the agent is directly between the defender and the ball by simply switching places on the field with the agent, provided the two agents are within one grid unit of each other.

Because keep-away soccer has not been used frequently in the past as a common MAS problem, and soccer has only begun to be used frequently for AI in the last decade, there are several issues, as mentioned above, that need to be dealt with. This is common with any problem, and once a problem is studied more, these issues will be worked out and standards agreed upon.

The layered learning application to GP is presented next, as we explain why keep-away soccer is a good testbed that illustrates its benefits.

Chapter 4

Layered Learning

Applying the layered learning paradigm to a problem consists of breaking that problem into a bottom-up hierarchy of subproblems. When the subproblems are solved in order, where each previous subproblem solution leads to the current subproblem's solution, the original problem is eventually solved. Figure 4.1 shows how individuals in a GP system are generated, evaluated and bred for the next generation, for normal, single-run genetic programming and for layered learning GP. This type of hierarchical solution is different than the hierarchical solution ADFs propose to find, which focus on code reuse and structure, not on how subtasks are learned.

4.1 Layered Learning Genetic Programming

Problems that attempt to simulate human behaviors, like robotic soccer and keepaway soccer, lend themselves well to a bottom-up decomposition. The reason for this is because human learning usually occurs in a bottom-up fashion of first learning the smaller tasks needed to solve a larger task. In fact, when the problem is of this type and we are already using a biologically motivated method like GP, it seems

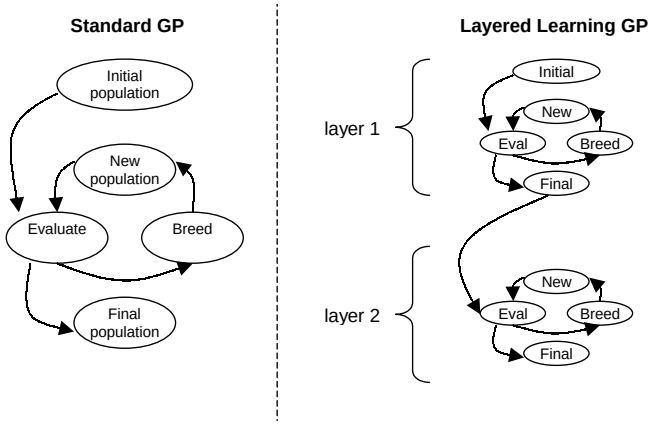


Figure 4.1: Individual propagation in GP and layered learning GP.

very natural to use a bottom-up decomposition of the problem that simulates human learning and allow GP to learn each one of the smaller problems.

Table 4.1 is a modified version of the table found in [22]. Each key principle of layered learning is correlated with a property of genetic programming for keep-away soccer. The first principle in the table is that learning from raw inputs is not tractable, i.e. the domain is so complex that given the domain primitives, the desired output can not be learned. In MAS systems, there is thought to be an inherent complexity of the problem that would overwhelm a learning system that just had access to the system primitives as input. When considering robot soccer in the SoccerServer, for instance, it is easy to see that with all the system complexities, it would be very difficult to start with the system primitives and learn high-level strategies that could compete with hand-coded strategies. For this reason, layered learning proposes to simplify the overall problem by learning smaller problems that represent the decomposed original problem.

The second principle in the table, that a bottom-up decomposition is given as an intuitive approach for learning tasks that are hoping to compete with human learned tasks. The third principle is straightforward for genetic programming, since each

Table 4.1: Key principles of layered learning and the GP keep-away soccer correlation.

Layered Learning	**Genetic Programming**
1. Learning from raw input is not tractable	MAS problems for GP are complex problems
2. A bottom-up decomposition is given	Humanistic learning problems have a natural bottom-up decomposition
3. Learning occurs independently at each level	GP applied to each layer is independent
4. Output of one layer feeds the next layer	Population in last generation of one layer is the next layer's initial population

specified layer represents a separate run in a GP system. And the last principle guides how information is propagated between runs of a GP system, the last population in one run becomes the next run's initial population. There are several issues that arise with the last principle. In the research here, we try two different types of transferring individuals from one run to another. The first type is to copy the best individual to fill the entire next population. The second type is to simply copy the entire previous generation from one layer to the next. These two methods of transferring individuals are discussed later. Other methods, like seeding ADFs with previous layer individuals, or using a tournament selection method as done for breeding, are possible ways to propagate individuals and are all areas of future research.

4.2 Methodology

When we modify standard GP for layered learning, we need to decide what is the learning objective at each layer, i.e. the fitness at each layer that drives the search for ideal individuals. As seen in [15], using a singular-objective fitness value often leads to the best performance, and is much easier than trying to define multi-objective fitness functions. While multiobjective fitness functions should allow GP to evolve more complex behaviors, it becomes difficult to decide what the multiobjective fitness should be and how important each fitness is to the solution. If one fitness is clearly more important than another, it is necessary to decide to what proportion that fitness is more important. While using a multiobjective fitness is a possibility, using a singular-objective fitness seems logical for layers of a layered learning system that represents a decomposition of a larger problem.

The last issue that needs to be addressed for layered learning in GP is that of transferring the population of the last generation of the previous layer to the initial population of the next generation. Because we want to evolve ideal individuals, and in every population there are certain individuals that have a better fitness than the other, we might want to copy that best individual many times to fill the initial population of the current layer. However, that duplication removes the diversity that was evolved from the previous layer, and seems counterintuitive, since the best individual might have scored the best fitness and may in fact only be a suboptimal solution. Therefore, we propose two experiments with layered learning GP, one that duplicates the best individual and one that simply copies the entire population.

Genetic programming is well-suited for the layered learning paradigm, as seen in this chapter. As with any new method, or hybrid method in this case, it remains to be seen that genetic programming benefits from the use of layered learning. In the next chapter, the experiment of learning keep-away soccer is described in detail

and the system parameters are given. These details should allow another researcher to replicate the keep-away soccer problem in the GP system used here, other GP systems, or for another machine learning technique like reinforcement learning.

Chapter 5

Experiment Design

Four initial experiments were chosen to investigate the performance of layered learning GP, standard GP (**SGP**), GP with ADFs (**ADFGP**), layered learning GP with the best individual duplicated to fill initial populations (**LL1GP**), and layered learning GP with the entire last population copied for the next initial population (**LL2GP**). SGP and ADFGP use the single fitness function of minimizing the number of turnovers that occur in a simulation. ADFGP allows each tree for kicking and moving to each have two additional trees that represent ADFs, where the first ADF can call the second ADF, and both have access to the normal function set, as in SGP. LL1GP and LL2GP both have two layers, the first layer's fitness function is to maximize the number of accurate passes, and the second layer's fitness function is to minimize the number of turnovers.

5.1 Five Preparatory Steps

To specify a GP experiment, the five major preparatory steps as described in [11] need to be addressed and for the research done here are summarized in Table 5.1. First, the terminal set needs to be specified. Terminals are the inputs into the system and can

Table 5.1: The five preparatory steps of a GP problem.

Step	Determine	Keep-away soccer setup
1	terminal set	see Table 5.2
2	function set	see Table 5.2
3	fitness function	turnovers(sgp,adfgp,layer2), 200-passes(layer1)
4	control parameters	see Table 5.3
5	stopping criterion	ideal fitness or maximum generations

be the actual primitives from the simulator, or functions that inputs those primitives and modifies them. For instance, in the problem here, the system primitives are the global location of agents on the grid, representing the field. But egocentric vectors, with magnitude and distance, are used as the terminal data type. To accomplish this, the terminals are functions, which compute the egocentric vectors using the system primitive information about which agent is currently being evaluated and the location of the agent and the location where the vector should point to. To keep the terminal set simple, egocentric vectors to the ball, defender agent, and each teammate, mate1 and mate2, are used. Appendex D has two samples representing an individual in the keep-away problem using the terminal and function sets. These are similar terminals that are used in previous RoboCup teams where GP was used.

The function set is determined next, and again, functions can vary the number of arguments they take, but all inputs are vectors. The return type of every function is a vector. The function set used here was chosen to be simple and to be similar to previous work done with GP in RoboCup competitions. The fitness function needs to be specified and is used to assign a fitness value to a GP individual after it has been evaluated in the simulator. The control parameters are used in the GP system that are responsible for breeding and evaluating individuals in each generation's population.

Table 5.2: Keep-away soccer terminal (egocentric vectors) and function set

terminals	functions(args)	Description
defender	rotate90(1)	rotate current vector 90 degrees counter-clockwise
mate1	random(1)	new random magnitude between 0 and current value
mate2	negate(1)	negate vector magnitude
ball	div2(1)	divide vector magnitude by 2
	mult2(2)	multiply vector magnitude by 2
	vadd(2)	add two vectors
	vsub(2)	subtract two vectors
	iflte(4)	if v1 $<$ v2 then v3, else v4 (comparing magnitudes)

The control parameters used here are from [12] and are standard control parameters for many GP benchmark problems. These parameters were also the ECJ [13] system defaults for GP problems. The last preparatory step for a GP problem is determining how a GP run will stop. Usually, a maximum number of generations is specified so that if an ideal fitness is not found before then, the run stops after the maximum generation. That stopping criterion is used here, and will stop if an ideal fitness is found or when the maximum generation is reached.

5.2 Control Parameters

Six variations of each experiment were developed that use the standard GP parameter settings as described in [12], summarized in Table 5.2. The maximum generations allowed per run and the population size are varied for the experiments. Maximum generation values are 51 and 101, and population sizes of 1000, 2000, and 4000. Six different runs then are done for each type of experiment, SGP, ADFGP, LL1GP and

LL2GP. The genetic operators crossover and reproduction create 90 and 10 percent of the next generation, respectively. Tournament selection is used of size 7 with maximum depth 17. Table 5.1 summarizes the function set used and is similar to function sets used in [15] and [1].

5.3 ECJ and the Simulator

The GP system used was developed by Sean Luke and is called Evolutionary Computation in Java [14][13] (ECJ). ECJ is a strongly typed GP system and allows for problems to be easily specified and evaluated. The simulator designed for keep-away soccer abstracts some of the low-level details of agents playing soccer from the TeamBots [28] simulator, which abstracts low-level details from the SoccerServer [2]. Abstractions in this way would allow the keep-away soccer simulator to be incorporated later to learn strategies for the TeamBots simulator and the SoccerServer.

In the SoccerServer, and TeamBots, players push the ball to maintain possession. To kick the ball, the player needs to be within a certain distance. For keep-away soccer, we eliminate the need for low-level ball possession skills and allow offensive agents to have possession of the ball. Once an agent has possession, possession is only lost when the ball is kicked, according to the evaluation of the agent's kick tree. Because vectors are used, which have direction and magnitude, this implementation would allow for dribbling actions to be learned where the agent simply passes it one unit away. This abstraction greatly simplifies the problem and still allows for a wide range of learned behaviors.

At each simulation step that allow agents to act, if the agent has possession of the ball, i.e. the agent is on top of the ball in the grid, the agent's kick tree is evaluated. Otherwise, the agent's move tree is evaluated.

Each evaluation of an individual in the simulator takes 200 timesteps, where the

Table 5.3: The GP control parameter settings that are typical for benchmark GP problems, from [12], Appendix D.

Two Major Parameters		
1	population size M	1,000 , 2,000 , 4,000
2	max generations to run G	51 , 101
Eleven Minor Numerical Parameters		
3	crossover probability	90%
4	reproduction probability	10%
5	choose internal points for crossover	90 %
6	max. size of programs	17
7	max. size for initial random programs	6
8	mutation probability	0%
9	permutation probability	0%
10	frequency of editing	0
11	encapsulation probability	0%
12	decimation condition	nil
13	decimation target percentage	0%
Eight Minor Qualitative Variables		
14	generative method for init random pop	ramped half-and-half
15	basic selection method	tournament, group size = 7
16	spousal selection method	tournament, group size = 7
17	adjusted fitness	not used
18	over-selection	not used
19	elitist strategy	not used
20	randomization of fitness cases	fixed(random) for all runs
21	structure preserving crossover	branch typing

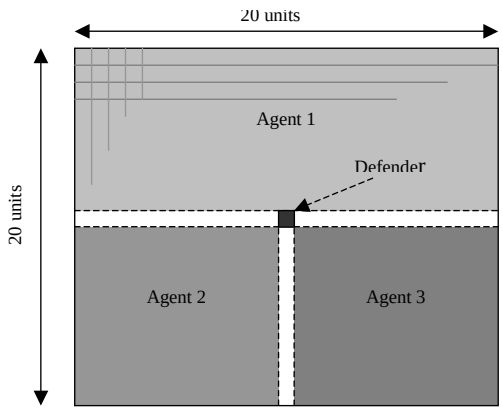


Figure 5.1: Field configuration for initial setup of agents and defender.

ball can move on each step, if passed. The defender moves on every other step, and all offensive agents move together on every fourth timestep. The initial setup of the simulation places the defender agent in the middle of a 20 by 20 unit grid. The field is then partitioned into three sections, the top half and the bottom left and right halves. Offensive agents are placed randomly within those sections, one in each, and the ball is placed a few units from one of the offensive agents, chosen at random. Figure 5.1 shows the field configuration and the areas where agents are initially placed. Because of all the randomness involved in the setup and evaluation, the same individual could produce different fitness values.

For layered learning experiments, 40 percent of the maximum number of generations are spent in layer 1 learning accurate passing without a defender present. To evaluate accurate passes, we count the number of passes which are passed to a location that is within 3 grid units of another agent. The fitness is then $200 - passes$, where there are 200 timesteps in a simulation and a fitness of 0 is best and ∞ the worst. The remaining 60 percent of generations are spent in layer 2 with a fitness value based on the number of turnovers that occur with a defender present.

Early runs of the system resulted in local optima being achieved, the most common

was all the offensive agents crowding the ball and preventing the defender from causing a turnover. To overcome this, the defender, if blocked from the ball, can move through an offensive agent without the ball by simple trading places with the agent if the two are within one unit on the grid.

Each experiment was run approximately 10 times and averages were taken across those runs. Running on a 16-processor 400 MHz Sun Ultra-Enterprise 10000 machine, evaluation of one generation took approximately 2 seconds for population size of 1000 and approximately 4-8 seconds for population size of 4000. Therefore, several hours are needed to collect results for only 10 runs of the experiments, when different experiments are run in parallel.

5.4 Developed Code

While ECJ was used as the GP system to breed, assign fitness, and collect statistics, the simulator is the heart of the system in that it allows individuals to be evaluated and produces the values that are the individuals' fitness values. Thus, to specify a GP problem in ECJ, it requires the development of a simulator for the evaluation of individuals. While there are already examples and samples in ECJ that can be reused, most problems require modifying existing classes for collecting statistics, individual evaluation, functions and terminals, and saving and loading populations as done for layered learning. Additionally, parameter files, which specify the control parameters and experimental setup for each run, need to be written. Thus, for the keep-away soccer problem, a class for evaluating individuals, for the basic data type of terminals, for visualization of the simulator, for managing statistics, and a class for managing populations from previous runs were all developed.

The classes that were developed for doing the evaluating, breeding and statistics required about 1750 lines of code, where about 750 of that was brand new code not

reused from any examples or samples. To code the functions and terminals required about 1000 lines of code and a parameter file for each of the standard GP setup, including each layered learning experiment, took about 100 lines of code and 350 lines of code were required for the ADF experiments.

There were a total of six different experiments run, and each experiment had 10 51x20 or 101x20 tables of values, one for each run. There were then approximately 330 tables of values produced. Several scripts were written to place these values into spreadsheets, combine data for similar runs, take averages, and to make graphs. The correct management of all the results required a fair amount of time and planning to efficiently analyze them. All the code, including for ECJ, to run the system, collect results, analyze and produce graphs, as well as an electronic version of this document, are located on the included CD-ROM.

Chapter 6

Results

6.1 Initial Experiments

6.1.1 SGP

The standard GP experiment converged to a good fitness, better than originally expected. After looking more closely at the problem, the simplifications made to the system allowed for a reasonable solution. Most important of these simplifications were allowing for two trees, a kick and move tree, and allowing for possession of the ball. Figure A.1 in Appendix A shows the learning curve for the best fitness of generation for the SGP and ADFGP experiments, the least number of turnovers that occurred in each of the 51 generations, where these numbers represent the average over 10 runs of the experiment. The plot of best fitness per generation is a standard plot for GP experiments, showing how quickly a GP run achieves a good fitness. To see how the whole generation of each run performs, a plot of the mean fitness per generation is usually used. Figure A.1 also gives the fitness in the last generation so that we may easily compare this with other runs.

For all experiments, the runs of 101 generations performed better and converged

better than the 51 generation runs. For comparative purposes, only the 101 generation runs are used. Graphs of best fitness for all experiments where runs with 51 generations were done can be found in Appendix A.

6.1.2 ADFGP

ADFGP experiments converged to fitness values in two clusters, one being better than SGP, and the other much worse. When observations of the individual size were accounted for, it originally appeared that the bad cluster contains individuals with about half the number of nodes as individuals in the good cluster. Prefiltering ADFGP runs based on individual size may be appropriate to remedy this, but since we are not explicitly studying ADFGP, we still use the averages here as this is only a hypothesized explanation of ADFGP. However, more experiments were done to collect 10 runs that were in the better cluster. When the best fitness graphs and the average nodes per individuals graphs were compared with these new runs, it became clear that prefiltering in this way would not work. The two clusters of good and bad best fitness did not correlate with the two clusters found in the average nodes per individual graph. This shows no correlation between the two attributes. Figure 6.1 shows these results. Thus for comparing ADFGP, the average of the original 10 runs, containing the two clusters, is still used. To see the performance of the best runs of ADFGP, 20 total runs of ADFGP were done and the 10 best runs were selected and averaged for the results, labeled **Good-ADFGP**, in Table 6.2, and shown in Figure 6.7.

6.1.3 LLGP

LL1GP, with duplicating the best individual from the previous layer, did much worse than SGP and ADFGP. LL2GP was competitive with SGP and ADFGP, remembering

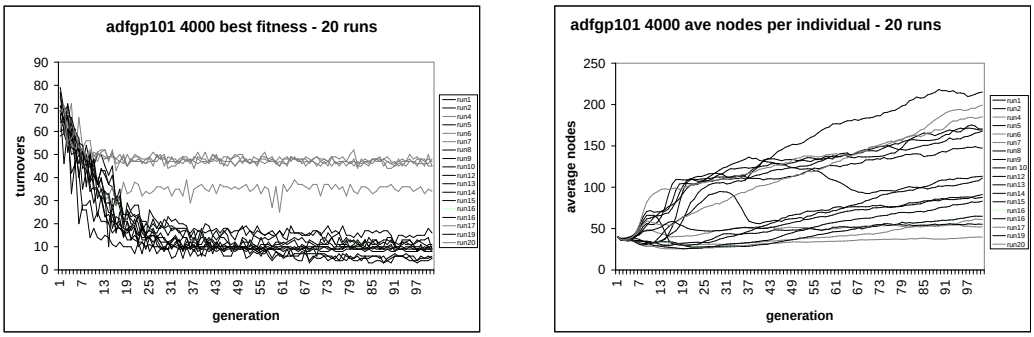


Figure 6.1: Best fitness and average nodes per individual for 20 runs of ADFGP.

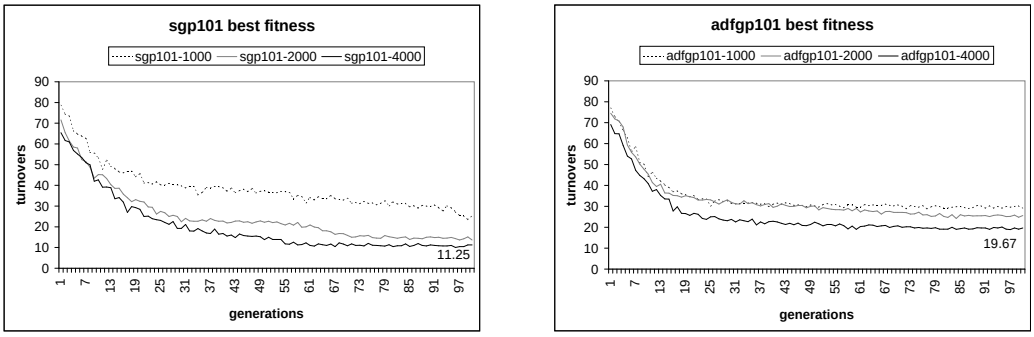


Figure 6.2: Best fitness for SGP and ADFGP.

that the initial 40 generations, are spent in layer 1 learning accurate passing.

The best fitness graphs for SGP, ADFGP, LL1GP, and LL2GP, with 101 generations, are shown in Figure 6.2, 6.3 and 6.4.

6.1.4 New Experiments

These results do not highlight a strength or weakness of layered learning for GP, except that we can get nearly the same solutions with LL2GP as with SGP and

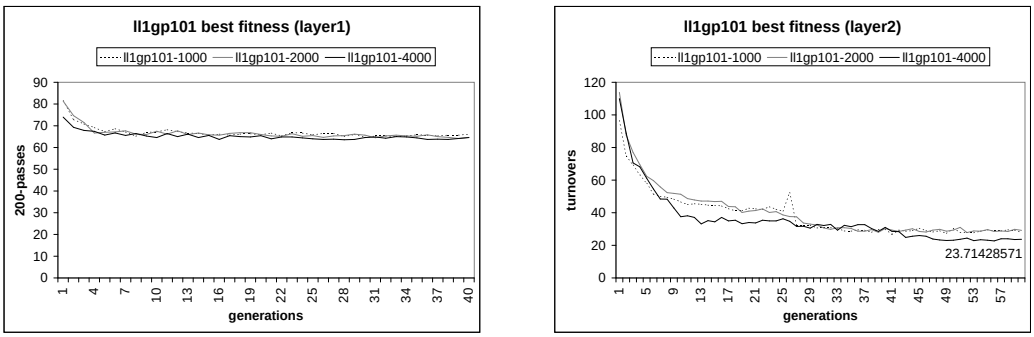


Figure 6.3: Best fitness for LL1GP, layers 1 and 2.

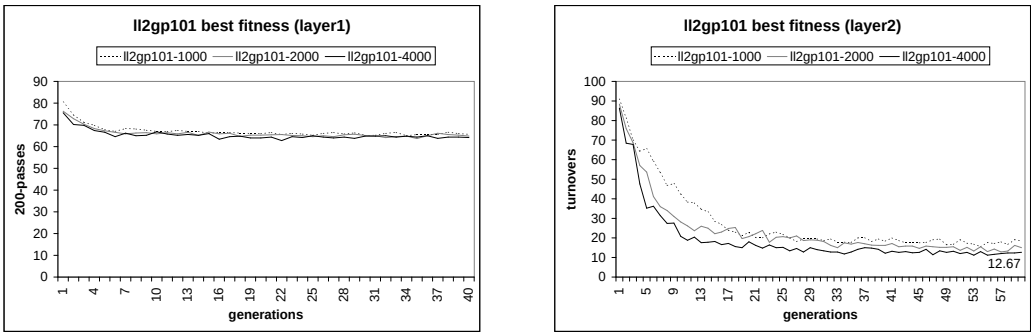


Figure 6.4: Best fitness for LL2GP, layers 1 and 2.

Table 6.1: Experiment similarities and differences for SGP, ADFGP, LL1GP, LL2GP, nLL2GP and n2LL2GP.

	sgp	adfgp	ll1gp	ll2gp	nll2gp	n2ll2gp
generations	51,101					
population size	1000,2000,4000					
number layers	1		2			
layer1 fitness	turnovers		200 - accurate passes			
layer2 fitness			turnovers			
% generations in layer1	100%		40%	40%	20%	10%
% generations in layer2			60%	60%	80%	90%

ADFGP. However, when we look at the learning curve for best fitness per generation of layer 1 in LL2GP, we notice that convergence takes place in about 15 generations and settles to the same value for the rest of the run. This hints that perhaps we do not gain anything from running for a total of 40 generations. Two new experiments are then developed to test this hypothesis.

6.2 New Layered Learning GP 1 and 2

New layered learning GP, nLL2GP and n2LL2GP, are exactly the same as LL2GP, except that for nLL2GP the first layer is only run for 20 generations, and the second is run for 81, totaling 101 generations. For n2LL2GP, the first layer is run for 10 generations and the second for 91. Table 6.1 summarizes the similarities and differences of all the experiments done here.

Figure 6.5 shows the learning curves for the new experiment, nLL2GP, with the fitness of the last generation labeled. Figure 6.6 shows the same learning curves for

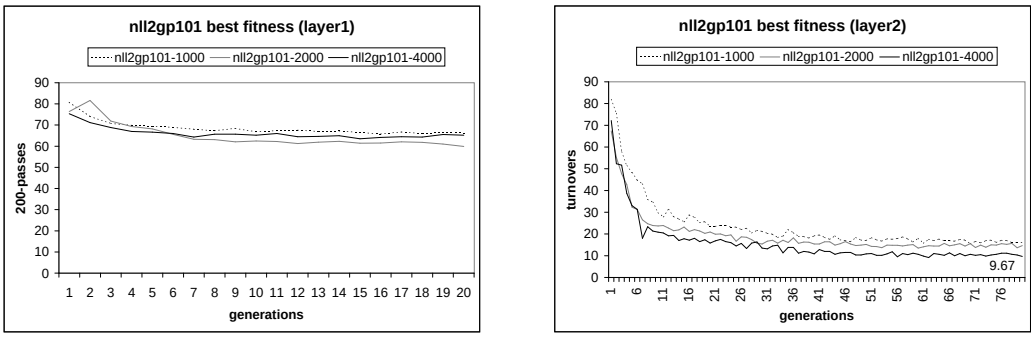


Figure 6.5: Best fitness for nLL2GP experiments, 101 gen.

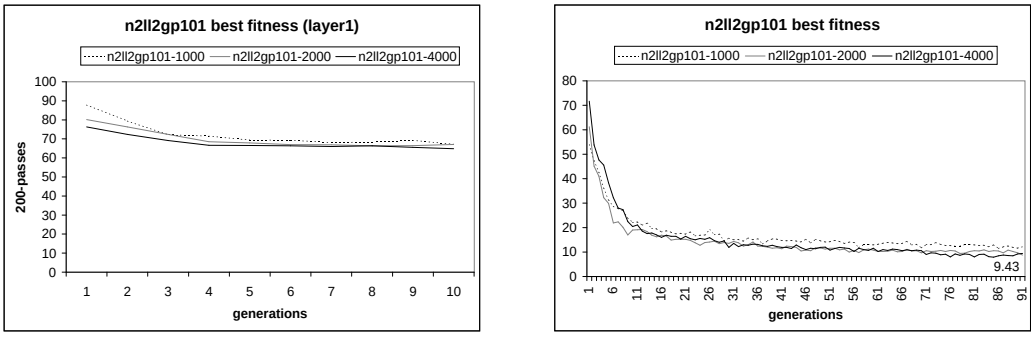


Figure 6.6: Best fitness for n2LL2GP experiment, 101 gen.

n2ll2gp. We see a steeper drop in fitness in n2LL2GP, layer 2, and a better resulting fitness. The nLL2GP experiment showed some improvement, but not as much as n2LL2GP. Lastly, Figure 6.7 shows the fitness for the 10 best runs of ADFGP selected from 20 total runs.

The same learning curves for mean fitness show that n2LL2GP, nLL2GP, SGP and ADFGP all produce the same approximate values, with n2LL2GP being slightly better. Appendix B gives the plots for the mean fitness for all experiments with 101 generation runs.

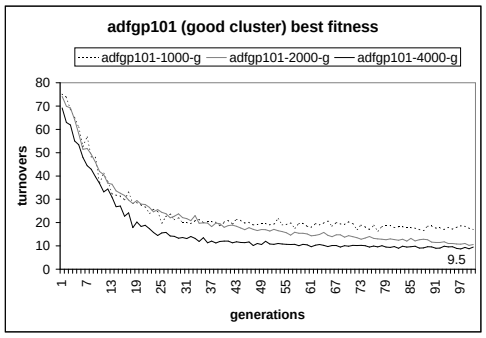


Figure 6.7: Best fitness for best runs of ADFGP, 101 gen.

These results suggest that a natural breakdown of the problem into subproblems, where GP solves each of the subproblems, could allow for a better converged fitness and speed up the learning over SGP. The standard deviation of the several runs of SGP, ADFGP, nLL2GP, n2LL2GP were 4.98, 17.45, 2.73 and 2.28 respectively, showing good stability for n2LL2GP. Table 6.2 gives some descriptive statistics for fitness and individual size across all experiments. Note that individual size is the average number of nodes, where each node represents a function, terminal, or ADF call, of an individual in a generation.

Examining best of run individuals show the emergence of several behaviors, moving without the ball to avoid defenders, passing to open agents, and spreading out across the field, i.e. not crowding other agents or the ball. All the results highlight several other areas of interesting and worthwhile research with layered learning and keep-away soccer, and for other domains as well.

Table 6.2: Data for experiments with population size=4000, max generations=101, and averaged over 10 runs. Good-ADFGP represents the average of the 10 best runs selected from 20 runs of ADFGP.

	sgp	adfgp	good adfgp	ll1gp	ll2gp	nll2gp	n2ll2gp
best fit. gen. 101	11.25	19.67	8.75	23.71	12.67	9.67	9.43
mean fit. gen. 101	66.89	60.21	64.27	82.03	64.64	74.78	70.39
ave. ind. size gen.101	228.74	113.25	123.07	161.71	171.40	217.36	249.21
1^{st} gen. fit.≤ 20	33	62	22	101	55	31	26
best fit. of run	9.0	16.56	6.83	19.29	9.0	7.32	5.78

Chapter 7

Conclusions

The hypothesis from the introduction is restated:

Hypothesis: The layered learning paradigm allows genetic programming to evolve more highly fit individuals in fewer generations than standard genetic programming by taking advantage of the natural decomposition of problems.

We showed that using layered learning for GP can evolve more highly fit individuals than standard GP. Additionally, layered learning GP allows for a natural decomposition of a large problem into subproblems. Each subproblem is then more easily solved with GP. The keep-away soccer problem is a good testbed for abstracting away the complexities of simulated soccer and allows for different GP methods to evolve good solutions for comparing methods. It is also an easily extended problem to the full game of soccer and transferred across platforms to other domains such as TeamBots and the SoccerServer from the simulator that was written here in ECJ.

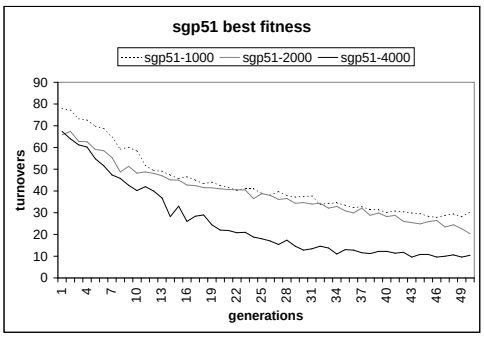
Intuitively, we can liken our success with layered learning in GP with the success of human soccer teams. Successful teams are usually made up of players with unique strategies, where learning took place in a bottom-up fashion. The n2LL2GP experiment simulates this kind of behavior, where we attempt to minimize the number of generations needed per layer. The results indicate that it is beneficial to learn com-

plex behaviors in a layered learning approach with GP, instead of standard GP, as it is easier to decide on fitness functions and natural to decompose the overall problem.

Thus, the hypothesis was shown to be correct by the experimental results found here. Additional research with this problem and other problems for layered learning in GP will need to be done to verify this finding. Some future work and extensions for this research are discussed next.

There are several extensions to this research that would be of interest. Developing a team for RoboCup competition using layered learning in GP would be a good way to test its ability more thoroughly. Studying other statistics about SGP, ADFGP, and n2LL2GP experiments could lead to other interesting conclusions, as would attempting to optimize better the number of generations needed in each layer. Diversity in populations is also an interesting issue, and whether layered learning promotes diversity. Other interesting modifications include developing heterogenous teams, adding additional lower and higher-level layers, and allowing ADFs in layered learning.

Appendix A



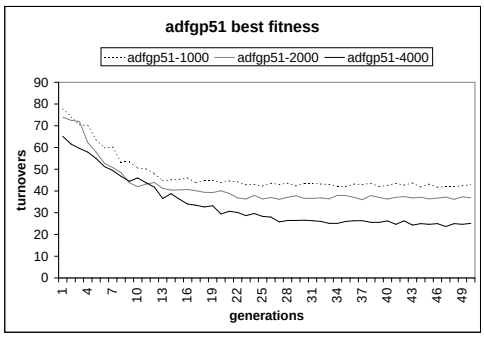
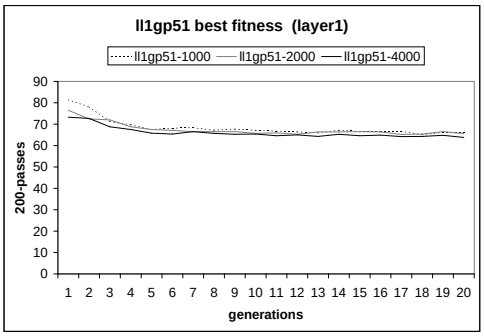


Figure A.1: Best fitness for SGP and ADFGP.



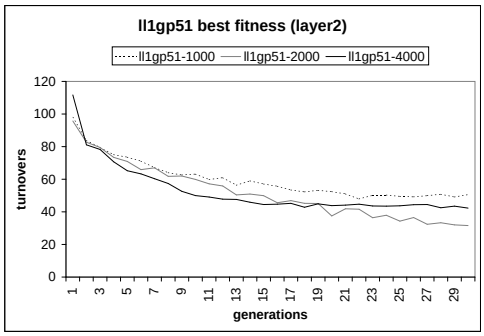


Figure A.2: Best fitness for LL1GP, layers 1 and 2.

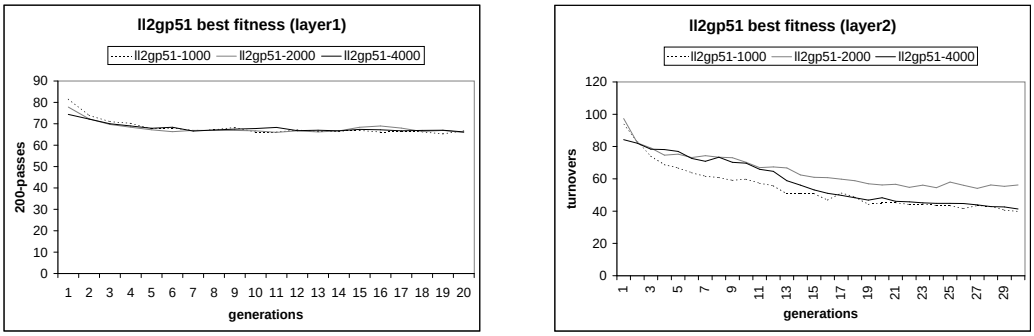


Figure A.3: Best fitness for LL2GP, layers 1 and 2.

Appendix B

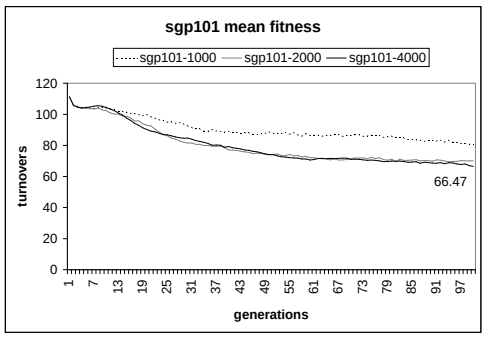


Figure B.1: Mean fitness for SGP.

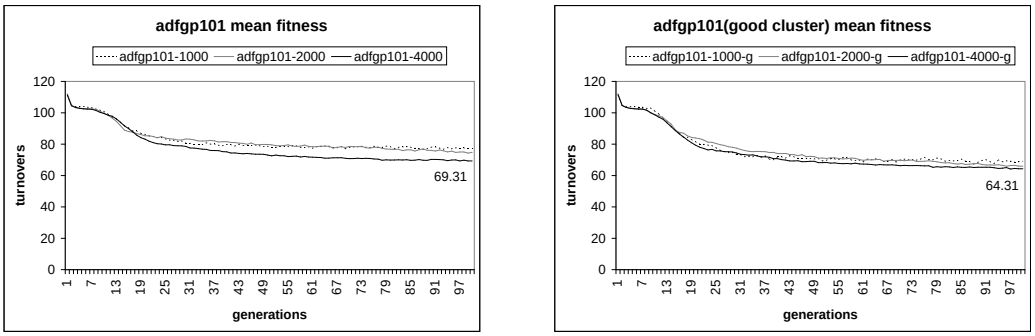
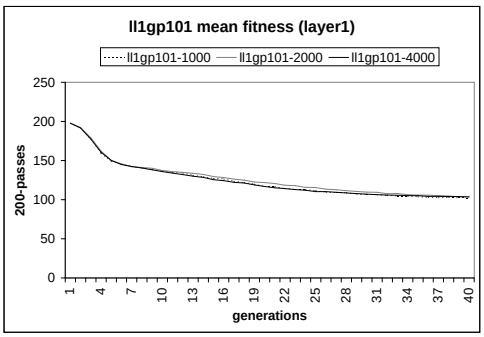


Figure B.2: Mean fitness for ADFGP and Good-ADFGP.



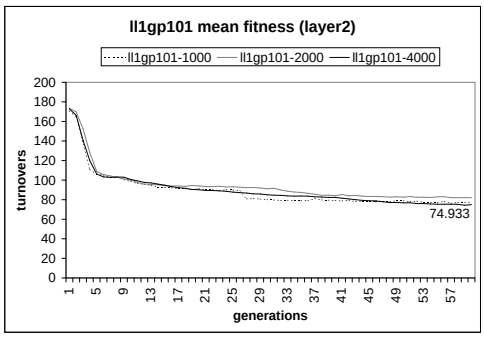
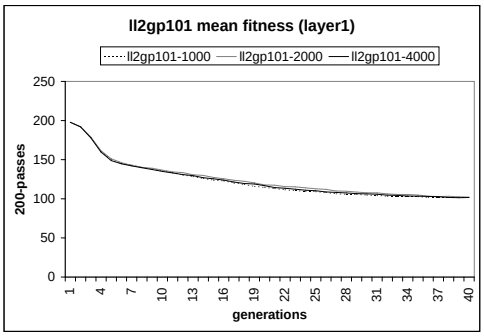


Figure B.3: Mean fitness for LL1GP, layers 1 and 2.



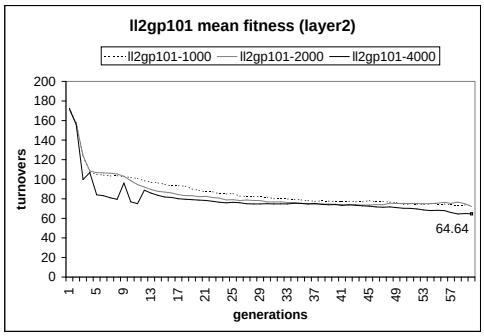


Figure B.4: Mean fitness for LL2GP, layers 1 and 2.

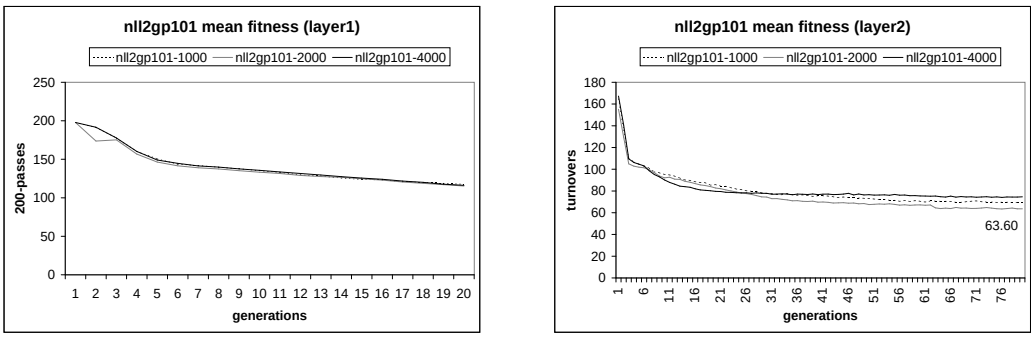


Figure B.5: Mean fitness for nLL2GP, layers 1 and 2.

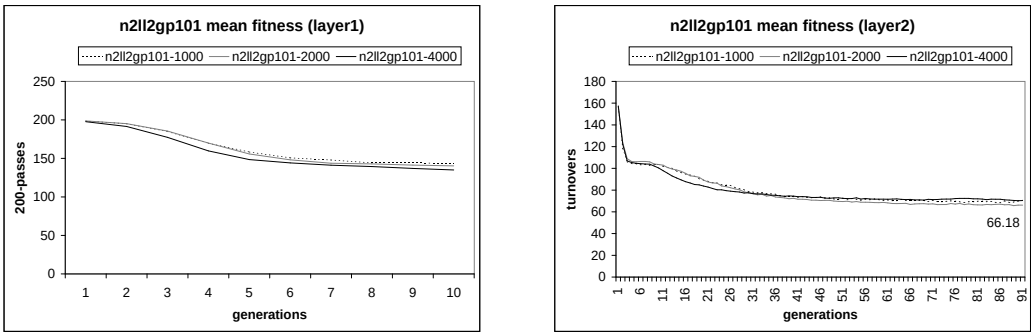


Figure B.6: Mean fitness for n2LL2GP, layers 1 and 2.

Appendix C

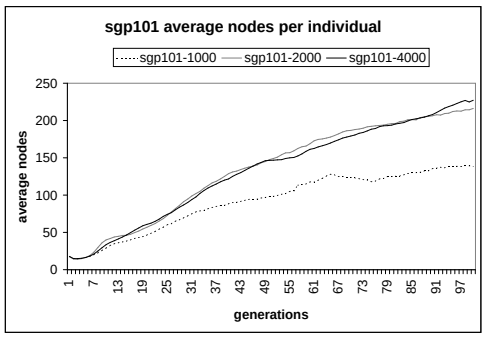
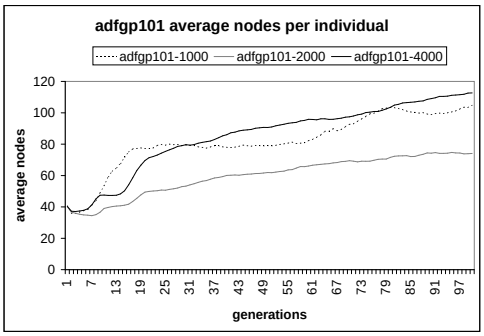


Figure C.1: Ave. number of nodes per individual for SGP.



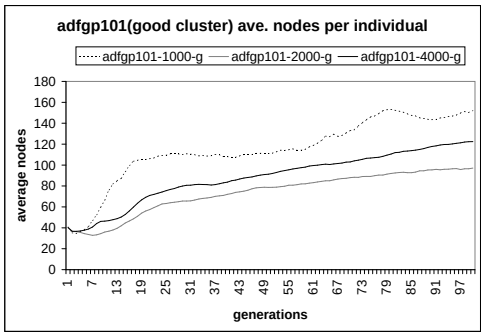


Figure C.2: Ave. number of nodes per individual for ADFGP and Good-ADFGP.

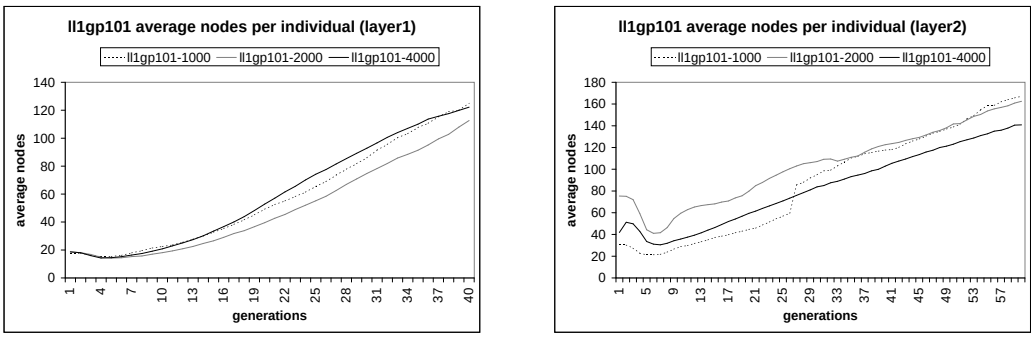


Figure C.3: Ave. number of nodes per individual for LL1GP.

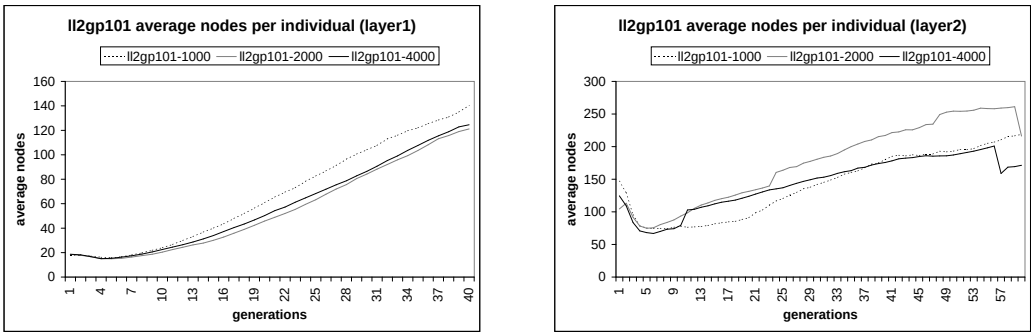


Figure C.4: Ave. number of nodes per individual for LL2GP.

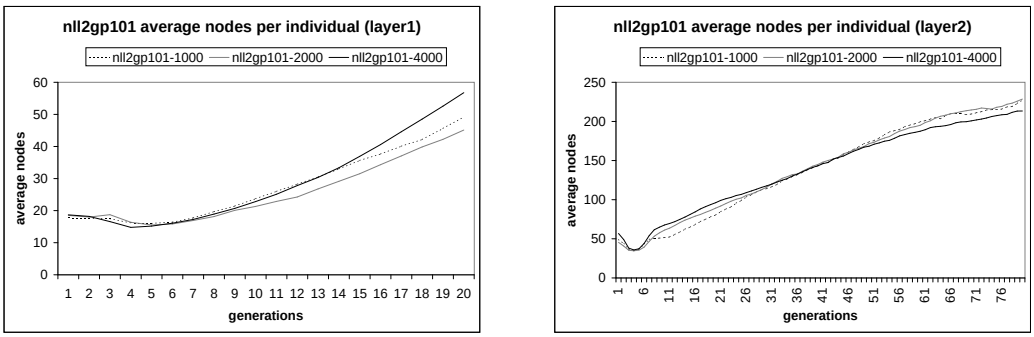


Figure C.5: Ave. number of nodes per individual for nLL2GP.

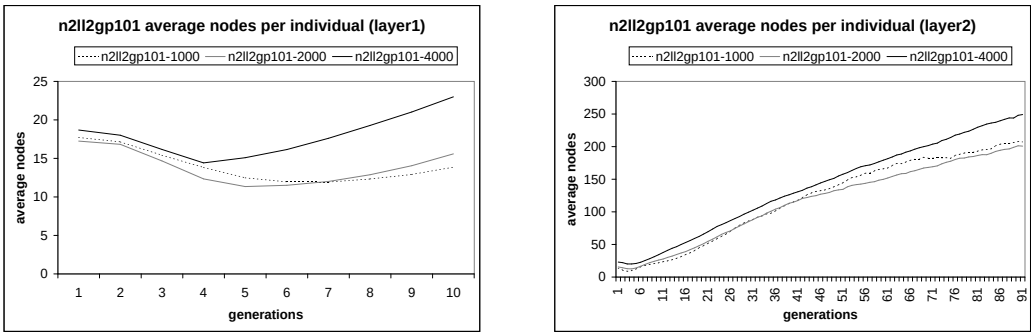


Figure C.6: Ave. number of nodes per individual for n2LL2GP.

Appendix D

Tree 0:	(vadd ball (iflte ball (mult2 ball) (vadd (vadd mate1 ball) (vsub mate2 mate2)) (mult2 (negate (negate (negate (iflte (iflte (iflte (vadd ball (rotate90 (vadd (vsub mate1 (div2 (div2 (div2 mate1)))) (vadd ball (iflte mate2 (random mate2) ball mate2))))) ball (iflte mate1 mate1 ball ball) (vsub mate2 mate1)) (vadd ball ball) mate1 ball) ball (iflte ball ball mate1 mate1) (vadd mate2 mate2))))))))
Tree 1:	(iflte (vsub ball mate1) (mult2 ball) (vsub mate1 mate1) (iflte mate1 (div2 (rotate90 (random mate2))) ball ball))

Figure D.1: Best-of-run individual from a n2LL2GP run. Tree 0 is the *kick-tree* and Tree 1 is the *move-tree*.

Tree 0:	(vadd (vadd (vadd (vadd mate1 ball) (vadd ADF0[2] ADF0[2])) (vadd (vadd ADF0[2] ADF0[2]) (mult2 (vadd (vadd (vadd mate1 ball) (vadd ADF0[2] ADF0[2])) (vadd (vadd ADF0[2] ADF0[2]) (mult2 (vadd mate1 ball))))))) (vadd ADF0[2] mate1))
Tree 1:	(iflte (mult2 ball) (negate (mult2 defender)) ball (negate (mult2 (iflte ball (rotate90 (iflte (iflte (mult2 ball) (negate (iflte (iflte ball (rotate90 defender) ADF2[4] mate2) (mult2 defender) (mult2 defender) (iflte ADF2[4] ADF3[5] ADF2[4] mate1))) ball (negate mate2)) (mult2 defender) ball (negate mate2))) ADF2[4] mate2))))
Tree 2:	(vadd (mult2 mate2) (vadd (vadd (mult2 mate2) (mult2 mate2)) (random defender)))
Tree 3:	(div2 (vsub defender ball))
Tree 4:	ball
Tree 5:	(vsub (mult2 (random (mult2 (random (mult2 defender))))) (vsub defender mate1))

Figure D.2: Best-of-run individual from a ADFGP run. Tree 0 is the *kick-tree*, Tree 1 is the *move-tree*. Trees 2 and 3 are ADFs for Tree 0, and trees 4 and 5 are ADFs for Tree 1.

References

[1] Andre, D., A. Teller. 1998. Evolving Team Darwin United. In M. Asada and H. Kitano, editors, *RoboCup-98: Robot Soccer World Cup II*, volume 1604 of LNCS. France. Springer Verlag.

[2] Andre, D. et al. 1999. Soccerserver Manual. Ver. 4, Rev. 02. Available through the World-Wide Web at http://www.robocup.org.

[3] Asada, M. et. al. 1999. Overview of RoboCup-98. In M. Asada and H. Kitano, editors, *RoboCup-98: Robot Soccer World Cup II*, volume 1604 of LNCS. France. Springer Verlag.

[4] Darwin, C. 1859. *The Origin of Species by Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life.* Mentor Reprint, 1958, NY.

[5] Fogel, D.B. 1998. *Evolutionary Computation, The Fossil Record: Selected Readings on the History of Evolutionary Algorithm.* IEEE Press.

[6] Holland, J.H. 1975. *Adaption in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* University of Michigan Press.

[7] Hsu, W. H., Auvil, L. S., Pottenger, W. M., Tcheng, D. and Welge, M. 1999. Self-Organizing Systems for Knowledge Discovery in Databases. In *Proceedings of the International Joint Conference on Neural Networks*. Washington, DC.

[8] Hsu, W. H., Cheng, Y., Guo, H., and Gustafson, S. 2000. Genetic Algorithms for Reformulation of Large-Scale KDD Problems with Many Irrelevant Attributes. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Las Vegas, NV.

[9] Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., Osawa, E. 1995. RoboCup: The Robot World Cup Initiative. In *Proceedings of the IJCAI-95 Workshop on Entertainment and AI/ALife*.

[10] Kitano, H., et al. 1997. The RoboCup Synthetic Agent Challenge 97. In *Proceedings of IJCAI-97 Conference*.

[11] Koza, J.R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press

[12] Koza, J.R. 1994. *Genetic Programming 2*. MIT Press

[13] Luke, S. 2000. ECJ: A Java-based Evolutionary Computation and Genetic Programming Research System. Version 4. Available through the World-Wide Web at http://www.cs.umd.edu/projects/plus/ec/ecj/.

[14] Luke, Sean. 2000. *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat*. Ph.D. Dissertation, Department of Computer Science, University of Maryland, College Park, Maryland.

[15] Luke, S. 1998. Genetic Programming Produced Competitive Soccer Softbot Teams for RoboCup97. In J. Koza et al, editors, *Proceedings of the Third Annual Genetic Programming Conference*. San Fransisco. Morgan Kaufmann.

[16] Luke, S. and L. Spector. 1996. Evolving Teamwork and Coordination with Genetic Programming. In J. Koza et al, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*. Cambridge. MIT Press.

[17] Luke, S., C. Hohn, J. Farris, G. Jackson, and J. Hendler. 1998. Co-evolving Soccer Softbot Team Coordination with Genetic Programming. In H. Kitano, editors, *RoboCup-97: Robot Soccer World Cup I*, volume 1395 of LNCS. Berlin. Springer-Verlag.

[18] Matsubara, H., Noda, I., Hiraku, K. 1997. Learning of Coorperative actions in multi-agent systems: a case study of pass play in Soccer. In *Adaption, Coevolution and Learning in Multiagent Systems: Papers from the 1996 AAAI Spring Symposium*. Menlo Park, CA. AAAI Press.

[19] Montana, D. J. 1995. Strongly Typed Genetic Programming. In *Proceedings of Evolutionary Computation*. volume 3, number 2. pp. 199-230.

[20] Rosca, J.P. and Ballard, D.H. Hierarchical Self-Orgainization in Genetic Programming. In *Proceedings of the Eleventh International Conference on Machine Learning*. pp.251-258. Morgan Kaufmann Publishers, Inc.

[21] Stone, P., Veloso M., Riley, P. 1999. The CMUnited-98 Champion Simulator Team. In M. Asada and H. Kitano, editors, *RoboCup-98: Robot Soccer World Cup II*, volume 1604 of LNCS. France. Springer Verlag.

[22] Stone, P., Veloso, M. 2000. Layered Learning. *Eleventh European Conference on Machine Learning*.

[23] Stone, P., Veloso, M. 1998. *A Layered Approch to Learning Client Behaviors in the RoboCup Soccer Server*. Applied Artificial Intelligence, volume 12.

[24] Stone, P., Veloso, M. 2000. *Multiagent Systems: A Survey from a Machine Learning Perspective*. Autonomous Robots, volume 8, number 3.

[25] Stone, P., Veloso, M. 1999. Team-Partitioned, Opaque-Transition Reinforcement Learning. In M. Asada and H. Kitano, editors, *RoboCup-98: Robot Soccer World Cup II*, volume 1604 of LNCS. France. Springer Verlag.

[26] Tambe, M. 1997. *Towards Flexible Teamwork*. Journal of Artificial Intelligence Research, volume 7, Pages 83-124.

[27] Tambe, M., Adibi, J., Alonaizon, Y., Erdem, A., Kaminka, G., Marsella, S. and Muslea, I. 1999. *Building agent teams using an explicit teamwork model and learning*. Artificial Intelligence, volume 110, pages 215-240.

[28] TeamBots software and documentation. Available through the World-Wide Web at http://www.teambots.org.