Using Genetic Programming and Decision Trees for Team Evolution

Siphesihle Philezwini Sithungu
Academy of Computer Science and Software Engineering
University of Johannesburg
Johannesburg, South Africa
siphesihles@uj.ac.za

Duncan Anthony Coulter
Academy of Computer Science and Software Engineering
University of Johannesburg
Johannesburg, South Africa
dcoulter@uj.ac.za

Elizabeth Marie Ehlers
Academy of Computer Science and Software Engineering
University of Johannesburg
Johannesburg, South Africa
emehlers@uj.ac.za

ABSTRACT

This paper presents work done to evolve soccer strategies through Genetic Programming. Each agent is controlled by an algorithm in the form of a decision tree to act on the environment given its percepts. Several experiments were performed and an analysis of the performance of the algorithm was documented afterwards. Experimental results showed that it is possible to implement soccer learning in a multi-agent system through Genetic Programming, although the evolution of higher-level soccer strategies is a more difficult task.

CCS CONCEPTS

• Computing methodologies→Genetic programming

KEYWORDS

genetic programming, decision trees, evolutionary learning

1 Introduction

Without much analysis, soccer might appear to be an easy game to play, with a negligible requirement for tactical approaches. However, at high levels of professional play, soccer involves complexity. Furthermore, a team's performance may be determined by several factors, such as the fitness levels and technical capabilities of players, overall team morale, the strategies applied [1] and environmental factors.

The complexity of soccer makes it a good testbed for multi-agent system research [2] [3]. Simulated soccer allows for two types of learning: *team learning*, where a single learner learns joint solutions, and *concurrent learning*, in which multiple agents learn simultaneously [4]. Soccer has both the characteristics of cooperative [5] (participating in a team) and competitive [6] (winning against opponents) multi-agent systems. The existence of internal competition within a team is also possibility because each agent may want to outperform other agents.

The work presented here proposes the use of Genetic Programming (GP) to evolve player behaviour so that it improves over time. The approach to be taken is that of concurrent learning, where each player in the field is represented by an agent that learns how to play the game intelligently enough. In order to achieve this aim, rules based on simple and primitive actions (i.e. choosing a direction) must be evolved to a point where each agent learns to play soccer. The idea is to avoid introducing any form of bias when defining the fitness function, as noted in [7]. Therefore, at the beginning of each experiment, no player is expected to possess any knowledge of soccer (i.e. to show no hand-coded coherent behaviour). As time progresses, basic improvement and emergent play should be observed.

Evolutionary Computation (EC) - better known as Evolutionary Algorithms (EA) - is a subfield of Computational Intelligence (CI) that encodes the processes of natural evolution. Charles Dar- win's theory of *natural selection* is considered the foundation of biological evolution and briefly states that: Individuals of a population compete in order to thrive in a world of finite resources. Those individuals possessing the most useful characteristics have a better chance of surviving and reproducing offspring .

Through reproduction, the characteristics of an individual are passed on to their offspring. Therefore, according to Darwin's theory, useful characteristics will be carried on to and inherited by subsequent generations, resulting in those characteristics being the most common across the population [8]. GP is part of the family of EAs and is inspired by the Genetic Algorithm (GA), which adopts the following concepts from biological evolution: (1) *fitness evaluation*, (2) *crossover* and (3) *mutation* [9]. In the GP context, a GA runs on a population computer programs rather than bit strings.

Genetic Programming (GP) deals with an important objective of computer science: creating computer programs that can solve problems without the provision of explicit instructions. This is referred to as *automatic programming*, a concept that refers to the automatic creation of computer programs that enable computers to solve problems.

An initial population of randomly generated computer programs made up of available programmatic elements is evolved by methods of natural selection in order to produce populations of improved programs [10]. At the end of each generation, computer programs in the form of trees are selected

CIIS'19, November, 2019, Bangkok, Thailand
 978-1-4503-7259-6...$15.00
https://doi.org/10.1145/3372422.3372430

for reproduction using a chosen selection mechanism (e.g. tournament, random or proportional selection [8]) where the *fitness* of each program is taken into account.

Crossover is used to perform the reproduction of *offspring*. To perform crossover, subtrees may be chosen from each parent and substituted with one another to produce two offspring programs for example. To perform mutation, a subtree may be chosen at random and replaced with a randomly generated subtree [11]. All the above-mentioned components have no domain knowledge and simply rely on the Darwinian principle of *survival of the fittest* [10]. A computer program can be represented as a tree (i.e. Abstract Syntax Tree [12]) or graph (i.e. Directed Acyclic Graph [13]).

The advantage of using GP to evolve trees is that we do not need to know the structure of an optimal tree in advance. Therefore, instead of concerning ourselves with task of building good solutions, the focus is on specifying a good fitness function for accepting candidate solutions [14]. Finally, Evolutionary Algorithms (EA) are highly customisable. This is an important feature because it serves as an advantage when designing an EA for the purpose of solving a specific problem [8]. The work presented here aims to use GP to evolve decision trees (DT) - which act as algorithms for the different agents. Each player in the soccer field will, therefore, be controlled by its dedicated decision tree.

The rest of the paper is structured as follows: Section 2 is a literature survey of relevant works. Section 3 describes the problem statement and Section 4 follows with the proposed solution. Section 5 reports on the experimental results and offers a critical analysis of the algorithm. Finally, Section 6 concludes the paper.

2 Literature Review

2.1 Generating Emergent Team Strategies through Genetic Algorithms in Football Video Games

Fernández-Leiva A., Cotta C. and Campaña Ceballos R. [15] proposed the use of a GA for improving the performance of computer-controlled opponents for human players in soccer video games. The aim of the work was to prove that evolutionary algorithms could perform better human-coded AI controllers in video games since it is becoming increasingly difficult to specify condition-action rules for player behaviour as video games continue to imitate real-life situations.

An agent was represented by its chromosome (a vector of k values, where k represents the number of situations that the agent might find itself in. Each index of the vector represented a specific situation, whereas the value in that index represented the action that the agent would perform for that situation). The type of GA used was a Steady State Genetic Algorithm (SSGA) that applied single-point crossover. The selection mechanisms used was binary tournament with an elitist replacement policy.

Experimental results showed that GAs can be used in video games since it was observed that the overall behaviour of the players did improve overtime. Furthermore, it was stated that using GAs in video games may lead to video games where computer-controlled players display emergent and entertaining behaviour. This is because computer-controlled agents would continuously face human opponents, which means that the algorithm would be consistently trained against naturally improving opponents [15].

2.2 A Softbot Team Whose Strategy is Learned through Genetic Programming

In [16] work was done to develop a softbot team that defeated opponents programmed using hand-coded strategies. The softbot team won the RoboCup Scientific Challenge Award. Furthermore, the EA used was able to evolve teams that could position themselves across the field, make passes, shoot at goals and coordinate behaviour to complement teammates. The EA used was GP and, accordingly, chromosomes were represented as trees instead of vectors. A tree represented an entire team, not a single player.

Two approaches were followed to construct a game-playing tree: In the first approach, a tree represented a heterogeneous team where different parts of the tree were assigned to different players. In the second approach, a tree represented a homogenous team instead. The homogenous performed better in the end.

In order to evaluate a team's fitness, the researchers applied a strategy that they referred to as *competitive fitness*. The idea behind competitive fitness was that a team's fitness was measured based on competition with other agents in the population (e.g. through a round-robin tournament where every team in the population plays against every other team).

The author pointed out that the function sets used were very biased. This inhibited the algorithm from reaching more generalised solutions that could maintain their own internal state. As such, the resulting teams consisted of purely reactive players whose decisions mainly relied on the current state of the environment. [16].

2.3 Genetic Programming for Robot Soccer

The work done in [11] presented the use of Strongly Typed Genetic Programming (STGP) in robot soccer. The selection mechanism used was elimination round-style tournament where two programs were randomly chosen to play against each other. The program that won the play-off was advanced to the next round.

A program's fitness was determined by how many rounds it won and the fittest individuals in a generation were played against each other in a best-of tournament. In the first experiment, the programs could only perform primitive soccer actions with the hope that they would evolve complex behaviours from them. The resulting programs did not show any intelligent behaviour; therefore, the winner of a tournament was

determined randomly. In the second experiment, higher level functions that correspond to complex soccer behaviour were introduced, but terminal actions remained the same as in the first experiment. This resulted in the programs showing simple soccer-playing behaviour.

During the third experiment, the same functions as in the first experiment were used including additional functions and terminal actions. Roulette wheel selection was used instead of tournament selection, and fitness was calculated as a weighted sum of the number of goals scored and executed kicks. The resulting teams showed interesting behaviour, although most of it was random [11].

2.4 Layered Learning in Genetic Programming for a Cooperative Robot Soccer Problem

The researchers in [2] used Layered Learning GP (LLGP) for learning keep-away soccer. A learning problem was divided into separate sub-problems that were solved in a bottom-up fashion. Therefore, the solution to the previous sub-problem became the initial population to the current sub-problem.

Four experiments were conducted. The first experiment used standard GP; the second experiment used Automatically Defined Functions GP (ADFGP). The third experiment used (LLGP) where the fittest individual was duplicated to fill the initial population for the next layer (LL1GP). The fourth experiment used LLGP where the entire population was used as the initial population of the next layer (LL2GP). According to the experimental results, LL1GP performed poorly when compared to standard GP and ADFGP, whereas LL2GP performed on par with SGP and ADFGP. Although LL2GP converged in the first 10 generations.

New Layered Learning Genetic Programming (nLL2GP) is a variation of LL2GP that uses 20 generations to train the first layer and 81 generations to train the second layer. nLL2GP performed slightly better than all the previous algorithms [2].

3 Problem Statement

The work presented here aims to use GP for team evolution through DTs. The secondary purpose is to discover how well a decision tree can perform as decision-making model for an agent performing a task in a real-time environment. The idea is that, even though agents may be initialized with purely random DTs, some of the decisions made by the agents will show some measure of intelligence.

Therefore, if GP is applied carefully to choose DTs that make favourable decisions - even though they were made randomly - new DTs can be derived that inherit the desired behaviour and further display interesting and complex behaviour due to random mutation.

Taking a closer look at the work presented in [15]: in order to add a new condition to the vector of possible situations as described in [15], one would have to redesign the vector. Therefore, it would be inconvenient to incorporate new behaviours. Our approach introduces more flexibility. Using a tree allows one to only add conditions to the existing pool of possible conditions without having to worry about restructuring the tree.

The work presented in [16] makes use of a single tree that represents an entire team. This means that when the tree is flawed, there is a high probability that the whole team will perform poorly. We propose that every agent is controlled by its own DT as this will lead to a more distributed structure without a single point of failure.

Finally, our approach might show resemblance to that of a Random Forest (RF) classifier albeit it is different. RF is an ensemble learning method where many classifiers (i.e. DTs) are generated - using a concept referred to as *bootstrap aggregation* - and whose predictions/classifications are aggregated to make a single classification [17]. In contrast to RF, our approach is based on the fact that each agent is controlled by one DT that makes decisions on behalf of only that agent. The decisions of other DTs are not aggregated to reach a decision for one agent. At the end of each generation, trees of desirable fitness are used to evolve tree which will also hopefully possess desirable fitness.

4 Proposed Solution

4.1 StratFinder Prototype

StratFinder (short for Strategy Finder) is the solution prototype that was developed for this work. The application consists of a simulation window that renders instances of soccer matches. Each soccer match is essentially a non-player game that consists of two teams meant to be in opposition.

Each team consists of 10 field players and there are no goalkeepers. Goalkeepers are not included because the aim is to evolve in-field game play without focusing on the added problem of evolving goalkeeping skills. However, goalkeepers could theoretically be added so that they behave according to predefined condition-action rules. Doing so would allow us to further examine how players would evolve if the game had goalkeepers. Please refer to Figure 1 for a screenshot of the simulation user interface.

Figure 1: StratFinder's simulation user interface.

The team in yellow must score goals at the right-hand side goal and defend the left-hand side goal. The team in black must score at the left-hand side goal and defend the right-hand side goal.

In order to win, a team must have scored more goals than the opposition by the end of the match. If the ball leaves the field, it goes back to the centre of the field and any player from either of the teams can kick it first when the simulation continues. Therefore, the throw-in and corner-kick rules of soccer do not apply. The offside rule also does not apply. These special conditions are already a part of the simulation.

4.2 Decision Tree Implementation

The DT used to control each player is a perfect binary tree. All internal nodes of a tree are function nodes (they represent conditions that the player might be in) that return a value of true or false. When a function node returns a value of *true* for a condition, the left child is evaluated; otherwise, the right child is evaluated. At the beginning of the first generation, all trees have the same depth. No evolutionary operator (whether it be crossover or mutation) can alter the structure of a tree. Please refer to Figure 2 for a depiction of a sample DT.

4.2.1 Function Nodes. There are 20 different conditions that a player can find itself in; this results in 20 different function nodes that initially have equal probability of existing anywhere in a DT except as terminal nodes (i.e. if a DT is 6 nodes deep, it will have 64 function nodes). Therefore, all conditions initially have a uniform probability of existing in any DT. Moreover, mutating the DTs will also ensure that the algorithm tests as many condition sequences as possible. Table 1 lists the possible Functions that can exist as internal nodes in a DT.

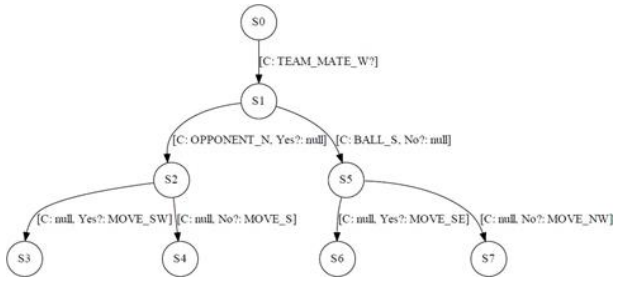

Figure 2: An example DT.

Table 1: A list of function nodes that can exist in a DT

Function Type	Function Name
Teammates	TEAMMATE_N
	TEAMMATE_S
	TEAMMATE_E
	TEAMMATE_W
Opponents	OPPONENT_N
	OPPONENT_S
	OPPONENT_W
	OPPONENT_E
Ball	BALL_N
	BALL_S
	BALL_W
	BALL_E
Touchline	TOUCHLINE_N
	TOUCHLINE_S
	TOUCHLINE_W
	TOUCHLINE_E
Goals	OWN_GOALS_CLOSE
	OPP_GOALS_CLOSE
Possession	IN_POSSESSION
	TEAMMATE_IN_POSSESSION
	OPP_IN_POSSESSION

4.2.1 Action Nodes. An action node represents the action that an agent will perform after having evaluated all preceding conditions. There are 8 different possible actions that an agent can perform, and they all have an equal probability of existing in any DT. Furthermore, an action node can only exist as a terminal (leaf) node. A player can move in 8 different directions. When a player is in possession of the ball, the player automatically kicks the ball in the direction it is moving in. Please refer to Table 2 for a list of the possible action nodes.

Table 2: A list of action nodes that can exist in a DT

Action Type	Action Name
Move	MOVE_N
	MOVE_S
	MOVE_W
	MOVE_E
	MOVE_NW
	MOVE_NE
	MOVE_SW
	MOVE_SE

4.3 Algorithm Implementation

There are three primary reasons for using GP for this problem: the first reason generally applies to all EAs. EAs can be used to solve optimisation problems for which there is no exact known solution/optimum. For example, there is no exact way in which players must behave in a soccer field in order to win a match. Different teams deploy different strategies to succeed. Therefore, the problem of evolving soccer strategies does not have a fixed or known solution.

The second reason is that GP allows for the use of trees which easily serve as natural visual representations of solutions: because of a tree's structure, it is not difficult for humans to see how the algorithm arrived at a certain decision (terminal node).

The third reason is that the worst-case computation time to traverse a binary tree is linear and occurs when the tree has taken the form of a list. Since the trees used in this algorithm do not change shapes, it is not possible for any of the trees to degenerate into lists. Therefore, the computation time to

evaluate the trees will always be logarithmic to the number of elements. Since a crossover operation on trees is an expensive approach [18] - because it involves creating copies of the desired subtrees in order to make offspring trees - ensuring a cheap evaluation of trees helps in making our approach viable.

EAs have the property that they can be used to solve a problem to which a fixed solution is not known but can instead be approximated. The approximation can be expressed in the fitness function of the algorithm by penalizing models that do not exhibit desirable behaviour and rewarding those that do, and then using selected models to reproduce similar models with a certain amount of diversity.

4.3.1 Chromosome Representation. Each DT represents a potential solution to the problem. A chromosome that represents good fitness is a DT that guides the agent towards optimum behaviour in the playing field. Therefore, a good solution is a DT that represents a sequence of steps that help the agent to contribute positively towards the team's chance of winning a match.

4.3.2 Fitness Evaluation. The fitness of each DT is evaluated at the end of each game or soccer match. Each DT's fitness is determined according to a performance score that is solely associated to it. The score of a DT is affected by a set of situations that it finds itself in (See Table 3).

Table 3: A depiction of rewards and penalties associated with being in each situation

Situation	Reward	Penalty
Moves outside of the field	0	50
Involved in a collision	0	50
In possession of the ball	1000	0
Scores a goal	10000	0
Teammate scores a goal	5000	0
Concedes a goal	0	5000

The reason fitness is calculated in this way is because we assume that either the model does not have decisions that can help it escape difficult situations, or it makes decisions that cause it to be in unfavourable situations. As shown in Equation 1, the fitness function can therefore be defined as the difference between rewards and penalties that an agent would have acquired by the end of one generation:

$$f(t) = R_{T,t} - P_{T,t} \quad (1)$$

where t is the generation counter, R_T and P_T are the total rewards and penalties for one generation, respectively.

4.3.3 Selection Strategy. At the end of each generation, the algorithm searches for candidates that fit the criteria for taking part in reproducing offspring for the next generation. The selection method used is tournament selection, and it is applied as follows for each team: at the end of the generation, 7 players are picked randomly. From the 7 players, 5 fittest players are then selected for reproduction. This strategy allows for some of the "worst" players to also have a chance of going to the next generation, although, the odds still favour the fittest players. Therefore, the algorithm leans towards elitism.

The algorithm has a considerable generation overlap since, at each iteration step (generation), only half of the population is replaced with new individuals. The 10 fittest members (5 from each team) are selected to reproduce 10 new offspring. This was done so that the performance of the players changes steadily over time. The algorithm uses two reproduction operators: crossover and mutation. Each team is evolved separately.

4.3.4 Reproduction. The crossover operator is used to produce offspring DTs from DTs selected in the previous step. The list of parent DTs is treated as a circular array. To perform crossover, two parent DTs that are next to each other in the list of parent DTs are selected to produce one offspring DT. Please refer to Figure 3 for an illustration of how two parent DTs are combined to produce an offspring DT.

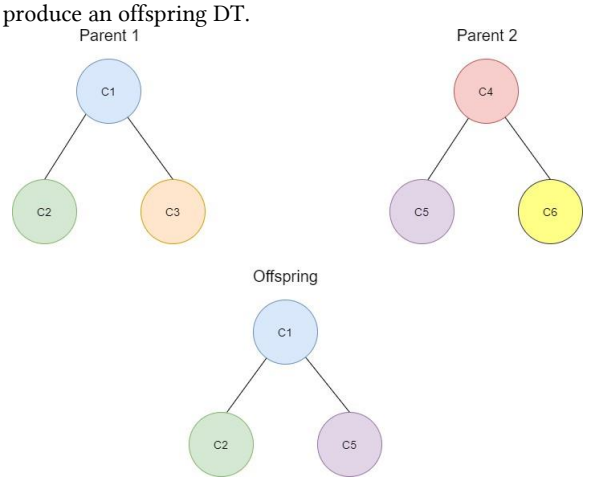

Figure 3: An illustration of how the crossover operation is carried out. The root of Parent 1 (C1) becomes the root of the offspring. The left child (C2) of Parent 1 becomes the offspring's left child. The left child (C5) of Parent 2 becomes the right child of the offspring.

The root node of the offspring DT is the root node of the first parent DT. The subtree extending from the left child of the first parent DT also becomes the left subtree of the offspring DT's root. The subtree that extends from the left child of the second parent DT's root becomes the right subtree of the offspring DT's root.

The mutation rate changes overtime with the change in the fitness landscape. At the start of the first generation, the mutation rate is 10%. At the end of each generation, the fitness of each selected DT is compared with the current overall best fitness. If the fitness of the DT in question is higher than the overall best fitness, the DT's fitness becomes the overall best fitness and the mutation rate is increased by 1%.

The reason for increasing the mutation rate is that the algorithm is still in the exploration phase. If the fitness of the DT in question is lower than or equal to the overall best fitness, the overall best fitness is not changed. However, the mutation rate is decreased by 1%. The reason for decreasing the mutation rate in this situation is because the algorithm may be converging towards a solution, therefore, exploitation should be favoured over exploration.

In order to perform mutation of a DT, Algorithm 1 is followed. Using Algorithm 1, a tree might be mutated as in Figure 4, for example. In Figure 4, node C2 and C5 were mutated as the algorithm traversed the tree in a random fashion.

Algorithm 1 Mutation Algorithm Pseudocode

```
Function mutate(tree)
    cursor ← tree.root;
    random ← random integer between 0 and 1; //Dice throw.
    if (random == 1)
        walk(cursor.leftChild);
    end if
    else
        walk(root.rightChild);
    end if
end function

Function walk(cursor)
    random ← random integer between 0 and 1;
    if (cursor is a leaf node)
        if (random == 1)
          //Mutate with random action.
            cursor ← a random action;
        end if
        return
    end if
    if (random == 1)
        //Mutate with random function.
        cursor ← a random function;
    end if
    if (cursor.leftChild != null & cursor.rightChild != null)
        random ← a random integer between 0 and 1;
        if (random == 1)
            walk(cursor.leftChild)
        end if
        else
            walk(cursor.rightChild)
        endif
    end if
end function
```

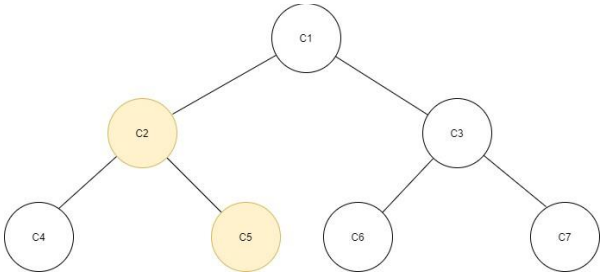

Figure 4: An example of mutation being performed on a DT. Each node is mutated with a uniform probability as the algorithm walks down the tree in a random fashion.

5 Experimental Results

The algorithm was tested for different tree depths: 4, 6, 8 and 10, and the experiments are titled Experiment 1, 2, 3 and 4 respectively. For tree depths 4 and 6, the evolutionary process was run for 1000 generations. For depths 8 and 10, the evolutionary process was run for 2000 generations. The aim was to discover any interesting emergent behaviour that would arise if the tree depth was increased.

5.1 Experiment 1

5.1.1 Performance of the Algorithm. For this experiment all the players in the field were initialised with DTs of depth size 4. Since, by the rules of the algorithm, all DTs are complete binary trees, all the DTs consisted of 31 nodes. Figure 5 and 6 show the resulting fitness curves for the black and yellow teams respectively, for this experiment.

After 1000 generations, the players did show some coordination and were able to recognise the location of the ball in the field, and even move towards the ball. However, if a player obtained possession of the ball, the player would consistently kick the ball in a single direction until it went out of the field.

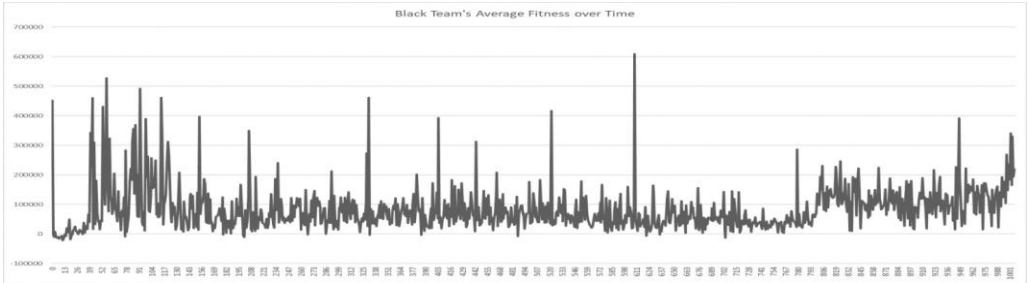

Figure 5: A curve showing the average fitness of the black team for each generation across 1000 generations for trees that are 4 nodes deep.

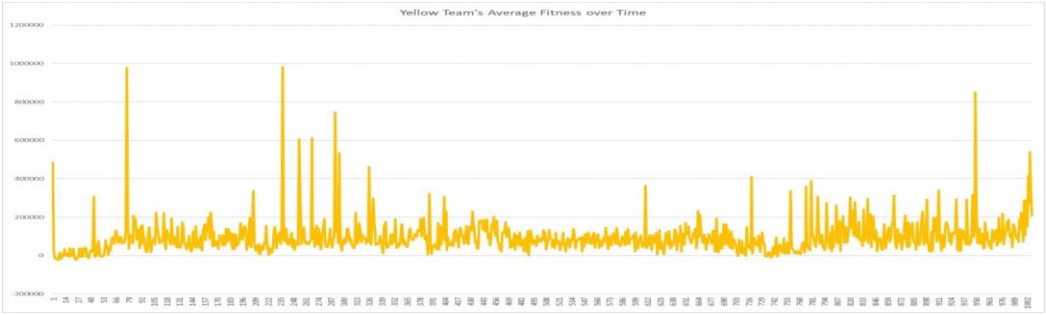

Figure 6: A curve showing the average fitness of the yellow team for each generation across 1000 generations for trees that are 4 nodes deep.

Each time the ball went out of the field, the players could not change the directions that they were moving in to follow the ball once again. This shows that the players had learnt to follow the ball at the start of the game, but they could not keep that up as the game continued.

As can be seen in Figure 5, the black team's average fitness quickly ascended and stopped going up at generation 65. In fact, there was a heavy drop of the team's average fitness in generation

78. The average fitness for the team finally plateaued from generation 117 onwards, although it followed an increasing trend from generation 800 going further. This resembles a pattern that is also observable in nature and is referred to as *punctuated equilibria* [19].

Another interesting observation is that the fitness of the yellow team also followed a similar trend although the fitness curve for the yellow team had less fluctuations. This means that the algorithm was able to exploit good regions in the search space for the yellow team.

5.1.2 Running Time of the Algorithm for each Evolution Sequence. Figure 7 shows the GP algorithm's running time for each evolution sequence for Experiment 1. An evolution sequence begins at the end of each generation. The following steps are performed: (1) chromosome selection, (2) crossover, (3) mutation and (4) returning of the new set players to the game state. The average running time of the algorithm was approximately 0.08 milliseconds.

5.2 Experiment 2

5.2.1 Performance of the Algorithm. For this experiment, the players were initialised with decision trees of depth 6 (127 nodes). Figure 8 and Figure 9 show the average fitness curves of the black and yellow teams respectively. Some interesting behaviour was observed for this experiment. Not only did the players chase the ball, but they had a sense of where their opponent's goals were. Some degree of competitiveness was observed. However, most of the players would collide with each other until the end of the match, leaving only a few players running around the field. Several players also went to the touchline and stayed there for the whole match.

As can be seen in Figure 8 and Figure 9, the two teams had overall higher fitness than when the DTs were 4 nodes deep. The fitness curve for the black team shows that the average fitness of the team increased at significant rate from generation 0 to generation 78, and then adopted a steady rise. This continued until generation 442, where the average fitness of the team again increased at a significant rate until it remained constant from generation 468 to generation 728.

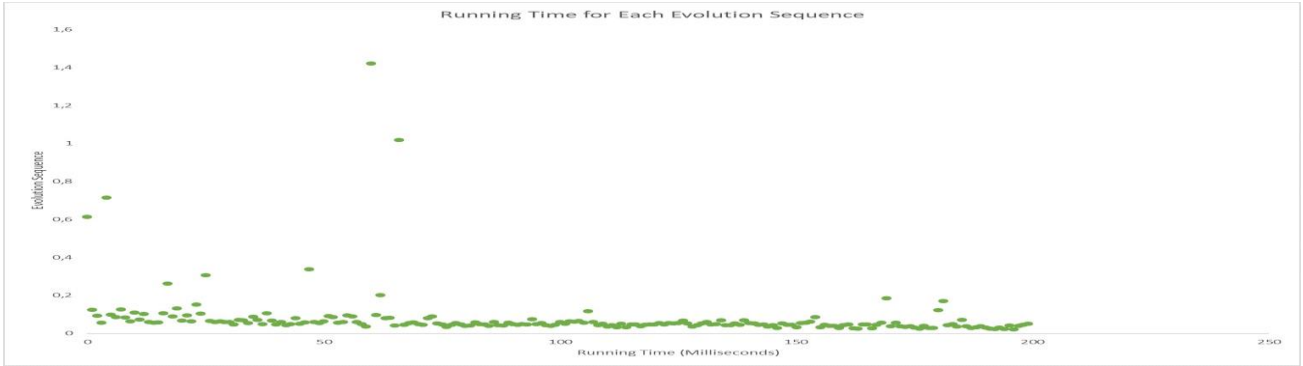

Figure 7: A scatter plot showing the running times (y-axis) for each evolution sequence (x-axis) when the trees were 4 nodes deep. The running times were recorded for the first 200 generations.

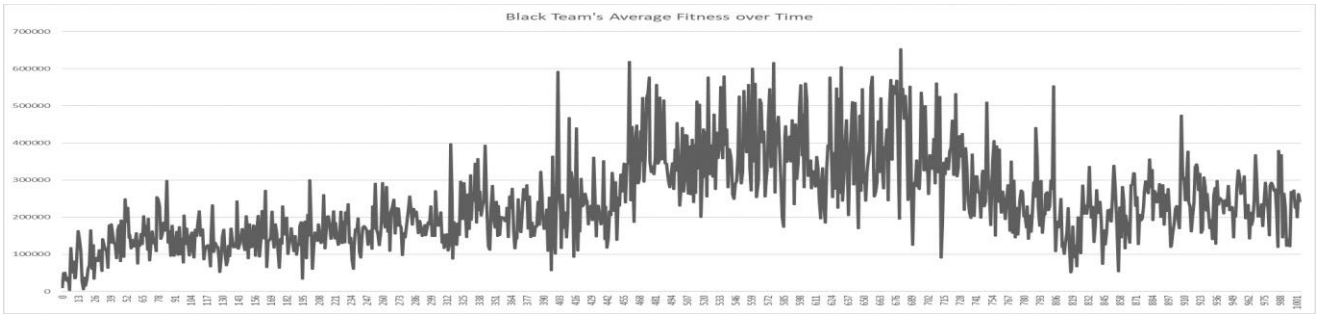

Figure 8: A curve showing the average fitness of the black team for each generation across 1000 generations for trees that are 6 nodes deep.

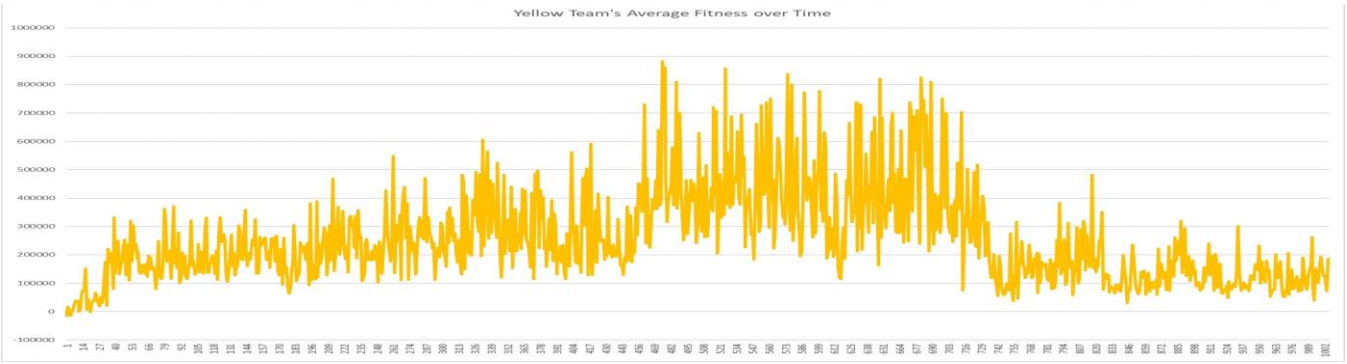

Figure 9: A curve showing the average fitness of the yellow team for each generation across 1000 generations for trees that are 6 nodes deep.

After generation 728, the average fitness of the team started dropping slowly until it maintained a it was constant again at generation 819. The same trend can be seen for the yellow team, although the changes took place at slightly different generations.

From an overall perspective, it was observed that increasing the depth of the DTs to depth 6 improved the performance of the players. This result can be justified by the fact that the players had more condition sequences to evaluate. Therefore, the players had the opportunity to make more complex decisions.

Furthermore, with a branching factor of 2, the resulting decision nodes for each DT in this experiment was 64. This provided the players with the opportunity to explore different decisions for different condition sequences than when they had only 16 different decisions to make in the previous experiment.

5.2.2 Running Time of the Algorithm for each Evolution Sequence. Please refer to Figure 10 for the average running time of the GP algorithm when evolving trees that are 6 nodes deep. On average, the algorithm took 0.01ms longer than when the DTs were 4 nodes deep. The difference in the number of nodes between the two tree sizes was 96.

5.3 Experiment 3

5.3.1 Performance of the Algorithm. For this experiment, each player's DT was given a depth of 8 nodes. Figure 11 and Figure 12 show the average fitness curves of the black and yellow teams respectively. From the 200th generation, players started moving towards the ball, although most players moved to the touchlines.

After 200 generations, the players always had a sense of where the ball was. Even if the ball went out of the playing area, the players could redirect to the ball, which would now be at the kick-off spot. Moreover, the competitiveness between the two teams could be observed in some instances. For example, if a player from the yellow team was in possession of the ball, then the player's teammates would try to stop any players from the team in black from moving close to the player in possession.

However, after 1500 generations, no improvement in gameplay could be observed. Instead, most of the players began to hardly move in some instances. Each player would rapidly move in opposite directions as if it was vibrating.

Although there were no improvements observed in terms of game play, the average fitness scores of the teams did not drop. This means that the algorithm converged to a local maximum. The semi-vibrational movement patterns of the players were the players' way of not colliding with each other and not moving outside of the field while being very close to the ball. Therefore, by the fitness function, they were satisfying all the requirements for not getting penalised.

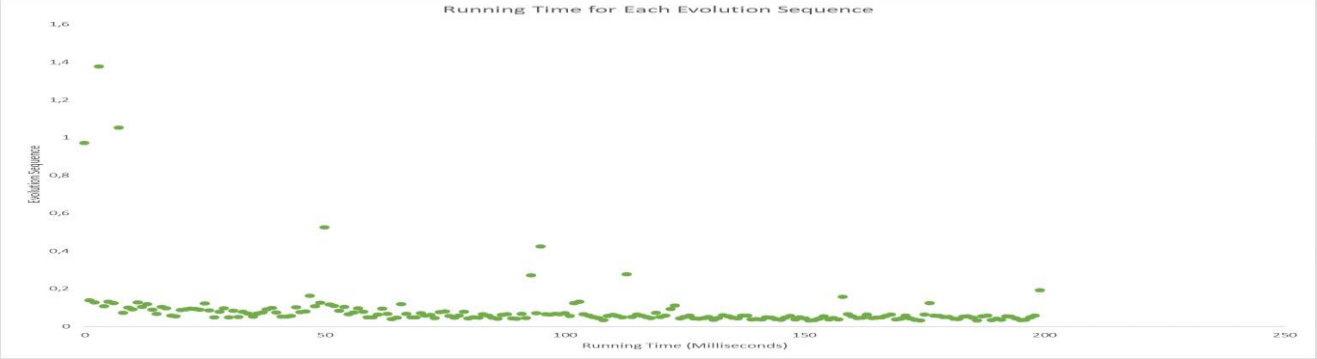

Figure 10: A scatter plot showing the running times (y-axis) for each evolution sequence (x-axis) when the trees were 6 nodes deep. The running times were recorded for the first 200 generations.

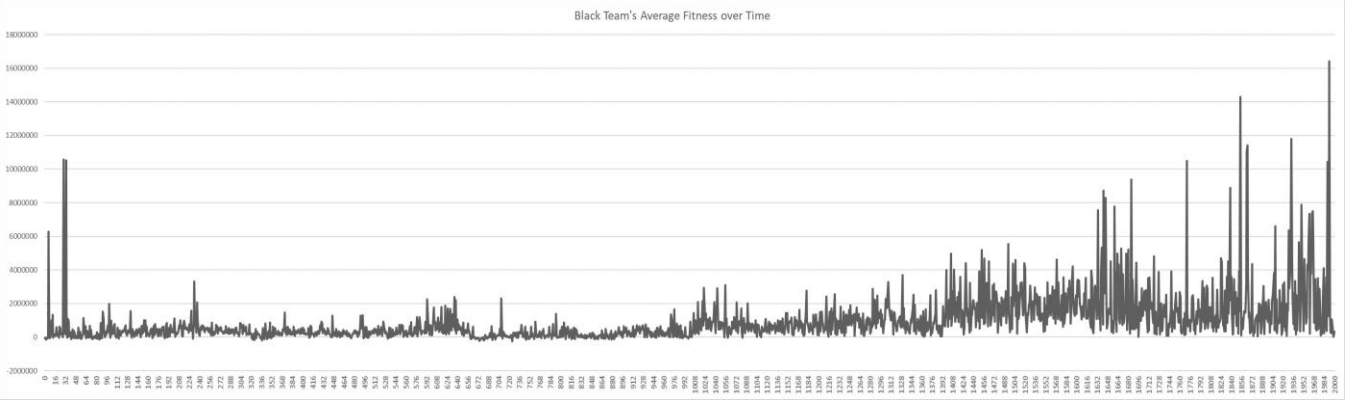

Figure 11: A curve showing the average fitness of the black team for each generation across 2000 generations for trees that are 8 nodes deep.

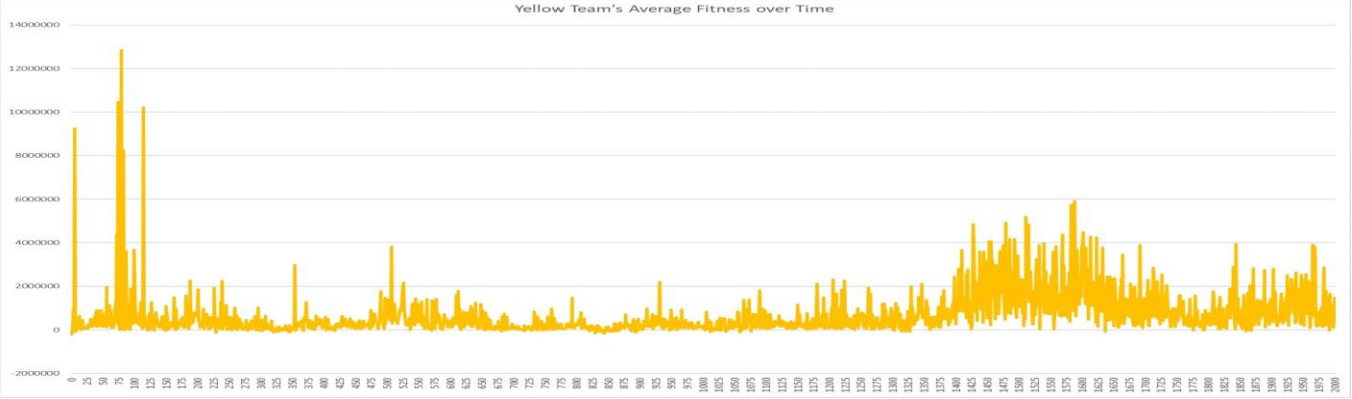

Figure 12: A curve showing the average fitness of the yellow team for each generation across 2000 generations for trees that are 8 nodes deep.

This led to the addition of a new penalty that was added in order to avoid this behaviour. Each player was given a time window. If the time window elapsed without the player having moved at least 2 pixels away from its current position, the player was penalised. As can be seen in Figure 11 and Figure 12, the average fitness over time for the two teams was constant from an overall perspective, until it started fluctuating heavily from 1400 onwards. However, from

both charts, it can be observed that the teams reached very high fitness levels for some generations.

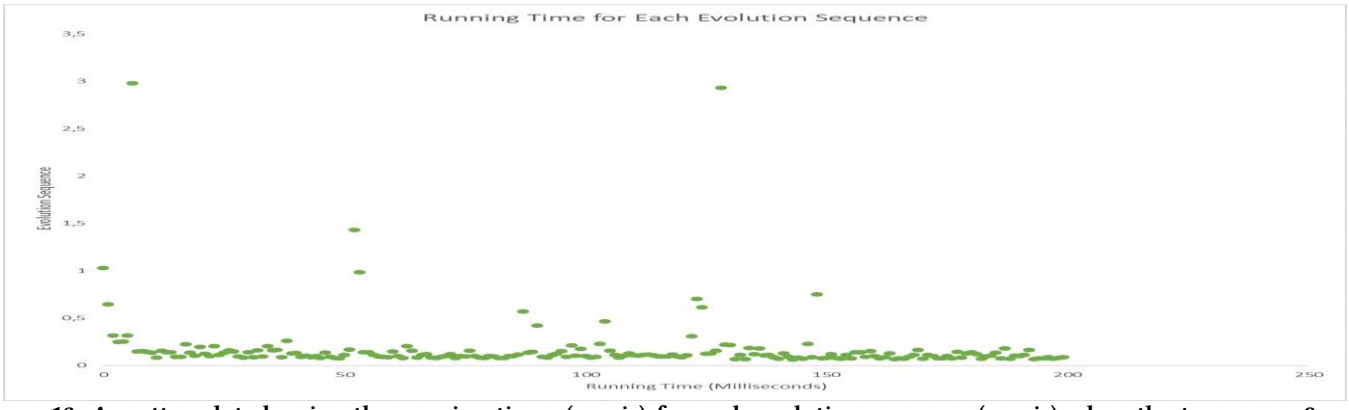

Figure 13: A scatter plot showing the running times (y-axis) for each evolution sequence (x-axis) when the trees were 8 nodes deep. The running times were recorded for the first 200 generations.

Between generations 75 and 100, the yellow team reached a team average fitness higher than 1200000. Between generations 1984 and 2000, the black team reached an average team fitness higher than 1600000. However, the algorithm could not exploit these good regions in the search space.

5.3.2 Running Time of the Algorithm for each Evolution Sequence. Figure 13 shows the average running time of each evolution sequence performed by the algorithm when the DTs were 8 nodes deep. The average running time of each evolution sequence in this experiment was 0.18 milliseconds. This is twice the average running time for Experiment 2, which was 0.09.

By increasing the depth of the tree by two, the average running time increased by 0.09 milliseconds. This is a significant increase, although justifiable considering that the DTs now had 384 more nodes.

5.4 Experiment 4

5.4.1 Performance of the Algorithm. For this final experiment, each DT was 10 nodes deep. Figure 14 and Figure 15 show the average fitness curves of the black and yellow teams respectively.

The resulting behaviour of the players was more sophisticated than in previous experiments. The players showed coordination and changed direction in order to chase the ball. Some emergent behaviour could be observed in some cases where a group of players would position themselves in a line, as if to form a defensive wall. Some players also showed coordinated movement with respect to each other. However, players struggled to score goals. This might be due to the competitiveness that the two teams had developed.

In previous experiments, the players could score goals easily, but in most cases, it was a repeated sequence of actions. Moreover, some players would repeatedly score own goals in one match. Therefore, this shows that it was not planned behaviour.

The time window constraint introduced in Experiment 3 improved the behaviour of the players on the pitch. However, the collisions between players was now a limiting factor. Two players would collide for the whole match without changing directions.

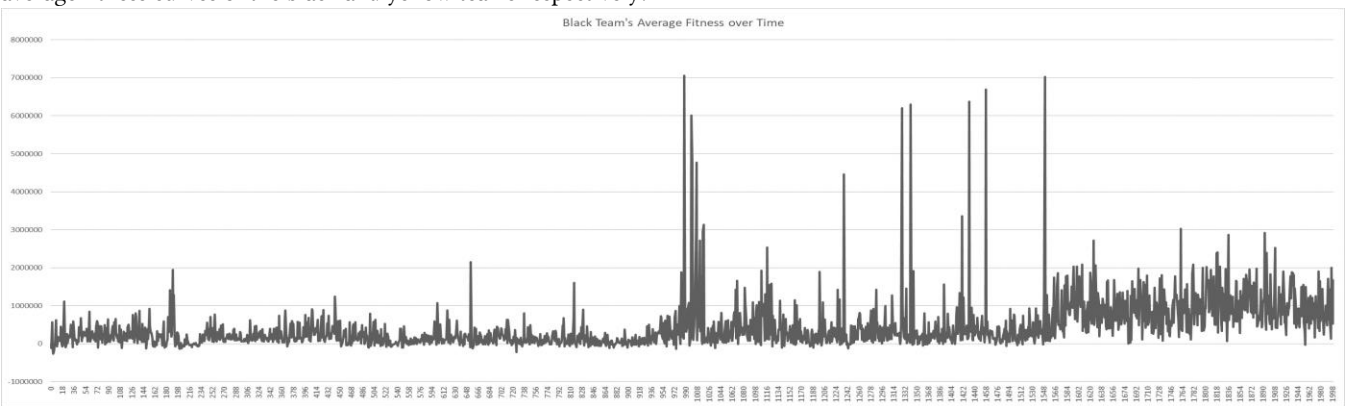

Figure 14: A curve showing the average fitness of the black team for each generation across 2000 generations for trees that are 10 nodes deep.

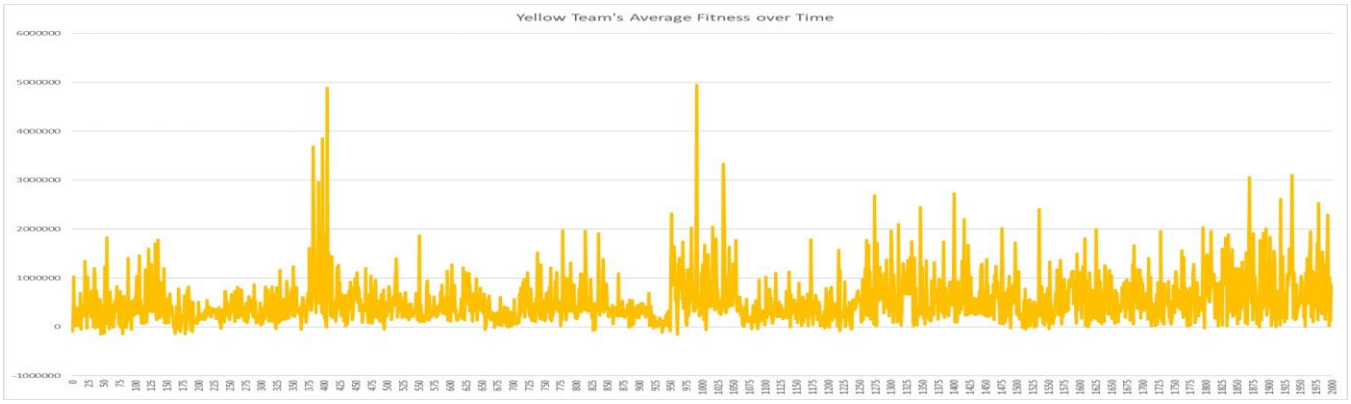

Figure 15: A curve showing the average fitness of the yellow team for each generation across 2000 generations for trees that are 10 nodes deep.

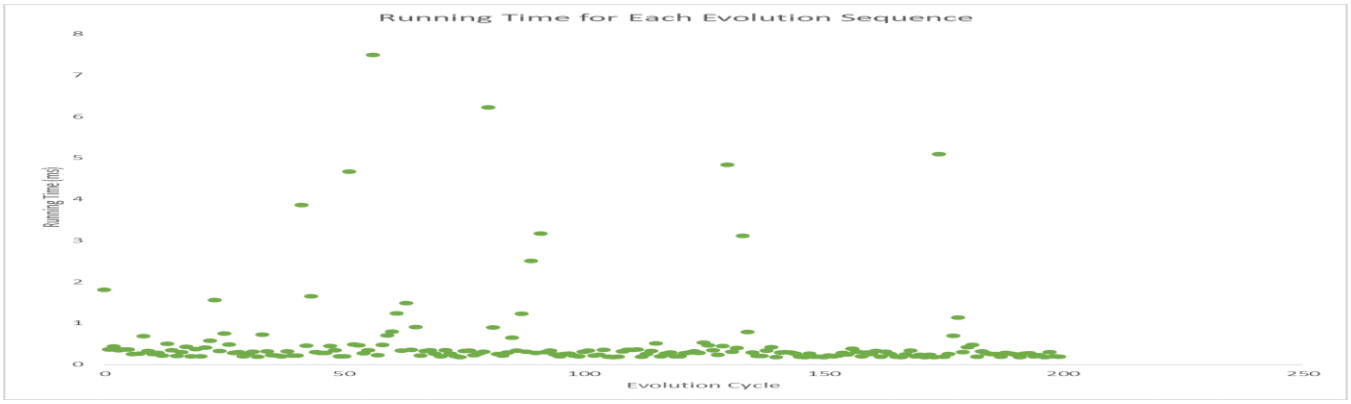

Figure 16: A scatter plot showing the running times (y-axis) for each evolution sequence (x-axis) when the trees were 10 nodes deep. The running times were recorded for the first 200 generations.

5.4.2 Running Time of the Algorithm for each Evolution Sequence. Figure 17 shows the average running time for each evolution sequence.

The average running time is 0.54. This is 0.36 milliseconds longer than in experiment 3. This is most certainly because the tree sizes were now more than two times bigger than in the previous experiment. The number of nodes had increased by 1536.

5 Conclusion

The work presented here has shown that it is possible to evolve emergent soccer strategies through GP. Four experiments were performed where, for each experiment, the DT depths were increased by 2. It was observed that the behaviour of the players on the pitch increased in complexity with the size of the trees. Although, some formations were observed when the trees were 10 nodes deep, they did not represent complex strategies. Table 4 compares the four approaches followed in the four experiments with regards to RTES (Running Time of each Evolution Sequence).

Although the average RTES was the highest for Experiment 4 (0.56 milliseconds), we still recommend the approach taken in Experiment 4 to perform evolution. This is because 0.56 milliseconds is not a significantly long time for one evolution sequence to take place in reality. Therefore, the approach in Experiment 4 is still feasible because the behaviour of the agents was much more intelligent than in the other experiments.

Table 4: A comparison of the four experiments.

Tree Depth	Generations	Average RTES
4	1000	0.08
6	1000	0.09
8	2000	0.18
10	2000	0.54

Further work needs to be done with regards to evolving DTs to a point where agents display emergent strategies. One of the limitations of evolving DTs without changing their shape is that

$O(logn)$ (where n is the number of nodes in the tree) conditions must always be evaluated before any decision is made. For some decisions to be made, only a few conditions need to be true. For example, say a player is in possession of the ball and in front of the opponent's goal. The only decision to make, after evaluating the respective conditions (IN_POSSESSION & OPP_GOALS_CLOSE) as being true, is to kick the ball into the net.

A limitation of implementing DTs as complete binary trees is that the size of a DT grows exponentially with its depth: with a branching factor of 2, the number of nodes at depth d become 2^d. Future work will focus on using DTs of variable shape and size in order to examine how the teams would evolve if all the DTs were not required to have the same shape and size.

REFERENCES

[1] Arni Arnason, Stefan B Sigurdsson, Arni Gudmundsson, Ingar Holme, Lars Engebretsen, and Roald Bahr. 2004. Physical fitness, injuries, and team performance in soccer. *Medicine & Science in Sports & Exercise* 36, 2 (2004), 278–285.

[2] Steven M Gustafson and William H Hsu. 2001. Layered learning in genetic programming for a cooperative robot soccer problem. In *European Conference on Genetic Programming*. Springer, 291–301.

[3] Peter Stone and Manuela Veloso. 2000. Layered learning. In *European Conference on Machine Learning*. Springer, 369–381.

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

[5] J-H Kim, H-S Shim, H-S Kim, M-J Jung, I-H Choi, and J-O Kim. 1997. A cooperative multi-agent system and its real time application to robot soccer. In *Proceedings of International Conference on Robotics and Automation*, Vol. 1. IEEE, 638–643.

[6] Franciszek Seredynski. 1997. Competitive coevolutionary multi-agent systems: The application to mapping and scheduling problems. *J. Parallel and Distrib. Comput.* 47, 1 (1997), 39–57.

[7] Sean Luke et al. 1998. Genetic programming produced competitive soccer softbot teams for robocup97. *Genetic Programming* 1998 (1998), 214–222.

[8] Andries P Engelbrecht. 2007. Computational intelligence: an introduction. John Wiley & Sons.

[9] Darrell Whitley. 1994. A genetic algorithm tutorial. *Statistics and computing* 4, 2 (1994), 65–85.

[10] John R Koza. 1997. Genetic programming. (1997).

[11] Vic Ciesielski, Dylan Mawhinney, and Peter Wilson. 2001. Genetic programming for robot soccer. In *Robot Soccer World Cup*. Springer, 319–324.

[12] Michel Chilowicz, Etienne Duris, and Gilles Roussel. 2009. Syntax tree fingerprinting for source code similarity detection. In *2009 IEEE 17th International Conference on Program Comprehension*. IEEE, 243–247.

[13] Julian F. Miller. 2011. *Cartesian Genetic Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 17–34. https://doi.org/10.1007/978-3-642-17310-3_2

[14] Sichun Wang. 2009. Solving the Optimal Solution of Weight Vectors on GP-Decision Tree. In *2009 Second International Conference on Intelligent Computation Technology and Automation*, Vol. 4. IEEE, 329–332.

[15] Antonio J Fernández, Carlos Cotta, and Rafael Campaña Ceballos. 2008. Generating Emergent Team Strategies in Football Simulation Videogames via Genetic Algorithms. In *GAMEON*. Citeseer, 120–128.

[16] Sean Luke. 1998. Evolving soccerbots: A retrospective. In Proceedings of the 12th Annual Conference of the Japanese Society for Artificial Intelligence. Citeseer.

[17] Andy Liaw, Matthew Wiener, et al. 2002. Classification and regression by randomForest. *R news* 2, 3 (2002), 18–22.

[18] Mihai Oltean. 2004. Solving even-parity problems using traceless genetic programming. In *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753)*, Vol. 2. IEEE, 1813–1819.

[19] Michael D Vose and Gunar E Liepins. 1991. Punctuated equilibria in genetic search. *Complex systems* 5, 1 (1991), 31–44.