Learning Strategies for Evolved Co-operating Multi-Agent Teams in Pursuit Domain

Gina Grossi

Submitted in partial fulfilment
of the requirements for the degree of

Master of Science

Department of Computer Science
Brock University
St. Catharines, Ontario

©*Gina Grossi,* 2017

Abstract

This study investigates how genetic programming (GP) can be effectively used in a multi-agent system to allow agents to learn to communicate. Using the predator-prey scenario and a co-operative learning strategy, communication protocols are compared as multiple predator agents learn the meaning of commands in order to achieve their common goal of first finding, and then tracking prey. This work is divided into three parts. The first part uses a simple GP language in the Pursuit Domain Development Kit (PDP) to investigate several communication protocols, and compares the predators' ability to find and track prey when the prey moves both linearly and randomly. The second part, again in the PDP environment, enhances the GP language and fitness measure in search of a better solution for when the prey moves randomly. The third part uses the Ms. Pac-Man Development Toolkit to test how the enhanced GP language performs in a game environment. The outcome of each part of this study reveals emergent behaviours in different forms of message sending patterns. The results from Part 1 reveal a general synchronization behaviour emerging from simple message passing among agents. Additionally, the results show a learned behaviour in the best result which resembles the behaviour of guards and reinforcements found in popular stealth video games. The outcomes from Part 2 reveal an emergent message sending pattern such that one agent is designated as the "sending" agent and the remaining agents are designated as "receiving" agents. Evolved agents in the Ms. Pac-Man simulator show an emergent sending pattern in which there is one agent that sends messages when it is in view of the prey. In addition, it is shown that evolved agents in both Part 2 and Part 3 are able to learn a language. For example, "sending" agents are able to make decisions about when and what type of command to send and "receiving" agents are able to associate the intended meaning to commands.

Acknowledgements

I would like to thank my supervisor, Dr. Brian Ross, for his guidance and insight throughout our work on this research. I would also like to thank my external examiner, Dr. Joseph Brown and my thesis committee members, Dr. Beatrice Ombuki-Berman, and Dr. Michael Winters, for their valuable feedback. Finally, I would like to thank every member of my family, and close friends for their endless support and encouragement during the entire time of my graduate studies. This accomplishment would not be have been possible without you.

Contents

1 Introduction **1**

1.1 Overview of Project Goal and Results 2

2 Background **4**

2.1 Genetic Programming 4

2.1.1 Strongly Typed GP 6

2.2 Multi-Agent Evolution 8

3 System Design **14**

3.1 System Overview 14

3.2 GP Tree Structure 14

3.3 Message Buffer System 17

3.3.1 Integration of GP Tree and Message Buffer System 19

3.4 Pursuit Domain 19

3.4.1 Integration of Pursuit Domain with GP 20

3.5 Ms. Pac-Man SDK 22

3.5.1 Integration of Ms. Pac-Man SDK with GP 24

4 Evolved Communication Protocols **27**

4.1 Problem and Environment 27

4.1.1 Learning Strategy 28

4.1.2 Communication Strategy and Communication Channel 29

4.1.3 Communication Protocols 30

4.2 Experiment Details 30

4.2.1 GP Language 31

4.2.2 Training and Testing Methods 34

4.2.3 Fitness Function 34

4.3 Results 35

4.3.1 Statistical Analysis 37
4.3.2 Emergent Behaviour 40
4.3.3 Summary of Results 57

5 Learning the Meaning of Commands **59**
5.1 Problem and Environment 59
5.1.1 Learning Strategy 60
5.1.2 Communication Strategy and Communication Channel 60
5.1.3 Communication Protocols 61
5.2 Experiment Details 63
5.2.1 GP Language 63
5.2.2 Training and Testing Methods 66
5.2.3 Fitness Function 66
5.3 Results 68
5.3.1 Send22 Protocol 68
5.3.2 SendAll Protocol 69
5.3.3 Statistical Analysis 69
5.3.4 Emergent Sending Patterns 72
5.3.5 Learning the Meaning of Commands 76
5.3.6 Summary of Results 91
5.4 Influence of Prey Movement Type in Training and Testing on Send22 Protocol 92
5.4.1 Training and Testing Types and Methods 93
5.4.2 Discussion of Results 95
5.4.3 Summary of Results 102

6 Agent Evolution in Ms. Pac-Man Environment **105**
6.1 Problem and Environment 105
6.1.1 Learning Strategy 107
6.1.2 Communication Strategy and Communication Channel 107
6.1.3 Communication Protocols 108
6.2 Experiment Details 109
6.2.1 GP Language 109
6.2.2 Training and Testing Methods 111
6.2.3 Fitness Function 113
6.3 Results 114
6.3.1 Statistical Analysis 114

6.3.2 Qualitative Analysis 116
6.3.3 Summary of Results 123

7 Conclusion **125**
7.1 Summary of Main Results 125
7.2 Future Work 127

Bibliography **133**

Appendices **134**

A Additional Experimental Analysis **134**
A.1 ANOVA Hypothesis 134
A.2 Evolved Communication Protocols: ANOVA Results 134
A.3 Learning the Meaning of Commands Part A: ANOVA Results 135
A.4 Learning the Meaning of Commands Part B: ANOVA results 135
A.5 Agent Evolution in Ms. Pac-Man Environment: ANOVA Results . . . 137

List of Tables

2.1 Pseudocode for Evolution in a GP System 6
2.2 Multi-Agent Learning Strategies [1] 9
2.3 Multi-Agent Communication Strategies [1] 9

3.1 Example of Strongly Type Language 15
3.2 Pseudocode for Root Node Evaluation 20

4.1 Communication Protocols 31
4.2 GP Parameters 32
4.3 Strongly Type Language 32
4.4 Terminal Set 33
4.5 Function Set 33
4.6 Test Fitness Summary Linear Prey (20 runs). 36
4.7 Test Fitness Summary Random Prey (20 runs). 39
4.8 Tukey Comparisons for Communication Protocols (Linear Prey) . . . 40
4.9 Tukey Comparisons for Communication Protocols (Random Prey) . 40
4.10 Message Sending Patterns (Prey Linear Movement) 41
4.11 SendAll Staircase Pattern: Agents Message Buffer Contents 41
4.12 Send22 Staircase Pattern: Agents Message Buffer Contents 45
4.13 Guard Behaviour Message Sending Pattern 46
4.14 Guard Behaviour: Agents Message Buffer Contents 47
4.15 Guard Behaviour Time Line 51
4.16 SendK Linear: GP Agents Possible Moves for Best Run (Run 20) . . 52
4.17 SendK Message Sending Pattern, Cycles 14-21 52
4.18 SendK: Agents Message Buffer Contents 55
4.19 Message Sending Patterns (Prey Random Movement) 57
4.20 SendAll Synchronized Message Pattern: Agents Message Buffer Contents 57
4.21 Send22 Agents Message Buffer Contents 58

5.1 Communication Protocols 62
5.2 GP Parameters 63
5.3 Strongly Type Language 64
5.4 Terminal Set 64
5.5 Function Set 65
5.6 Send22 Fitness Summary (20 Test Runs) 69
5.7 SendAll Fitness Summary (20 Test Runs) 69
5.8 Tukey Comparisons for Send22 Protocols 72
5.9 Tukey Comparisons for SendAll Protocols 72
5.10 Message Sending Patterns for SendAll 73
5.11 Message Sending Patterns for Send22 74
5.12 Learning Meaning of Commands: Send22C1orC2 Run 13 Time Line 90
5.13 Send22C1orC2 Test Run 13 Message Buffer 91
5.14 Prey Movement Types in Training and Testing 93
5.15 Fitness Summary Linear Training (20 Test Runs) 94
5.16 Fitness Summary Random Training (20 Test Runs) 95
5.17 Fitness Summary Linear-Random Training (20 Test Runs) 96
5.18 Tukey Comparisons for Training Movement Types 98
5.19 Tukey Comparisons for Testing Movement Types 99

6.1 Communication Protocols 108
6.2 GP Parameters 110
6.3 Strongly Type Language 110
6.4 Terminal Set 111
6.5 Function Set 112
6.6 Send22 Fitness Summary (20 Test Runs) 115
6.7 Tukey Comparisons for Send22 Protocols 116
6.8 Sending Patterns for Send22C1, Send22C1orC2, and Send22PvA . . . 118
6.9 Send22C1orC2 Test Run 7 Message Buffer 123

A.1 ANOVA Hypothesis 134
A.2 Prey Linear Movement ANOVA results 134
A.3 Prey Random Movement ANOVA results 135
A.4 Send22 Protocol Types ANOVA results 135
A.5 SendAll Protocol Types ANOVA results 135
A.6 Training: Prey Linear Movement Types ANOVA results 136
A.7 Training: Prey Random Movement Types ANOVA results 136

A.8 Training: Prey Linear/Random Movement Types ANOVA results . . 136
A.9 Testing: Prey Linear Movement Types ANOVA results 136
A.10 Testing: Prey Random Movement Types ANOVA results 137
A.11 Testing: Prey Linear/Random Movement Types ANOVA results . . . 137
A.12 PacMan Testing: Send22 ANOVA results 137

List of Figures

2.1 Example of a GP Parse Tree 5
2.2 GP Tree Before Crossover 7
2.3 GP Tree After Crossover 7

3.1 Legend used for GP Tree Diagrams 16
3.2 Example of GP Tree using Strongly Typed Language 16
3.3 Major Components of Message Buffer System 17
3.4 GP and Message Buffer System Integration 21
3.5 Example of Pursuit Domain Simulator 21
3.6 GP and Pursuit Domain System Integration 23
3.7 Example of Ms. Pac-Man Simulator 24
3.8 GP and Ms. Pac-Man System Integration 26

4.1 Pursuit Domain Environment 28
4.2 Top-level GP Structure 29
4.3 Prey Linear Movement - Training Fitness for all Communication Types 36
4.4 Prey Linear Movement - Training Fitness for Top Performers 37
4.5 Prey Linear Movement - Training Fitness for Best and Worst Performers 38
4.6 Prey Random Movement - Training Fitness for all Communication Types 39
4.7 Staircase Pattern: SendAll Agent 1 & 3's branches 42
4.8 Staircase Pattern: SendAll (Agent 1 (purple)) 43
4.9 Staircase Pattern: SendAll Cycle 11 44
4.10 Staircase Pattern: SendAll Cycle 12 44
4.11 Staircase Pattern: Send22 (Agent 3 (green)) 45
4.12 Guard Behaviour GP: Example of Agent 2's C1 branch 46
4.13 Guard Behaviour GP: Example of Agent 1's C1 branch 48
4.14 Guard Behaviour: Agent 2 Sends out Messages. 49
4.15 Guard Behaviour: Agent 1 uses LRM data. 49
4.16 Guard Behaviour Cycle 13. 50

4.17 Guard Behaviour Cycle 15. 50
4.18 Guard Behaviour Cycle 17. 51
4.19 SendK GP Sub-tree Example 53
4.20 SendAll Random GP Sub-tree Example 56

5.1 Top level GP Structure 60
5.2 Send22 - Training 68
5.3 SendAll - Training 70
5.4 Send22 SendAll - Training 71
5.5 Common GP Structure 74
5.6 SendAllC1 Run 2 Agent 0 76
5.7 SendAllC1 Run 2 Agent 1 76
5.8 SendAllC1 Run 2 Agent 2 77
5.9 SendAllC1 Run 2 Agent 3 77
5.10 Common GP Structure to Decide To Send Specific Commands 79
5.11 SendAllC1C2 Agent 0 Sub-tree, Run 15 79
5.12 Send22C1C2 Agent 2 & Agent 3 Sub-trees, Run 12 81
5.13 SendAllC1orC2 Agent 0 & Agent 1 Sub-trees, Run 19 82
5.14 SendAllC1orC2 Agent 2 & Agent 3 Sub-trees, Run 19 83
5.15 Send22C1orC2 Agent 0 & Agent 1 Sub-trees, Run 13 84
5.16 Send22C1orC2 Agent 2 & Agent 3 Sub-trees, Run 13 85
5.17 Deciding Which Command to Send: Send22orC1C2, Run 13 86
5.18 Understanding Received Commands: Send22orC1C2, Run 13 86
5.19 Learning the Meaning of Commands: Send22C1orC2 Run 13, **Cycle 1** 87
5.20 Learning the Meaning of Commands: Send22C1orC2 Run 13, **Cycle 9** 87
5.21 Learning the Meaning of Commands: Send22C1orC2 Run 13, **Cycle 11** 88
5.22 Learning the Meaning of Commands: Send22C1orC2 Run 13, **Cycle 13** 88
5.23 Learning the Meaning of Commands: Send22C1orC2 Run 13, **Cycle 17** 89
5.24 Learning the Meaning of Commands: Send22C1orC2 Run 13, **Cycle 22** 89
5.25 Send22 Protocol - Linear Training (_L_L, _L_R, _L_LR) 94
5.26 Send22 Protocol - Random Training (_R_L, _R_R, _R_LR) 95
5.27 Send22 Protocol - Linear/Random Training (_LR_L, _LR_R, _LR_LR) 96
5.28 Send22C1_L_L Agent 0 & Agent 1 Sub-trees, Run 8 100
5.29 Send22C1_L_L Agent 2 & Agent 3 Sub-trees, Run 8 101
5.30 Send22C1orC2_L_L Agent 0 & Agent 1 Sub-trees, Run 17 103
5.31 Send22C1orC2_L_L Agent 2 & Agent 3 Sub-trees, Run 17 104

6.1 Agents' FOV in Ms. Pac-Man Environment 106
6.2 Agents' Starting Positions in Ms. Pac-Man 113
6.3 Send22 - Training for C1, C1orC2, PvA 115
6.4 Decision Sub-trees for Send22C1, Send22C1orC2, and Send22PvA . . 119
6.5 Send22C1 Typical Behaviour 119
6.6 Send22C1orC2 Typical Behaviour 120
6.7 Send22PvA Typical Behaviour 120
6.8 Send22C1orC2 Agent 0 & 1 C0 & C2 Branches 121
6.9 Send22C1orC2 Run 7, **Cycles 130 - 147** 122

Chapter 1

Introduction

In artificial intelligence (AI), an *agent* is known as an entity that exhibits autonomy, and performs actions based on feed back from the environment [1]. An environment containing more than one agent is known as a *multi-agent* system. In this type of system, agents with limited information about their world interact with one another to complete a task [1]. Evolving co-ordinating behaviour strategies for agents is a central issue in multi-agent systems research. It has been found that evolved strategies can be applied to many real world applications in which agent coordination is necessary (e.g. robots working together to complete a task) [2]. Recently, there is growing interest in research using evolved behaviour strategies to aid in the development of scripted enemy AI for commercial games [3].

As seen in the work by Reverte *et al.* [2], an appropriate test bed for multi-agent systems is the predator-prey pursuit problem (the pursuit domain). The pursuit domain contains multiple agents, known as "predators", who have the job of working together to chase and capture an agent, known as the "prey". The job of the prey is to evade the predators (not be captured). A typical scenario consists of an infinite, discrete world in the form of a toroidal grid containing 4 predators and 1 prey. Agents move sequentially and are not allowed to occupy the same cell [2]. Using several variants of the pursuit domain, early work by Haynes *et al.* [4] and Denzinger and Fuchs [5] show that agents can successfully learn to co-ordinate movements.

Research has shown that communication is effective in multi-agent systems where little information about the environment is known. Using the pursuit domain, Iba [6], and Kam-Chuen and Giles [7] demonstrate that multiple predator agents can successfully learn to use a new simple command language in order to capture a prey agent. Iba [6] also shows that communication is effective in co-ordinating robot navigation. In earlier work, Yanco and Stein [8] develop an adaptive communication

protocol for co-operating mobile robots. In more recent work, case based reasoning is used by Kou *et al.* [9] in a predator-prey scenario in which each "predator" agent, using limited communication, must learn to capture the evader.

Previous work using genetic programming (GP) shows that complex behaviour can emerge from simple interactions among agents. Tanev *et al.* [10] demonstrate the emerging surrounding behaviour of agents developed from proximity defined interactions of predator agents in the pursuit domain. In other work, Zhang and Cho [11] explore the idea that realistic complex tasks require more than one type of emergent behaviour to solve a problem and implement a robust fitness measure ("fitness switching") to encourage the emergence of multiple behaviours.

In addition to the pursuit domain, learning strategies in game environments have also been studied. Luke *et al.* [12] present a competitive learning strategy using genetic programming to co-evolve agents that are members of a soccer team. Also in a simulated soccer game, Kou *et al.* [13] use a predefined language to allow a "coach" agent to co-ordinate movement of "player" agents. In the game of Ms. Pac-Man, Ms. Pac-Man and ghost team controllers are co-evolved by Cardona *et al.* [14] and GP is used by Alhejali and Lucas [15] to evolve Ms. Pac-Man behaviours using training camps. Research by Kadlec [16] uses genetic programming to optimize non player characters' (NPCs) behaviour in "Death Match" and "Capture the Flag" game modes in the commercial game of Unreal Tournament 2004 (UT) [17].

1.1 Overview of Project Goal and Results

The goals of this research are to investigate how well genetic programming can evolve predator agents that can learn a language consisting of generic commands, and how well they can communicate using this language in order to learn the behaviour of tracking prey. These goals are achieved by focusing on the following communication and learning strategies in a predator-prey scenario.

The communication strategy in this study uses a learned language [1]. The language consists of at most 3 generic commands, *C0*, *C1*, and *C2*. Commands, along with simple environment data, are sent from one agent to another through a message passing communication channel. An agent learns to associate a meaning to each command through evolving branches of its GP sub-tree. Depending on the experiment, each agent has 2 or 3 child branches (command trees) where each branch is associated with one command. The evaluation of an agent's child branch relies on whether the agent has received a message and the type of command (*C1* or *C2*) within the

message.

The learning strategy uses a fully co-operative implementation with a global fitness measure [1]. The predator agents work together to complete their common goal of first finding, and then following a prey (as closely as possible). The global fitness measure is a minimization function which calculates the total distance between all predator agents and the prey over a limited period. The motivation for this fitness function is to compare how different communication protocols perform in allowing the predators to track the prey's movement. Agents collaborate to minimize the global fitness value using a heterogeneous team based learning strategy such that each agent uses its own learning algorithm to evolve [1].

Similar to Reverte *et al.* [2], predator agents have little knowledge of the environment. They do not know the location of the prey unless they are in field of view (FOV) of the prey and they do not know the location of other predators. However, similar to robot agents in Iba [6], they do know the relative (nearest, second nearest and farthest) direction of other predator agents. Predator agents can send messages (that *may* contain prey information if they are in FOV of prey) to other predator agents at any time.

The following gives a brief outline of the organization of the remainder of the thesis. Chapter 2 gives a brief background of research and terminology used in this paper. Chapter 3 describes the major systems used throughout this research and gives an explanation of how they are integrated.

Chapter 4 examines different communication protocols [1] used for predator agents evolved to learn a language within the Pursuit Domain Development (PDP) Toolkit [19]. Two experiments are performed. The first experiment, *Prey Linear Movement*, tests the ability of predator agents to find and follow a linear moving prey. The second experiment, *Prey Random Movement*, tests the ability of predator agents to find and follow a random moving prey.

Chapter 5 focuses on an enhanced GP language and fitness measure for the *Prey Random Movement* problem from Chapter 4 and is divided into two parts. Part one focuses on the top performing protocols found in Chapter 4. Part two determines whether or not the success of predator agents is affected in test runs that use a different prey movement pattern than used in training.

Chapter 6 tests the GP language defined in Chapter 5 in the "real world" game environment of Ms. Pac-Man[20].

The final chapter reviews the results of the thesis and suggests future work.

[1]This chapter gives a more detailed explanation of the work presented in CIG 2017 [18].

Chapter 2

Background

2.1 Genetic Programming

The goal of genetic programming (GP) is to search a set of potential computer programs which, when executed, produce desired behaviour [21]. GPs can be thought of as a subset of genetic algorithms (GA). To define population members, genetic algorithms use individual structures represented as fixed strings. In contrast, population members for GPs are computer programs. The computer programs are arranged in a hierarchical tree structure which can evolve and can vary in size, shape and complexity [21]. To measure fitness, GPs execute the computer program for the specific population member. As shown in Figure 2.1, the structure of a genetic program is represented as a parse tree. Internal tree nodes represent functions (e.g. add (Add) or subtract (Sub)...) needed to solve the problem and leaf nodes represent data for these functions (e.g. variables and constants). According to Eberhart and Shi [21], in order to define a genetic program the following five steps must be performed:

1. **Specify the Terminal Set**: The terminal set consists of the data or variables to be used as input for the functions in the GP. An example could be location and/or direction as input for a problem dealing with movement of an agent.

2. **Specify the Function Set**: The function set consists of the set of functions which can be used to solve the problem. Functions can have one or more inputs. The inputs can be results of other functions or terminals (see Figure 2.1). Some examples could be arithmetic operators (addition, subtraction etc...) or Boolean operators (AND, NOT, etc...). Other examples could be more complex (having up to four inputs or more) such as if-then-else statements.

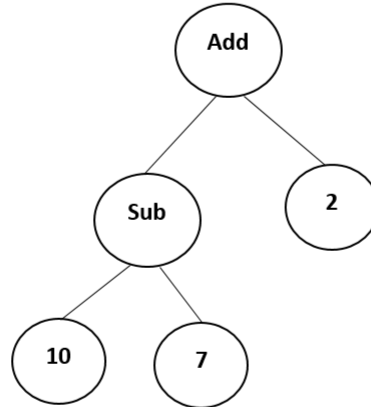


Figure 2.1: Example of a GP Parse Tree

3. **Specify the Fitness Measure**: The fitness measure is used to measure the performance of a GP individual. An example of a fitness measure could be the score a program receives in a game.

4. **Select the Control Parameters**: Two main control parameters include the population size and the maximum number of generations. In addition, the following parameters could be used as well: reproduction probability, crossover probability and maximum depth of tree allowed.

5. **Specify the Termination Conditions**: The termination condition is usually determined by the maximum number of generations control parameter. After all generations are complete, the best individual is determined by the one which scored the best using the fitness measure.

Table 2.1 shows the pseudocode for the steps in evolution using a GP system as described by Poli *et al.* [22]. Through an *Initialize()* function the first step initializes a random population of GP individuals. For each generation, until a maximum generation value ($MaxGen$) (or another terminating condition) is reached the following steps occur.

Each member of the population is evaluated through the *Evaluate()* function. This function determines the fitness score of each GP individual and returns the best performing individual found so far. Either 1 or 2 individuals are selected for reproduction based on their fitness score and a probability setting using the *SelectIndividualsForReproduction()* function. Reproduction uses genetic operators (crossover or mutation) to create new members of the population. Based on another probability

Table 2.1: Pseudocode for Evolution in a GP System

```
Evolve(){
    Initialize();
    int generation = 0;
    GPIndividualType bestIndividualSoFar = null;
    Loop {
          bestIndividual = Evaluate();
          SelectIndividualsForReproduction();
          CreateNewIndividuals();
          generation++;
          if (generation > MaxGen) then break;
     }
    return bestIndividualSoFar;
}
```

setting, the *CreateNewIndividuals()* function performs either crossover or mutation on the selected individuals.

Crossover is performed by using the selected individuals as 2 parent computer programs. A point in the tree structure of each parent is chosen as a crossover point. The crossover points can be either the root, function or terminal nodes. The crossover exchange involves swapping the crossover root point (along with its entire sub-tree) of one of the parent trees with the crossover root point (along with its entire sub-tree) of the other parent tree (see Figures 2.2 and 2.3). If mutation is selected to create a new individual, then a new child program is generated by randomly changing a part of 1 selected parent program. Weaker individuals of the population are replaced with new individuals created after crossover and mutation[22].

The evolution process is repeated for each generation until the maximum number of generations is reached or until a terminal condition has been met. The best individual (based on fitness score) is returned.

2.1.1 Strongly Typed GP

According to Montana [23], strongly typed GP is used to apply constraints to genetic programming. It is particularly useful in controlling the tree structure of a GP. This is done by a applying a type to each terminal and by applying a type to every argument and return value of each function defined in the GP language. Creation processes such as initialization, random expressions, crossover and mutation, generate GP trees

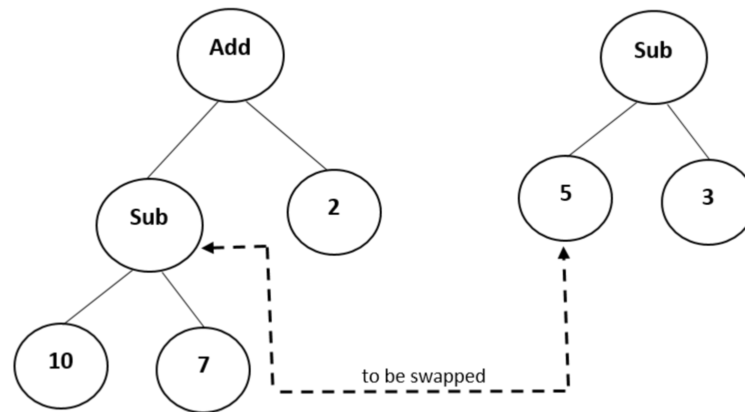


Figure 2.2: GP Tree Before Crossover

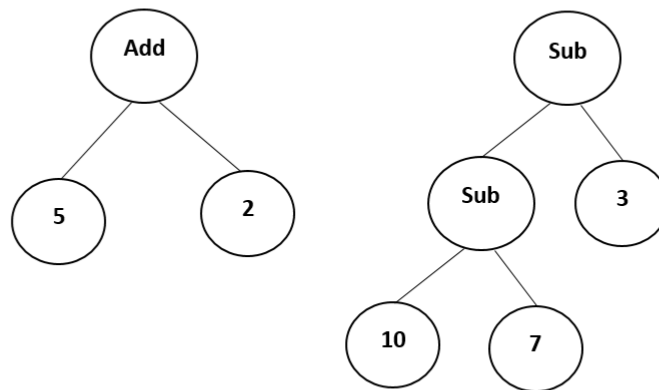


Figure 2.3: GP Tree After Crossover

so that the types of each terminal, function arguments, and function return types are not violated.

For example, as stated in [23] and [22], suppose a GP system has 2 types: *numeric* and *Boolean*. The GP language would have terminals and functions that return either a numeric value or a Boolean value. An example could be a *getDistance()* function which returns the numeric distance to a prey and *ifPreyAhead()* function which returns true if the prey is within view or false if the prey is not within view. A third function, *if(Boolean,numeric,numeric)* has 3 arguments. The first argument is a Boolean type, and the second and third values are numeric types. This function returns a value with a numeric type. If the first argument evaluates to true, it returns the first numeric value otherwise it returns the second numeric value. Suppose the GP system generates a new GP individual using crossover and the crossover point is the

Boolean argument of the *if(Boolean,numeric,numeric)* function on the first parent. The crossover point on the second parent must be a terminal or function that returns a Boolean (e.g. the *ifPreyAhead()* function) so that the return value matches the type of the crossover point in the first parent [22]. In this way, using strongly typed GP gives the programmer control over GP structures created by the GP system.

2.2 Multi-Agent Evolution

Multi-agent learning and communication has been studied for years. The work in [24] provide real world practical examples of uses for multi-agent systems. At first glance, it is sometimes difficult to sort out the differences between multi-agent learning, (co-operative vs competitive) and communication. The following provides a more detailed background of some of the research in multi-agent evolutionary learning and communication.

Pannait and Luke [1] provide an early survey of multi-agent based research and give a good explanation of the differences between multi-agent learning strategies and communication strategies. They also show how co-operative and competitive strategies contribute to multi-agent learning. Tables 2.2 and 2.3 provide a brief summary of their findings. More recent work by [25] focuses on critically reviewing interaction protocols for multi-agents.

As shown in Table 2.2, multi-agent evolutionary learning strategies can involve one or more learning agents having individual (local) and/or common (global) fitness goals. The agents' learning can take place as part of a team or happen concurrently. Team based learning can be homogeneous or heterogeneous. Team members of a homogeneous team use the same learning algorithm to evolve, while team members of a heterogeneous team use their own learning algorithm to evolve. Concurrent learning can happen in either a fully co-operative scenario, where agents work together to complete a goal, or in a competitive scenario where agents compete against each other (co-evolution) to achieve a goal. Concurrent learning can also happen in a hybrid version of the scenarios in which learning can occur both in an individual scenario and in a co-operative scenario.

Communication strategies, as seen in Table 2.3, are concerned with the actual language (commands) used to communicate and the communication channel. The language can be direct or indirect. Direct language can use hard-coded commands or a learned language. In the case of hard coded commands, agents learn commands that have pre-defined meanings. In the case of a learned language, agents learn to associate

Table 2.2: Multi-Agent Learning Strategies [1]

Learning Strategies	Description	Number of Agent Learners	Individual Fitness	Common Fitness
Team	Homogeneous	1	No	Yes
	Heterogeneous	1 or more	No	Yes
Concurrent	Fully co-operative: *agents always work together*	1 or more	No	Yes
	Partially co-operative: *agents sometimes work together*	1 or more	Yes	Yes
	Competitive: *agents compete with each other (Co-evolution)*	1 or more	Yes	Yes

Table 2.3: Multi-Agent Communication Strategies [1]

Communication Strategies	Types	Communication Channel
Direct	Hard Coded: *use pre-defined commands* Learned Language: *learn meaning of commands*	Message Board or Message Passing
Indirect	Implicit transfer of information: *from agent to agent through modification of the environment. Inspired by insects social use of pheromones.*	Footstep trail Breadcrumb trail Hints through object placement

meaning to commands through various trials. Indirect communication is inspired by insects use of pheromones and involves the implicit transfer of information from agent to agent through the modification of the environment.

The communication channel, shown in Table 2.3, involves the mechanism used to communicate commands. Using direct communication, can involve a central message board, such as a blackboard, in which all agents can read/write commands to a

global post, or it can involve message passing, in which individual agents directly send and receive messages to/from each other. Using indirect communication, the communication channel can involve leaving footsteps, bread crumb trails, or placing hints in the environment.

Iba [6] made early observations in the evolutionary learning of communicating agents. He used genetic programming to evolve a learned language separately in both a robot navigation scenario and predator-prey scenario. In both scenarios a message passing system was used to send and receive commands between agents. In the robot navigation scenario, the following three experiments were performed to control robots' navigation through a grid: a heterogeneous team of agents with communication, a heterogeneous team without communication and a homogeneous team with communication. Team fitness was used to measure how quickly a team of agents could move to its goal. The results demonstrated that a heterogeneous team with communication outperformed a heterogeneous team that did not use communication. However, a homogeneous team with communication performed almost as well as the heterogeneous team with communication.

In the predator-prey scenario Iba [6] used a homogeneous team with common goals of "distance to the prey" and "capture the prey" to measure fitness. In this scenario, he used three experiments to study the importance of a learned language. The experiments involved teams that used no communication, teams that learned commands through communication and teams that learned communication within a field of view (that is, only team members within a certain range could communicate with each other). An overhead cost for communication was applied to the fitness measure. The results of this scenario showed that teams with learned communication commands were superior in fitness. However, the average number of successes (capturing of prey) were similar in all three experiments for the following reasons. The team without communication could capture prey in some cases because the prey moved about the grid in a random fashion, however in most cases they could not capture the prey. The teams with communication could close in on the prey quickly but were stopped once they were very close to the prey (due to the burden cost of communication). This scenario showed that communication is necessary when predators are far from the prey, but once they are near and in view of the prey, communication is not necessary and the communication cost often becomes a burden.

Kam-Chuen and Giles [7] used a genetic algorithm to compare the influence the message length has on an agent's ability to learn a pre-defined language in the predator-prey scenario. Each predator agent evolved individually using a fitness based

on the average performance of all predators. A message board was used to communicate commands and information such as bearing/distance of prey and actions (move North, East, South and West). The first experiment evolved new strategies comparing the message lengths of 0 (no communication), 1 (1 command in message) and 2 (2 commands in message). In the second experiment, new strategies were grown from successful strategies. In this experiment, strategies using a message length of 2 were evolved from successful strategies created from message lengths of 1. The results demonstrated that new strategies (which evolved on their own) with message lengths of 1 performed just as well as strategies that were grown from successful strategies (therefore, growing the length of the message from 1 to 2 did not increase the performance).

Reverte *et al.* [2] applied neuro-evolution to evolve a co-ordination protocol for predator agents in the predator-prey scenario. Each agent evolved separately using a neural network to determine the next best move for the agent. Communication was done through a hard-coded set of commands using message passing. However, message passing was limited to agents within each other's field of view. A global fitness uses the number of collisions (i.e. two agents try to occupy the same spot on the grid) and the number of cycles it takes to capture the prey, to measure fitness. The results showed that using neuro-evolution significantly reduces the number of collisions compared to trials that did not use neuro-evolution.

Luke *et al.* [12] present a competitive learning strategy using genetic programming to co-evolve agents that are members (players) of a soccer team. This research was entered in the first RoboCup workshop in 1997. In this study, each agent learned individually using 2 action trees. In order to perform an action, the program for each agent used a set of state rules to decide which tree to use. The first tree consisted of simple actions such as making a kick. The second tree consisted of moving a player. In this research, two experiments were conducted to test the performance of a homogeneous team (each player uses the same set of two trees) against the performance of a heterogeneous team (each player uses its own set of two trees). Agents had limited communication with other team members. Fitness evaluation used the performance of the entire team (number of wins). The researchers reported that although they believed that the heterogeneous teams would out perform the homogeneous teams, they could not complete the heterogeneous team trials (due to time constraints) before the RoboCup challenge.

Yanco and Stein [8] developed an adaptive communication protocol for co-operating mobile robots. The co-operative task was to learn a language to co-ordinate move-

ment of the robots. Learning was done by all agents, however, one agent was given the role as a leader while the remaining agents were given the roles as followers. The leader was aware of task specification (environmental cues) and communicated commands to the followers. Followers were only aware of the commands sent to them. Fitness was team based so that all members of the team (both followers and leaders) were expected to perform the appropriate actions. The leader was tasked with learning to interpret environmental cues in order to give the appropriate commands to the followers. The followers were tasked with learning to interpret the commands sent to them in order to perform the appropriate action. Commands sent by the leader robot were determined to be appropriate if and only if the follower robots took the appropriate action when that signal was received. Results demonstrated that robots were easily able to develop a shared language. Robots were also able to adapt to a changing environment in a time that was comparable to the original learning environment. The results also showed that increasing the language size (in a range starting from 2 commands up to a size of 20 commands) increased the learning time exponentially. Increasing the team size from 2 to 3 members also increased the learning time significantly.

Barrett *et al.* [26] present (to the best of their knowledge) one of the first solutions that create autonomous agents capable of demonstrating ad hoc teamwork in an open complex teamwork domain. Members of an ad hoc team can effectively cooperate with multiple teammates on a set of collaborative tasks [26]. This work studies a range of algorithms for on-line behavior generation in which a single agent must collaborate with a range of teammates in the pursuit domain.

As described earlier, there is a strong research interest in the opportunities of using learning algorithms in video games. In May 2012, leading experts met in Schloss Dagstuhl in Saarland, southern Germany, to discuss future research directions and challenges in the field of artificial and computational intelligence in commercial video games [3]. The purpose of the two day seminar was to bring together academic researchers and game developers to discuss challenges in computational intelligence and to identify areas of potential future research. The seminar was divided into the following areas of research: Search in Real Time Video Games, Pathfinding in Games, Learning and Game AI, Player Modeling, Procedural Content Generation, General Video Game Playing, Developing a Video Game Description Language and Artificial and Computational Intelligence for Games on Mobile Devices. Based on their research interest and expertise, members of the seminar discussed the challenges within each specific area in the *Dagstuhl Reports* [3]. Later, follow ups were written

in each specific area. The Learning and Game AI follow up report listed several opportunities for learning algorithms in commercial games [27].

One of the opportunities found by Muñoz-Avila *et al.* [27] is a learning algorithm's ability to make timely decisions. A difficult challenge for AI in games is based on two issues: game AI is normally given little CPU time compared to other game systems (e.g. pathfinding), and the time to develop game AI is relatively short (other development tasks such as graphics and level design take precedence) [27]. Machine learning offers the possibility of improvement. For example, analyzing game logs of game traces allows the possibility of machine learning techniques to tune the game AI. Using this strategy, as stated by Muñoz-Avila *et al.* [27], Tesauro [28] created a system capable of producing game play strategies that were considered highly competent and LeeUrban *et al.* [29] created a learning system for a squad of bots that adapts to the opposing teams strategy. The Dagstuhl seminar and follow up reports are important because they show that along with commercial games becoming increasingly popular, there is also a growing interest in research to understand how learning algorithms can benefit the commercial games industry.

Chapter 3

System Design

3.1 System Overview

This research, in different experiments, uses the Pursuit Domain Development Toolkit [19] and the Ms. Pac-Man Development Toolkit [20] to create predator and prey agents in the pursuit domain. Genetic programming, in the Java Evolutionary Computation (ECJ) [30] environment, is used to evolve predator agents so that they learn how to find and follow a prey agent. Predator agents are allowed to communicate using a message buffer system developed in ECJ using Eclipse [31]. The following sections describe the systems and their integration for use in this work.

3.2 GP Tree Structure

This study uses the Java programming language and the Java Evolutionary Computation (ECJ) Toolkit [30] to conduct GP experiments. In order to create a heterogeneous team of predator agents, a strongly typed language [22] is used so that each predator agent evolves its own sub-tree. Table 3.1 shows an example of the strongly typed language used as a base to build the varying GP languages in this study. The following list describes the types in the language.

- **ROOT**: This type acts as the root node of the GP tree. It requires 4 **SIM** types as input.
- **SIM**: The SIM type serves as a command tree for a specific agent. It represents an agent's sub-tree and requires, at the basic level, two **EXPR** types as input. The number of input nodes can vary depending on the experiment. An agent

Table 3.1: Example of Strongly Type Language

ROOT	$::= (SIM, SIM, SIM, SIM)$
SIM	$::= CommandTree(EXPR, EXPR)$
EXPR	$::= Left \mid Right \mid Up \mid Down \mid Stay \mid$
	$::= IfGrtEql(NIL, NIL, EXPR, EXPR)$
	$::= Send(EXPR)$
NIL	$::= North \mid South \mid West \mid East \mid$
	$::= Goal \mid LRM$
	$::= Add(NIL, NIL)$

evaluates only one of the **EXPR** sub-trees resulting in one movement step for the agent.

- **EXPR**: This type represents one branch of an agent's sub-tree. It contains the expressions used to determine a movement for the agent. **EXPR** Functions can have **EXPR** or **NIL** types as input and will evaluate only one of its **EXPR** inputs. For example, the IfGrtEql expression shown in Table 3.1 calculates the length of the first two **NIL** input vectors. If the first length is greater than the second length, then the function evaluates the third **EXPR** input expression otherwise, it evaluates the fourth **EXPR** input expression. The Send function is used to send messages and has only one **EXPR** type as input. This input type is evaluated after a message is sent. The language is structured to ensure that the evaluation of the top-level **EXPR** type results in one terminal **EXPR** movement type (i.e. *Up, Down, Left, Right, or Stay*).

- **NIL**: The NIL type represents a directional vector on a 2D (x, y) grid. Examples of terminal expressions are the directional vectors *North, South, West, East* and *Stay* terminals nodes. Other examples include the *Goal* and *LRM* terminal expressions which hold directional information to the prey. **NIL** Functions accept only **NIL** types as input (see Add function in Table 3.1).

Figure 3.1 shows a legend for GP tree structure diagrams used throughout this document. Rectangular nodes represent the non terminal **EXPR** type in the language. Circle nodes represent the terminal **EXPR** (i.e. *Up, Down, Left, Right, or Stay*) type and hexagonal nodes represent the **NIL** type.

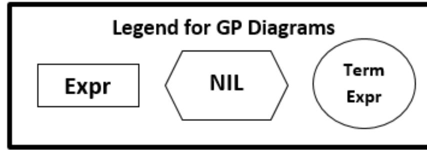


Figure 3.1: Legend used for GP Tree Diagrams

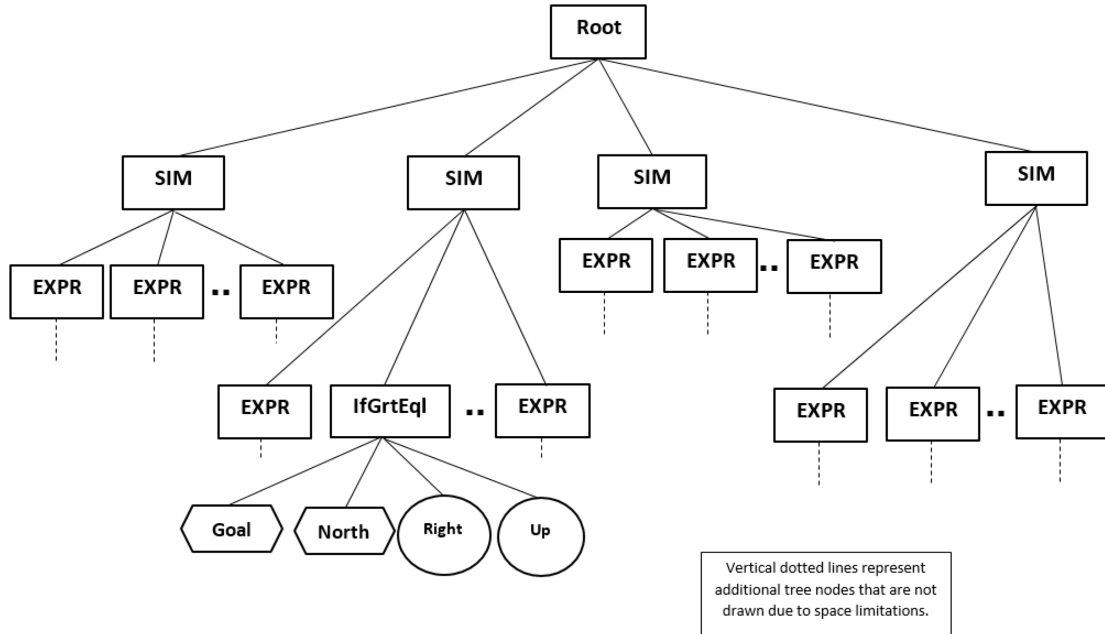


Figure 3.2: Example of GP Tree using Strongly Typed Language

Figure 3.2 shows the top level of the GP tree structure using the language defined in Table 3.1. This figure shows 4 SIM type agents as children (inputs) to the root node. Each SIM agent has at least 2 or more EXPR type children. The evaluation of an EXPR child results in one movement step for the agent. For example, Figure 3.2 shows that the second SIM node (from the left) has the IfGrtEql expression as its second child. The evaluation of the IfGrtEql expression results in the agent moving either *Right* or *Up*. The GP tree is structured so that each evaluation of a SIM child node ends at a movement node (i.e. *Up, Down, Left, Right, or Stay*). The SIM agent chooses to evaluate only one of its EXPR children. This results in each SIM agent moving one step for each evaluation.

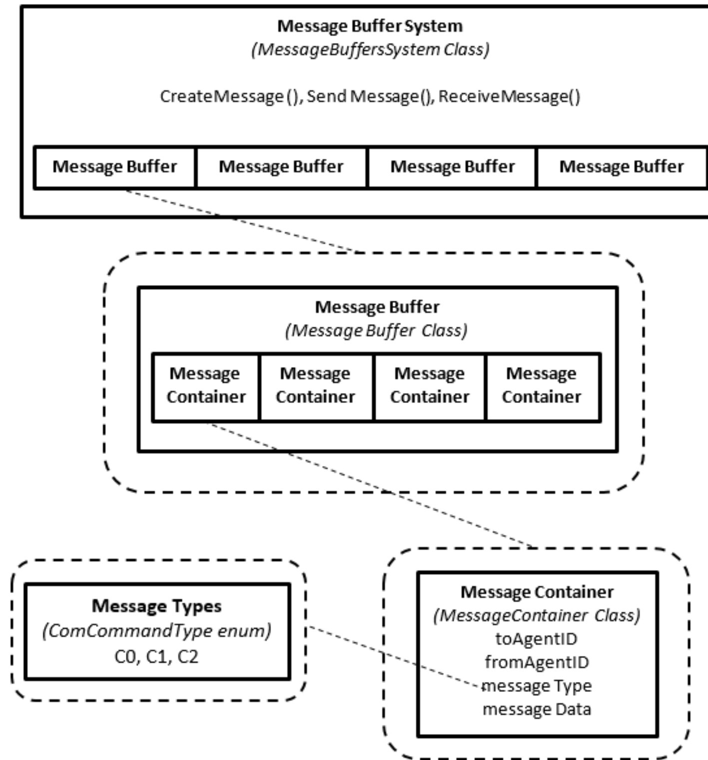


Figure 3.3: Major Components of Message Buffer System

3.3 Message Buffer System

Communication between predator agents is achieved through the message buffer system. This system is implemented in ECJ [30] for easy integration with GP sub-tree classes. Figure 3.3 shows the major components of the message buffer system.

The *MessageContainer* class holds all the data for one message. The data is explained below:

- **int toAgentID**: Holds the ID of the agent receiving message.
- **int fromAgentID**: Holds ID of the agent sending message.
- **CommCommandType messageType**: Holds the message type (enumerated type).
- **CommandDataType messageData**: Contains all data for a message including a 2D directional vector variable.

The *MessageBufferSystem* class is the main class. It contains a list of message buffers (one for each agent) and gives the ability to create, send and receive messages

to the message buffers through the CreateMessage, SendMessage and ReceiveMessage functions listed below.

- **CreateMessage(int to, int from,**
  CommCommandType inMessageType,
  CommandDataType inMessageData): Creates and returns a message (of type *MessageContainer*) with necessary data to send the message.
  Parameters:
  - ***to***: agent to send message,
  - ***from***: agent which is sending message,
  - ***inMessageType***: the message command to send,
  - ***inMessageData***: the data to send in the message.
- **SendMessage(MessageContainer message)**: Sends message to agent defined in message.
  Parameter:
  - ***message***: message containing the sending agent, receiving agent, message command and message data.
- **ReceiveMessage(int agentID)**: Removes and returns first message from an agent's message buffer.
  Parameter:
  - ***agentID***: specifies which agent's message buffer the message should be removed from.

The information for each agent's message buffer is stored in the *MessageBuffer* class. The message buffer is a list of type *MessageContainer* and operates as a queue. When a message is sent to an agent's message buffer (via the *MessageBufferSystem.Send* function) the message is added to the end of the message buffer. When a message is received (via the *MessageBufferSystem.Receive* function) a message is removed from the front of the message queue. Although the size of the message buffer can be dynamic, in this study the maximum number of messages it can hold is fixed to 4. If the message buffer is full when a message is added, then the first message in the list is removed (since it is the oldest message) to make room for the new message.

3.3.1 Integration of GP Tree and Message Buffer System

The integration of the GP Tree and Message Buffer System is seen in Figure 3.4. The evaluation of a GP tree begins at the root node. The root node evaluates each of the four agent sub-trees consecutively always starting with Agent 0 and ending with Agent 3.

Before the evaluation of its sub-tree, an agent uses the *MessageBufferSytem.Receive* function to remove the next message in its message queue. If there is a message then a global data variable which holds the last received message (LRM) for the agent is updated with the message data and the corresponding branch, determined by the message type (C1 or C2), is evaluated. If there is no message in the agent's message queue, then the LRM global variable and message type are set to default values and the C0 branch of the agent is evaluated. Table 3.2 shows the pseudocode in the *Evaluate* function for the root node of the GP tree.

Steps 1 to 3 in Figure 3.4 show what happens when one agent sends a message to another agent. Agent 2, upon evaluation of its *Send* node sends a message (with directional information as the message data and the C1 command as the message type) to Agent 1. This places a message in Agent 1's message buffer (see Step 1). On Agent 1's next turn, before evaluating its sub-tree, it will use the *MessageBuffer-Sytem.Receive* function to remove the message from its buffer and to update its LRM global data (see Step 2). The message type in the message data determines what branch will be evaluated. In this case, the message type is C1 therefore Agent 1 evaluates its C1 branch (see Step 3).

3.4 Pursuit Domain

Experiments in Chapter 4 and Chapter 5 use the Pursuit Domain Package (PDP) by Kok and Vlassis [19]. According to Reverte *et al.* [2], PDP is a toolkit which simulates the predator-prey problem. It includes one prey agent and four predator agents and allows the modification of parameters to instantiate different experimental scenarios. Its environment consists of a grid in which agents are allowed to move to any adjacent cell (9 possible options) in one time step. This research uses this toolkit to allow predators to learn commands in order to find and follow the prey. Figure 3.5 shows an example of this environment.

In this study a 20 x 20 grid is used, where 1 cell = 1 unit of distance. Predator agents have a field of view (FOV) = 2 cells, shown as the grey square around each

Table 3.2: Pseudocode for Root Node Evaluation

```
Evaluate() {
   int i = 0;
   Loop {
      LRM[i] = MESSAGE_DEFAULT;
      CommCommandType type = C0;
      MessageContainer message = MessageBufferSystem.ReceiveMessage(i,0);
      if(message) then
      {
         LRM[i] = message.messageData;
         type = message.messageType;
      }
      evaluate(Agent[i].SubTree[type]);
      i++;
      if (i > 3 ) then break;
   }
}
```

agent in Figure 3.5. All four predator agents and the prey can move to only one cell (Up, Down, Left, Right or Stay) in 1 time cycle. The grid is toroidal such that if any agent's next move is outside of the grid boundary, then it will move to the next cell on the opposite edge (i.e. the movements are wrapped at the edges). There are no collisions in the environment. Predator agents are allowed to share the same cell as the prey agent, however if a predator agent moves to a cell that is occupied by another predator agent then both predator agents are penalized by being placed at their original starting positions. In order to allow predator agents to find and follow the prey agent, they are given a specific number of movements (update cycles). Predator agents are always given the full number of update cycles even if predators share the same cell (i.e. "capture") as the prey.

3.4.1 Integration of Pursuit Domain with GP

The integration of the Pursuit Domain Toolkit and the GP environment is done through the class *PDPGameWorld*. This class is shown as the center box in Figure 3.6. The *PDPGameWorld* class is responsible for initializing the Pursuit Domain simulator and acts as a wrapper class between the GP environment and the Pursuit Domain simulator, giving both systems access to required data and functions. The *UpdateAgents* function in this class is used to signal the update of the position of

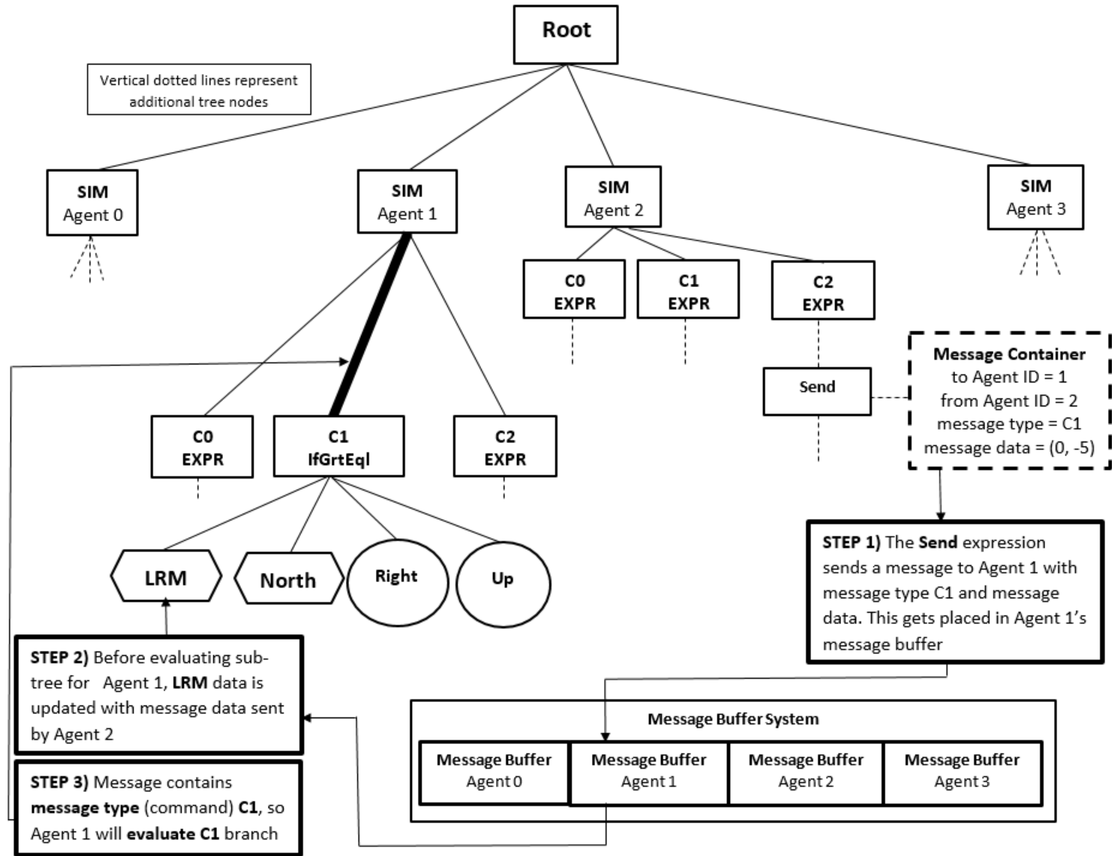


Figure 3.4: GP and Message Buffer System Integration

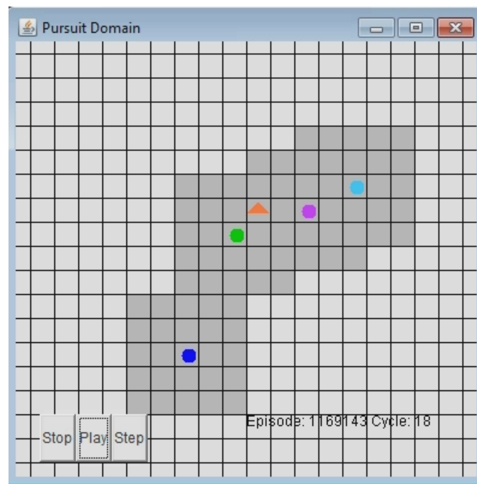


Figure 3.5: Example of Pursuit Domain Simulator
Prey Agent (orange), Predator Agent 0 (light blue), Predator Agent 1 (purple)
Predator Agent 2 (dark blue), Predator Agent 3 (green)

agents in the simulator.

The Agent class (see bottom boxes in Figure 3.6) in the Pursuit Domain simulator

controls the behaviours of agents. This is done through the *DetermineMovement* function in this class. The purpose of this function is to update the agents in the simulator by moving them one cell on the grid.

A second message buffer system object is used for direct communication between corresponding agents in the GP environment and agents in the simulator (see top box in Figure 3.6). Using this second message buffer system, when an agent (in the GP environment) evaluates a movement terminal node such as "Up" then a message is sent to the corresponding agent's message buffer in the simulator environment (see Step 1 in Figure 3.6). After all agents have evaluated their sub-trees, the GP *evaluate* function calls the *UpdateAgents* function from the *PDPGameWorld* class (see Step 2 in Figure 3.6). The *UpdateAgents* function calls the *SimulatorUpdate* function which is responsible for updating the prey agent and predator agents in the simulator (see Step 3 in Figure 3.6). The *SimulatorUpdate* function does this by calling the *DetermineMovement* function first for the prey agent and then for the predator agents (see Steps 4 & 5 in Figure 3.6). The *DetermineMovement* function checks the message buffer for the simulator agent and moves the agent in the direction defined in the message.

3.5 Ms. Pac-Man SDK

To further investigate results found using the Pursuit Domain, a popular simulator of the game Ms. Pac-Man is used in Chapter 6. This simulator, developed for the Ms. Pac-Man vs Ghost Team Competition [20], has been widely used in research as a test bed for evolving both Ms. Pac-Man (prey agents) and Ghosts (predator agents) (see Fig 3.7). Some of this research, such as the work by Cardone *et al.* [14], uses competitive co-evolution strategies to evolve both Ms. Pac-Man and Ghosts. Others, such as Shrum and Miikkulainen [32], and Alhejali and Lucas [33] [34], focus on investigating different learning strategies to evolve controllers solely for Ms. Pac-Man. This research uses the Ms. Pac-Man simulator to evolve behaviours for the predator Ghosts. The Ghosts in this study will need to learn the meaning of commands in order to find and follow the prey (Ms. Pac-Man) as it moves randomly in the environment.

The Ms. Pac-Man environment is in the shape of rectangular grid with dimension of 28 (width) x 30 (height) cells. Each cell contains a pill and the four corner cells contain power pills. The goal of the game is for Ms. Pac-Man to collect as many pills (points) as she can without getting caught (entering the same cell as one of the Ghosts). If Ms. Pac-Man collects a power pill then the game changes from chase

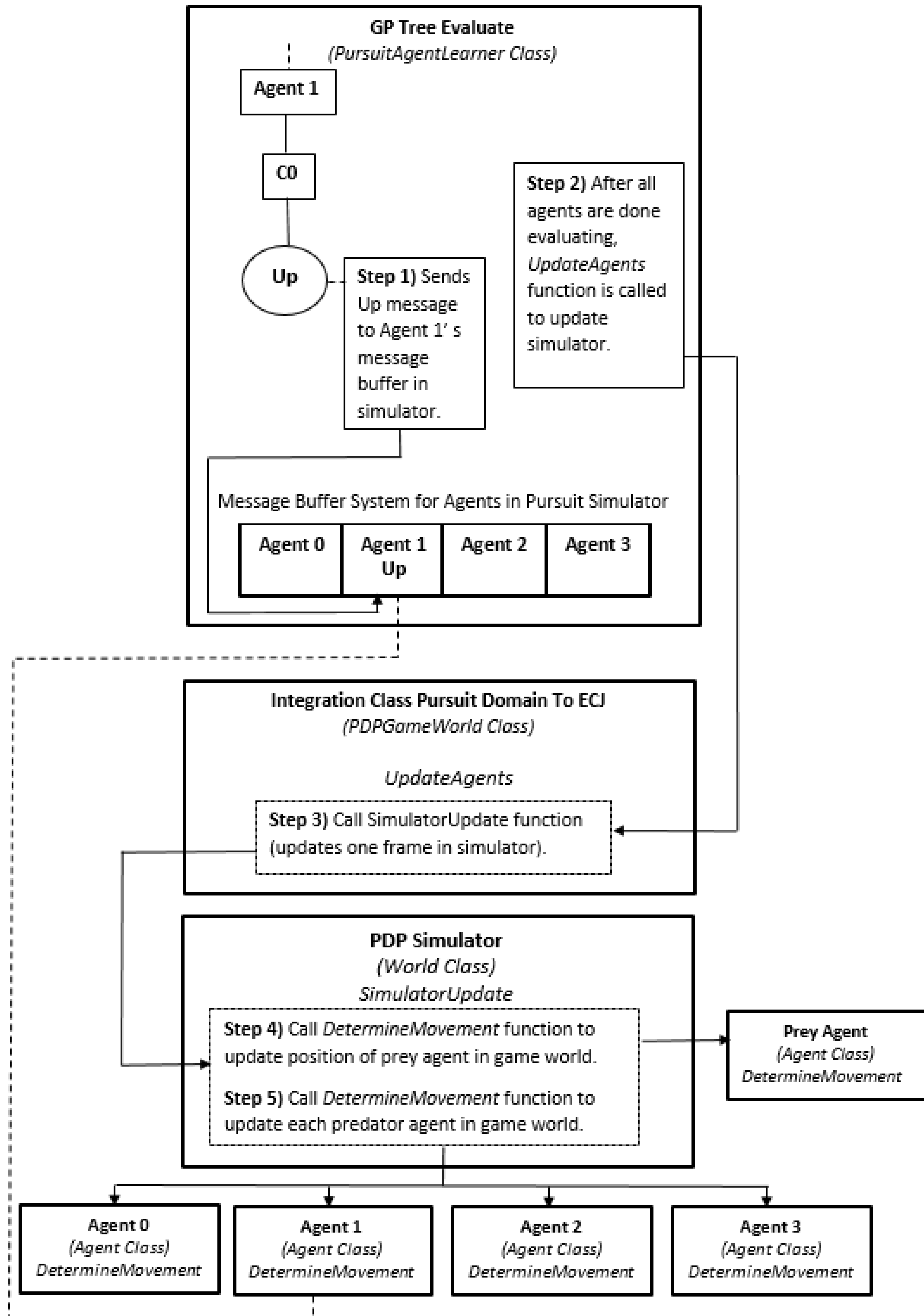


Figure 3.6: GP and Pursuit Domain System Integration
The evaluation of the GP tree updates one frame of the simulator

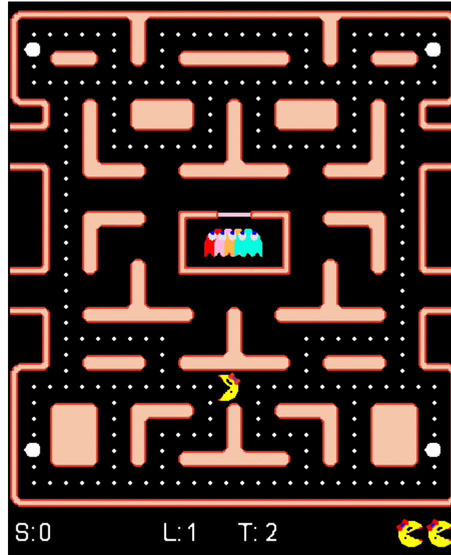


Figure 3.7: Example of Ms. Pac-Man Simulator
Agent 0 "Blinky" (red Ghost), Agent 1 "Pinky" (pink Ghost)
Agent 2 "Clyde" (orange Ghost), Agent 3 "Inky" (blue Ghost)

mode to retreat mode. In chase mode, Ghosts chase Ms. Pac-Man and can destroy Ms. Pac-Man if they catch her. In retreat mode, Ghosts retreat back to the center of the grid (to their lair). In this mode if Ms. Pac-Man catches a Ghost then the Ghost is destroyed. Retreat mode lasts for a set amount of time. Most of the game is played in chase mode.

The Ms. Pac-Man environment has many collisions (walls) that block agents' movements. To keep agents within the grid, all edges are blocked with walls except for 2 specific grid spaces on the vertical edges of the grid. Agents can use these spaces to wrap around the opposite side of the grid. Ghost agents may occupy the same grid cell without any penalties.

In order to allow predator Ghosts to find and follow Ms. Pac-Man, the game is limited to chase mode. The Ghosts continue to pursue Ms. Pac-Man (even if one Ghost "catches" Ms. Pac-Man) within a defined period. In this study Ms. Pac-Man does not die when it occupies the same cell as a Ghost.

3.5.1 Integration of Ms. Pac-Man SDK with GP

The class that integrates the GP system with the Ms. Pac-Man ToolKit is called *PacManToECJ* and is represented in the center box in Figure 3.8. This class is responsible for initializing the Ms. Pac-Man environment and is used as a wrapper class between the GP environment and the Ms. Pac-Man simulator, giving both

systems access to required data and functions. This class is also used to hold the movement actions set by the GP tree for each agent in an array called "Direction". Additionally, it contains the function *AdvancePacManSimOneFrame* used to update agents' data from the GP environment to the Pac-Man simulator. This function is called after the evaluation of all the agents' sub-trees is complete in the root node *evaluate* function.

Developers have access to Ms. Pac-Man and Ghost behaviours by overriding the controller classes (*PacManControllerBase* and *GhostControllerBase*) in the Ms. Pac-Man Toolkit (see bottom right boxes in Figure 3.8). The purpose of the controller classes is to update the movement action data for the Ghosts and Ms. Pac-Man. In this study, the actions for the Ghosts' controller are determined by the GP tree. The *Tick* function for the Ms. Pac-Man controller sets the movement action for this agent and is called before the game update function, *AdvanceGame*. The *AdvanceGame* function is called from the game update function in the simulator class (*PacManSimulator*) (see bottom left box in Figure 3.8). Using agents' action data, this function updates game variables and the positions of the Ghosts and Ms. Pac-Man in the game world.

Steps 1 to 6 in Figure 3.8 show the process of how the GP environment interacts with the *PacManToECJ* class and the Ms. Pac-Man simulator in order to move a Ghost in the "Up" direction. When an agent evaluates a terminal movement node such as "Up" then it sets the corresponding slot (using the agent's number) in the Direction array to "Up" (see Step 1). This results in the Direction array containing the next movement action for each agent after all agents have evaluated their sub-trees. Once this occurs the *AdvancePacManSimOneFrame* function updates the Ghosts actions (using the Direction array) in the Ms. Pac-Man simulator (see Steps 2 & 3). After updating the Ghosts actions, it calls the simulator's update function: *PacManSimulatorUpdateOneFrame* (see Step 4). This function is responsible for calling the *Tick* function for the Ms. Pac-Man controller (updating the next random action for Ms. Pac-Man) and to advance the game by one frame by calling the *AdvanceGame* function (see Steps 5 & 6).

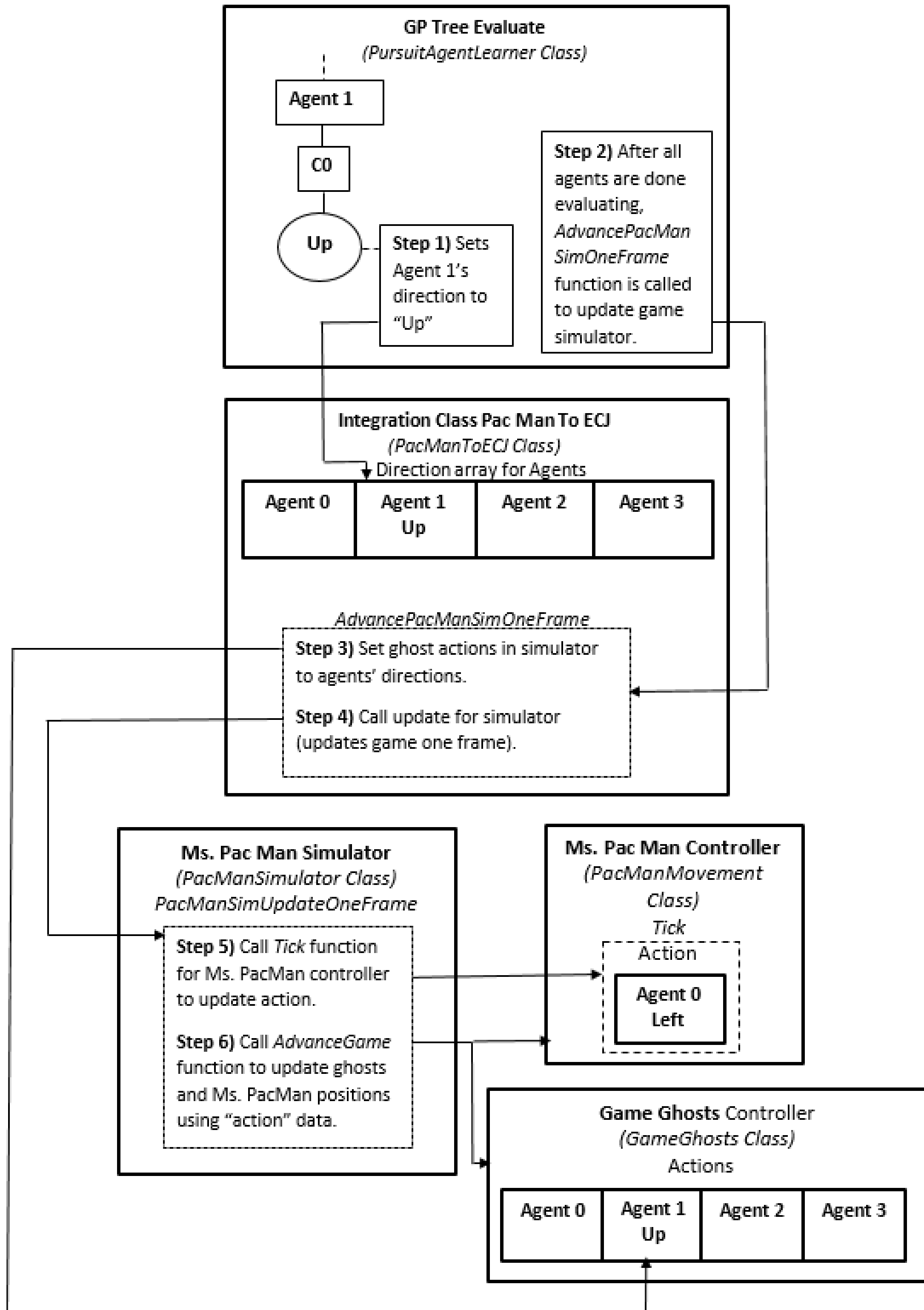


Figure 3.8: GP and Ms. Pac-Man System Integration
The evaluation of the GP tree updates one frame of the simulator

Chapter 4

Evolved Communication Protocols

4.1 Problem and Environment

This study continues the investigation of using GP to evolve emerged behaviours in a multi-agent system. Using the pursuit domain and a co-operative learning strategy, multiple predator agents are tasked to learn the meaning of a simple set of commands with the goal of first finding and then following a prey.

Different communication protocols are compared in two experiments, with each experiment defining a different type of movement for the prey. In the first experiment, *Prey Linear Movement*, all communication protocols are trained and tested in the pursuit domain environment. The prey, starting from a random position within its own start area, moves linearly (Up ↑) on the grid. In the second experiment, *Prey Random Movement*, the environment is the same however, the prey moves in a random pattern (Up, Down, Left or Right).

Experiments in this chapter use the Pursuit Domain Package (PDP) by Kok and Vlassis [19]. This environment consists of a grid in which agents are allowed to move to every adjacent cell. In this study a 20 x 20 grid is used, where 1 cell = 1 unit of distance. Predator agents have a field of view (FOV) = 2 and is measured using euclidean distance. The initial starting position of the prey is chosen at random and is confined to the prey's starting area on the grid. The prey's starting area, as well as each predator's starting area, are seen in Figure 4.1. The starting areas are arranged so that they don't overlap (similar to Iba [6] and Haynes *et al.* [35]). Agent 0 is shown as light blue, Agent 1 is shown as purple, Agent 2 is shown as dark blue, Agent 3 is shown as green and the Prey (triangle) is shown as orange. The green lines on the grid outline the confined starting areas of the prey and each agent. The yellow cells on the grid define each agent's FOV. All four predator agents and the prey can

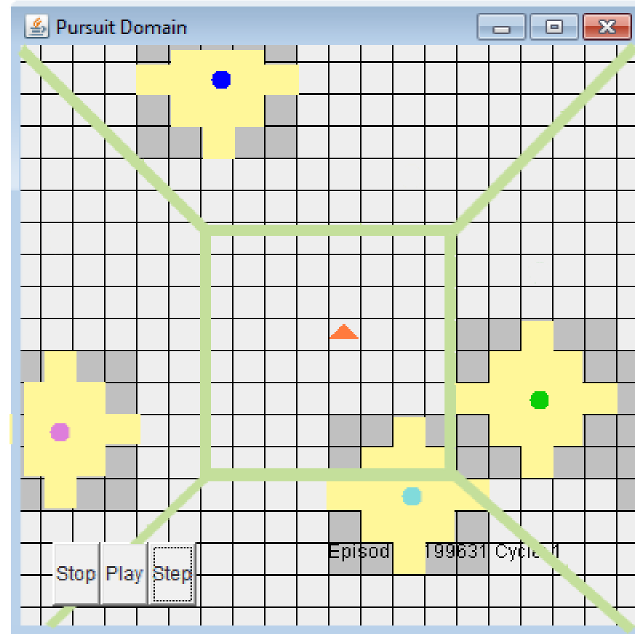


Figure 4.1: Pursuit Domain Environment

move to only one cell (Up, Down, Left, Right or Stay) in 1 time cycle. The grid is toroidal such that if the prey's (or any predator's) next move is outside of the grid boundary, then it will move to the next cell on the opposite edge (i.e. the movements are wrapped at the edges).

Results of this study show emergent behaviour in the top performing communication protocols in both experiments. Specifically, a synchronized alternating message sending pattern emerges from simple message passing among predator agents. In addition, the learned behaviour and collaboration of agents in the best result resembles the behaviour of guard and reinforcements that can be found in popular stealth video games (e.g. *Metal Gear Solid (MGS)*[36]).

4.1.1 Learning Strategy

The learning strategy is fully co-operative such that predator agents work as a heterogeneous team using a global fitness measure. The predator agents' common goal is to first find, and then follow the prey as closely as possible. A minimization function for the global fitness measure is used. This function calculates the total distance between all predator agents and the prey over a limited period. Using different communication protocols, the motivation for this fitness function is to compare how agents communicate to achieve their goal of tracking the prey.

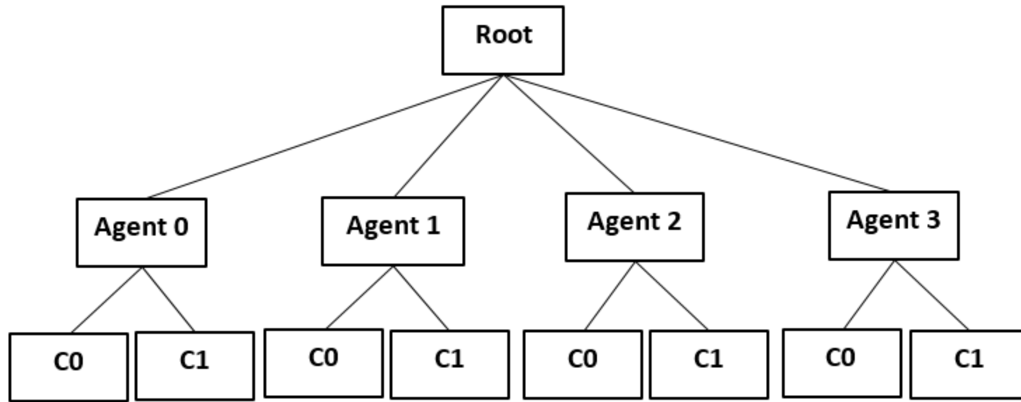


Figure 4.2: Top-level GP Structure

4.1.2 Communication Strategy and Communication Channel

The learned language in this study consists of two generic commands, *C0* and *C1*. A message passing communication channel is used to send the *C1* command, along with simple environment data, from one agent to another. Agents learn to associate a meaning to the *C0* and *C1* commands by evolving corresponding branches of its GP tree. Each agent has 2 child branches (command trees) where each branch is associated with one command. The first command, *C0*, is evaluated when the agent has no messages in its message buffer. The second command, *C1*, is evaluated when the agent has at least one message. Figure 4.2 shows the overall GP structure used in both experiments for this study.

The communication channel uses a message passing system in which each agent has its own message buffer queue. Upon evaluation of its command tree, an agent may send no messages, one message (with C1 command) or many messages (each with C1 command) to another agent. Once a message is sent it is placed in the receiving agent's message buffer. Before evaluation, if there is a message in the agent's buffer, the message is removed and a data variable for the agent is updated with the message data from the removed message. This data contains directional information about the prey if the sending agent was in FOV of they prey at the time the sending agent sent the message. A maximum number of 4 messages can be held in each message buffer. At the time of receiving a message, if an agent's message buffer is full then the first message (oldest message) in the buffer is removed and the newly received message is added to the end of the buffer.

The evaluation of all four agents occurs in 1 time cycle and each evaluation results in one movement (Up, Down, Left, Right or Stay) on the grid. The GP structure

consists of a heterogeneous team of agents such that each agent uses its own tree.

4.1.3 Communication Protocols

A communication protocol defines the method by which a sending agent sends a message to a receiving agent's message buffer. Seven different types of message passing are examined including agents sending messages individually, as a leader or as a member of a team. Table 4.1 defines the communication protocols. In this table, Agents 1 to 4 are shown as A0, A1, A2 and A3. The description of how message passing is accomplished for each communication protocol is as follows:

- ***Send22***: Two teams of two agents. A0 and A1 form one team and A2 and A3 form the other team. Each agent sends to its partner only.
- ***Send21***: Two teams of two agents. A0 and A1 form one team and A2 and A3 form the other team. A0 sends to A1, A2 sends to A3.
- ***SendLine***: Each agent sends to one other agent only (except for the last agent) in the form of a line such that A0 sends to A1, A1 sends to A2 and A2 sends to A3.
- ***SendLine2D***: Each agent sends to two other agents (except for the first and last agents) in the form of a line such that A0 sends to A1, A1 sends to A0 and A2, A2 sends to A1 and A3 sends to A2.
- ***Send13***: The first agent A0 (acting as a leader), sends to all other agents A1, A2 and A3.
- ***SendAll***: Each agent sends to every other agent.
- ***SendK***: A set of "send" commands (similar to the ones used by robot agents in Iba [6]) that allow the agent, A, to send to other agents based on proximity. *SendKN0* allows A to send to its nearest agent, *SendKN1* allows A to send to its second nearest agent, and *SendKN2* allows A to send to its farthest agent.

4.2 Experiment Details

This section describes the experiment details for two experiments, *Prey Linear Movement* and *Prey Random Movement*. Each experiment uses the same settings for the

Table 4.1: Communication Protocols

Communication Protocols	Method of Message Passing
Send22	A0 $\leftrightarrow$ A1 A2 $\leftrightarrow$ A3
Send21	A0 $\rightarrow$ A1 A2 $\rightarrow$ A3
SendLine	A0 $\rightarrow$ A1 $\rightarrow$ A2 $\rightarrow$ A3
SendLine2D	A0$\leftrightarrow$A1$\leftrightarrow$A2$\leftrightarrow$ A3
Send13	A0 $\rightarrow$ A1, A2, A3
SendAll	A0 $\rightarrow$ A1, A2, A3 A1 $\rightarrow$ A0, A2, A3 A2 $\rightarrow$ A0, A1, A3 A3 $\rightarrow$ A0, A1, A2
SendK (similar to Iba [6])	
SendKN0	A $\rightarrow$ nearest agent
SendKN1	A $\rightarrow$ 2nd nearest agent
SendKN2	A $\rightarrow$ farthest agent

GP parameters found in Table 4.2. The initial method to create trees uses Koza's *Ramped half-and-half* method made available by [30]. To create trees, the builder method picks GROW 50% of the time and FULL the other 50% of the time. It uses a range *(Min-Max Tree size ramp)* to determine the size of the tree. A random value within the range is picked and is used as the maximum tree size for the FULL approach and is used as the tree size for the GROW approach [30].

The fitness function, GP language, testing and training methods are also the same for the two experiments and are described below. The only difference between the two experiments lies in the movement of the prey. In the first experiment, *Prey Linear Movement*, the prey starts from a random position within its own start area and moves linearly (Up $\uparrow$) on the grid. In *Prey Random Movement*, the prey moves in a random pattern (Up, Down, Left or Right) on the grid. All agents move only one step per time cycle. Both experiments tested each communication protocol individually.

4.2.1 GP Language

The GP language is limited in order to allow high-level behaviours to emerge. The GP structure in Figure 4.2 is created using the strongly typed language from Table 4.3.

Table 4.2: GP Parameters

GP Parameter	Value
Initial Tree Method	Koza's *Ramped half-and-half* [30], [22]
Min-Max Tree size (ramp)	4-6
Population size	1000
Generations	125
Selection	Tournament, size = 4
Crossover	90%
Mutation	10%
Runs per experiment	20

Table 4.3: Strongly Type Language

ROOT	$::= (SIM, SIM, SIM, SIM)$
SIM	$::= CommandTree(EXPR, EXPR)$
EXPR	$::= Left \mid Right \mid Up \mid Down \mid Stay \mid$
	$::= IfGrtEql(NIL, NIL, EXPR, EXPR) \mid$
	$::= IfDist(NIL, NIL, EXPR, EXPR) \mid$
	$::= IfDot(NIL, NIL, EXPR, EXPR) \mid$
	$::= Send(EXPR)$
COM	$::= C1$
NIL	$::= Goal \mid AgentDir0 \mid AgentDir1 \mid AgentDir2 \mid$
	$::= Add(NIL, NIL) \mid$
	$::= Sub(NIL, NIL) \mid$
	$::= Rotate90(NIL) \mid$
	$::= S2(NIL) \mid$
	$::= S1/2(NIL) \mid$
	$::= Reverse(NIL) \mid$
	$::= North \mid South \mid East \mid West \mid$
	$::= LRM$

Terminal Set

The terminal set used is defined in Table 4.4. Movement commands (Up, Down, Left, Right and Stay) are sent directly to the agent as a result of the evaluation of its command tree. The language is typed such that only 1 movement is sent per evaluation. The direction vectors, North, South, West and East contain the unit vector of each direction. There are two communication commands, *C0 and C1*. *C0* is used as the default command if no messages have been sent to an agent and *C1* is the command used when messages are sent by agents. The Goal terminal gives the direction to the agent only if the agent is within field of view (FOV) of the

Table 4.4: Terminal Set

Name	Description
Up,Down,Left,Right,Stay	move commands: $\uparrow, \downarrow, \leftarrow, \rightarrow$
North,South,West,East	$(0,1),(0,-1),(-1,0),(1,0)$
C1	communication command
Goal	direction from prey to agent
	if agent is within FOV
AgentDir0-2 (similar to Iba [6])	direction from nearest(0))
	2nd nearest (1) and farthest agent(2)
LRM	Last Received Message

Table 4.5: Function Set

Function	Description
Root	returns the evaluated value of the entire tree
SimAgent	returns the evaluated value of one agent
Add	vector addition
Sub	vector subtraction
S2	scales vector by 2
S1/2	scales vector by 1/2
Rotate90	rotates vector by 90 degrees
Reverse	multiplies vector by -1
IfGrtEql	compares the length of two vectors
IfDot	calculates the dot product of two vectors
IfDist	checks if agent is withing FOV of prey
Send	sends a message to another agent
	(see Communication Protocols, Table 4.1)

prey, otherwise, it gives a default direction of (40,40). Directions to the nearest, 2nd nearest and farthest agent are given in AgentDir0, AgentDir1 and AgentDir2 respectively (similar to the ones used for robot navigation in Iba [6]). The terminal node to hold data for the last message removed from an agent's message buffer is named Last Received Message (LRM). Before evaluation of its tree, an agent checks its message buffer. If it contains messages, the first message is removed and its data is set to the LRM variable to be used for that evaluation cycle. If there are no messages in its buffer, the LRM node is set to the default vector (40,40).

Function Set

The function set is seen in Table 4.5. Functions include mathematical operations on two dimensional vectors, logical operations and message sending commands. Each logical operation consists of at least two child nodes [6]. The result of the logical

operation will evaluate only one of the child nodes. For *IfGrtEql*, if the length of the first input vector is $\geq$ to the length of the second input vector, then the third child node is evaluated, otherwise, the fourth child is evaluated. In *IfDot*, the result of the dot product of the first two input vectors is calculated. If $0 < result \leq 1$ then the third child node is evaluated, otherwise, the fourth child is evaluated. For *IfDist*, if the agent is within FOV of the prey, then the first child node is evaluated, otherwise, the second child is evaluated. Finally, the message sending command is listed as *Send*. In both experiments, all seven communication protocols, as listed in Table 4.1, are tested individually. For each test, the *Send* command is replaced with the specific communication protocol. For all communication protocols, when an agent issues a send command it first checks to see if it is within FOV of the prey. If it is within FOV, the message data sent to the receiving agent contains the direction from the receiving agent to the prey, otherwise, it contains a default value of (40,40).

4.2.2 Training and Testing Methods

The training and testing of a GP individual consists of cycles and episodes. Training and testing begin with the agents and prey starting in a random position within their own area on the grid (see Figure 4.1). In one cycle, all four agents evaluate their command tree once, one at a time. Each evaluation results in one movement of the agent, where one movement equates to one (cell) on the grid (and one unit in distance). After 30 cycles, one episode is complete (i.e. 1 episode = 30 cycles). In training, each GP individual is given 10 episodes and the positions of the agents/prey are reset to the original starting position after each episode is complete. The test run uses the GP individual with the best fitness in training. This GP individual is tested with 30 episodes instead of 10 and the test run sets a new random start position for each agent and the prey before a new episode begins.

4.2.3 Fitness Function

Fitness is measured by finding the sum of episode fitness scores. The episode fitness is the sum of each of the agent's distance to the prey in 30 cycles, where each cell on the grid represents 1 unit of distance. GP individuals with better fitness scores will minimize the distance sum as agents track (keep as close as possible to) the prey.

Equation (4.1) shows the total distance fitness calculation used in training:

$$TotDist = \sum_{k=1}^{q} \sum_{j=1}^{m} \sum_{i=0}^{3} \sqrt{(A_i.x - P.x)^2 + (A_i.y - P.y)^2} \quad (4.1)$$

Here, A_i represents the location of $Agent_i$, where $i = 0...3$, P is the location of the prey, m represents the number of cycles and q is the number of episodes. We set q to 10 in training and to 30 in testing in our experiments. Similar to Equation (4.1), the test run fitness measures the average distance of all the episodes as seen in Equation (4.2):

$$AveDist = \frac{TotDist}{q} \quad (4.2)$$

4.3 Results

This section compares the performance of all communication types. In order to better understand reasons for differences in performance, the top fitness of the best and worst communication type are analyzed for each prey movement type (linear and random). If significant differences are found in the performance between communication types, the data is further analyzed in search of emergent behaviour.

The training results for linear movement of the prey are seen in Figures 4.3, 4.4, and 4.5. These figures display the average adjusted fitness performance of each generation in 20 runs, with a total of 125 generations per run. The average fitness (best fitness in orange colour range and mean fitness in blue colour range) is shown for each communication type. In order to better identify best and worst performers in training fitness, Figure 4.4 shows two of the top performers (SendAll and Send22) and Figure 4.5 shows both the best performer (SendAll) and the worst performer (SendK) for linear movement of the prey.

The training results for random movement of the prey are seen in Figure 4.6. This figure displays the average fitness performance of each generation in 20 runs, with a total of 125 generations per run. The average fitness (best fitness in orange colour range and mean fitness in blue colour range) is shown for each communication type.

Table 4.6 displays the performance of test runs for a linear moving prey. It shows the best individual GP found in each training run by listing the minimum fitness, the maximum fitness, and the average fitness of 20 test runs. In the same manner, Table 4.7 displays the performance of test runs for a random moving prey.

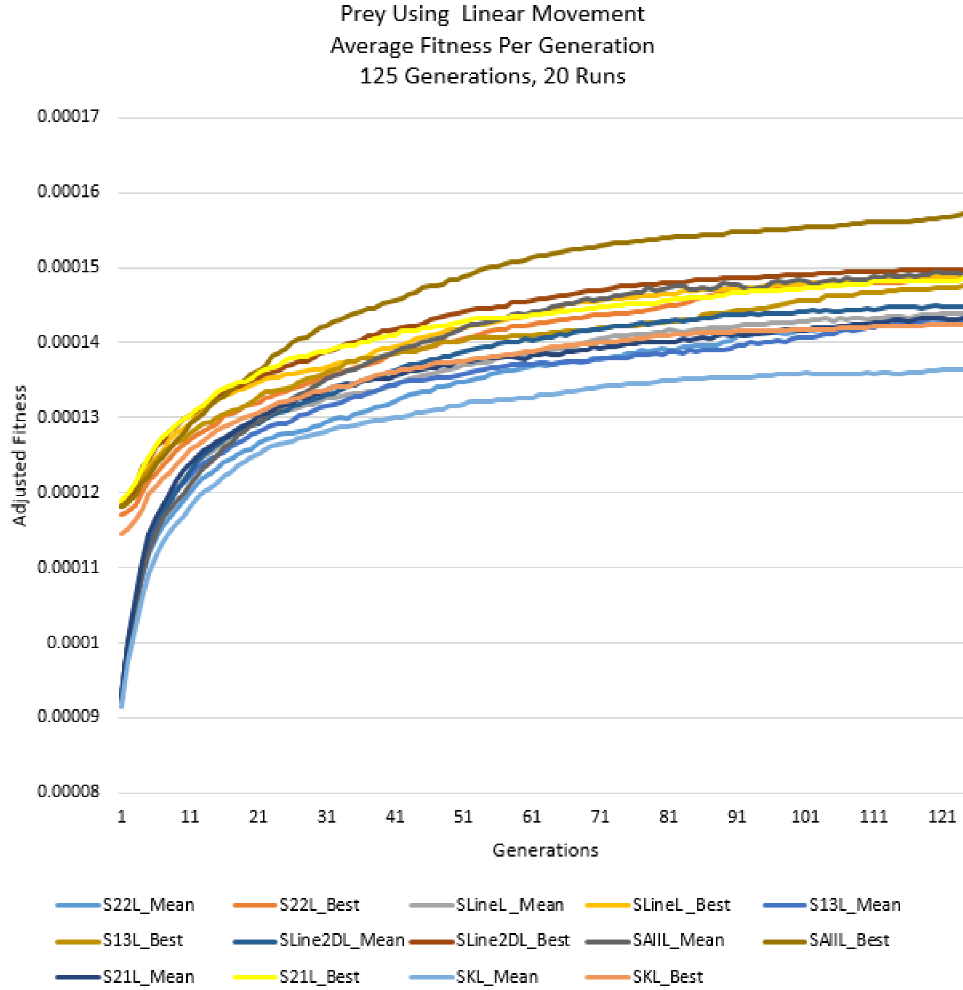


Figure 4.3: Prey Linear Movement - Training Fitness for all Communication Types *SendAll, SendLine, SendLine2D, Send22, Send21, Send13, and SendK*

Table 4.6: Test Fitness Summary Linear Prey (20 runs).

Prey Movement	Communication Type	Min Fitness of 20 runs	Ave Fitness of 20 runs	Max Fitness of 20 runs
Linear	SendAll	707	788	852
	Send22	727	789	878
	Send21	713	801	876
	Send13	712	803	849
	SLine2D	747	806	875
	SLine	728	804	920
	SendK	783	827	868

Fitness is a minimization function. See Equation (4.2).

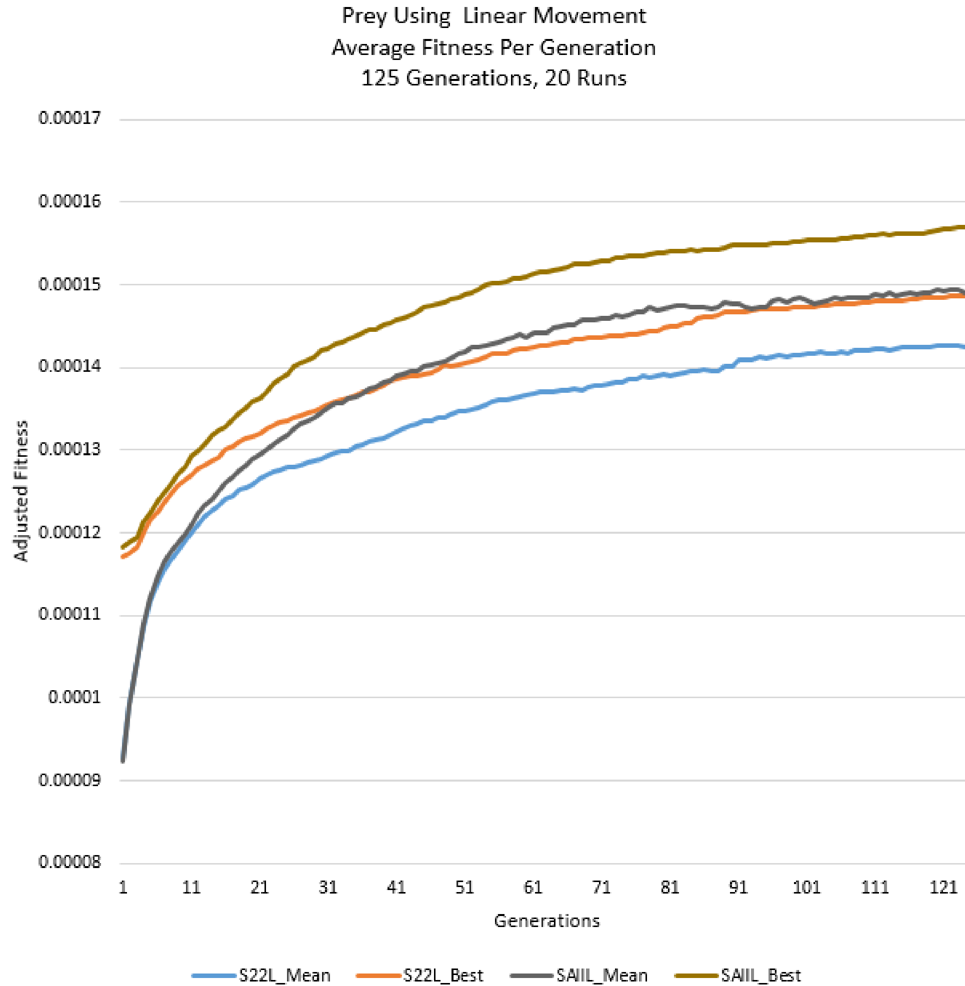


Figure 4.4: Prey Linear Movement - Training Fitness for Top Performers
SendAll and Send22

4.3.1 Statistical Analysis

The average test fitness values in Table 4.6 reveal that for linear movement of prey, SendAll and Send22 are the top performers with scores of 788 and 789 respectively, and SendK is the worst performer with a score of 827. The results for the random movement of the prey in Table 4.7 show that there are small differences in the average fitness values across all communication protocols with an average fitness range from 720 to 739. Again SendAll is seen as the top performer with a score of 720 and interestingly, Send22 is the worst performer with a score of 739. To verify the significance of the results, the One-Way ANOVA test (using Minitab [37]) with a 95% confidence interval is used. The ANOVA test uses a two-tailed T-test with seven factors, where each factor represents one communication protocol including the final test fitness for

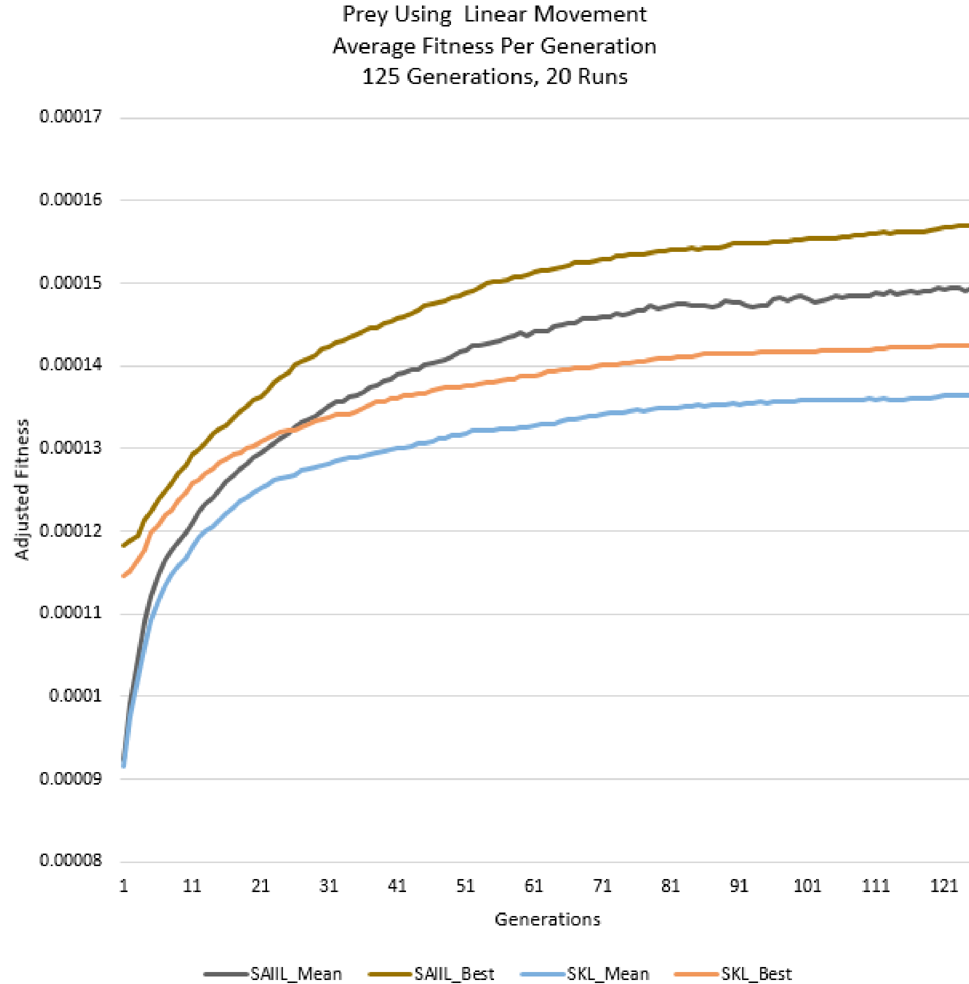


Figure 4.5: Prey Linear Movement - Training Fitness for Best and Worst Performers *SendAll and SendK*

each of the 20 test runs. The ANOVA test results for linear movement of the prey show that the $P - Value < \alpha$ (see Appendix A) indicating that there is a significant difference in the fitness results. The ANOVA test results for random movement of the prey show that the $P - Value > \alpha$ (see Appendix A) indicating that there is a not a significant difference in the fitness results for random movement of prey.

To identify which factors have significantly different means in the linear movement of the prey, the Tukey method [38] for multiple comparisons is used for the communication protocols with results shown in Table 4.8. In these tables, protocols which do not share the same group letter indicate that their range of difference of mean does not contain a zero. Thus, protocols labelled with different letter groups are considered to be significantly different [39]. The top performers (SendAll and Send22) and the worst performer (SendK) for linear movement of prey do not share

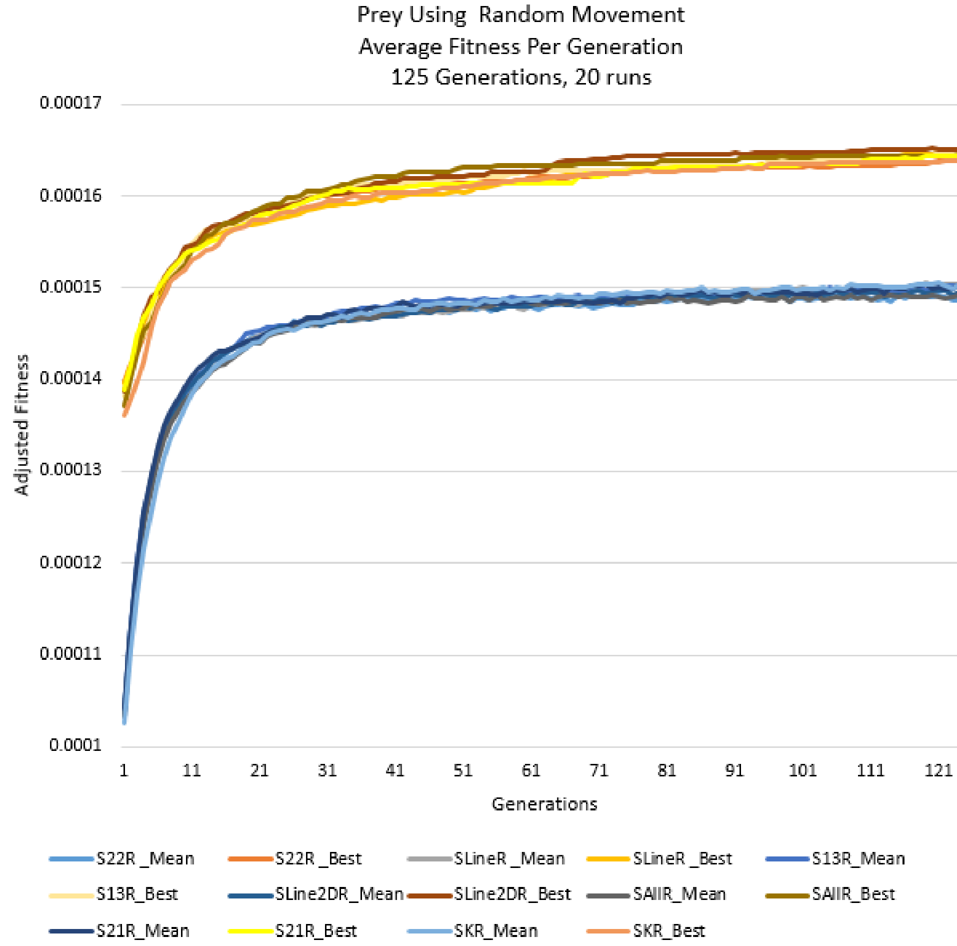


Figure 4.6: Prey Random Movement - Training Fitness for all Communication Types *SendAll, SendLine, SendLine2D, Send22, Send21, Send13, SendK*

Table 4.7: Test Fitness Summary Random Prey (20 runs).

Prey Movement	Communication Type	Min Fitness of 20 runs	Ave Fitness of 20 runs	Max Fitness of 20 runs
Random	SendAll	638	720	776
	Send21	683	729	805
	SLine2D	682	730	763
	SLine	674	735	774
	SendK	672	735	772
	Send13	673	735	767
	Send22	698	739	803

Fitness is a minimization function. See Equation (4.2).

the same letter group. This reveals that there is a significant difference between the top performers and the worst performer.

Table 4.8: Tukey Comparisons for Communication Protocols (Linear Prey)

Communication Protocol	Group Letter	Average Fitness
SendK	A	827
SendLine2D	AB	806
SendLine	AB	804
Send13	AB	803
Send21	AB	801
Send22	B	789
SendAll	B	788

Protocols that do not share the same letter are significantly different

Table 4.9: Tukey Comparisons for Communication Protocols (Random Prey)

Communication Protocol	Group Letter	Average Fitness
Send22	A	739
SendLine	A	735
SendK	A	735
Send13	A	735
SendLine2D	A	730
Send21	A	729
SendAll	A	720

Protocols that do not share the same letter are significantly different

Table 4.9 shows the Tukey method for ANOVA test results in the random movement of the prey. This table verifies that there is not a significant difference in the results for random movement of the prey because all protocols share the same letter group.

4.3.2 Emergent Behaviour

The results show that most tests evolved competent agents that are able to find and follow the prey. Many of the communication protocols did not produce significant differences in fitness scores or perceived behaviours. However, some experiments did regularly evolve interesting behaviours that show high-levels of coordination among agents. These behaviours are highlighted in the remainder of this section.

Table 4.10: Message Sending Patterns (Prey Linear Movement)

SendAll Run 14

Sender		*Receivers*	*Description*
A0	→	*A1, A2, A3*	*(rarely sends)*
A1	→	*A0, A2, A3*	*(sends with A3 every other cycle)*
A2			*(never sends)*
A3	→	*A0, A1, A2*	*(sends with A1 every other cycle)*

Send22 Run 15

Sender		*Receivers*	*Description*
A0			*(never sends)*
A1			*(never sends)*
A2	→	*A3*	*(sends every other cycle)*
A3	→	*A2*	*(sends every other cycle)*

Table 4.11: SendAll Staircase Pattern: Agents Message Buffer Contents

SendAll(Linear) Test Run 14, Cycles 9-13

Cycle	From Agent	Agent 0 Message LRM	From Agent	Agent 1 Message LRM	From Agent	Agent 2 Message LRM	From Agent	Agent 3 Message LRM
9	1	(40,40)			1	(40,40)		
10	3	(40,40)	3	(40,40)	3	(40,40)	1	(40,40)
					1	(40,40)		
11	1	(40,40)			1	(40,40)		
12	3	(40,40)	3	(40,40)	3	(40,40)	1	(1,-6)
					1	(3,-11)		
13	1	(-4,-9)			1	(3,-11)	1	(1,-5)
					1	(3,-10)		

Synchronized Alternating Sending Pattern

Table 4.10 describes the coordination of message passing among top individuals of the SendAll and Send22 communication protocols. It was created by examining the contents of the message buffers before each agent evaluated their command tree. This table demonstrates that SendAll (Run 14) has two agents (Agents 1 and 3) that almost always send out messages to all other agents and has two agents (Agents 0 and 2) that rarely/never send messages. Table 4.11 shows the message buffer data for this run. This data shows that Agents 1 and 3 coordinate their message passing by synchronizing their “sends” so that they send a message to all other agents on the same cycle, every other cycle. In this table it is seen that Agent1 and Agent 3 have a message from each other every other cycle.

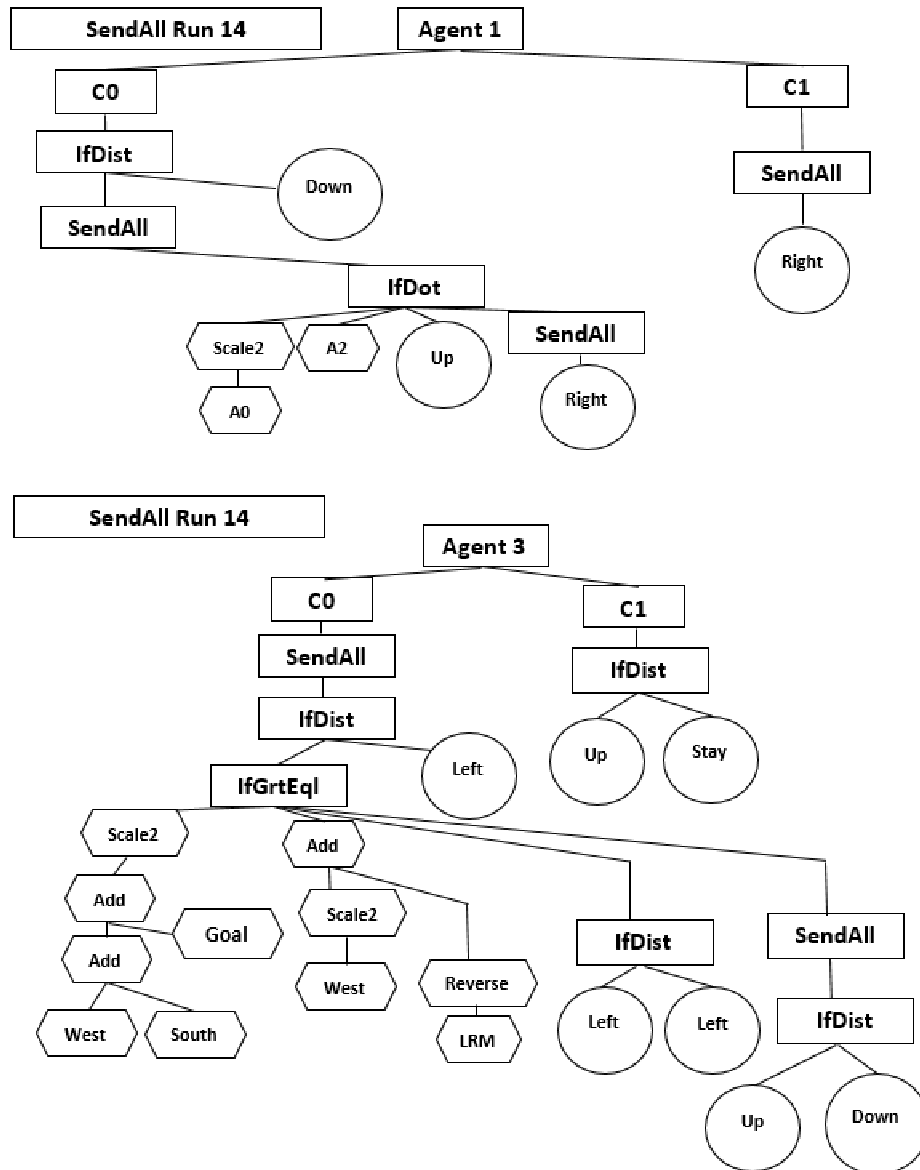


Figure 4.7: Staircase Pattern: SendAll Agent 1 & 3's branches
SendAll Run 14

This synchronization causes Agents 1 and 3 to alternate the evaluation of their C0 and C1 branch each cycle. This coordination is an advantage for these agents because the required movement nodes (to achieve their goal of finding the prey) are divided between both branches.

For example, looking at the GP sub-tree structures for Agent 1 and Agent 3 in Figure 4.7 for Run 14, it is seen that in Agent 1's C0 branch, it has nodes to move Up, Down, and Right, and in its C1 branch has a node to move Right. Based on Agent's

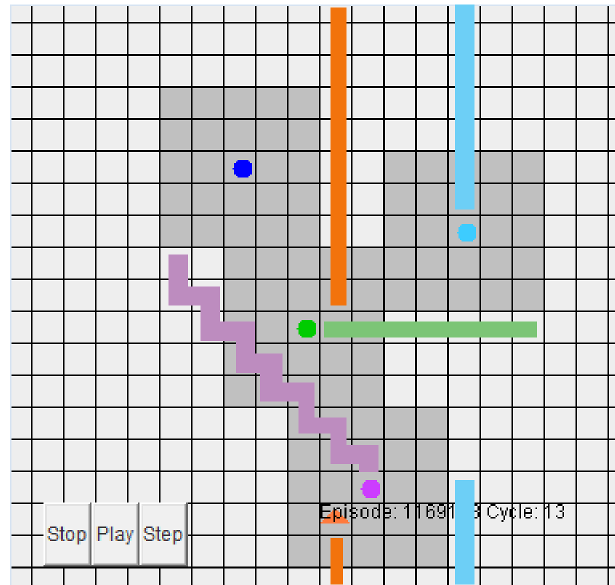
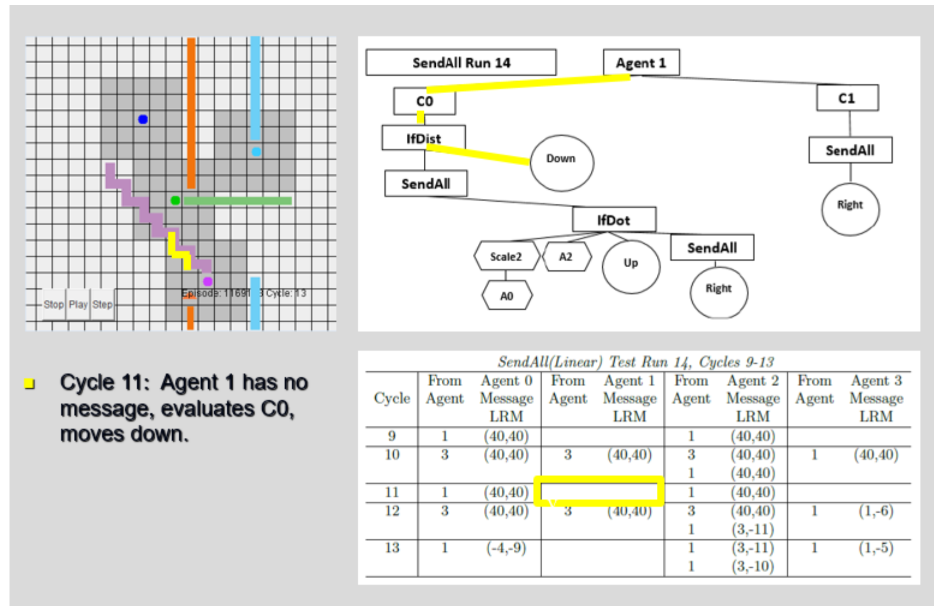


Figure 4.8: Staircase Pattern: SendAll (Agent 1 (purple))

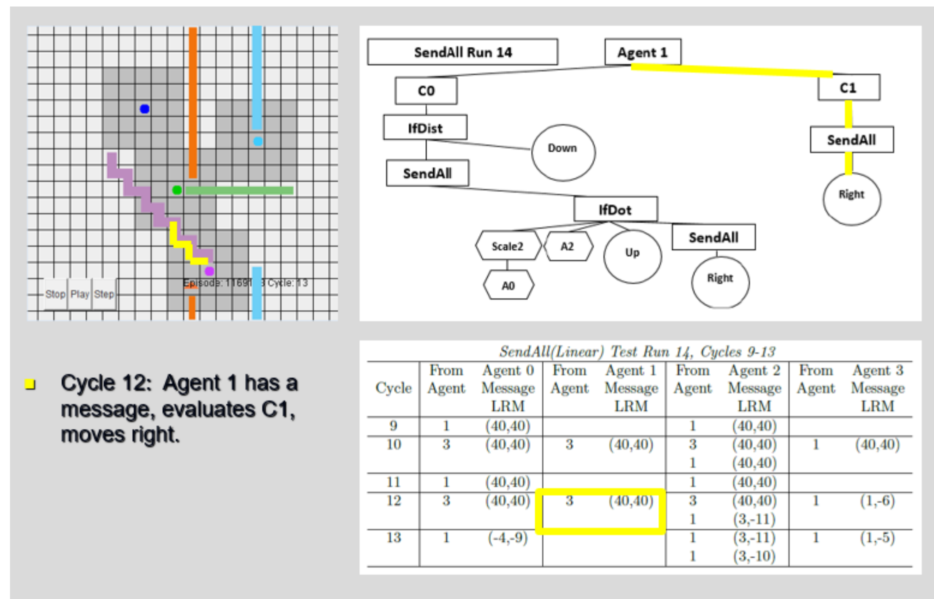
1 starting position area in Figure 4.1, Agent 1 must learn to move Right (towards center of grid) in order to find the prey, and it must learn to move Up in order to track the prey. The alternating pattern of message receiving allows Agent 1 to use two directions to find the prey. Switching the evaluation of its C0 and C1 branches each cycle, causes Agent 1 to move Right (C1 branch) then Down (C0 branch). This synchronization causes Agent 1 to move in a distinct staircase pattern until it finds the prey as seen in Figure 4.8. Once it finds the prey it is able to move in the Up direction to follow the prey (not shown). Figures 4.9 and 4.10 give a good example of how Agent 1 alternates the evaluation of its C0 and C1 branches to move in a staircase pattern to find the prey.

Table 4.10 also shows the message sending pattern of all agents in Send22 (Run 15). This data shows that Agents 0 and 1 never send messages to each other. However, it does show a synchronized alternating sending pattern between Agent 2 and Agent 3. Table 4.12 shows the message buffers for this run. In this table it is seen that Agent 2 and Agent 3 use alternate cycles to send messages to each other. In turn, this causes Agents 2 and 3 to evaluate their C0 or C1 branch every other cycle. Similar to previous tests, this causes Agent 3 to move in a distinct staircase pattern until it finds the prey. The staircase pattern for this run is shown in Figure 4.11. Once Agent 3 finds the prey it is able to move in the Up direction to follow the prey (not shown).



Cycle	From Agent	Agent 0 Message LRM	From Agent	Agent 1 Message LRM	From Agent	Agent 2 Message LRM	From Agent	Agent 3 Message LRM
9	1	(40,40)			1	(40,40)		
10	3	(40,40)	3	(40,40)	3 1	(40,40) (40,40)	1	(40,40)
11	1	(40,40)			1	(40,40)		
12	3	(40,40)	3	(40,40)	3 1	(40,40) (3,-11)	1	(1,-6)
13	1	(-4,-9)			1 1	(3,-11) (3,-10)	1	(1,-5)

Figure 4.9: Staircase Pattern: SendAll Cycle 11



Cycle	From Agent	Agent 0 Message LRM	From Agent	Agent 1 Message LRM	From Agent	Agent 2 Message LRM	From Agent	Agent 3 Message LRM
9	1	(40,40)			1	(40,40)		
10	3	(40,40)	3	(40,40)	3 1	(40,40) (40,40)	1	(40,40)
11	1	(40,40)			1	(40,40)		
12	3	(40,40)	3	(40,40)	3 1	(40,40) (3,-11)	1	(1,-6)
13	1	(-4,-9)			1 1	(3,-11) (3,-10)	1	(1,-5)

Figure 4.10: Staircase Pattern: SendAll Cycle 12

Guard Behaviour through Collaboration

A behaviour found in the video game series *Metal Gear Solid (MGS)*[36] is a guard protecting an area. Generally, the guard protects an area by remaining in a defined area. Once the guard spots an intruder, it begins to follow the intruder and notify other guards for reinforcement. Upon notification, reinforcement guards track (and attack) the intruder. The agents from the best test run in the *Prey Linear Movement*

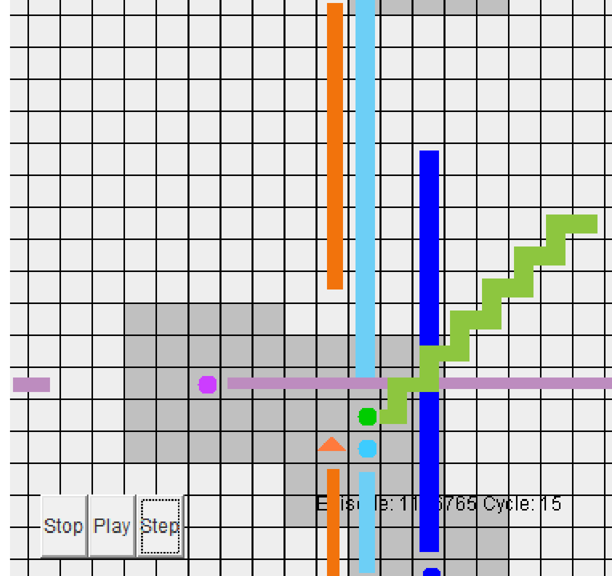


Figure 4.11: Staircase Pattern: Send22 (Agent 3 (green))

Table 4.12: Send22 Staircase Pattern: Agents Message Buffer Contents

Send22(Linear) Test Run 15, Cycles 11-18

Cycle	From Agent	Agent 0 Message LRM	From Agent	Agent 1 Message LRM	From Agent	Agent 2 Message LRM	From Agent	Agent 3 Message LRM
11							2	(40,40)
12					3	(40,40)		
13							2	(40,40)
14					3	(40,40)		
15							2	(40,40)
16					3	(-3,4)		
17							2	(40,40)
18					3	(-3,-12)		

experiment for the SendAll communication protocol, evolved a simple form of this guard and reinforcement behaviour.

In the best test run (Run 15), predator agents collaborate through message sending and learn to rely on data sent by other agents to find and follow the prey. Table 4.13 shows the message sending pattern for all four agents in this run. This table was created by examining the contents of the message buffers before each agent evaluated its command tree. It is seen that Agent 0 (A0) and Agent 3 (A3) send a message to all other agents almost every cycle, Agent 2 (A2) sends multiple messages at once when it is in view of the prey, and Agent 1 (A1) does not send any messages at all.

Additionally, it is seen in Figure 4.12 that Agent 2's C1 branch does not send a

Table 4.13: Guard Behaviour Message Sending Pattern

SendAll(Linear) Run 15

Sender		*Receivers*	*Description*
$A0$	$\rightarrow$	$A1, A2, A3$	*(always sends)*
$A1$			*(never sends)*
$A2$	$\rightarrow$	$A0, A1, A3$	*(multiple sends when in view of prey)*
$A3$	$\rightarrow$	$A0, A1, A2$	*(always sends)*

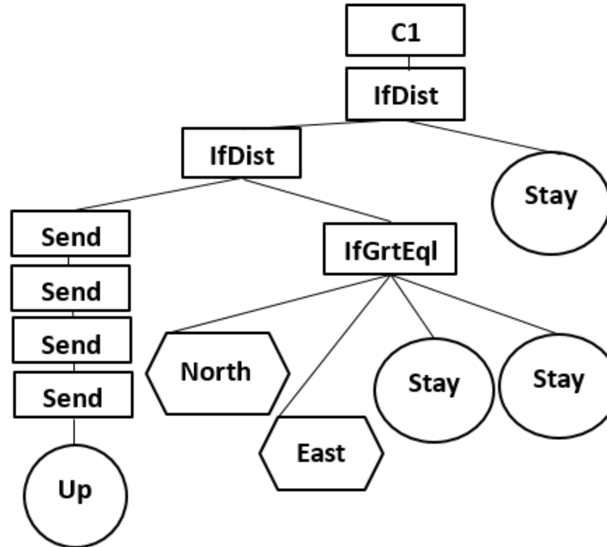


Figure 4.12: Guard Behaviour GP: Example of Agent 2's C1 branch
Agent 2 sends multiple messages to all Agents when it sees the prey.

message and does not move (Stays) if it is not in view of the prey. However, if it is in view of the prey, Agent 2 sends out four messages at once to every other agent. Because Agent 2 is in view of the prey, the data in the message contains the direction to the prey relative to the receiving agent.

Table 4.14 shows the message buffers for this run. It is seen that the message buffers for Agents 0, 1 and 3 contain messages from Agent 2 (Cycles 5-17) and the message data contains directional information to the prey.

The C1 branch for Agent 1 is seen in Figure 4.13. Upon evaluation of Agent 1's C1 branch, the LRM node will be updated with directional information to the prey (from the message data). This branch shows that, using the IfDist expression, if Agent 1 is not within FOV of the prey it uses the LRM node with the IfDot expression to decide to move Right or Left. If it is within FOV of the prey then it moves Up (to follow the prey). An example of this is demonstrated in Figures 4.14 and 4.15.

The guard and reinforcement behaviour is shown in the cycle time line found in

Table 4.14: Guard Behaviour: Agents Message Buffer Contents

SendAll(Linear) Run 15, Cycles 4-17

Cycle	From Agent	Agent 0 Message LRM	From Agent	Agent 1 Message LRM	From Agent	Agent 2 Message LRM	From Agent	Agent 3 Message LRM
4	3	(40,40)	0	(40,40)	0	(40,40)	2	(-7,6)
	3	(40,40)	3	(40,40)	3	(40,40)	2	(-7,6)
	3	(40,40)	3	(40,40)	3	(40,40)	2	(-7,6)
	3	(40,40)	0	(40,40)	0	(40,40)	2	(-7,6)
5	2	(3,-5)	2	(8,7)	0	(40,40)	2	(-6,7)
	2	(3,-5)	3	(40,40)	3	(40,40)	2	(-6,7)
	3	(40,40)	3	(40,40)	3	(40,40)	2	(-6,7)
	3	(40,40)	0	(40,40)	0	(40,40)	2	(-6,7)
12	2	(3,-11)	2	(1,-6)	0	(40,40)	2	(1,-6)
	2	(3,-11)	3	(40,40)	3	(40,40)	2	(1,-6)
	3	(40,40)	3	(40,40)	3	(40,40)	2	(1,-6)
	3	(40,40)	0	(40,40)	0	(40,40)	2	(1,-6)
13	2	(3,-9)	2	(0,-5)	0	(40,40)	2	(1,-5)
	2	(3,-9)	3	(40,40)	3	(40,40)	2	(1,-5)
	3	(40,40)	3	(40,40)	3	(40,40)	2	(1,-5)
	3	(40,40)	0	(40,40)	0	(40,40)	2	(1,-5)
15	2	(3,-5)	2	(0,-3)	0	(40,40)	2	(1,-3)
	2	(3,-5)	3	(40,40)	3	(40,40)	2	(1,-3)
	3	(40,40)	3	(40,40)	3	(40,40)	2	(1,-3)
	3	(40,40)	0	(40,40)	0	(40,40)	2	(1,-3)
17	2	(3,-1)	2	(2,-1)	0	(40,40)	2	(1,-1)
	2	(3,-1)	3	(40,40)	3	(40,40)	2	(1,-1)
	3	(40,40)	3	(40,40)	3	(40,40)	2	(1,-1)
	3	(40,40)	0	(40,40)	0	(40,40)	2	(1,-1)

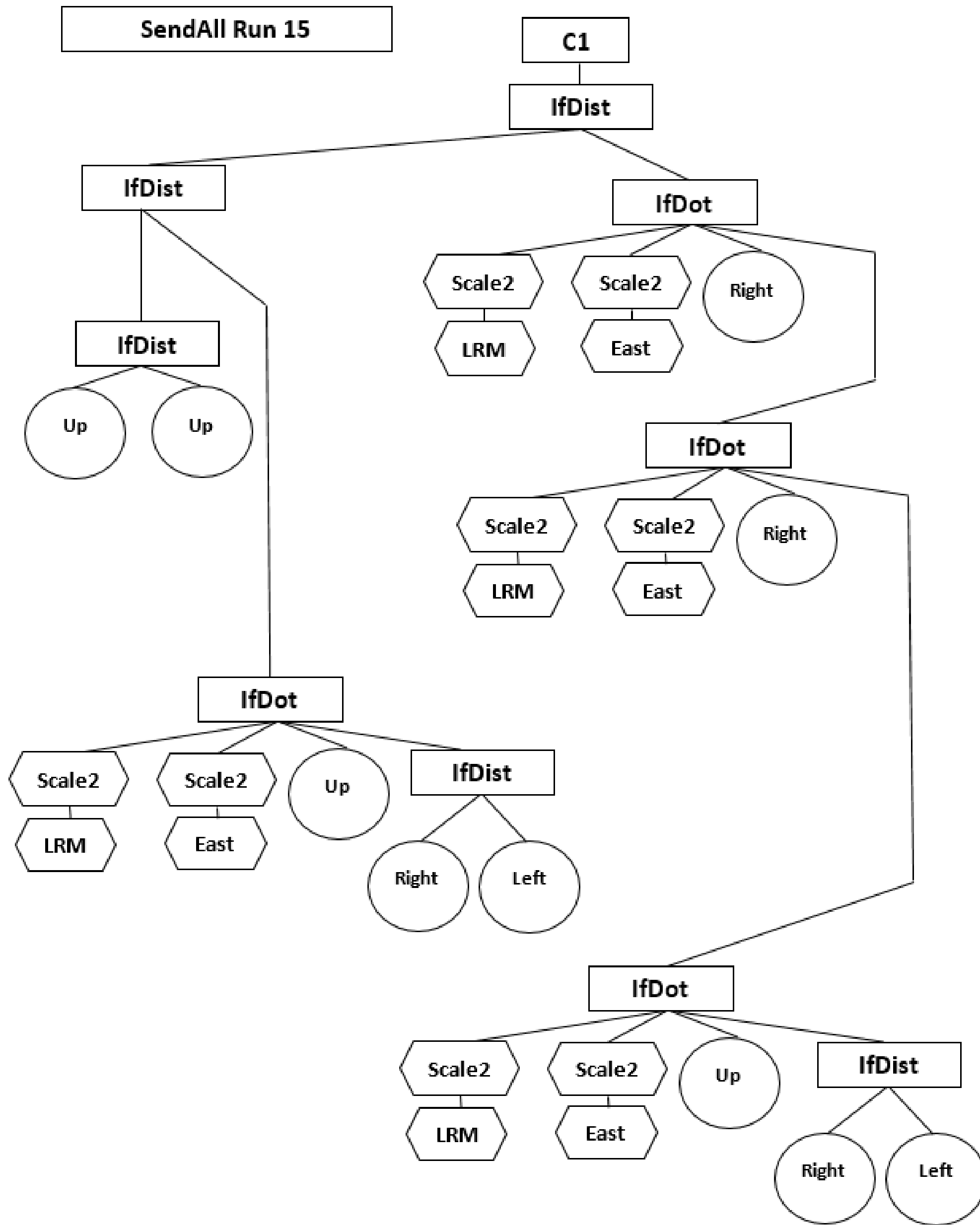
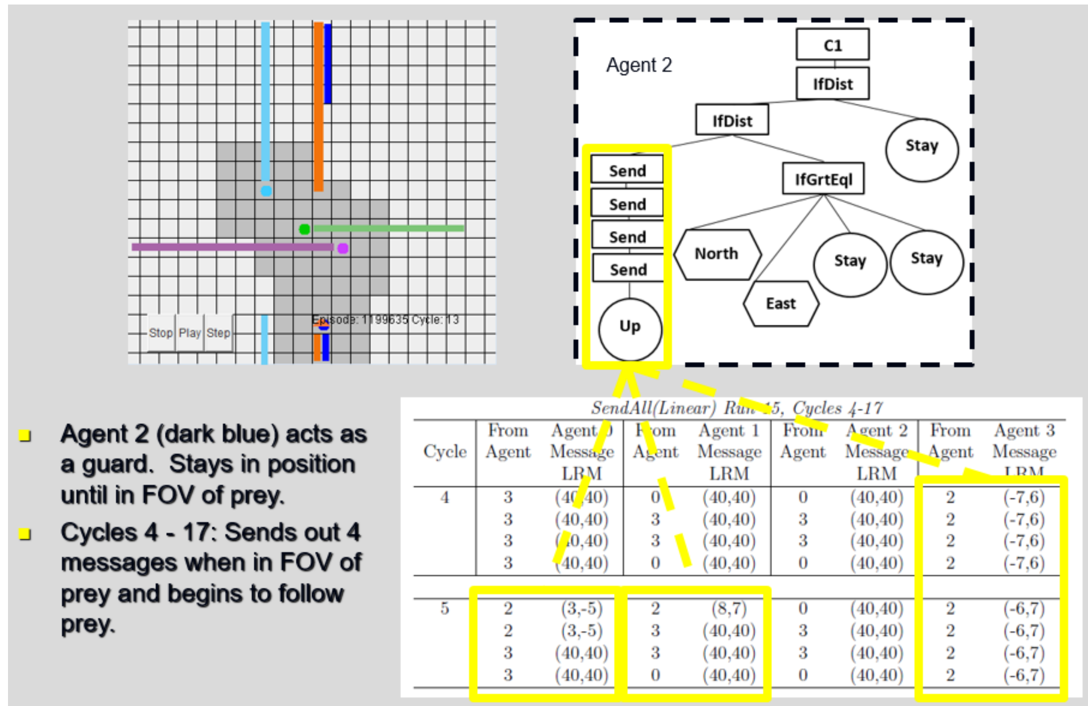


Figure 4.13: Guard Behaviour GP: Example of Agent 1's C1 branch
Agent 1 uses the LRM node to decide to move Right or Left.

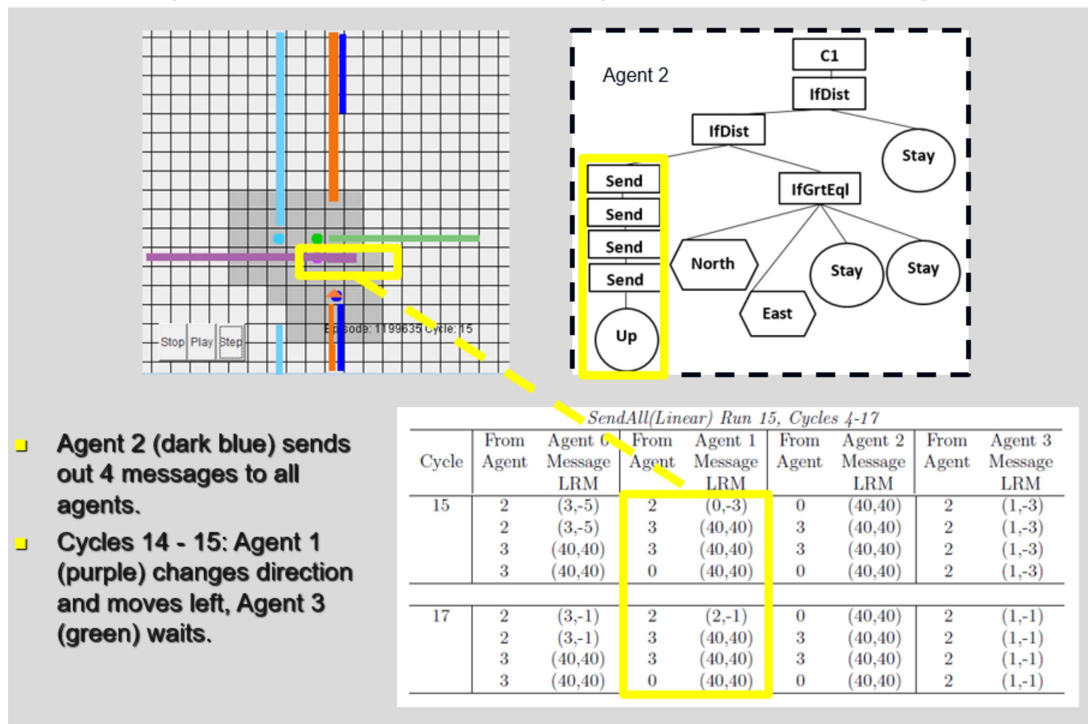
Table 4.15 and Figures 4.16, 4.17 and 4.18. Agent 2 (dark blue) acts a "guard" and "stays" until it views the prey. When it sees the prey it sends multiple messages to all other agents. These messages help Agent 1(purple) find and follow the prey. Initially, Agent 1 is seen moving Right in Cycles 0-12. After Cycle 13, Agent 1 begins to move Left (see Figures 4.17 & 4.18). Therefore, it seems that Agent 1 is using the LRM data in its message buffer to decide to move Right or Left and once it is within view of the prey it moves Up to follow the prey (not shown).



SendAll(Linear) Run 15, Cycles 4-17

Cycle	From Agent	Agent 0 Message LRM	From Agent	Agent 1 Message LRM	From Agent	Agent 2 Message LRM	From Agent	Agent 3 Message LRM
4	3	(40,40)	0	(40,40)	0	(40,40)	2	(-7,6)
	3	(40,40)	3	(40,40)	3	(40,40)	2	(-7,6)
	3	(40,40)	3	(40,40)	3	(40,40)	2	(-7,6)
	3	(40,40)	0	(40,40)	0	(40,40)	2	(-7,6)
5	2	(3,-5)	2	(8,7)	0	(40,40)	2	(-6,7)
	2	(3,-5)	3	(40,40)	3	(40,40)	2	(-6,7)
	3	(40,40)	3	(40,40)	3	(40,40)	2	(-6,7)
	3	(40,40)	0	(40,40)	0	(40,40)	2	(-6,7)

Figure 4.14: Guard Behaviour: Agent 2 Sends out Messages.



SendAll(Linear) Run 15, Cycles 4-17

Cycle	From Agent	Agent 0 Message LRM	From Agent	Agent 1 Message LRM	From Agent	Agent 2 Message LRM	From Agent	Agent 3 Message LRM
15	2	(3,-5)	2	(0,-3)	0	(40,40)	2	(1,-3)
	2	(3,-5)	3	(40,40)	3	(40,40)	2	(1,-3)
	3	(40,40)	3	(40,40)	3	(40,40)	2	(1,-3)
	3	(40,40)	0	(40,40)	0	(40,40)	2	(1,-3)
17	2	(3,-1)	2	(2,-1)	0	(40,40)	2	(1,-1)
	2	(3,-1)	3	(40,40)	3	(40,40)	2	(1,-1)
	3	(40,40)	3	(40,40)	3	(40,40)	2	(1,-1)
	3	(40,40)	0	(40,40)	0	(40,40)	2	(1,-1)

Figure 4.15: Guard Behaviour: Agent 1 uses LRM data.

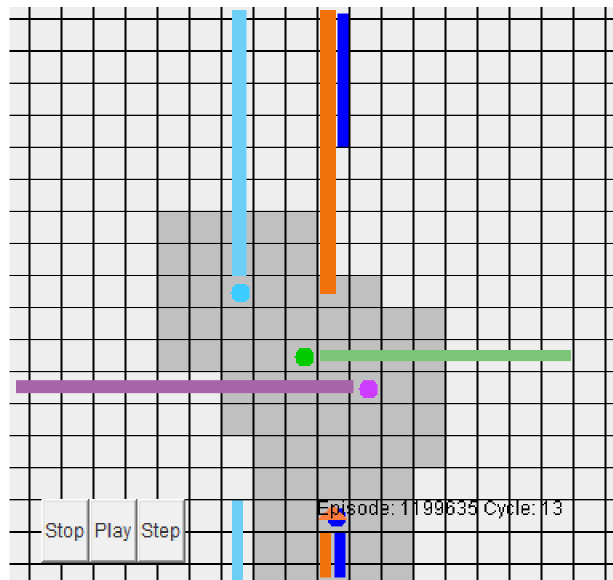


Figure 4.16: Guard Behaviour Cycle 13.
Coloured lines show the path of the agent from Cycle 0-13. Agent 2 (dark blue) is in FOV of prey and sends messages notifying agents of the prey's location. At Cycle 13, Agent 1 (purple) using LRM data, changes direction and begins to move left.

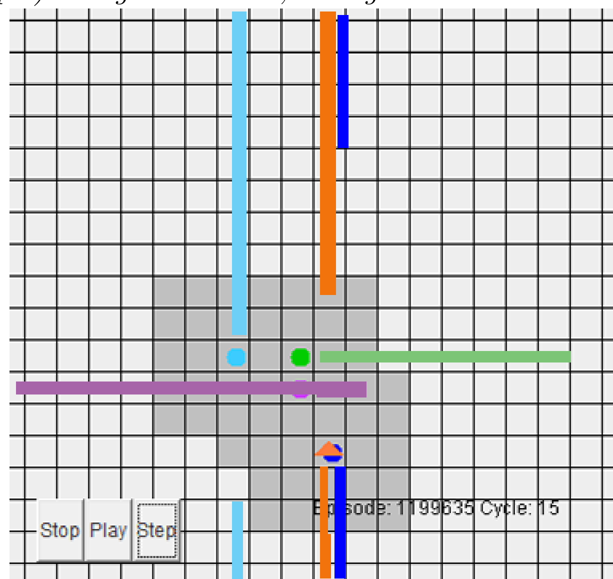


Figure 4.17: Guard Behaviour Cycle 15.
Coloured lines show the path of the agent from Cycle 0-15. Using LRM data sent from Agent 2 (dark blue), Agent 1 (purple) stops moving left and waits for prey.

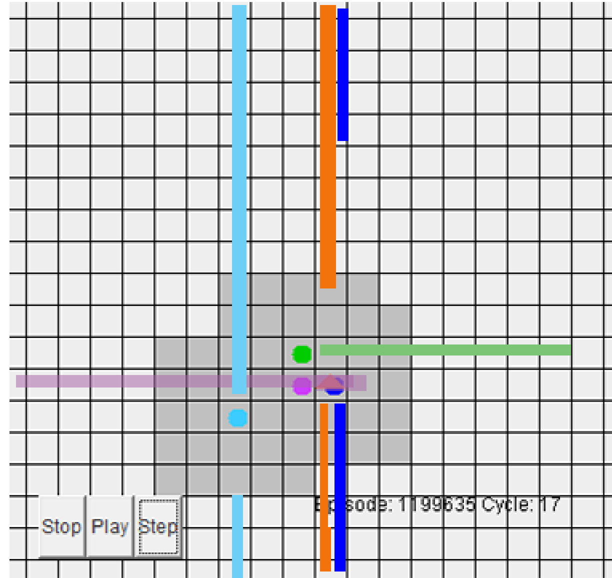


Figure 4.18: Guard Behaviour Cycle 17.
Coloured lines show the path of the agent from Cycle 0-17.
3 agents are in FOV of prey and follow prey in the next cycles (not shown).

Table 4.15: Guard Behaviour Time Line

Time Line Event	Cycle(s)	Description
0 (not shown)	0-4	*Agent 0 moves Down,* *Agent 1 moves to the Right,* *Agent 2 waits in position,* *Agent 3 moves to the Left*
1 (not shown)	5	*Agent 2 enters FOV of prey, floods all other agents' message buffers with prey location data*
2 (not shown)	5-12	*Agent 0 continues to move Down,* *Agent 1 continues to moves to the Right,* *Agent 2 follows prey and sends messages,* *Agent 3 continues to move to the Left*
3 (see Fig. 4.16)	13	*Using LRM data from Agent 2* *Agent 1 begins to move Left and Agent 3 Stays,* *Agent 0 continues to move Down,* *Agent 2 follows prey and sends messages*
4 (see Fig. 4.17)	15	*Using LRM data from Agent 2* *Agent 1 and 3 wait to be in FOV of prey,* *Agent 0 continues to move Down,* *Agent 2 follows prey and sends messages*
5 (see Fig. 4.18)	17	*Agent 1, 2 and 3 are all in FOV* *of prey and follow prey in the Up direction*

Table 4.16: SendK Linear: GP Agents Possible Moves for Best Run (Run 20)

	Agent 0		Agent 1		Agent 2		Agent 3	
SendK	C0	C1	C0	C1	C0	C1	C0	C1
Run 20	D S	U D	R	U S	U D R S	U S	L R S	L R

U = Up, D = Down, L = Left, R = Right or S = Stay

Table 4.17: SendK Message Sending Pattern, Cycles 14-21

Sender		*Receivers*	
$A0$	$->$	$SendKN0, SendKN2$	*A2, A3*
$A1$	$->$	$SendKN0$	*A2*
$A2$	$->$	$SendKN0, SendKN1$	*A0,A1*
$A3$	$->$	$SendKN0, SendKN1, SendKN1$	*A0,A1,A2*

Agent 0 (A0), Agent 1 (A1), Agent 2 (A2), Agent 3 (A3)
The sender's receiver depends on the closest (SendKN0),
2nd closest (SendKN1), and farthest agent (SendKN2) at send time.

Prey Linear Movement Worst Performer: SendK

It has been shown that the SendK protocol is the worst performer when the prey moves in a linear pattern. This section analyzes the best run (Run 20) of the SendK protocol to determine reasons for its poor performance.

Table 4.16 shows the possible movements for each agent based off the GP tree for Run 20. This data shows that Agent 0's C0 (no message) branch and C1 (message) branch allows for the possibility for movement in the Down/Stay (C0) and Up/Down(C1) directions. This is good because due to its start position, Agent 0 must move in the Up or Down direction in order to find and follow the prey in its linear movement pattern. Similarly, the GP structures for the remaining agents show that they all have the possibility of moving in their required directions. Thus, based off the GP structure alone, all agents in SendK's best run have the possibility of moving in the proper direction to achieve their goal of finding and following the prey.

Table 4.17 shows the message sending pattern for all four agents for Run 20. This table was created by examining the GP structure of the agents for Run 20 and by examining the content of each agent's message buffer before the agent's tree was evaluated in each cycle. This table shows that messages are sent to the nearest, second nearest and farthest agent at the time a message is sent. In Cycles 14 -21, Agent 0 sends to its nearest and farthest agents, Agent 2 and Agent 3. Agent 1 sends to its nearest agent, Agent 2. Agent 2 sends to its nearest and second nearest agents, Agent 0 and Agent 1. Agent 3 sends to all agents, Agent 0, Agent 1 and Agent 2.

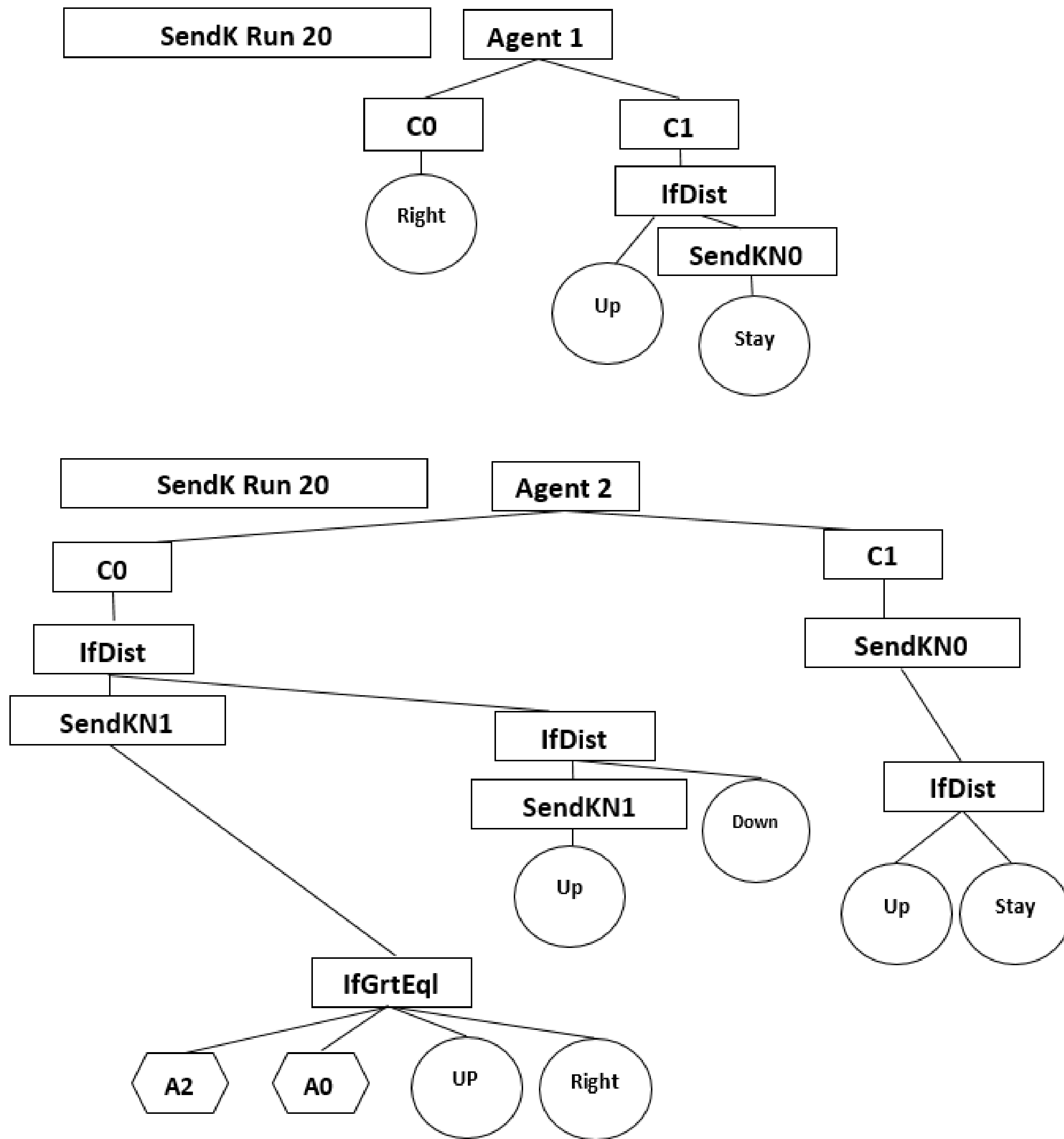


Figure 4.19: SendK GP Sub-tree Example
SendK Run 20, Agents 1 & 2

An example of the message buffer contents can be found in Table 4.18 and examples of sub-trees for Agent 1 and Agent 2 can be seen in Figure 4.19. Cycles 14-20 of the message buffers in Table 4.18 show that Agent 2 is in FOV of the prey and sends messages (containing relative directional data to the prey) to Agent 1 (see Agent 1's message buffer contents). Looking at Figure 4.19, it is seen that Agent 1's C1 branch does not contain an LRM node so it is unable to use the information sent to it by Agent 2.

The message buffer shows that Agent 1 and Agent 2 send LRM data to each other

from Cycles 16 - 20. However, neither of the agents use this data because they do not have the LRM node in their C1 branches. The only advantage they have in receiving a message from one another is that their C1 branch contains the proper nodes (IfDist) to allow them to follow the prey (if they do come within of the prey).

It is seen in Run 20 that most agents in SendK are able to follow the prey once they come into view of the prey. Receiving messages indirectly helps agents follow the prey, but because no agents have the LRM node, receiving messages does not help them find the prey. This may be one of the reasons SendK did not perform as well as SendAll and Send22.

Prey Random Movement

The solutions in the *Prey Random Movement* experiment did not perform as well as they did in the *Prey Linear Movement* experiment. This section analyzes the results to determine the reason for the poor performance. Table 4.19 shows the message sending pattern for all four agents for the top performing protocol, SendAll. Figure 4.20 shows the sub-trees for Agent 0 and Agent 3 of Run 12 and Table 4.20 shows the message buffer contents for this run.

Table 4.19 shows that Agent 1 and Agent 2 rarely send messages at all. However, Agent 0 and Agent 3 send one message to all other agents every other cycle so that Agents 1 and 2 always have a message in their buffer and Agents 0 and 3 have a message in their buffer every other cycle. Table 4.20 shows that the message buffers for Agent 0 and Agent 3 contain messages from each other every other cycle causing them to evaluate their C0 and C1 branches every other cycle.

Unlike previous examples, this synchronization does not help Agent 3 because as seen in Figure 4.20, its C0 and C1 branches combined contain only 2 (Left and Right) of the 4 required movements to follow the prey. Similarly, Agent 0's C0 and C1 branches contain only 2 (Up and Left) of the 4 required movements. Thus, Agent 1 and Agent 3 can find and follow the prey only as the prey moves in 2 directions. When viewing all episodes in the video playback for test Run 12, it is seen that some of the agents are able to find and follow the prey using 2 or 3 directions. However, in most cases agents fail to continue to follow the prey when the prey moves in a direction that is not included in either its C0 or C1 branch.

Table 4.19 shows the message sending pattern of all agents in Send 22 (Run 3). This table shows that Agent 0 and Agent 1 rarely send messages to each other. Agents 2 and 3 often send messages to each other every other frame, similar to the alternating synchronized sending pattern that is seen in previous experiments.

Table 4.18: SendK: Agents Message Buffer Contents

SendK (Linear) Test Run 20, Cycles 14-20

Cycle	From Agent	Agent 0 Message LRM	From Agent	Agent 1 Message LRM	From Agent	Agent 2 Message LRM	From Agent	Agent 3 Message LRM
14			2	(-1,-4)	3	(40,40)		
			3	(40,40)	1	(40,40)		
					0	(40,40)		
					1	(40,40)		
15			3	(40,40)	0	(40,40)		
			2	(-1,-3)	1	(40,40)		
			3	(40,40)	0	(40,40)		
					1	(40,40)		
16			2	(-1,-3)	0	(40,40)	0	(40,40)
			3	(40,40)	1	(40,40)		
			2	(-1,-2)	3	(40,40)		
					1	(1,0)		
17	3	(40,40)	3	(40,40)	1	(40,40)		
	3	(40,40)	2	(-1,-2)	3	(40,40)		
			2	(-1,-1)	1	(1,0)		
					1	(1,0)		
18	3	(40,40)	2	(-1,-2)	1	(1,0)		
			2	(-1,-1)	1	(1,0)		
			2	(-1,-1)	3	(40,40)		
					1	(1,0)		
19			2	(-1,-1)	1	(1,0)	0	(40,40)
			2	(-1,-1)	3	(40,40)		
			2	(-1,-1)	1	(1,0)		
					3	(40,40)		
20	1	(-2,-2)	2	(-1,-1)	3	(40,40)		
	3	(40,40)	2	(-1,-1)	1	(1,0)		
	3	(40,40)	2	(-1,-1)	3	(40,40)		

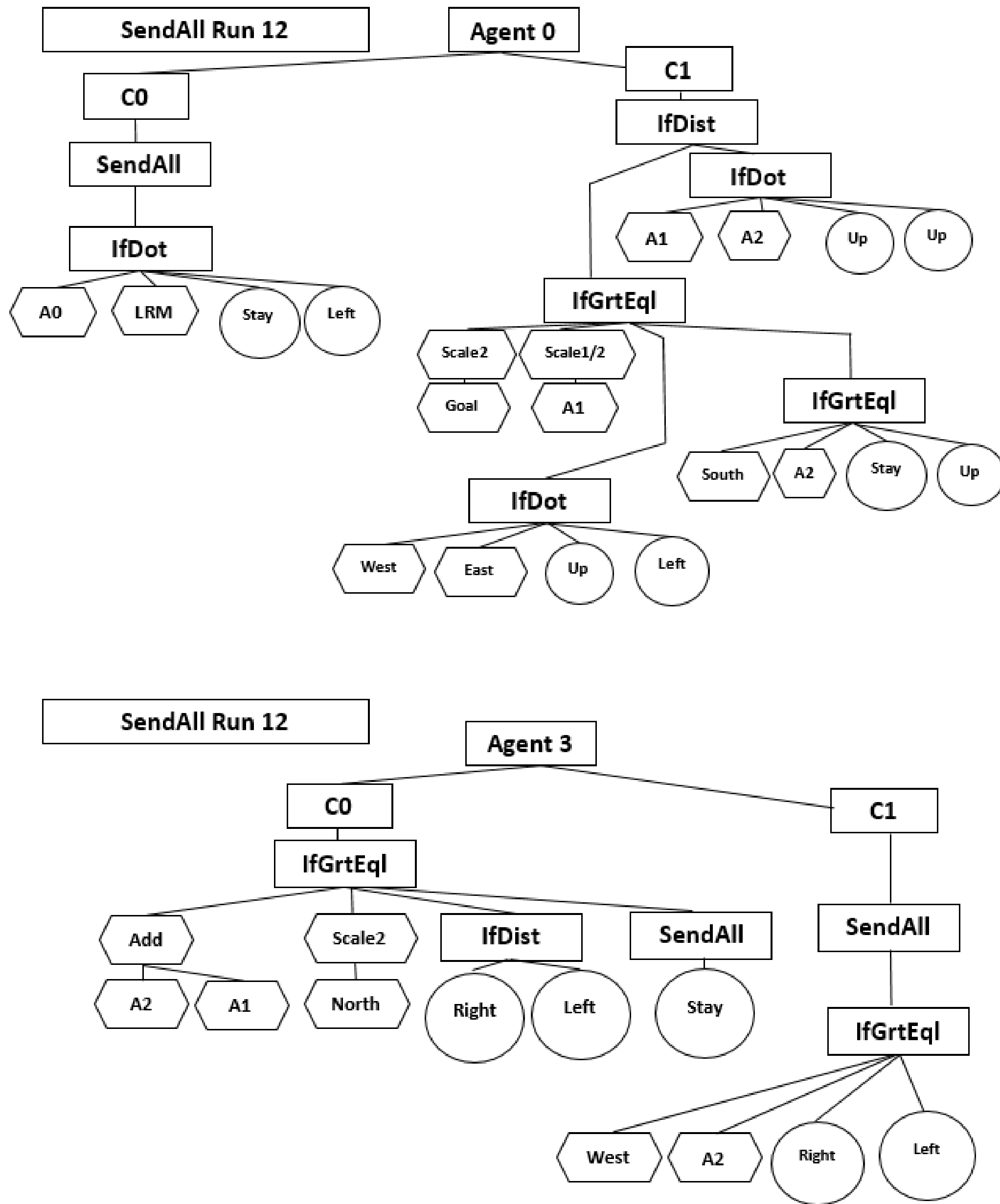


Figure 4.20: SendAll Random GP Sub-tree Example
SendAll Run 12, Agents 0 & 3

In this case, alternating the message "sends" does not seem to help Agents 2 and 3. The test run playback, along with the message buffers, show that when Agent 3 is in view of the prey, it stops sending messages to Agent 2.

Table 4.21 shows the message buffers for Run 3. Cycles 20 - 22 and Cycles 27 - 28 show the Agent 2 and Agent 3 have a message from one another every other cycle. For a short time, from Cycles 23 - 26, Agent 3 is in FOV of the prey and stops sending messages to Agent 2. Thus, the fact that Agent 3 sees the prey is lost to its

Table 4.19: Message Sending Patterns (Prey Random Movement)

SendAll Run 12

Sender		*Receivers*	*Description*
A0	→	*A1, A2, A3*	*(sends every other cycle)*
A1	→	*A0, A2, A3*	*(rarely sends)*
A2	→	*A0, A1, A3*	*(rarely sends)*
A3	→	*A0, A1, A2*	*(sends every other cycle)*

Send22 Run 3

Sender		*Receivers*	*Description*
A0			*(rarely sends)*
A1			*(rarely sends)*
A2	→	*A3*	*(sends every other cycle)*
A3	→	*A2*	*(sends every other cycle)*

Table 4.20: SendAll Synchronized Message Pattern: Agents Message Buffer Contents

SendAll(Random) Run 12, Cycles 17-22

Cycle	From Agent	Agent 0 Message LRM	From Agent	Agent 1 Message LRM	From Agent	Agent 2 Message LRM	From Agent	Agent 3 Message LRM
17	3	(40,40)	3	(40,40)	3	(40,40)		
18			0	(40,40)	0	(40,40)	0	(40,40)
19	3	(-2,-2)	3	(7,-3)	3	(1,-1)		
20			0	(40,40)	0	(40,40)	0	(40,40)
21	3	(-2,-2)	3	(6,-2)	3	(2,-2)		
22			0	(40,40)	0	(40,40)	0	(40,40)

partner. The unpredictable random movement of the prey causes the agents to easily move out of view of the prey. Agents do not remain in FOV for a long period of time and may make it difficult for evolving agents to learn how to notify other agents.

4.3.3 Summary of Results

This chapter continues the research of using GP in a multi-agent system. Using the pursuit domain and a co-operative learning strategy, multiple predator agents learn the meaning of communication commands in order to achieve their goal of finding and following a prey. A number of different communication protocols are examined in two different experiments, *Prey Linear Movement* and *Prey Random Movement*. The outcome of this chapter reveals an emergent behaviour of a synchronized alternating sending pattern among predator agents. This synchronized behaviour helps

Table 4.21: Send22 Agents Message Buffer Contents

Send22 (Random) Run 3, Cycles 20-28

Cycle	From Agent	Agent 0 Message LRM	From Agent	Agent 1 Message LRM	From Agent	Agent 2 Message LRM	From Agent	Agent 3 Message LRM
20							2	(40,40)
21					3	(40,40)		
22							2	(40,40)
23							2	(40,40)
24							2	(40,40)
25							2	(40,40)
26							2	(40,40)
27					3	(40,40)		
28							2	(40,40)

the coordination of agents in the top performing communication protocols in the *Prey Linear Movement* experiment. The *Prey Random Movement* experiment provides a more difficult problem and results show that the synchronized message sending is not as effective. Top performers in the *Prey Linear Movement* experiment are successful in achieving the goals of this research. Agents are able to locate and track the prey using commands and message data sent by other agents. In addition, the best result shows the learned behaviour and collaboration of agents resemble the behaviour of guards and reinforcements that can be found in popular stealth video games (e.g. *Metal Gear Solid (MGS)*[36]).

Chapter 5

Learning the Meaning of Commands

5.1 Problem and Environment

This chapter focuses on finding a better solution for the *Prey Random Movement* problem discussed earlier in Section 4.3.2. It was shown that the characteristics of the environment for the *Communication Protocols* experiment were not sufficient enough to allow the predator agents to follow the prey when it moved randomly in all four directions (either Up, Down, Left or Right). The solution for this experiment focuses on the Send22 and SendAll protocols and uses the same settings as found in Section 4.2 for the Pursuit Domain environment [19]. However, it augments the GP Language, the fitness measure and the type of messages that can be sent to each agent. The goal is to see if the agents can learn the meaning of commands in order to track the prey.

The results show an emergent message sending pattern in both protocols. The emergent behaviour is such that one agent is designated as the "sending agent" and all other agents are designated as "receiving" agents. It is found that in most cases receiving agents are able to associate meaning to commands received from the sending agent. Using a more enriched language than seen previously, top performers use a common sub-tree that influence the agents in achieving their goal of finding and following the prey. Other solutions, although not among the top performers, are able to produce more complex trees with common sub-trees that allow for decision making by the sending agent. In these solutions receiving agents are able to associate specific meaning to sent commands as well as successfully use message data.

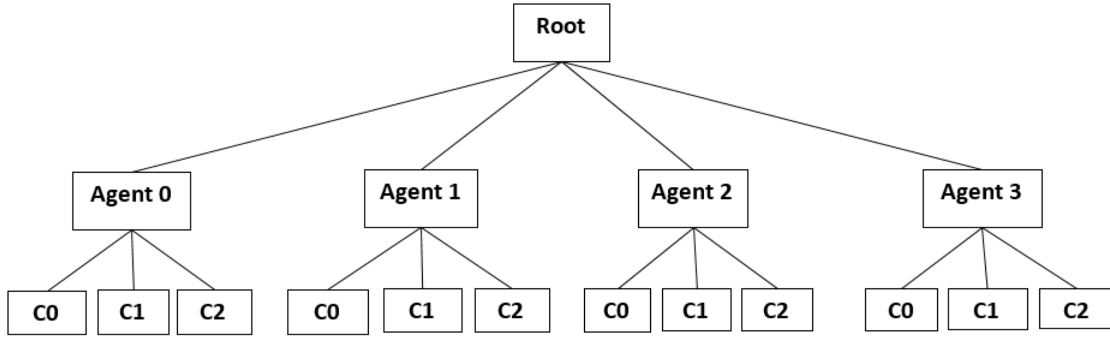


Figure 5.1: Top level GP Structure
C0, C1 and C2 are command branches, only one branch is evaluated each cycle

5.1.1 Learning Strategy

Similar to Section 4.1.1, the learning strategy uses a fully co-operative implementation with a global fitness measure. The predator agents work together towards the common goal of following the prey (as closely as possible). The global fitness is a minimization function calculated over a limited time period. The sum of the distance between all predator agents to the prey is used for fitness. A base penalty value for each predator agent that is not within FOV of (and not following) the prey is added to the fitness sum. If an agent is within distance of the prey and moves in the same direction as the prey for a defined period of time then that agent does not receive a penalty. The motivation for this fitness function is to compare how well the Send22 and SendAll protocols perform in allowing the predators to first find, and then track, the prey's random movement. Agents collaborate to minimize the global fitness value using a heterogeneous team-based learning strategy such that each agent uses its own evolved strategy.

5.1.2 Communication Strategy and Communication Channel

The communication strategy in this study uses a learned language similar to the one used in Section 4.1.2. However, one additional command is added for a total of three generic commands. The commands are *C0*, *C1* and*C2*. The *C1* and *C2* commands, along with simple environment data, are sent individually from one agent to another through a message passing communication channel. An agent learns to associate a meaning to each command through evolving branches of its GP tree.

Figure 5.1 shows the top-level of the GP structure. The root node contains 4 child nodes each representing a GP tree for one agent. Each agent node contains its

own message buffer and either one, two or three child branches (depending on the protocol), *C0*, *C1* and *C2*. The order of evaluation for agents is from left to right, starting at Agent 0 and ending with Agent 3. At the time of an agent's evaluation, the message buffer is checked. If the message buffer contains one or more messages then the agent checks the type of the command in the latest message. If the command is *C1*, then the agent evaluates its *C1* child; if the command is *C2*, then the agent evaluates its *C2* child; otherwise, if there is no message in the buffer, the agent evaluates its *C0* child. The evaluation of agents occurs in 1 time cycle and results in one movement per agent (Up, Down, Left, Right or Stay) on the grid. The GP structure consists of a heterogeneous team of agents such that each agent uses its own tree.

5.1.3 Communication Protocols

A communication protocol defines the method by which a sending agent sends a message to a receiving agent's message buffer. Three different message types are examined for Send22 and SendAll and each is listed in Table 5.1 as their own communication protocol. Message passing for Send22 and SendAll is the same as described in Chapter 4 with the addition of message types.

- **Send22**: Two teams of two agents. Agent 0 and Agent 1 form one team and Agent 2 and Agent 3 form the other team. Each agent sends to its partner only.
- **SendAll**: Each agent sends to every other agent.

The message types are C1, C1C2, and C1orC2. Depending on the protocol, agents can send only one of the following types of command:

- **C1**: Sends only C1 commands.
- **C1C2**: Sends C1 and C2 commands each at different times.
- **C1orC2**: Sends either a C1 or C2 command (depending if agent is within FOV of prey at each send).

In the *C1* protocol, only one send message is used to send a message with the C1 command. Similar to the experiments in Chapter 4, if the agent is within FOV, the message data sent to the receiving agent contains the direction from the receiving agent to the prey, otherwise it contains a default value of (4000, 4000). For the *C1C2* protocol, two send message commands are used. SendC1 is used to send the

Table 5.1: Communication Protocols

Communication Protocols	Description
Send22C1	One send command only.
	Agents send C1 command.
Message Command(s)	C1
Message Data	$if agent <= FOV\ then$ send dir to prey
	$else$ send default dir
Send22C1C2	Send commands: Send22C1 and Send22C2.
	Agents send C1 command on Send22C1.
	Agents send C2 command on Send22C2.
Message Command(s)	C1, C2
Message Data	$if agent <= FOV\ then$ send dir to prey
	$else$ send default dir
Send22C1orC2	Agents send either C1 or C2 command.
Message Command(s)	C1 or C2
Message Data	$if agent <= FOV\ then$ send dir to prey with C2
	$else$ send default dir with C1
SendAllC1	One send command only.
	Agents send C1 command.
Message Command(s)	C1
Message Data	$if agent <= FOV\ then$ send dir to prey
	$else$ send default dir
SendAllC1C2	Send commands: SendAllC1 and SendAllC2.
	Agents send C1 command on SendAllC1.
	Agents send C2 command on SendAllC2
Message Command(s)	C1, C2
Message Data	$if agent <= FOV\ then$ send dir to prey
	$else$ send default dir
SendAllC1orC2	Agents send either C1 or C2 command.
Message Command(s)	C1 or C2
Message Data	$if agent <= FOV\ then$ send dir to prey with C2
	$else$ send default dir with C1

C1 command and SendC2 is used to send the C2 command. For both of these send commands, if the agent is within FOV, the message data sent to the receiving agent contains the direction from the receiving agent to the prey, otherwise it contains a default value of (4000, 4000). In the *C1orC2* protocol, one send message command is used (SendC1orC2) to send either a C1 command or a C2 command. If the agent is within FOV, the message data sent to the receiving agent contains the direction from the receiving agent to the prey with the C2 command, otherwise the message

Table 5.2: GP Parameters

GP Parameter	Value
Initial Tree Method	Koza's *Ramped half-and-half* [30], [22]
Min-Max Tree size (ramp)	7-12
Population size	1000
Generations	125
Selection	Tournament, size = 4
Crossover	90%
Mutation	10%
Runs per experiment	20

data sent contains a default value of (4000, 4000) with the C1 command.

5.2 Experiment Details

All GP parameters can be found in Table 5.2. Each experiment uses the same settings for the GP parameters as seen in Section 4.2 with the exception of the *Min-Max Tree size (ramp)*. This parameter is increased to allow for larger tree sizes and to better accommodate the top level structure of the GP (containing 3 levels on its own) including the root node at level one, the agents' nodes at level two, and finally the command nodes at level 3 (see Figure 5.1). The fitness function, GP language, testing and training methods are described below. In this experiment, the prey moves in a random pattern (Up, Down, Left or Right) on the grid. Both agents and prey move only one step per time cycle. Each communication protocol is tested individually (see Table 5.1).

5.2.1 GP Language

In order to construct the GP structure as shown in Figure 5.1 the strongly typed language from Table 5.3 is used.

Terminal Set

The terminal set used is defined in Table 5.4. Movement commands (Up, Down, Left, Right and Stay) are sent directly to the agent as a result of the evaluation of its command tree. The language is typed such that only 1 movement is sent per

Table 5.3: Strongly Type Language

ROOT	$::= (SIM, SIM, SIM, SIM)$
SIM	$::= \mathit{CommandTree}(EXPR, EXPR, EXPR)$
EXPR	$::= \mathit{Left} \mid \mathit{Right} \mid \mathit{Up} \mid \mathit{Down} \mid \mathit{Stay} \mid$
	$::= MoveForward \mid$
	$::= MoveInDir(NIL) \mid$
	$::= IfGrtEql(NIL, NIL, EXPR, EXPR) \mid$
	$::= IfDist(NIL, NIL, EXPR, EXPR) \mid$
	$::= IfDot(NIL, NIL, EXPR, EXPR) \mid$
	$::= \mathit{Send}(EXPR)$
NIL	$::= \mathit{Goal} \mid \mathit{AgentDir} \mid \mathit{AgentDir}0 \mid \mathit{AgentDir}1 \mid \mathit{AgentDir}2 \mid$
	$::= \mathit{Add}(NIL, NIL) \mid$
	$::= \mathit{Sub}(NIL, NIL) \mid$
	$::= \mathit{Rotate}90(NIL) \mid$
	$::= \mathit{Reverse}(NIL) \mid$
	$::= North \mid \mathit{South} \mid \mathit{East} \mid West \mid$
	$::= LRM$

Table 5.4: Terminal Set

Name	Description
Up,Down,Left,Right,Stay	move commands: $\uparrow, \downarrow, \leftarrow, \rightarrow$
North,South,West,East	$(0,1), (0,-1), (-1,0), (1,0)$
Goal	direction from prey to agent
	if agent is within FOV
AgentDir	the direction the agent is currently facing
AgentDir0-2 (similar to Iba [6])	direction from nearest(0))
	2nd nearest (1) and farthest agent(2)
LRM	Last Received Message
MaxDist	returns a large distance value (4000)
MoveForward	moves agent one step in current direction

evaluation. The direction vectors, North, South, West and East contain the unit vector of each direction. The Goal terminal gives the direction to the agent only

Table 5.5: Function Set

Function	Description
Root	returns the evaluated value of the entire tree
SimAgent	returns the evaluated value of one agent
Add	vector addition
Sub	vector subtraction
Rotate90	rotates vector by 90 degrees
Reverse	multiplies vector by -1
IfGrtEql	compares the length of two vectors
IfDot	calculates the dot product of two vectors
IfDist	checks if agent is withing FOV of prey
MoveInDir	moves in direction of input vector
	and sets current agent's direction
SendC1, SendC1C2, SendC1orC2	(see Communication Protocols, Table 5.1)

if the agent is within field of view (FOV) of the prey, otherwise it gives a default direction of (4000, 4000). The AgentDir terminal node holds the current direction the agent is facing (either North, South, East or West). Directions to the nearest, 2nd nearest and farthest agent are given in AgentDir0, AgentDir1 and AgentDir2 respectively (similar to the ones used for robot navigation in Iba [6]). The terminal node to hold data for the last message removed from an agent's message buffer is named Last Received Message (LRM). Before evaluation of its tree, an agent checks its message buffer. If it contains messages, the first message is removed and its data is set to the LRM variable to be used for that evaluation cycle. If there are no messages in its buffer, the LRM node is set to the default vector (4000, 4000). The MaxDist node holds a constant value of 4000 representing a large distance value and finally MoveForward is a movement command and results in the agent moving one step in its current direction.

Function Set

The function set is seen in Table 5.5. Functions include mathematical operations on two dimensional vectors, logical operations and message sending commands. The functions are the same as the ones used in Section 4.2.1 with the addition of the movement function, MoveInDir. Message sending commands are listed as *SendC1, SendC1C2,and SendC1orC2*. The communication commands are *C0, C1 and C2.*

C0 is used as the default command if no messages have been sent to an agent and *C1 and C2* are the commands used when messages are sent by agents. The type of command that is sent to an agent depends on the protocol that is being tested, *C1*, *C2*, and *C1orC2*. For example, the Send22C1 protocol will send the C1 command to its partner. If the sending agent is within FOV then the message data will contain the direction to the prey otherwise it will contain the default direction value (4000, 4000). For a more detailed description see Table 5.1.

5.2.2 Training and Testing Methods

The training and testing of a GP individual consists of cycles and episodes. The training and testing methods for this experiment are the same as listed in Section 4.2.2. As seen earlier, training and testing begin with the agents and prey starting in a random position within their own area on the grid (See Figure 4.1). In one cycle, all four agents evaluate their command tree once in sequence, one after the other. Each evaluation results in one movement of the agent, where one movement equates to one (cell) on the grid (and one unit in distance). After 30 cycles, one episode is complete (i.e. 1 episode = 30 cycles). In training, each GP individual is given 10 episodes and the positions of the agents/prey are reset to the original starting position after each episode is complete. The test run uses the GP individual with the best fitness in training. This GP individual is tested with 30 episodes instead of 10 and the test run sets a new random start position for each agent and the prey before a new episode begins.

5.2.3 Fitness Function

The total fitness value is measured by accumulating sums from cycles to episodes. In each cycle of an episode, the sum of two fitness values is used as a fitness measure for that cycle. The total sum of each cycle fitness is used as the episode fitness. Finally, the total sum of each episode fitness score is used as the final fitness value for each individual GP.

The first fitness value calculated for each cycle is the sum of each of the agent's distance to the prey, where each cell on the grid represents 1 unit of distance. Equation (5.1) calculates each agent's distance to the prey where A is the agent's position and P is the prey's position.

$$fAgentDist(A) = \sqrt{(A.x - P.x)^2 + (A.y - P.y)^2} \tag{5.1}$$

The second fitness value used in the cycle fitness measure is the total base penalty value. This value represents a specific penalty (10 points) for each predator agent in each cycle. For example, in any given cycle, each agent that is not within FOV of the prey or chooses not to move in the same direction as the prey is penalized by adding 10 points to the total base penalty value for that cycle. If the agent is in FOV of the prey and moves in the same direction as the prey then it is not penalized. Thus, if all four agents choose to move in the same direction as the prey and are in FOV of the prey, the total base penalty value for that cycle would be zero. However, if none of the agents were in FOV of the prey or if none of the agents chose the same direction as the prey then the total base penalty value for the cycle would be 40.

Equation (5.2) shows how the total base penalty value is calculated in each cycle where *lpMove* is the prey's last move (Up, Down, Left or Right), *aMove* is the move chosen by the agent in that cycle and *FOV* is a function that returns *true* if the agent is within view of the prey.

$$fBasePenatly(lpMove, aMove) = \begin{cases} 0 & \text{if } (lpMove == aMove) \&\& (FOV) \\ 10 & \text{otherwise.} \end{cases} \tag{5.2}$$

Equation (5.3) shows the total distance fitness calculation used in training. Here, $APos_i$ represents the location of $Agent_i$, where $i = 0...3$, m represents the number of cycles and q is the number of episodes. We set q to 10 in training and to 30 in testing in our experiments.

$$TotFit = \sum_{k=1}^{q} \sum_{j=1}^{m} \sum_{i=0}^{3} \Big(fAgentDist(APos_i) + fBasePenatly(lpMove, aMove_i) \Big) \tag{5.3}$$

Similar to Equation (5.3), the test run fitness measures the average distance of all the episodes as seen in Equation (5.4).

$$AveDist = \frac{TotFit}{q} \tag{5.4}$$

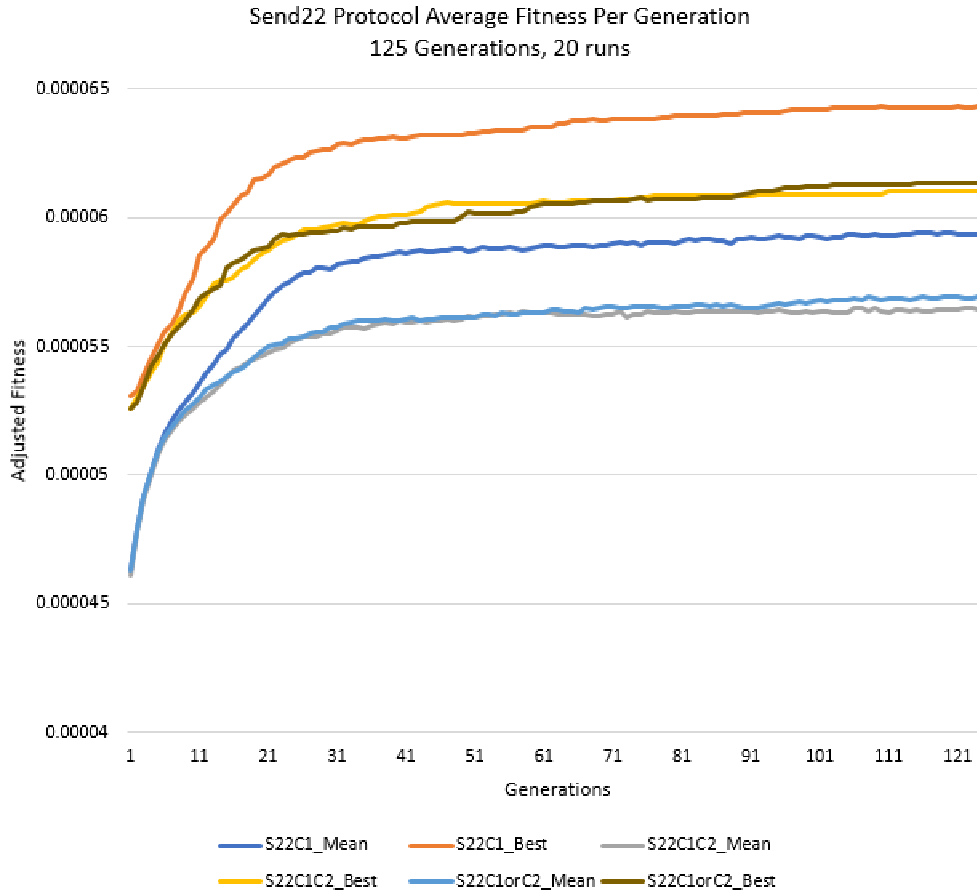


Figure 5.2: Send22 - Training

5.3 Results

In this section the performance results of the communication protocols Send22 and SendAll with different message types (see Table 5.1) are shown. Using random movement for the prey for each protocol, the training results are first displayed followed by the testing results.

5.3.1 Send22 Protocol

The training results for Send22C1, Send22C1C2 and Send22C1orC2 are shown in Figure 5.2. This figure displays the average fitness performance of each generation in 20 runs, with a total of 125 generations per run. The average fitness (best fitness in orange colour range and mean fitness in blue colour range) is shown for each protocol. Here it is seen that the best performer in training is Send22C1.

Results for test runs are shown in Table 5.6. This table displays the performance

Table 5.6: Send22 Fitness Summary (20 Test Runs)

Communication Type	Min Fitness	Average Fitness	Max Fitness
Send22C1	1681	1805	1964
Send22C1C2	1721	1883	2031
Send22C1orC2	1609	1850	2032

Fitness is a minimization function. See Equation (5.4).

Table 5.7: SendAll Fitness Summary (20 Test Runs)

Communication Type	Min Fitness	Average Fitness	Max Fitness
SendAllC1	1641	1790	2023
SendAllC1C2	1701	1905	2094
SendAllC1orC2	1734	1842	1974

Fitness is a minimization function. See Equation (5.4).

of test runs for the best individual GP found in each training run by listing the minimum fitness, the maximum fitness and the average fitness for 20 test runs.

5.3.2 SendAll Protocol

The training results for SAll22C1, SendAllC1C2, and SendAllC1orC2 are shown in Figure 5.3. This figure displays the average fitness performance of each generation in 20 runs, with a total of 125 generations per run. The average fitness (best fitness in orange colour range and mean fitness in blue colour range) is shown for each protocol. Here it is seen that the best performer in training is SendAllC1.

Results for test runs are shown in Table 5.7. This table displays the performance of test runs for the best individual GP found in each training run by listing the minimum fitness, the maximum fitness and the average fitness for 20 test runs.

5.3.3 Statistical Analysis

In this section the statistical performance results are discussed. In each section the training results are first discussed followed by testing results.

In Figure 5.2 it is seen that for training in the Send22 communication protocols, the Send22C1 (dark orange line for average of best fitness, dark blue line for average of mean fitness) outperformed all other types. Send22C1C2 and Send22C1orC2 training fitness values are fairly equal throughout the generations with Send22C1orC2 finishing just slighlty above Send22C1C2 in the final generations. Training fitness for the SendAll communication protocols is seen in Figure 5.3. This data shows that

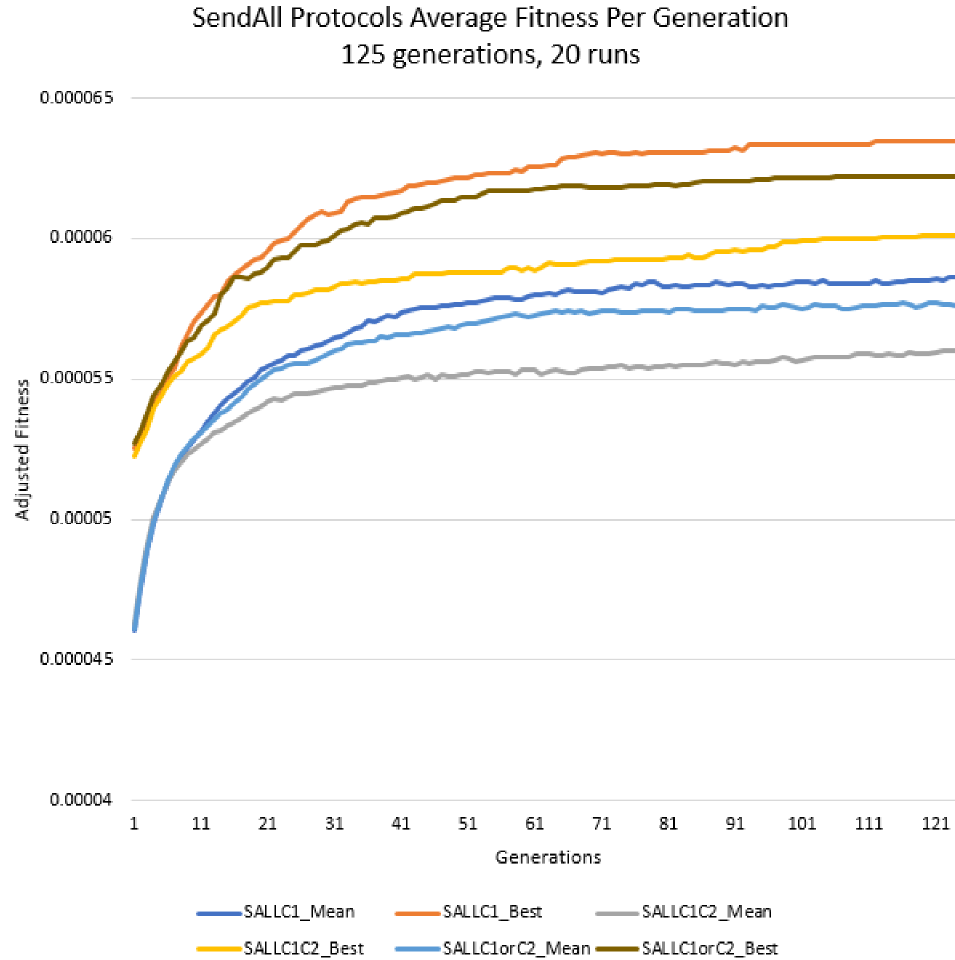


Figure 5.3: SendAll - Training

SendAllC1 (dark orange line for average of best fitness, dark blue line for average of mean fitness) finished first in performance while SendAllC1orC1 finished second clearly above SendAllC1C1.

Figure 5.4 compares the top performers in training for Send22 and SendAll communication protocols. Here is is seen that Send22C1 outperforms SendAllC1. Figure 5.4 also shows the second best performers in their own categories, Send22CC1orC2 and SendAllC1orC2. The training graph shows that Send22C1orC2 initially performs just as well as SendAllC1orC2 in early generations, but at around generation 31 SendAllC1orC2 begins to outperform Send22C1orC2.

The test results show a correlation with the training results for each protocol type. Table 5.6 shows that Send22C1 has the lowest average fitness of 1805 of all the Send22 communication types with Send22C1orC2 finishing second with a score of 1850. Test results in Table 5.7 show that, similar to the training results, SendAllC1

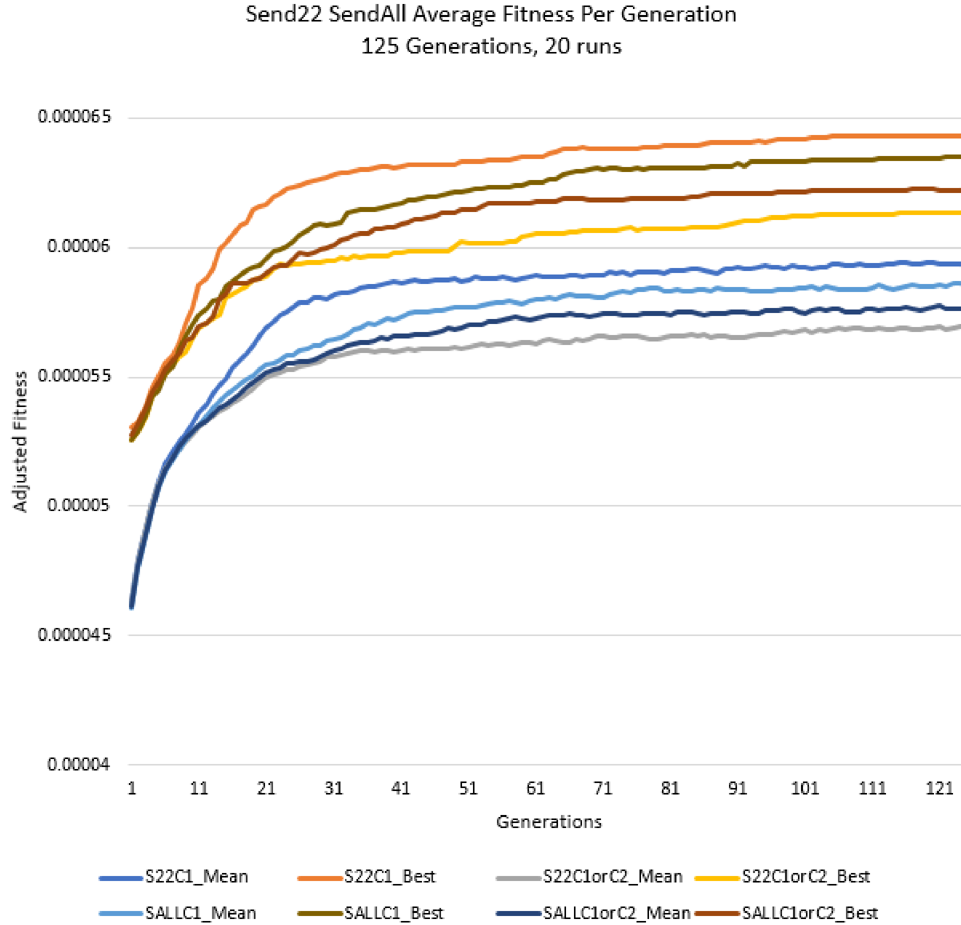


Figure 5.4: Send22 SendAll - Training

also outperforms all other SendAll types with the lowest average fitness score of 1790 and SendAllC1orC2 finishes second with a score of 1842.

To verify the significance of the test results the One-Way ANOVA test (using Minitab [37]) with a 95% confidence interval is used. The ANOVA test uses a two-tailed T-test with 3 factors, where each factor represents one communication protocol including the final test fitness for the 20 test runs. In the Send22 protocol, the ANOVA test results in the $P{-}Value < \alpha$ (see Appendix A) indicating that there is a significant difference in the test results for Send22C1, Send22C1C2 and Send22C1orC2 as shown in Table 5.6. Similarly, the ANOVA test results for the SendAll protocol show the $P - Value < \alpha$ indicating that there is also a significant difference in the test results for SendAllC1, SendAllC1C2 and SendAllC1orC2 as shown in Table 5.7.

In order to identify which factors have different means the Tukey method [38] for multiple comparisons is used for both the Send22 and SendAll protocols with results shown in Table 5.8 and Table 5.9 respectively. In these tables, protocols which do not

Table 5.8: Tukey Comparisons for Send22 Protocols

Communication Protocol	Group Letter	Average Fitness
Send22C1C2	A	1883
Send22C1orC2	AB	1850
Send22C1	B	1805

Protocols that do not share the same letter are significantly different

Table 5.9: Tukey Comparisons for SendAll Protocols

Communication Protocol	Group Letter	Average Fitness
SendAllC1C2	A	1905
SendAllC1orC2	AB	1842
SendAllC1	B	1791

Protocols that do not share the same letter are significantly different

share the same group letter indicate that their range of difference of mean does not contain a zero. Thus, protocols labelled with different letter groups are considered to be significantly different [39].

For the Send22 protocol, Table 5.8 shows that the mean for factor Send22C1 is significantly better than the mean for Send22C1C2 because they do not share the same letter group. However, Send22C1 is not significantly different than Send22C1orC2 because they are in the same letter group [37]. The SendAll protocol shows the same pattern in the Tukey test in Table 5.9. Here it is seen that the factor SendAllC1 is significantly better than SendAllC1C2 because they are not in the same letter group. Again, it is seen that SendAllC1 is not significantly better than SendAllC1orC2 because they share the same letter group [39].

5.3.4 Emergent Sending Patterns

This section discusses the qualitative aspects such as GP tree structure, message sending patterns, and playback of some of the best test runs in order to understand why top GPs are able to outperform others. This examination reveals an emergent pattern in message sending that is seen at least once in almost all the communication protocols. Also, it reveals that although the statistical analysis of the test data determined that the C1 and C1orC2 protocols (in both Send22 and SendAll) are not significantly different, the C1orC2 protocol is seemingly more successful in allowing

Table 5.10: Message Sending Patterns for SendAll

SendAllC1 Run 2

Sender		*Receivers*	*Description*
A0			*(never sends)*
A1			*(never sends)*
A2	→	*A0, A1, A3*	*(sends 3 C1 commands when not in FOV of prey)*
A3			*(never sends)*

SendAllC1C2 Run 15

Sender		*Receivers*	*Description*
A0	→	*A1, A2, A3*	*(always sends C1 command)*
A1			*(never sends)*
A2			*(never sends)*
A3			*(never sends)*

SendAllC1orC2 Run 19

Sender		*Receivers*	*Description*
A0			*(never sends)*
A1	→	*A0, A2, A3*	*(almost always sends)*
A2			*(never sends)*
A3			*(never sends)*

predator agents to learn the meaning of commands through message passing. Likewise, although it was shown that the C1C2 protocol (for both Send22 and SendAll) was statistically worse than the C1 protocol, the evolved agents using the C1C2 protocol seem to be more successful in learning meaning of commands compared to the C1 protocol.

An emergent sending pattern is found at least once among the top 3 performers of each protocol. The common pattern is such that one agent is responsible for sending messages (i.e. it does not receive messages) while the remaining agents are responsible for only receiving messages (i.e. they do not send messages). Examples of this emergent message sending pattern for each protocol are shown in Tables 5.10 and Table 5.11.

Table 5.10 shows that Agent 2 is the only agent sending messages in the SendAllC1 protocol, Agent 0 is the only agent sending messages in the SendAllC1C2 protocol, and Agent 1 is the only agent sending messages in the SendAllC1orC2 protocol.

Table 5.11 shows that Agent 0 is the only agent sending messages in the Send22C1 protocol, Agent 3 is the only agent sending messages in the Send22C1C2 protocol,

Table 5.11: Message Sending Patterns for Send22

Send22C1 Run 17

Sender		*Receivers*	*Description*
*A*0	→	*A*1	*(almost always sends)*
*A*1			*(never sends)*
*A*2			*(never sends)*
*A*3			*(never sends)*

Send22C1C2 Run 17

Sender		*Receivers*	*Description*
*A*0			*(never sends)*
*A*1			*(never sends)*
*A*2			*(never sends)*
*A*3	→	*A*2	*(always sends)*

Send22C1orC2 Run 11

Sender		*Receivers*	*Description*
*A*0			*(never sends)*
*A*1	→	*A*0	*(sends to A1 when not in FOV of prey)*
*A*2			*(never sends)*
*A*3			*(never sends)*

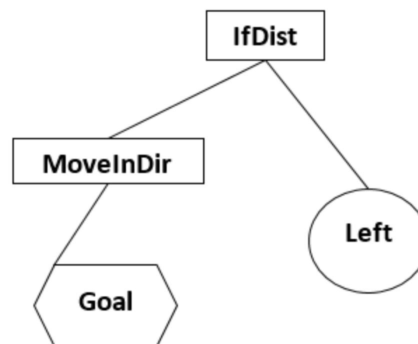


Figure 5.5: Common GP Structure
This GP structure allows agents to find and follow the prey on their own.

and Agent 1 is the only agent sending messages in the Send22C1orC2 protocol.

By interpreting the data, the agents mentioned in the tables above are designated as sending agents because they do not receive messages from any other agents. In each protocol, all other agents are designated as receiving agents because they do not send messages.

A GP structure that is influential to the agents in reaching their goal is typically found in one of the receiving agents command trees (either C0, C1 or C2 sub trees). This structure is shown in Figure 5.5. Upon evaluation of this sub-tree, an agent first checks if it is within FOV of the prey using the IfDist expression. If it is within range of the prey then it moves in the direction of the prey using the MoveInDir expression with the Goal vector node as input (*MoveinDir(Goal)*) (*note: the Goal node will only have valid direction to the prey if the agent is within FOV of the prey*). If the agent is not within FOV of the prey then the agent chooses another direction to move (in any one of the four possible directions of Up, Down, Left or Right).

In many cases the test runs show that this sub-tree is all that is needed for an agent to achieve its goal of finding and following the prey. Using the enriched GP language, the predator agents seem to learn that they will succeed in their task by assigning one agent as the "sender" agent (which sends the same command) and assigning the remaining agents as "receiver" agents each having this sub-tree in the corresponding command branch (either in C1 or C2).

An example of a typical GP showing this emergent sending/receiving pattern is shown in Figures 5.6 to 5.9. These figures represent each agent's GP structure for the best run using the SendAllC1 protocol. Figures 5.6 and 5.7 show that Agent 0 and Agent 1 both have the influential sub-tree (see Figure 5.5) as part of their C1 branch. Figure 5.8 displays the GP structure for the sending agent, Agent 2. It is seen that Agent 2 sends 3 messages (with the C1 command) to the receiving agents, Agent 0, Agent 1, and Agent 3, when it is not in FOV of the prey. Finally, Figure 5.9 shows that Agent 3's tree does not contain the influential sub-tree.

This section shows an emergent sending pattern in both the Send22 and SendAll protocols. The top agents are able to simplify message sending so that even agents using the SendAll protocol (which allows all agents to send to every other agent) learn that an efficient method to achieve their goal is to have one "sending agent" that sends the same command to "receiving agents". In addition, most of the "receiving" agents have the corresponding command branch (sent by the sending agent) which contains the influential sub-tree found in Figure 5.5 to help it follow the prey. On a very basic level, receiving agents are applying a meaning to the C1 command. But in many cases, agents' contain the influential expression *MoveInDir(Goal)* in more than one command sub-tree (see Figures 5.6, 5.7 and 5.8). Therefore, it seems that this expression is essential in the success of the Send22C1 and SendAllC1 protocols.

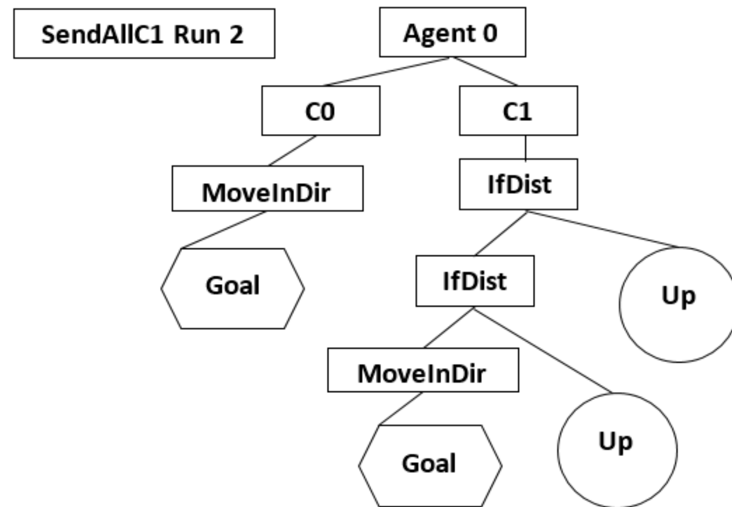


Figure 5.6: SendAllC1 Run 2 Agent 0
This "receiver" agent evaluates its C1 branch when it receives C1 commands from Agent 2.

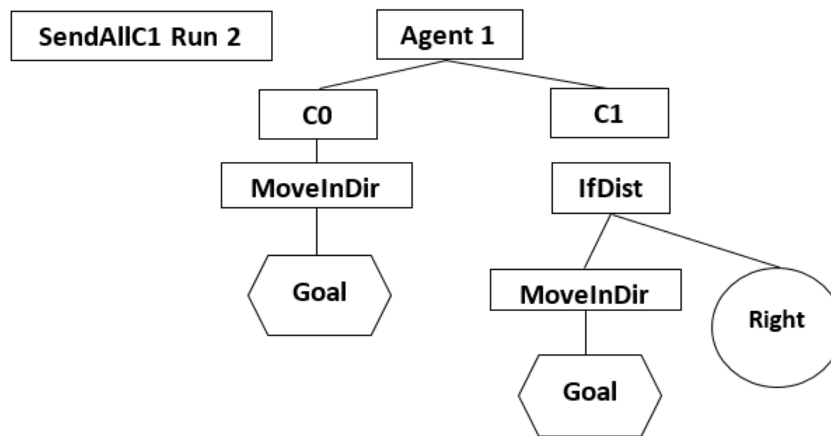


Figure 5.7: SendAllC1 Run 2 Agent 1
This "receiver" agent evaluates its C1 branch when it receives C1 commands from Agent 2.

5.3.5 Learning the Meaning of Commands

The previous section showed that, through an emergent sending/receiving pattern, agents were able to associate a meaning to a command that was sent by another agent. The Send22C1 and SendAllC1 protocols were able to outperform the the C1C2 and C1orC2 protocols in test runs. However, this section shows that the GP trees created

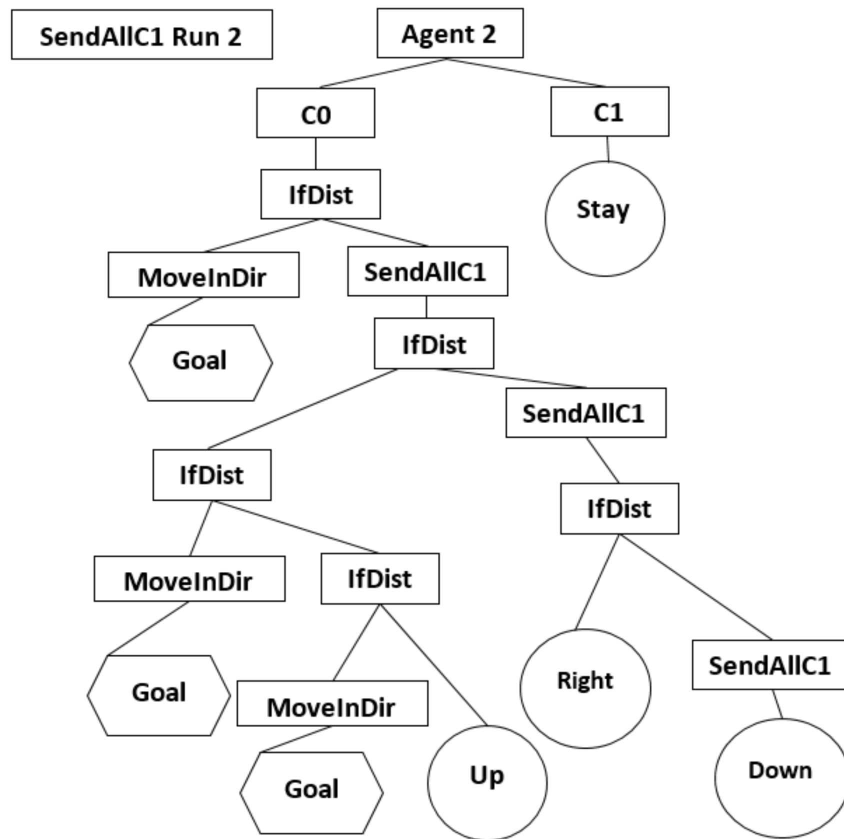


Figure 5.8: SendAllC1 Run 2 Agent 2
This "sender" agent sends 3 C1 commands when it is not in FOV of prey.

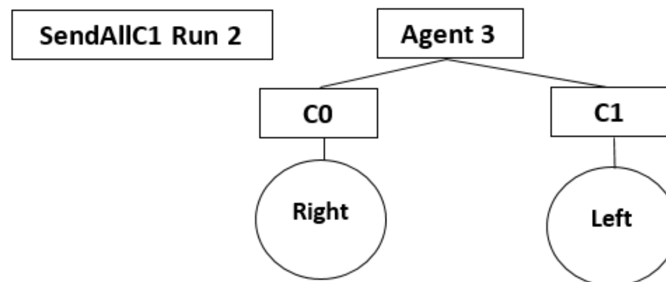


Figure 5.9: SendAllC1 Run 2 Agent 3
This "receiver" agent evaluates its C1 branch when it receives C1 commands from Agent 2.

using the C1C2 and C1orC2 protocols are more complex than the trees created by the C1 protocols and in some cases are better able to associate specific meaning to commands and use information sent by other agents.

As seen in the previous section the SendAllC1 protocol typically creates simple sub-trees for receiving agents (see Figures 5.6 to 5.9). These sub-trees are able to apply a meaning to the command C1. Upon receipt of a C1 command, receiving agents use the common GP structure found in Figure 5.5 to move in the direction of the prey (if the agent is in FOV of the prey). In general, agents' tree structures are simple because they do not require decision making about what types of commands to send and they do not require message data (through the LRM node) in order find and follow the prey.

Agents' trees created using the C1C2 and C1orC2 protocols are more complex and seem to be able to make decisions on what types of command to send and when to send them. Receiving agents are able to apply meaning to received commands and are able to successfully use received message data (through the LRM node). To demonstrate this, an example from each of SendAllC1C2, SendAllC1orC2, Send22C1C2, and Send22C1orC2 are examined.

C1C2 - Deciding Which Command to Send

A typical example of decision making found in a sending agent's sub-tree in the C1C2 protocol is seen in Figure 5.10. This sub-tree allows an agent to decide whether to send a C1 command or a C2 command. Using the IfDist expression the decision is based off whether the agent is within FOV of the prey. If the agent is within FOV of the prey, then it sends a C1 command (using SendAll or Send22, depending on the protocol) with message data containing information about the relative direction to the prey. As explained previously, the message data is stored in the LRM node upon receipt of the message. If the agent is not within the FOV of prey, then the sending agent sends a C2 command with message data containing a default direction of (4000, 4000).

This common structure is found in Agent 0's C2 branch in one of SendAllC1C2's top runs, Run 15. Figure 5.11 shows Agent0's GP sub-tree. Unfortunately, because Agent 0 never receives any messages, it never gets the chance to evaluate the C2 branch and therefore does not actually make a decision on which command to send. Instead, it sends C1 messages to all the other agents every cycle by evaluating its C0 branch. Despite this, this example is shown to demonstrate that SendAllC1C2 creates complex sub-trees that have the potential for decision making when sending.

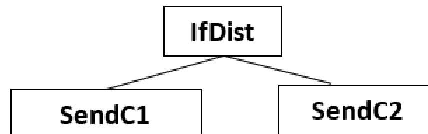


Figure 5.10: Common GP Structure to Decide To Send Specific Commands
Allows agents to send specific commands (C1 or C2).

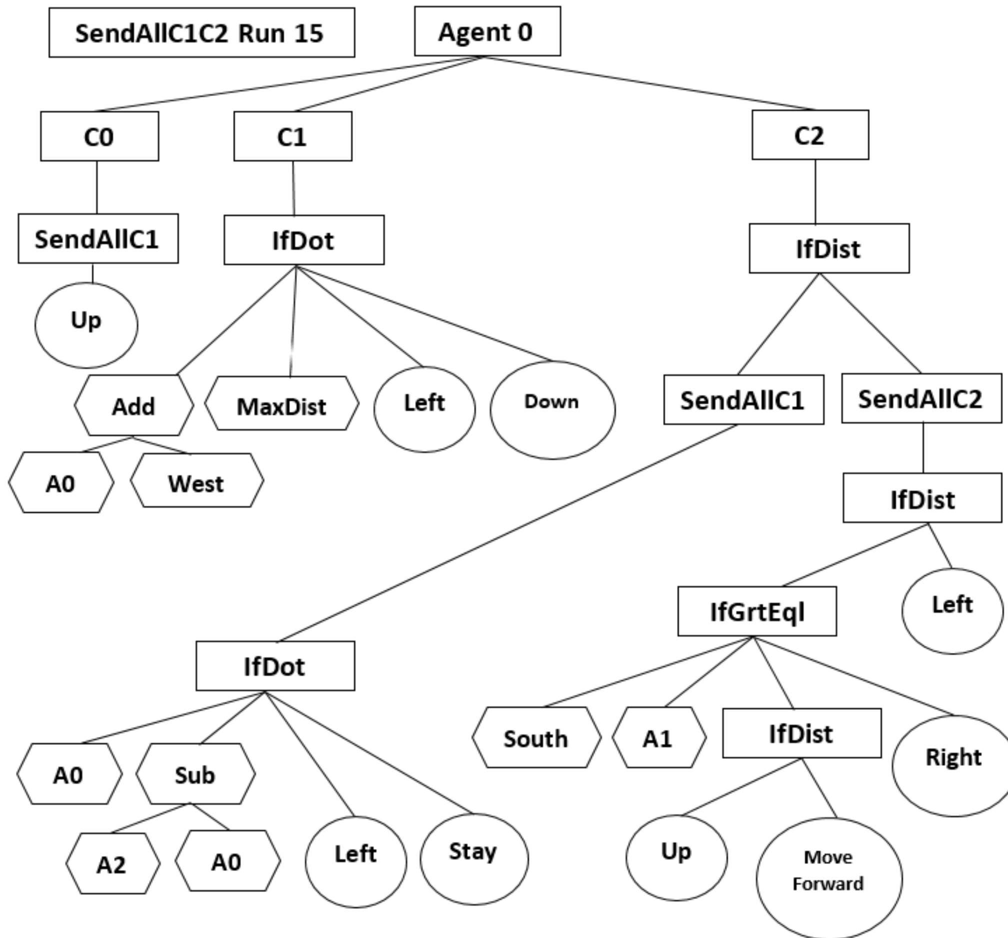


Figure 5.11: SendAllC1C2 Agent 0 Sub-tree, Run 15
A0...A2 = AgentDir0...AgentDir2 Terminals

A top performer, Run 12, in Send22C1C2 protocol is actually able to make decisions when sending commands. Figure 5.12 shows the sub-trees for Agent 2 and Agent 3 in this run. In Figure 5.12, the C0 branch for Agent 2 (the sending agent) allows Agent 2 to decide which command to send based on whether it is in FOV of the prey. Similar to the explanation above, a C1 command (with relative directional data about the prey) is sent when Agent 2 is in FOV of the prey and a C2 command

(with default data) is sent when Agent 2 is not in FOV of the prey. The sub-tree for the receiving agent, Agent 3, is also seen if Figure 5.12. Depending on the message type sent to Agent 3, it will evaluate either its C1 or C2 branch. Figure 5.12 shows that the C1 branch for Agent 3 contains the LRM node (which upon receipt of the C1 command will contain relative directional information about the prey). Thus by using the LRM node in C1, Agent 3 has learned to associate a meaning to the C1 command. This is of interest because the sending agent, Agent 2, through its decision making gives a meaning to the C1 command by sending this command only when it is in FOV of the prey. In turn, the receiving agent is able to understand the meaning of C1 and uses the LRM node only in its C1 branch. Ideally, the C1 branch of Agent 3 would use the MoveInDir expression with the LRM as input (*MoveInDir(LRM)*) but it is encouraging that the C1 branch evolved to use the LRM node.

C1orC2 - Understanding The Meaning of Received Commands

The C1orC2 results for both SendAll and Send22 show that receiving agents are able to apply meaning to the C2 command and are able to successfully use the LRM data to move in the direction of the prey. In the C1orC2 protocol, decision making on what command to send is handled in the GP language via the Send function. If the sending agent is within FOV of the prey, then the send function will send a C2 command with relative directional data to the prey, otherwise it will send a C1 command with default directional data.

Agents' sub-trees for a top performer, Run 19, in the SendAllC1orC2 protocol is shown in Figures 5.13 and 5.14. In this run, Agent 1 is the sending agent. As seen in Figure 5.13, in its C0 branch, Agent 1 uses the SendAll expression to send a message to every other agent each cycle. Either a C1 or C2 command will be sent, as described above.

Receiving agents, Agent 2 and Agent 3, shown in Figure 5.14, use their C1 branches to move in a specific direction if they are not within FOV of the prey but, if they are in FOV they use the *MoveInDir(Goal)* expression. Thus, these agents are associating the C1 command with moving in the direction of the goal (not using the LRM data which, based on the send criteria, would not have useful information in this branch).

The remaining receiving agent shown in Figure 5.13, is Agent 0. It uses the *MoveInDir(LRM)* expression in its C2 branch. This agent learns to associate its C2 branch with having relative directional information to the prey via the LRM node. When Agent 0 receives a C2 command it may move in the direction of the prey

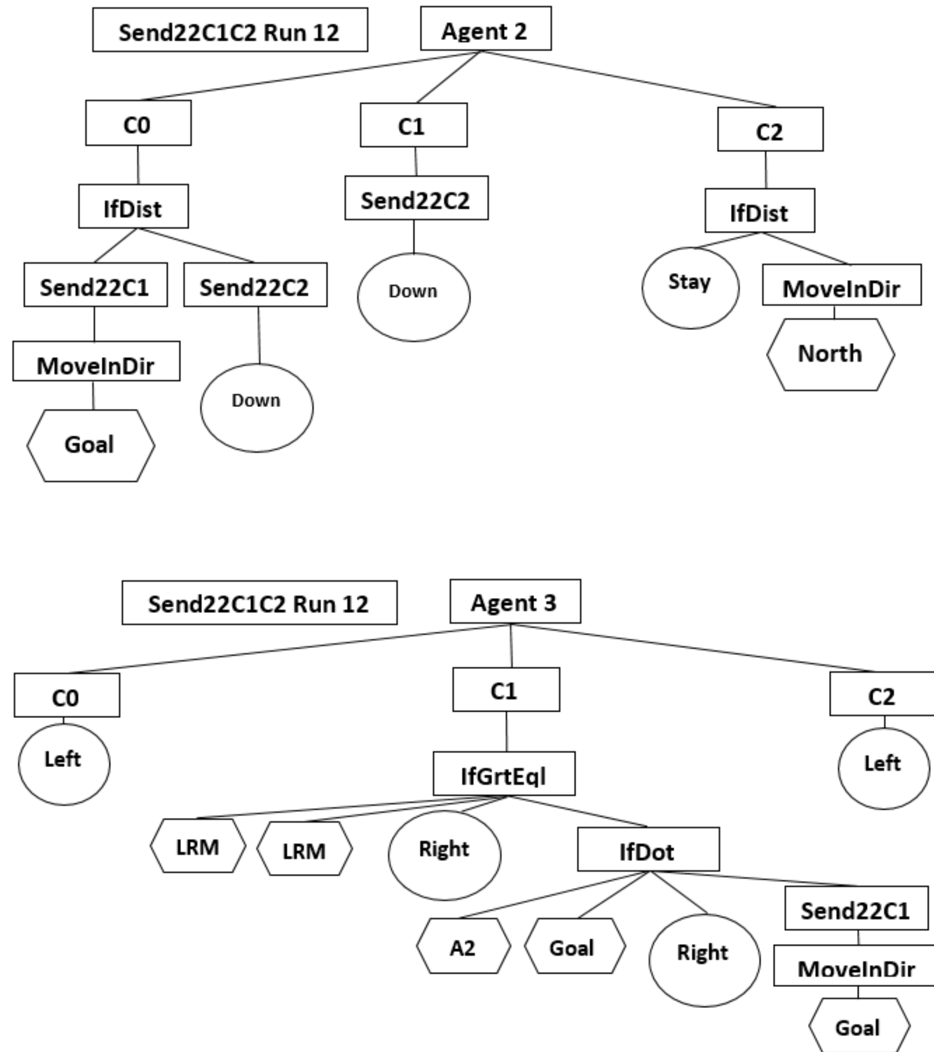


Figure 5.12: Send22C1C2 Agent 2 & Agent 3 Sub-trees, Run 12
A2 = AgentDir2 Terminal

(depending on other conditions in its C2 sub-tree).

Figures 5.15 and 5.16 show agents' sub-trees for one of the best examples of an agent learning the meaning of commands. This example usese the Send22C1orC2 (Run 13). In this example, a sending agent makes decisions about when to send specific commands and a receiving agent is able to associate specific meaning to those commands.

In Figure 5.15, it is seen that Agent 0 and Agent 1 have quite simple sub-trees and do not communicate at all with each other. However, in Figure 5.16 it is seen that Agent 2 and Agent 3 communicate such that Agent 2 is the sender and Agent 3 is the receiver.

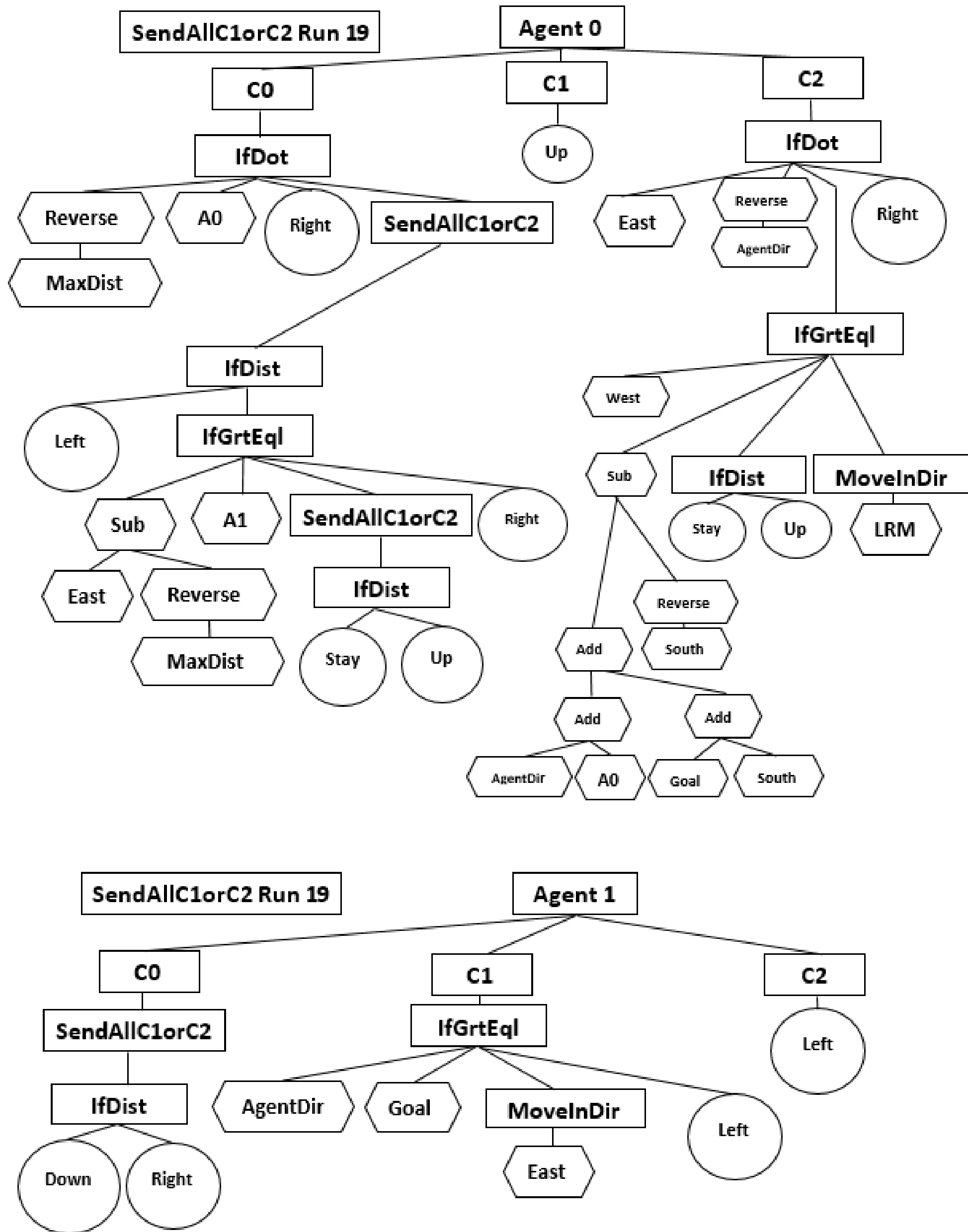


Figure 5.13: SendAllC1orC2 Agent 0 & Agent 1 Sub-trees, Run 19
Agent 1 is the sending agent and sends messages every cycle.
Agent 0 is a receiving agent and moves in direction of prey using LRM node in C2.
A0...A2 = AgentDir0...AgentDir2 Terminals

Despite the fact that the Send command is programmed to decide which command to send, Agent 2 uses the IfGrtEql expression to send a command only when it is in FOV of the prey. Using this expression ensures that Agent 2 will only send a command (the C2 command) when it is in view of the prey and it will never send a C1 command. Thus, Agent 2 decides when to send the C2 command. Figure 5.17

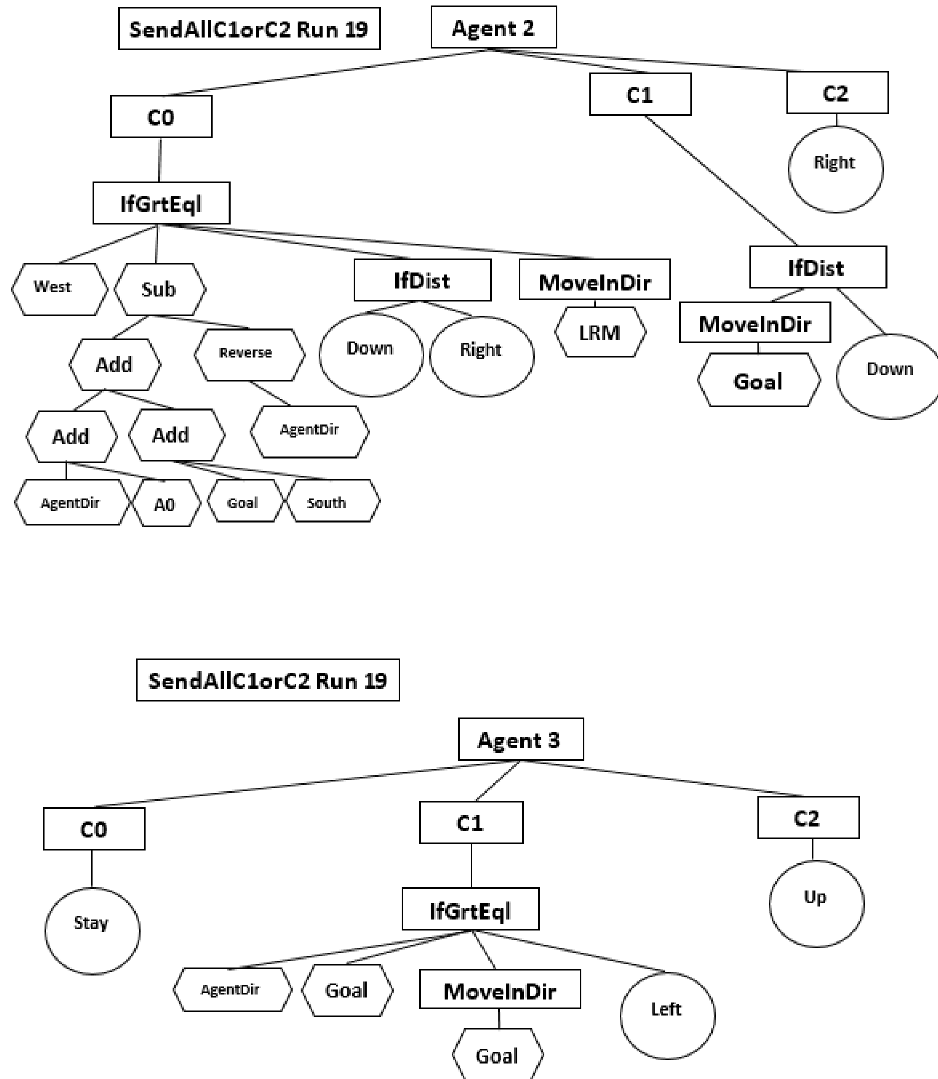
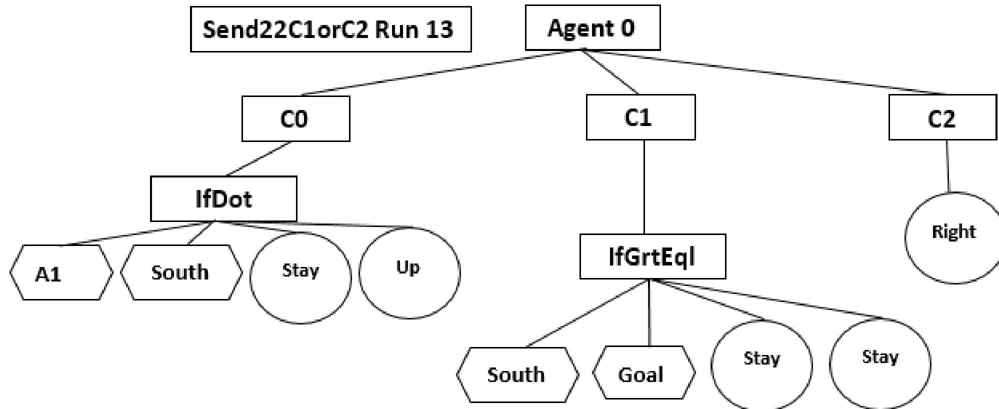


Figure 5.14: SendAllC1orC2 Agent 2 & Agent 3 Sub-trees, Run 19
Agent 2 and Agent 3 are receiving agents and evaluate their C1 or C2 branch each cycle. A0...A2 = AgentDir0...AgentDir2 Terminals

shows how Agent 2 decides to send messages to Agent 3 using its C0 branch. It also shows that the message buffer for Agent 3 contains useful information about the prey (sent from Agent 2).

Agent 3 will evaluate its C0 branch when it does not have a message from Agent 2 and it will evaluate its C2 branch when it does have a message. As seen in Figure 5.16, the C2 branch for Agent 3 contains the *MoveInDir(LRM)* expression as part of its sub-branches. The LRM data will contain relative directional information to the prey. Therefore, Agent 3 associates the C2 command with moving in the direction of the prey. Figure 5.18 shows how Agent 3 uses the contents of its message buffer



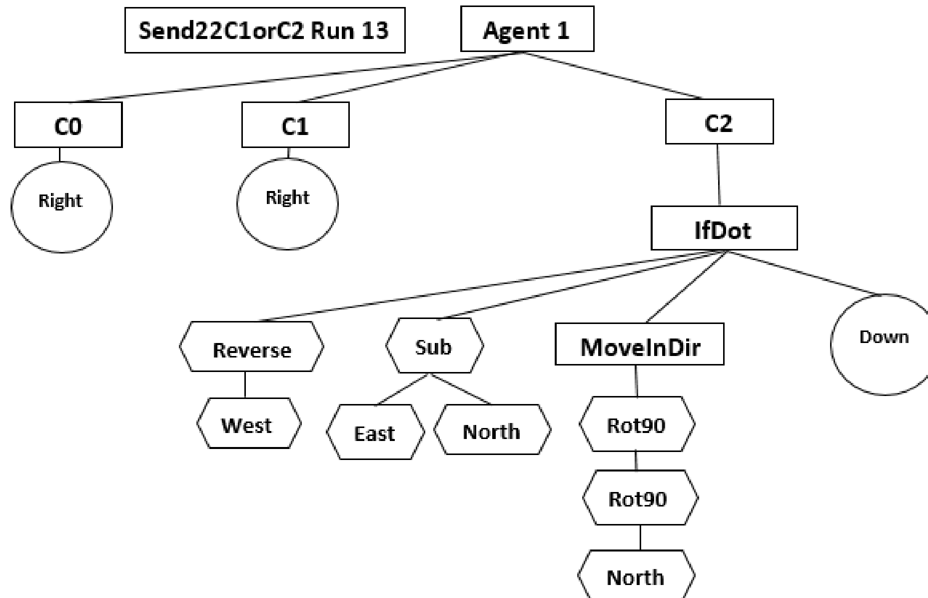


Figure 5.15: Send22C1orC2 Agent 0 & Agent 1 Sub-trees, Run 13
Agent 0 and Agent 1 do not communicate.
A0...A2 = AgentDir0...AgentDir2 Terminals

and its C2 branch to move towards the prey. Together Figure 5.17 and 5.18 are of interest because they show how Agent 2 specifically makes the decision to send the C2 command only when Agent 2 is in FOV of the prey and Agent 3 learns to give the correct meaning to the C2 command by using the message data to move towards the prey.

The communication between Agent 2 and Agent 3 can be seen in more detail by examining images from the playback for Run 13 and by examining the message buffers

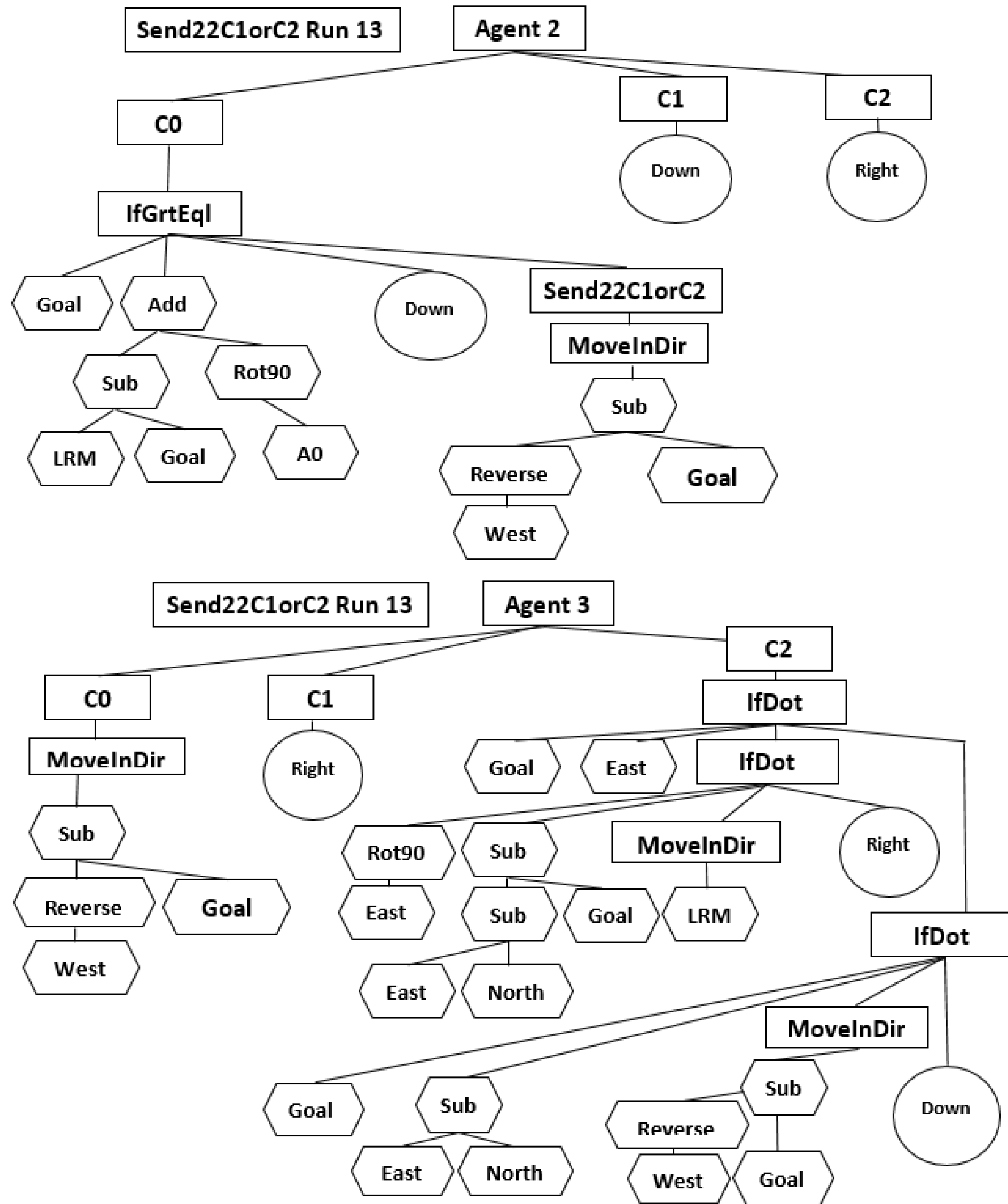
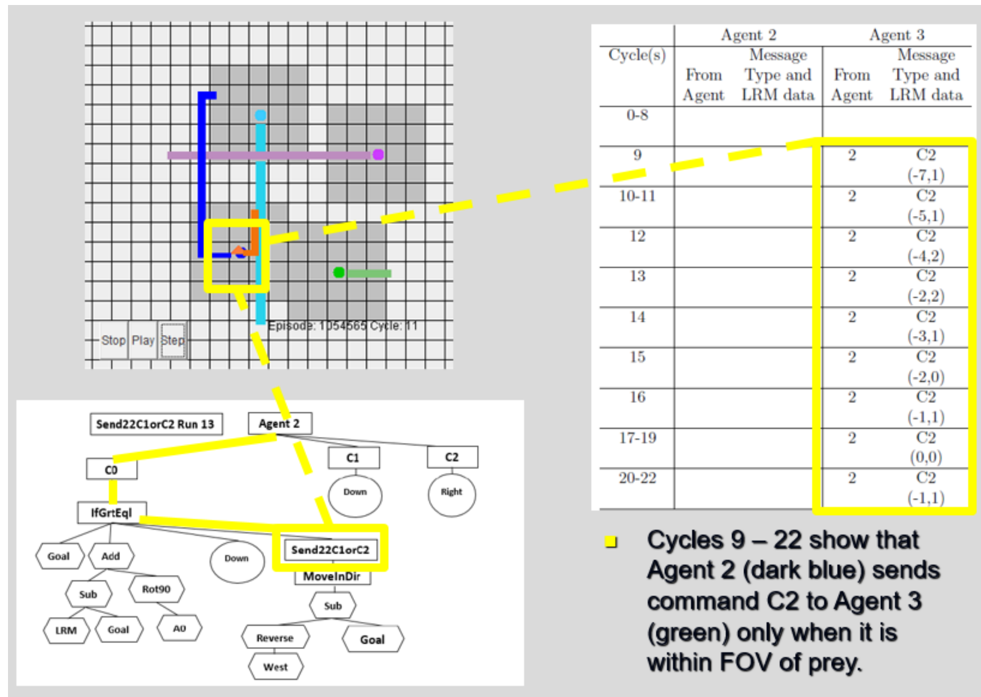


Figure 5.16: Send22C1orC2 Agent 2 & Agent 3 Sub-trees, Run 13
Agent 2 is the sender and Agent 3 is the receiver.
A0...A2 = AgentDir0...AgentDir2 Terminals

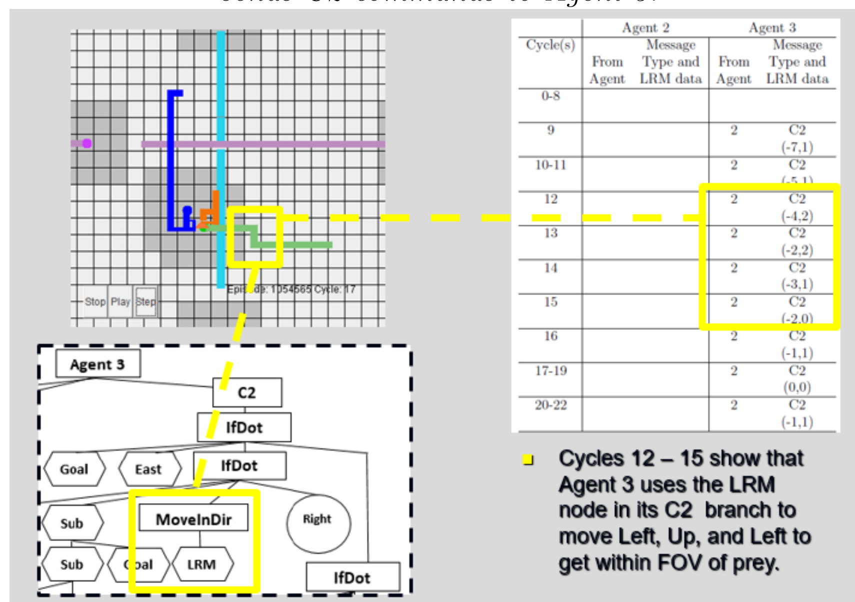
of these two agents. Figures 5.19 to 5.24 show the playback images from Cycles 0 - 22 from an episode in test Run 13. In these figures, coloured lines show the path of the agent from their starting position to the current cycle. Table 5.12 describes the actions of the agents in these figures as events in a time line. Table 5.13 shows the contents of the message buffers for these two agents during the same time period.

Figures 5.19 shows the starting positions of the agents at Cycle 0. Here no agents



Cycle(s)	Agent 2 From Agent	Agent 2 Message Type and LRM data	Agent 3 From Agent	Agent 3 Message Type and LRM data
0-8				
9			2	C2 (-7,1)
10-11			2	C2 (-5,1)
12			2	C2 (-4,2)
13			2	C2 (-2,2)
14			2	C2 (-3,1)
15			2	C2 (-2,0)
16			2	C2 (-1,1)
17-19			2	C2 (0,0)
20-22			2	C2 (-1,1)

Figure 5.17: Deciding Which Command to Send: Send22orC1C2, Run 13
Agent 2 decides to send ONLY when in FOV of prey and sends C2 commands to Agent 3.



Cycle(s)	Agent 2 From Agent	Agent 2 Message Type and LRM data	Agent 3 From Agent	Agent 3 Message Type and LRM data
0-8				
9			2	C2 (-7,1)
10-11			2	C2 (-5,1)
12			2	C2 (-4,2)
13			2	C2 (-2,2)
14			2	C2 (-3,1)
15			2	C2 (-2,0)
16			2	C2 (-1,1)
17-19			2	C2 (0,0)
20-22			2	C2 (-1,1)

Figure 5.18: Understanding Received Commands: Send22orC1C2, Run 13
Agent 3 evaluates C2 branch and uses LRM to move in direction of prey.

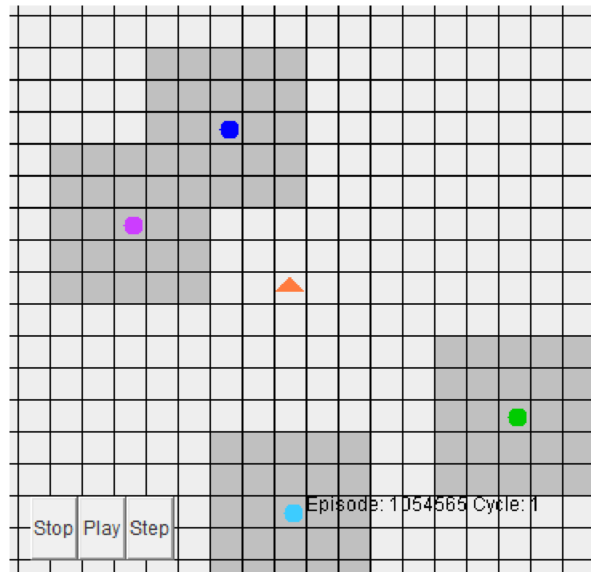


Figure 5.19: Learning the Meaning of Commands: Send22C1orC2 Run 13, **Cycle 1**
Agents at starting positions. Agent 0 (light blue), Agent 1(purple), Agent 2 (dark blue), Agent 3(green)

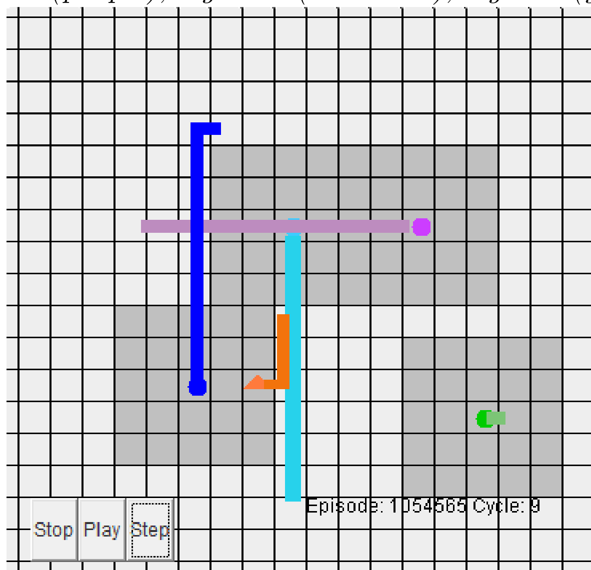


Figure 5.20: Learning the Meaning of Commands: Send22C1orC2 Run 13, **Cycle 9**
Agent 2 is in FOV of the Prey and begins to send messages to Agent 3. Agent 3 uses the message data from Agent 2 to move in direction of Prey. Agent 0 (light blue), Agent 1(purple), Agent 2 (dark blue), Agent 3(green)

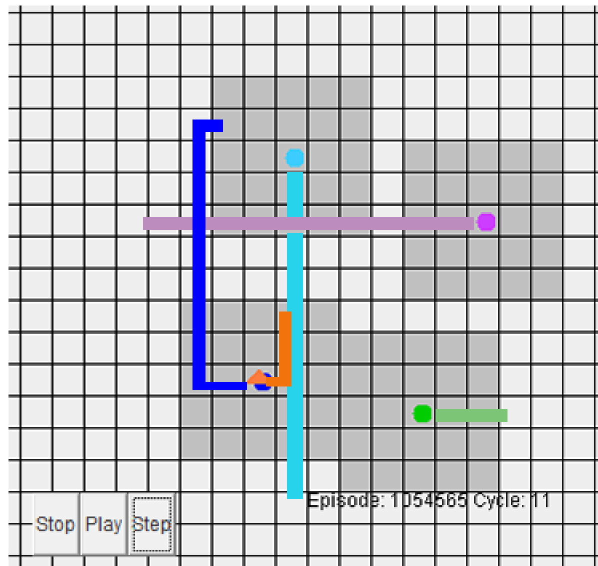


Figure 5.21: Learning the Meaning of Commands: Send22C1orC2 Run 13, **Cycle 11**
Agent 2 stays within FOV of Prey and continues to send messages to Agent 3. Agent 3 uses the message data from Agent 2 to move in direction of Prey. Agent 0 (light blue), Agent 1(purple), Agent 2 (dark blue), Agent 3(green)

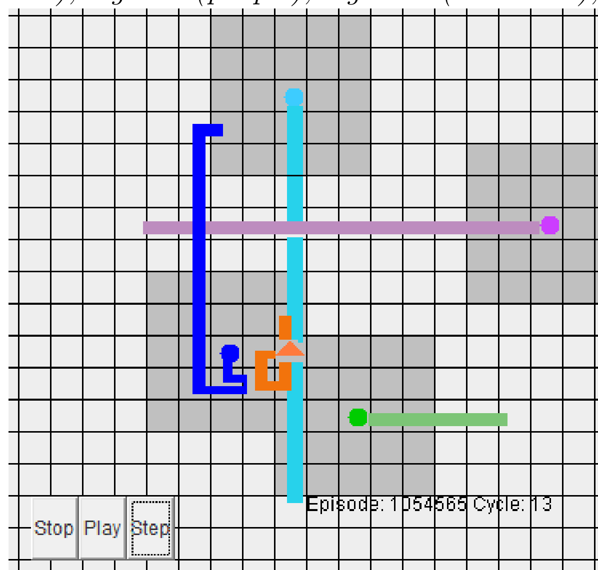


Figure 5.22: Learning the Meaning of Commands: Send22C1orC2 Run 13, **Cycle 13**
Agent 2 stays within FOV of Prey and continues to send messages to Agent 3. Agent 3 uses the message data from Agent 2 to move in direction of Prey. Agent 0 (light blue), Agent 1(purple), Agent 2 (dark blue), Agent 3(green)

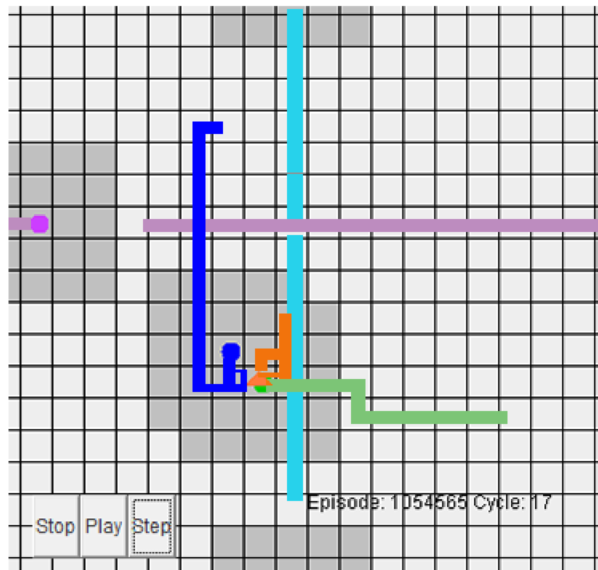


Figure 5.23: Learning the Meaning of Commands: Send22C1orC2 Run 13, **Cycle 17**
Agent 2 stays within FOV of Prey and continues to send messages to Agent 3. Agent 3 uses the message data from Agent 2 to move in direction of Prey. Agent 0 (light blue), Agent 1(purple), Agent 2 (dark blue), Agent 3(green)

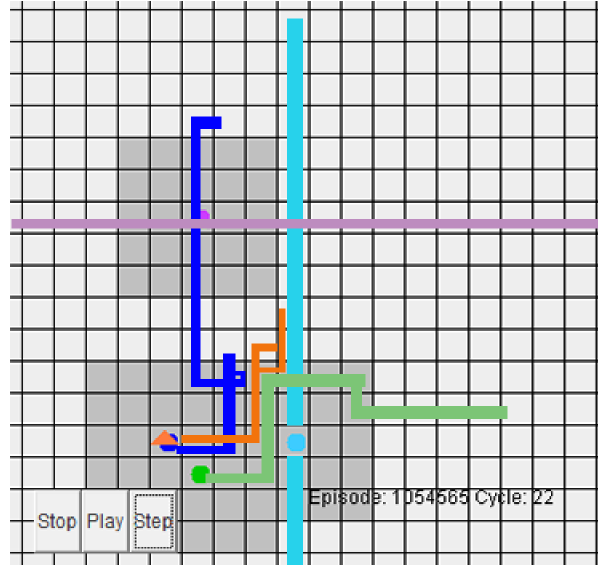


Figure 5.24: Learning the Meaning of Commands: Send22C1orC2 Run 13, **Cycle 22**
Agent 2 stays within FOV of Prey and continues to send messages to Agent 3. Agent 3 uses the message data from Agent 2 to move in direction of Prey. Agent 0 (light blue), Agent 1(purple), Agent 2 (dark blue), Agent 3(green)

Table 5.12: Learning Meaning of Commands: Send22C1orC2 Run 13 Time Line

Time Line Event	Cycle(s)	Description
0 (see Fig. 5.19)	1	Agents at starting positions
1 (not shown)	2-8	*Agent 0 moves Up. Agent 1 moves to the Right. Agent 2, not in FOV (doesn't send messages), moves Left then Down. Agent 3 waits.*
2 (see Fig. 5.20)	9	*Agent 2 enters FOV of prey, sends message to Agent 3 (message contains prey location data). Agent 3 moves towards the direction of prey.*
3 (see Fig. 5.21)	10-11	*Agent 0 continues to move Up. Agent 1 continues to move to the Right. Agent 2 follows prey and sends message to Agent 3. Using LRM data from Agent 2, Agent 3 moves in the direction of prey.*
4 (see Fig. 5.22)	12-13	*Agent 0 continues to move Up. Agent 1 continues to move to the Right. Agent 2 follows prey and sends message to Agent 3. Using LRM data from Agent 2, Agent 3 moves in the direction of prey.*
5 (see Fig. 5.23)	14-17	*Agent 0 continues to move Up (wraps around grid). Agent 1 continues to move to the Right (wraps around grid). Agent 2 follows prey and sends message to Agent 3. Agent 3 moves in the direction of prey. Agent 3 reaches prey at Cycle 17.*
6 (see Fig. 5.24)	18-22	*Agent 0 continues to move Up. Agent 1 continues to move to the Right. Agent 2 follows prey and sends message to Agent 3. Using LRM data from Agent 2, Agent 3 moves in the direction of prey.*

have messages in their buffers and their moves are made by evaluating their C0 branches. Figure 5.20 displays the path each agent takes from Cycles 0-9. At Cycle 9, Agent 2 (dark blue) is in FOV of the prey and remains in view of the prey until Cycle 30.

At Cycle 9, Agent 2 begins to send messages to Agent 3. Table 5.13 shows that the contents of Agent 3's message buffer contains one message from Agent 2 in each cycle (from Cycle 9 to Cycle 22). Each message contains the message command C2

Table 5.13: Send22C1orC2 Test Run 13 Message Buffer

Cycle(s)	Agent 2		Agent 3	
	From Agent	Message Type and LRM data	From Agent	Message Type and LRM data
0-8				
9			2	C2 (-7,1)
10-11			2	C2 (-5,1)
12			2	C2 (-4,2)
13			2	C2 (-2,2)
14			2	C2 (-3,1)
15			2	C2 (-2,0)
16			2	C2 (-1,1)
17-19			2	C2 (0,0)
20-22			2	C2 (-1,1)

Agents' Message Buffer Contents at each cycle. (Cycles 0-22)
All agent buffers are empty except for Agent 3's buffer.

and message data (LRM) with relative directional data to the prey. From Cycles 9-22 Agent 3 evaluates its C2 branch causing it to move in the direction of the prey. This movement can be seen in Figures 5.20 to 5.24. Looking at the GP structure, message buffers and playback of the test runs show that the Send22C1orC2 protocol is successful in allowing agents to associate meaning to commands and message data that allow them to achieve their goal of finding and following the prey.

5.3.6 Summary of Results

Statistical analysis shows that the C1 protocol outperforms the C1C2 protocol and it peforms equally as well as the C1orC2 protocol. Qualitative analysis shows that there is an emergent behaviour in message passing that produce "sender" and "receiver"

agents across all protocols. The qualitative analysis also shows that the C1C2 and C1orC2 protocols produce more complex sub-trees for the predator agents than the C1 protocols. The success of the simpler GPs created by the C1 protocol is influenced by the *MoveInDir(Goal)* expression. The success of the more complex GPs created by the C1C2 and C1orC2 protocols is based on their ability to create "sender" agents that make decisions on which type of command to send and "receiver" agents that are able to associate the correct meaning of the sent command. Although there is room for improvement (i.e. not all agents learn to communicate or associate meanings to commands), the C1C2 and C1orC2 protocols show promise when it comes to the ability of agents to learn the meaning of commands. Future work could investigate whether increasing the grid size from 20x20 to 40x40 would affect the emergent behaviour in message passing or if it would affect the influence of *MoveInDir(Goal)*.

5.4 Influence of Prey Movement Type in Training and Testing on Send22 Protocol

The previous sections in this chapter examined the ability of evolved predator agents to learn the meaning of commands with the goal of finding and following a random moving prey. The results showed that the C1 and C1orC2 protocols performed equally as well. The C1 protocol produced simple GP sub-trees that were influenced by the *MoveInDir(Goal)* expression and the C1orC2 protocols produced more complex sub-trees that were able to associate specific meanings to commands of which the meaning was determined by sending agents.

This section, using the Send22C1 and Send22C1orC2 protocols and the same experiment details described earlier in this chapter, investigates whether changes in the prey's movement in training and testing influence the predator agents' ability to achieve their goal. For example, some of the questions asked in creating this experiment were: How does training predator agents with a linear moving prey affect the test results when the prey moves linearly, randomly or uses both types of movement (i.e. switching from linear movement only to random movement only) in the test runs? Likewise, how does training predator agents with a random moving prey affect the test results when the prey moves with linear movement, random movement or switches from one to the other in testing? Finally, how does training predator agents with a prey that switches from linear movement to random movement affect the test results when the test runs use a prey that moves linearly only, randomly only or

Table 5.14: Prey Movement Types in Training and Testing

Training & Testing Types	Training Episodes (10 total) Prey Movement Type(s)	Testing Episodes (30 total) Prey Movement Type(s)
SendC1_L_L	10 episodes = linear	30 episodes = linear
SendC1_L_R	10 episodes = linear	30 episodes = random
SendC1_L_LR	10 episodes = linear	1st 15 episodes = linear & 15 episodes = random
SendC1_R_L	10 episodes = random	30 episodes = linear
SendC1_R_R	10 episodes = random	30 episodes = random
SendC1_R_LR	10 episodes = random	1st 15 episodes = linear & 2nd 15 episodes = random
SendC1_LR_L	1st 5 episodes = linear & 2nd 5 episodes = random	30 episodes = linear
SendC1_LR_R	1st 5 episodes = linear & 2nd 5 episodes = random	30 episodes = random
SendC1_LR_LR	1st 5 episodes = linear & 2nd 5 episodes = random	1st 15 episodes = linear & 2nd 15 episodes = random

The Send22 and Send22C1orC2 protocols are examined using the above prey movement types in training and testing runs.

switches from one movement type to the other type of movement?

5.4.1 Training and Testing Types and Methods

Table 5.14 gives a description of all the training and testing combinations of prey movement types examined in this experiment. The training and testing methods are the same as listed in previous sections of this chapter. The fitness function and the total number of cycles and episodes in training and testing are also the same as discussed previously. However, depending on the type of training and testing being examined, the number of episodes dedicated to the prey's linear movement and the prey's random movement are different and are found in Table 5.14. The training results for linear, random and linear-random training are shown in Figures 5.25, 5.26 and 5.27 respectively. The testing results for linear, random and linear-random testing are shown in Tables 5.15, 5.16 and 5.17 respectively.

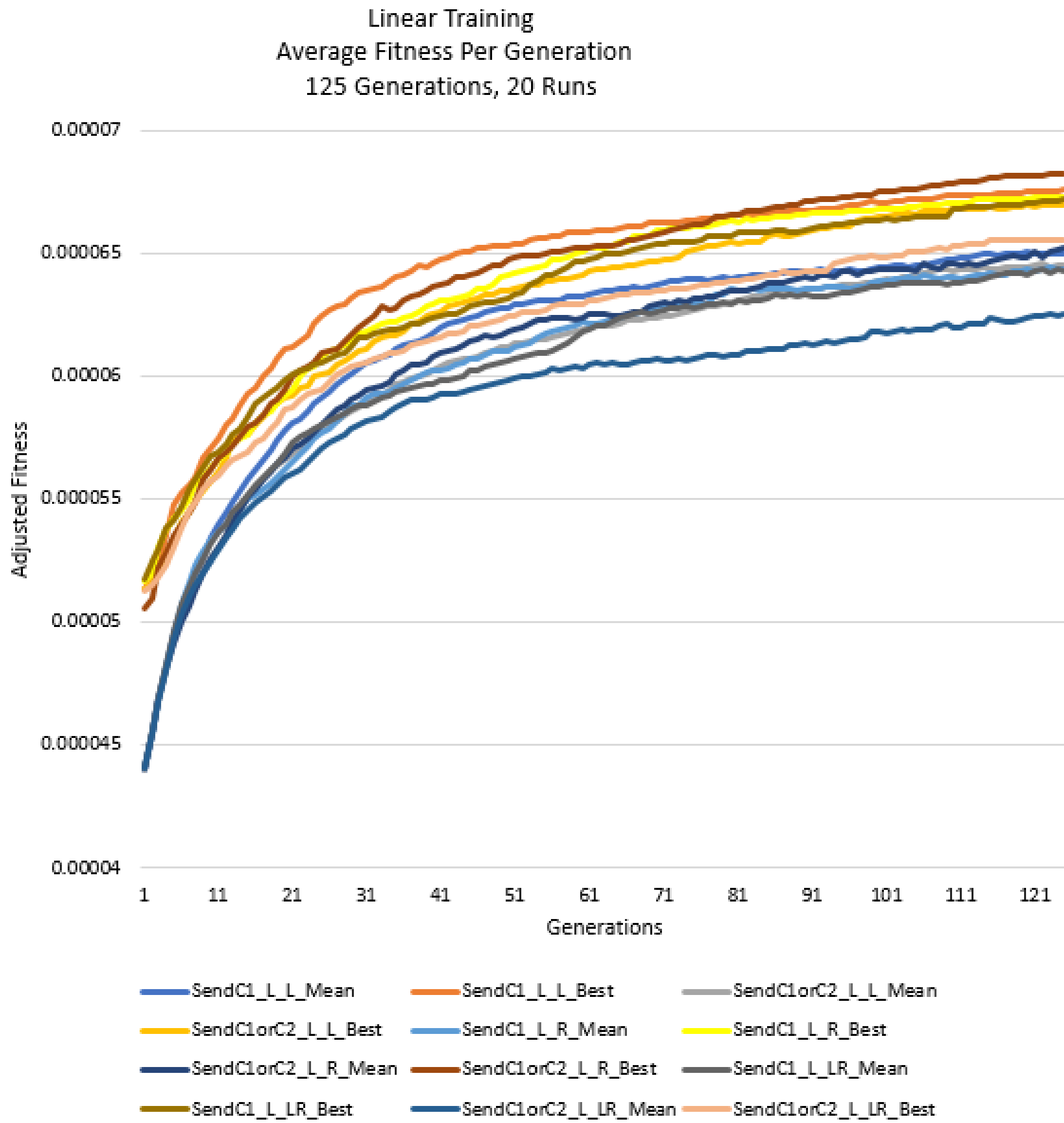


Figure 5.25: Send22 Protocol - Linear Training (_L_L, _L_R, _L_LR)

Table 5.15: Fitness Summary Linear Training (20 Test Runs)

Communication Type	Min Fitness	Average Fitness	Max Fitness
Send22C1_L_L	1618	1791	1959
Send22C1_L_R	1918	2025	2190
Send22C1_L_LR	1747	1932	2120
Send22C1orC2_L_L	1620	1823	2010
Send22C1orC2_L_R	1916	2020	2196
Send22C1orC2_L_LR	1757	1919	2159

Fitness is a minimization function. See Equation (5.4).

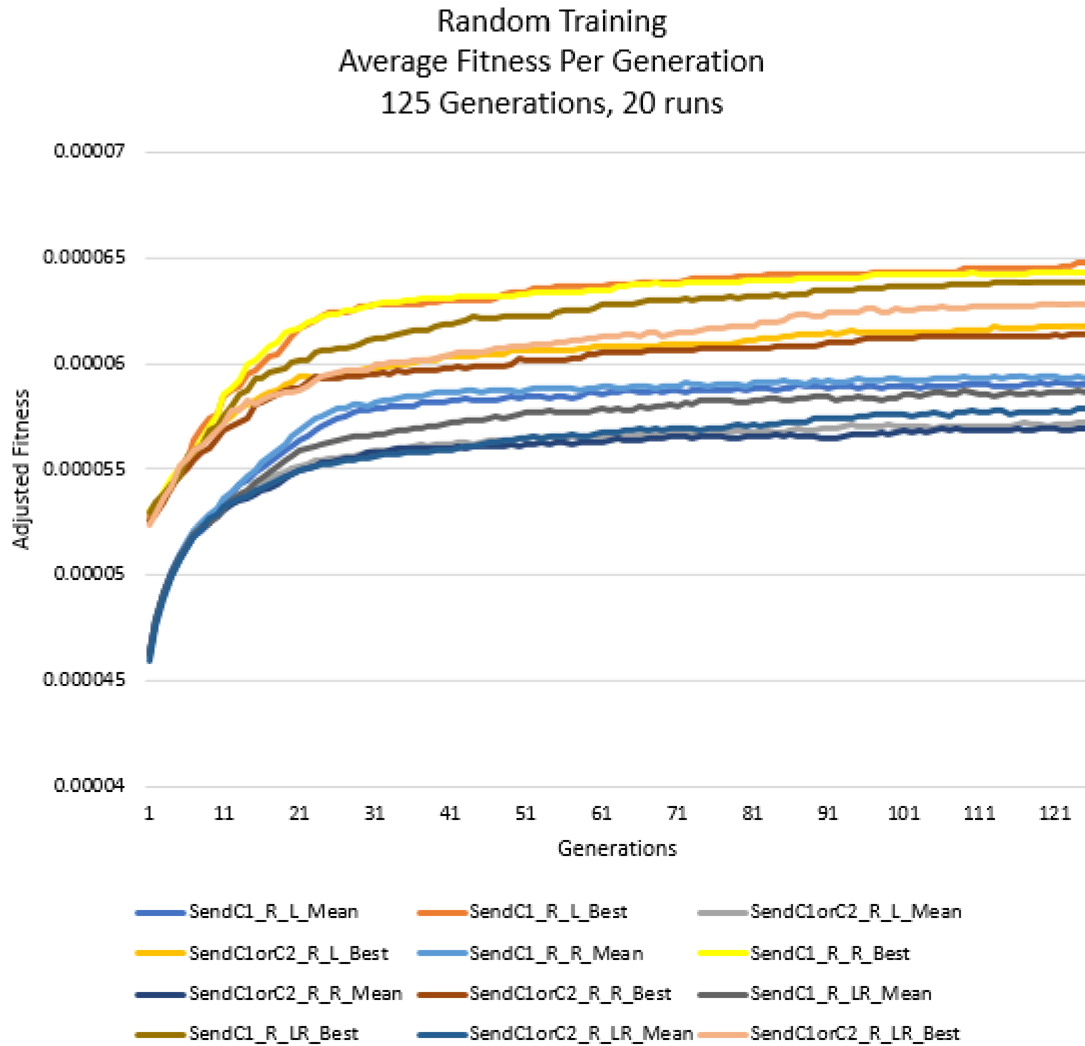


Figure 5.26: Send22 Protocol - Random Training (_R_L, _R_R, _R_LR)

Table 5.16: Fitness Summary Random Training (20 Test Runs)

Communication Type	Min Fitness	Average Fitness	Max Fitness
Send22C1_R_L	1917	2099	2338
Send22C1_R_R	1681	1805	1964
Send22C1_R_LR	1717	1957	2120
Send22C1orC2_R_L	1980	2151	2432
Send22C1orC2_R_R	1609	1850	2032
Send22C1orC2_R_LR	1757	1964	2163

Fitness is a minimization function. See Equation (5.4).

5.4.2 Discussion of Results

This section discusses both the statistical and qualitative aspects of the best and worst results from above.

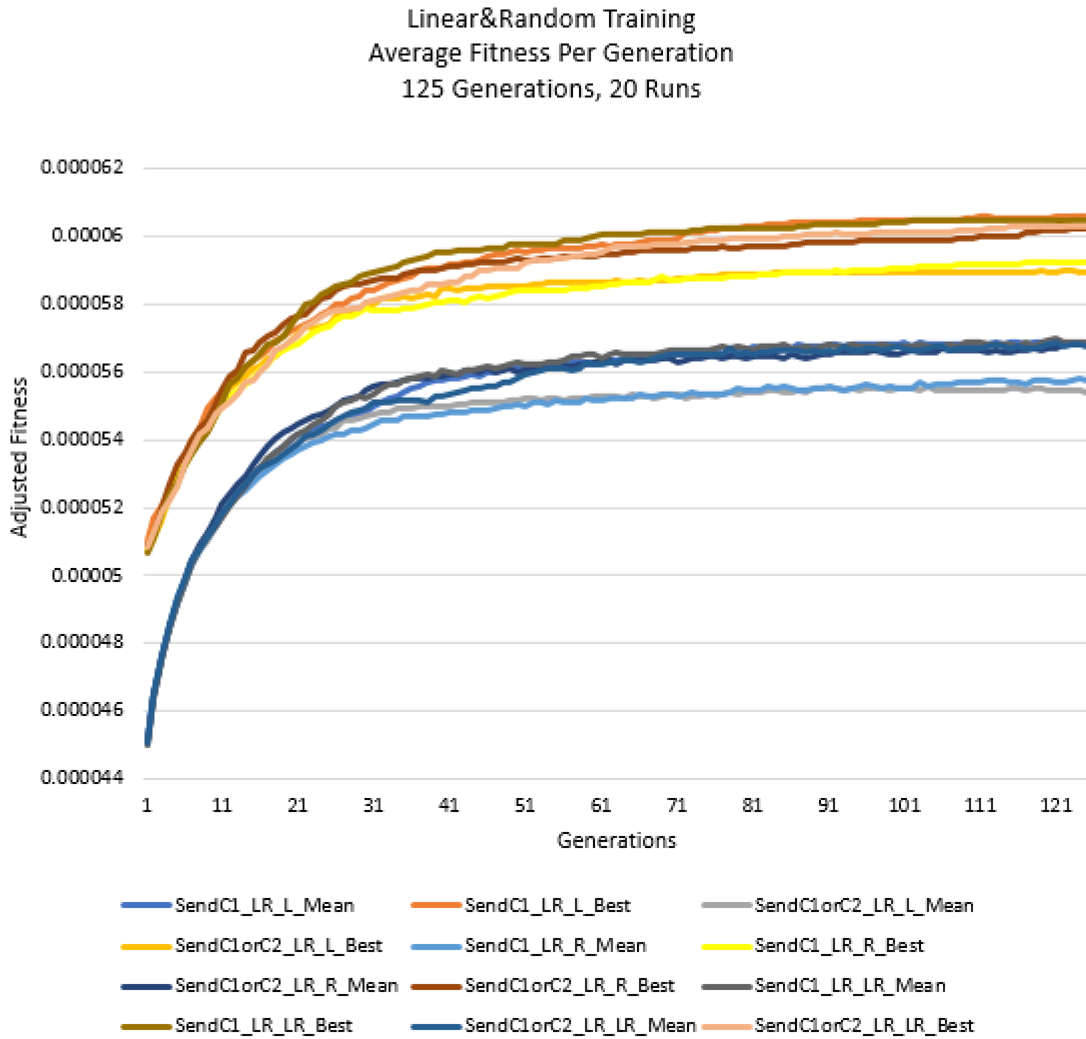


Figure 5.27: Send22 Protocol - Linear/Random Training (_LR_L, _LR_R, _LR_LR)

Table 5.17: Fitness Summary Linear-Random Training (20 Test Runs)

Communication Type	Min Fitness	Average Fitness	Max Fitness
Send22C1_LR_L	1732	1918	2239
Send22C1_LR_R	1758	1929	2149
Send22C1_LR_LR	1790	1907	2147
Send22C1orC2_LR_L	1794	1941	2262
Send22C1orC2_LR_R	1718	1912	2052
Send22C1orC2_LR_LR	1750	1932	2182

Fitness is a minimization function. See Equation (5.4).

Statistical Analysis

Figure 5.25, shows the linear training results for Send22C1 and Send22C1orC2. This graph shows that the performance of the best fitness (orange range) are fairly equal

across all protocols in training except for protocol SendC1orC2_L_LR which appears lower on the graph (light orange colour). Figure 5.26 shows that the best fitness (orange range of colours on graph) and mean fitness (blue range of colours on graph) results of the Send22C1 protocol outperforms the best of the Send22C1orC2 protocol. Finally, Figure 5.27 show that all protocols perform equally as well in later generations except for SendC1_LR_R and SendC1orC2_LR_L.

The test run results shown in Tables 5.15 and 5.16 show that the average fitness value for the Send22C1_L_L type (1791) outperforms all other types when the prey moves in a linear direction in training and that the average fitness value for the Send22C1_R_R type (1805) outperforms all other types when the prey moves in a random direction in training. Similarly, the test run results for Table 5.17 show that the average fitness value for the Send22C1_LR_LR type (1907) outperforms all other types when the prey moves first moves in a linear direction and then in a random direction in training.

To verify the significance of the test results the One-Way ANOVA test (using Minitab [37]) with a 95% confidence interval is used. The ANOVA test uses a two-tailed T-test with 6 factors, where each factor represents one training/testing type of each communication protocol (i.e. 3 training/testing types for each protocol, Send22C1 and Send22C1orC2). In linear training (see Table 5.15), the ANOVA test results in the $P - Value < \alpha$ (see Appendix A) indicating that there is a significant difference in the data. Similarly, the ANOVA test for random training (see Table 5.16) shows the $P - Value < \alpha$ indicating that there is also a significant difference in this data. However, the ANOVA test results in $P - Value > \alpha$ for the linear/random training types (see Table 5.17) do not show a significant different in the results.

Additionally, ANOVA tests were run on the same data above but the data was regrouped into *testing prey movement* types instead of *training prey movement* types. This data was organized such that all test runs with linear prey movement type were in the first group, all test runs with random prey movement type were in the second group and finally all test runs with linear and random prey movement were in a third group. An ANOVA test was performed on each group. For example, the ANOVA test for the *testing random prey movement* group included: Send22C1_L_R, Send22C1orC2_L_R, Send22C1_R_R, Send22C1orC2_R_R, Send22C1_LR_R, and Send22C1orC2_LR_R.

The ANOVA results for the *testing prey movement* types were similar to the results for the *training prey movement* types. That is, there is a significant difference in *testing prey movement* types for linear movement only and random movement only.

Table 5.18: Tukey Comparisons for Training Movement Types

ANOVA Groups	Communication Types	Group Letter	Average Fitness
	Send22C1_L_R	A	2025
	Send22C1orC2_L_R	A	2020
Training Linear Movement	Send22C1_L_LR	B	1931
	Send22C1orC2_L_LR	B	1918
	Send22C1_L_L	C	1790
	Send22C1orC2_L_L	C	1822
	Send22C1_R_L	A	2098
	Send22C1orC2_R_L	A	2150
Training Random Movement	Send22C1_R_LR	B	1957
	Send22C1orC2_R_LR	B	1963
	Send22C1_R_R	C	1805
	Send22C1orC2_R_R	C	1850
	Send22C1_LR_L	A	1918
	Send22C1orC2_LR_L	A	1941
Training Linear & Random Movement	Send22C1_LR_R	A	1929
	Send22C1orC2_LR_R	A	1911
	Send22C1_LR_LR	C	1906
	Send22C1orC2_LR_LR	C	1931

Types that do not share the same letter are significantly different.

But there is not a significant difference in test runs when the prey moves in both a linear and random movement (see Appendix A).

In order to identify which factors in the ANOVA tests have different means the Tukey method [38] for multiple comparisons is used for both the *training prey movement* and *testing prey movement* types. The results are shown in Tables 5.18 and 5.19 respectively. In these tables, types that are similar share the same group letter. Types that are significantly different (i.e. their interval (range of difference of mean) does not contain a zero) do not share the same group letter [39]. These tables show that in both the *training prey movement* and *testing prey movement* types the _L_L types for both Send22C1 and Send22C1orC2 share the same group letter. The _L_R types for both Send22C1 and Send22C1orC2 share another group letter and the _L_LR types for both Send22C1 and Send22C1orC2 share their own group letter.

Table 5.19: Tukey Comparisons for Testing Movement Types

ANOVA Groups	Communication Types	Group Letter	Average Fitness
	Send22C1_R_L	A	2098
	Send22C1orC2_R_L	A	2150
Testing Linear Movement	Send22C1_LR_L	B	1918
	Send22C1orC2_LR_L	B C	1941
	Send22C1_L_L	D	1790
	Send22C1orC2_L_L	C D	1822
	Send22C1_L_R	A	2025
	Send22C1orC2_L_R	A	2020
Testing Random Movement	Send22C1_R_R	B C	1805
	Send22C1orC2_R_R	C D	1850
	Send22C1_LR_R	B	1929
	Send22C1orC2_LR_R	B	1911
	Send22C1_LR_LR	A	1918
	Send22C1orC2_LR_LR	A	1941
Testing Linear & Random Movement	Send22C1_LR_LR	A	1929
	Send22C1orC2_LR_LR	A	1911
	Send22C1_LR_LR	C	1906
	Send22C1orC2_LR_LR	C	1931

Types that do not share the same letter are significantly different.

The same pattern is found across all movement combinations. That is, significant differences are found in how the groups are arranged in training/testing movement types. As seen in the test data earlier, the best of these types (with the lowest fitness value) is in the Send22_L_L in linear training/testing tests and Send22_R_R in random training/testing tests. Although the Send22C1_LR_LR type outperformed the others in the training/testing tests, there was not a significant difference found in these results.

Qualitative Analysis

The statistical analysis shows that the Send22C1_L_L protocol is the best performer (in average fitness value) in both the *training prey movement* and *testing prey move-*

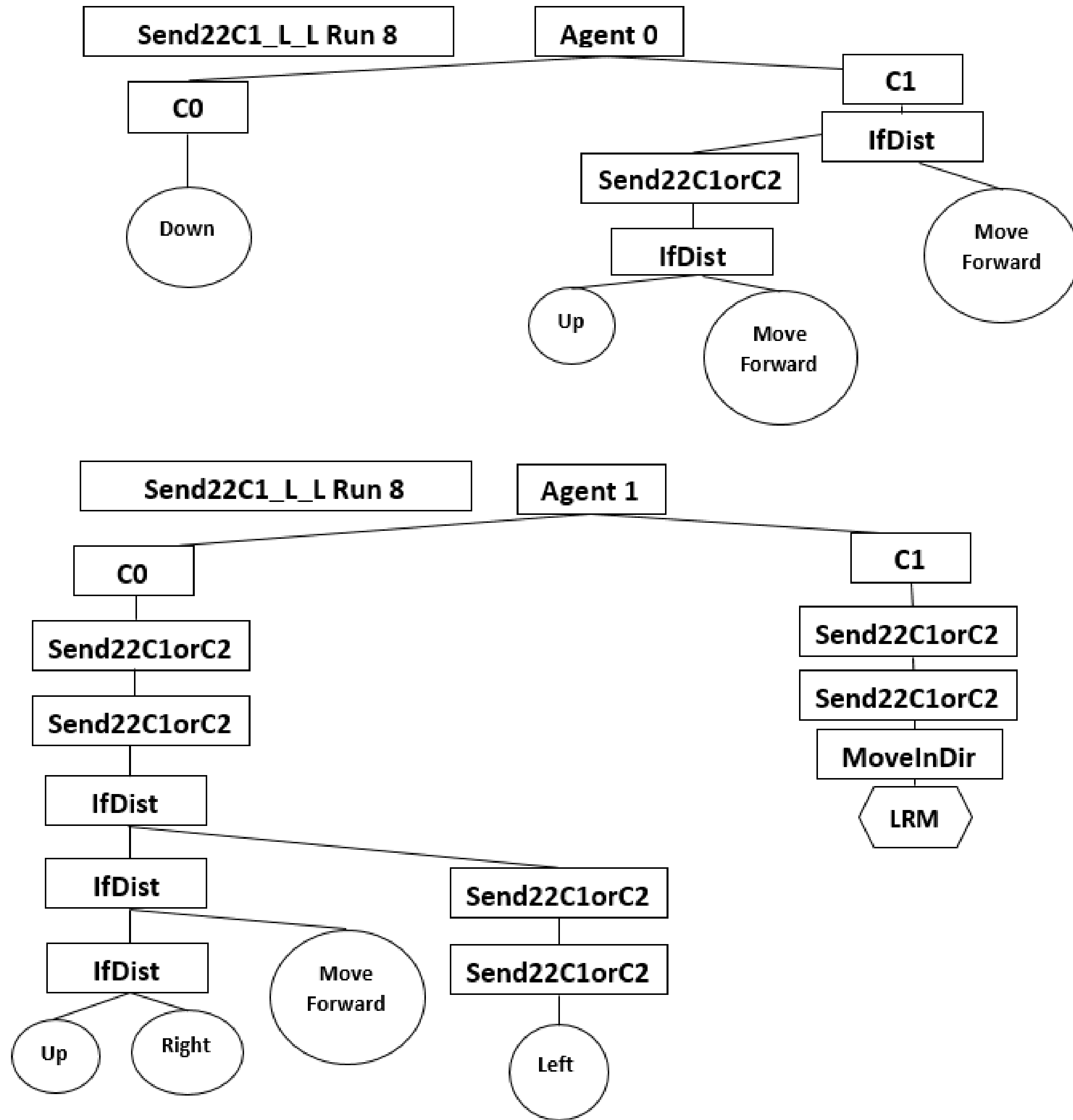


Figure 5.28: Send22C1_L_L Agent 0 & Agent 1 Sub-trees, Run 8
Agent 0 uses C1 branch to send to Agent 1 when in FOV of prey.
Agent 1 uses the LRM node in C1 branch to move towards prey.
A0...A2 = AgentDir0...AgentDir2 Terminals

ment types. It also shows that the best performers for each group type were the ones which used the same prey movement type for both training and testing. For example, the linear training group showed that the _L_L types (linear movement in training and testing) were the best of their group, the random training group showed that the _R_R types (random movement in training and testing) were the best of their group and the _LR_LR types all did equally as well in the linear/random training group.

The GP structures and message buffers of the top performers for the Send22C1_L_L

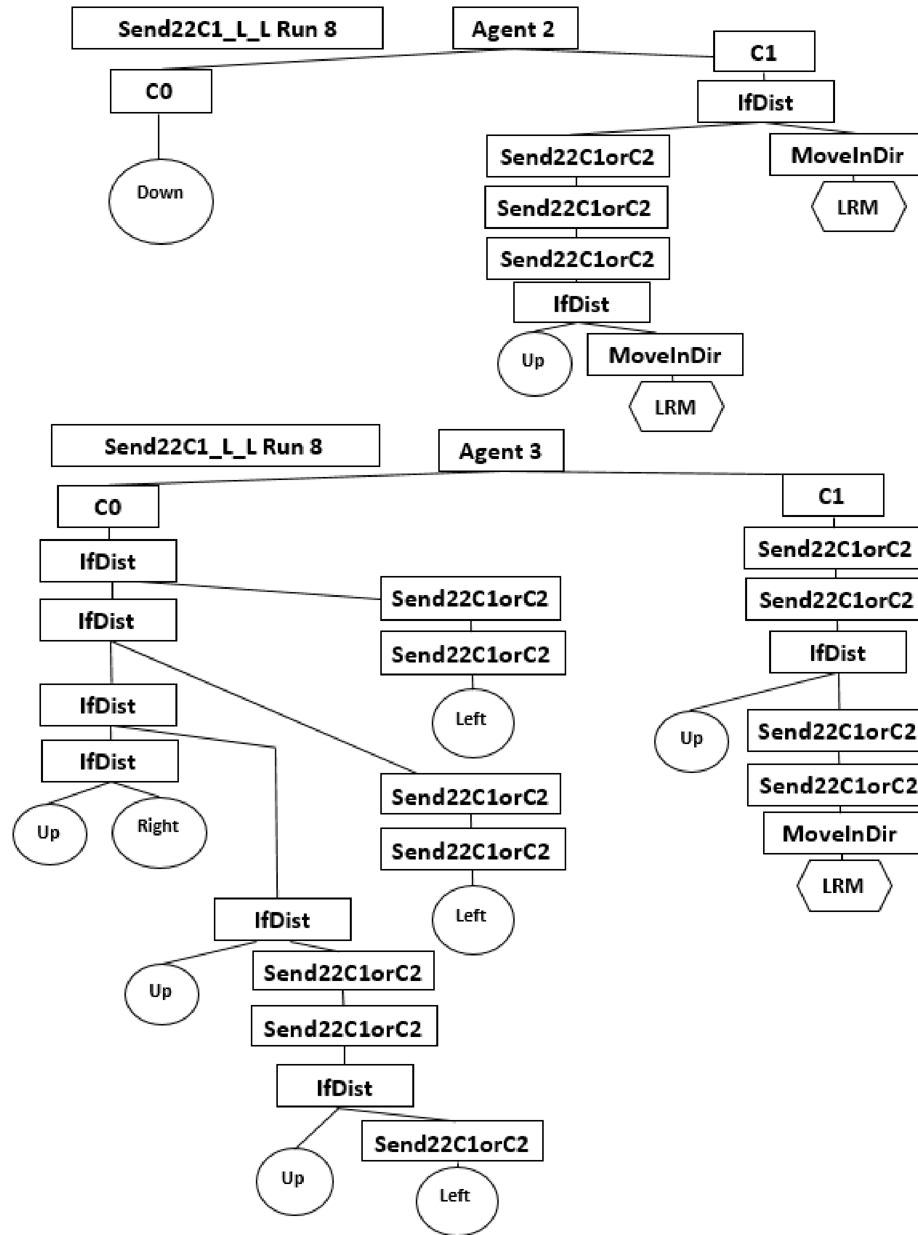


Figure 5.29: Send22C1_L_L Agent 2 & Agent 3 Sub-trees, Run 8
Agent 2 uses C1 branch to send multiple times to Agent 3 when in FOV of prey.
Agent 3 uses the LRM node in C1 branch to move towards prey.
A0...A2 = AgentDir0...AgentDir2 Terminals

and Send22C1orC2_L_L show that these types produce trees that allow predator agents to communicate, make decisions and associate meaning to commands as well as use message data that is sent to them. Figures 5.28 and 5.29 show the GP tree for the best run, Run 8, for Send22C1_L_L and Figures 5.30 and 5.31 show the GP tree for the best run, Run 17, for Send22C1orC2_L_L. These figures, as well as the mes-

sage buffers, demonstrate that all predator agents communicate with their "partner" except for Agent 3 in Send22C1orC2_L_L.

It is seen in Figure 5.28 that Agent 0 uses the IfDist expression to send a message to Agent 1 only if it is in FOV of the prey. Agent 1, upon receipt of the C1 command from Agent 0, will use its C1 branch to move in the direction of the prey (using the *MoveInDir(LRM)* expression). Similar to Agent 0, Agent 2 sends multiple messages to Agent 3 only if Agent 2 is in FOV of the prey (see Figure 5.29). These messages will have directional information to the prey. Agent 3 uses this information in its C1 branch to move in the direction of the prey (again using the *MoveInDir(LRM)* expression). Thus, receiving agents are able to associate the proper meaning of the command determined by the sending agents.

The same type of communication and association of commands is found in Figures 5.30 and 5.31 for Send22C1orC2_L_L. Figure 5.30 shows that Agent 1 sends six messages to Agent 0 (flooding its message buffer, similar to the guard behaviour found in Section 4.3.2) if Agent 1 is in FOV of the prey. The C2 branch for Agent 0 contains the *MoveInDir(LRM)* expression. Like previous examples, this will help Agent 0 reach the prey if it is not already in FOV of the prey. Figure 5.31 demonstrates that Agent 2 only sends a message to Agent 3 if Agent 2 is in FOV of the prey. This allows Agent 3 to evaluate its C2 branch only when there is valid data in the LRM node. This C2 branch does contain the *MoveInDir(LRM)* expression, therefore the messages sent to Agent 3 by Agent 2 help Agent 3 find the prey if it is not in FOV of the prey. Again in this communication type, receiving agents are able to associate the correct meaning of the command (and data) determined by the sending agents.

5.4.3 Summary of Results

The data shows that the _L_L types for Send22C1 and Send22C1orC2 both create GPs that allow agents to communicate successfully. Sending agents make decisions on when (and what type of command) to send. Receiving agents are able to associate meanings to the commands and are able to use the message data to help them find the prey. The results also show that training is important and does influence the success of test runs. For example, training agents with a linear moving prey does not help agents when the prey moves randomly, or linearly/randomly in test runs. Likewise, training agents with a random moving prey does not help the agents when the prey moves linearly or linearly/randomly in test runs.

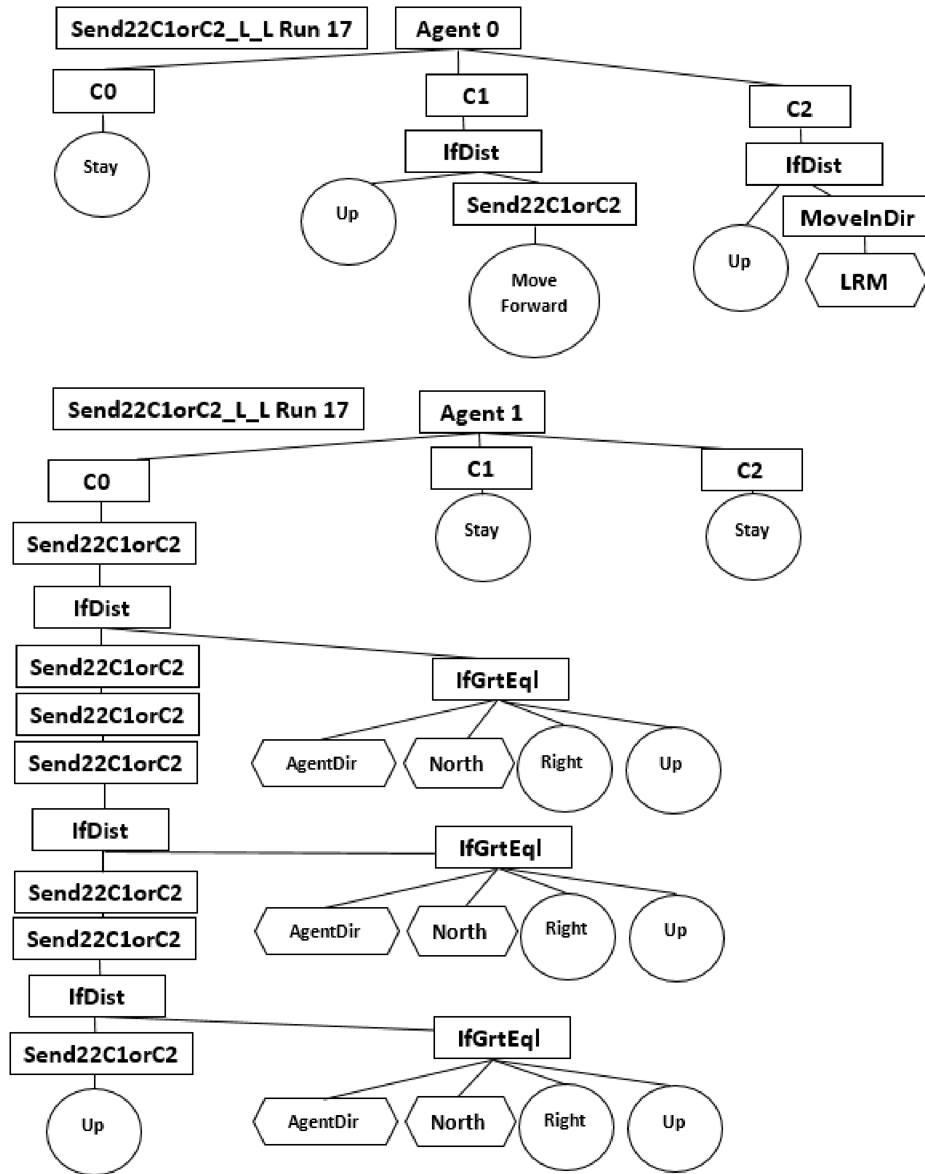


Figure 5.30: Send22C1orC2_L_L Agent 0 & Agent 1 Sub-trees, Run 17
Agent 0 uses the LRM node in C2 branch to move towards prey.
Agent 1 sends 6 messages to Agent 0 when in FOV of prey.
A0...A2 = AgentDir0...AgentDir2 Terminals

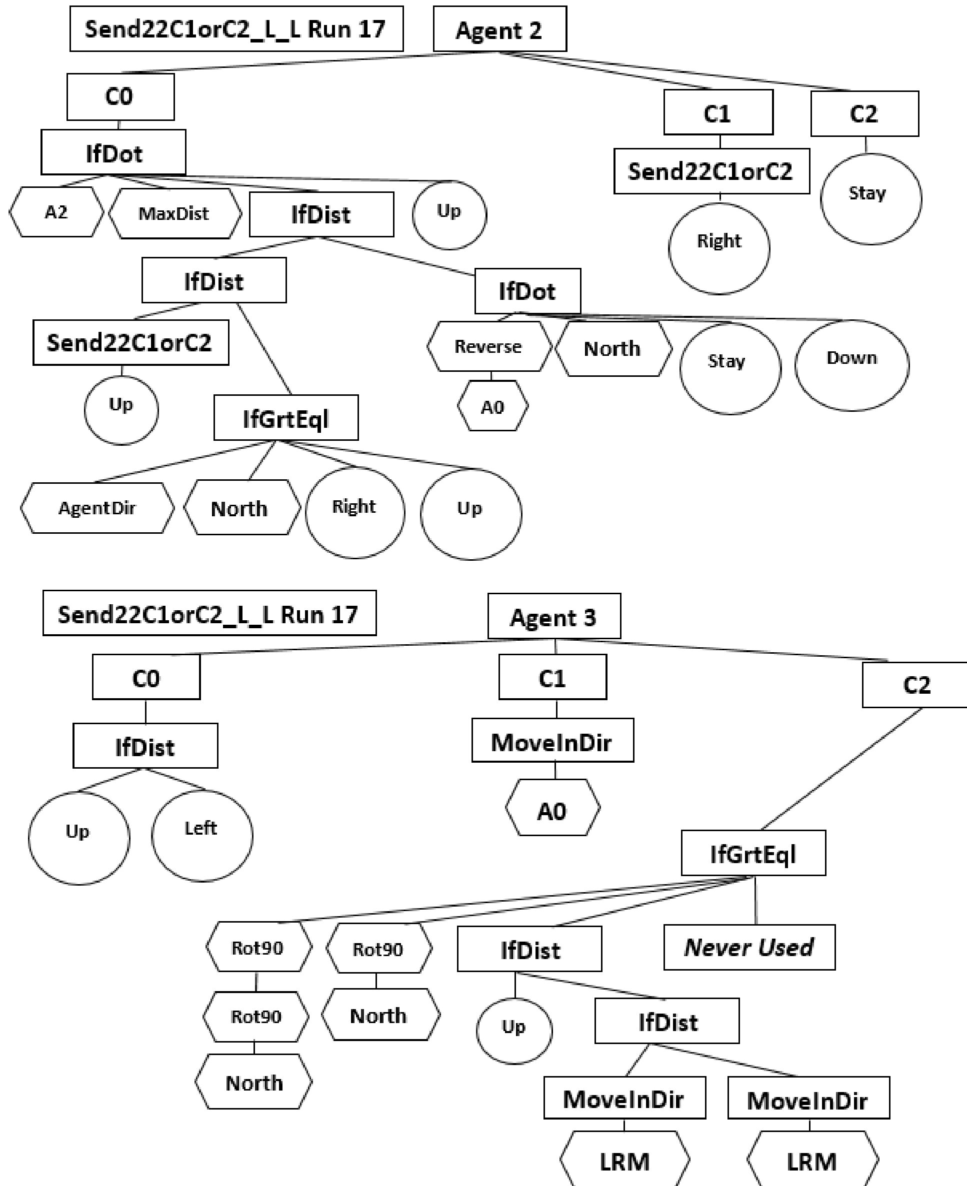


Figure 5.31: Send22C1orC2_L_L Agent 2 & Agent 3 Sub-trees, Run 17
Agent 2 uses C0 branch to send to Agent 3 when in FOV of prey.
Agent 3 uses the LRM node in C2 branch to move towards prey.
A0...A2 = AgentDir0...AgentDir2 Terminals

Chapter 6

Agent Evolution in Ms. Pac-Man Environment

6.1 Problem and Environment

The goal of this experiment is to see if predator agents, using a similar GP language as in Section 5.2, can learn to find and follow the prey as it moves randomly in the Ms. Pac-Man game environment [20]. The solution for this experiment focuses on the top protocols, Send22C1 and Send22C1orC2, found in Chapter 5. A new version of the Send22 protocol type, Send22PvA is also examined. The results show that the GP Language does not allow the predator agents to follow the prey as well as it did in the Pursuit Domain simulator [19]. However, agents are able to learn the meaning of commands and an emergent pattern in the agents' sub-trees reveals there is one ghost agent (of 4 ghosts) that is able to make decisions (similar to evolved agents in Section 5.3.5) and only sends commands to its partner when it is in FOV of the prey.

The Ms. Pac-Man environment is shown in Figure 6.1. In this environment, ghosts act as the predator agents and Ms. Pac-Man acts as the prey. There are a few differences in the Ms. Pac-Man environment compared to the Pursuit Domain environment. They are:

1. The Pursuit Domain is toroidal along all grid spaces of all the edges of the grid. This means when agents move off of a vertical or horizontal edge they can "wrap" from the current edge to the opposite edge. The Ms. Pac-Man environment has this characteristic only in 2 specific grid spaces on the vertical edges of the grid (see Figure 6.1).

Figure 6.1: Agents' FOV in Ms. Pac-Man Environment
Agent 0 "Blinky" (red ghost), Agent 1 "Pinky" (pink ghost)
Agent 2 "Clyde" (orange ghost), Agent 3 "Inky" (blue ghost)

2. The Pursuit Domain in this study has no collisions thus, agents can move once to any grid space in the Up, Down, Left or Right directions per time step. The only rule to this is predator agents are penalized if two agents try to occupy the same grid space in the same time step. If this happens the two agents are reset to their original starting positions on the grid. The Ms. Pac-Man environment has many collisions (walls) that block agents' movements. In this study, if agents attempt to move into a grid space containing a collision then they do not move on that time step. However, agents may occupy the same grid cell without any penalties.

3. In the Pursuit Domain, agents start in random positions in their own areas of the grid with the prey's area in the center. In the Ms. Pac-Man environment, predator agents (ghosts) start in the same position in their own area (lair) at the center of their grid and the prey (Ms. Pac-Man) starts in the same spot below the agents' area.

4. The Pursuit Domain in this study uses a grid size of 20 x 20 cells. The Ms. Pac-Man grid size is 28 (width) x 30 (height) cells.

5. The visible range for agents in the Pursuit Domain is 2, where the distance unit

of measure is 1 pixel (approximately 1x1 pixels = 1 cell on grid). The visible range for agents in the Ms. Pac-Man environment is 16 and is shown as grey circles around each ghost in Figure 6.1. The distance unit of measure is 1 pixel (distance between dots on grid is approximately 4 pixels).

In order to keep the Ms. Pac-Man environment similar to the Pursuit Domain, the Ms. Pac-Man and ghost agents are limited to Chase Mode, in which the ghosts continue to pursue Ms. Pac-Man (even if one ghost "catches" Ms. Pac-Man) within a defined period. Thus, in this version Ms. Pac-Man does not die when it occupies the same cell as a ghost and the entire 200 cycles are always completed in one episode.

The outcome of this experiment shows that the solution used in Chapter 5 in the Pursuit Domain environment is not as successful in the Ms. Pac-Man environment. It is found that collisions in the Ms. Pac-Man environment may contribute to the agents' shortcomings in learning. However, similar to the results in Section 5.3.5, a successful result in this chapter shows that agents are able to learn the meaning of commands and an emergent pattern in the agents' sub-trees reveals that in each protocol experiment there is at least one agent that learns to make decisions about when and what type of command to send.

6.1.1 Learning Strategy

Similar to previous experiments in this study, the learning strategy uses a fully co-operative implementation with a global fitness measure. The predator agents work together towards the common goal of following the prey (as closely as possible). The global fitness is a minimization function calculated over a limited time period. The total distance between all predator agents is used for the fitness measure. A base penalty value is added for each agent that is not within distance of the prey for a defined time period. The motivation for this fitness function is to compare how well the protocols perform in allowing the ghosts to first find, and then follow, the prey's random movement. Agents collaborate to minimize the global fitness value using a heterogeneous team based learning strategy such that each agent uses its own learning algorithm to evolve.

6.1.2 Communication Strategy and Communication Channel

The communication strategy in this study uses a learned language exactly as seen in Section 5.1.2. As a reminder, the commands are *C0*, *C1* and *C2*. The *C1* and

Table 6.1: Communication Protocols

Communication Protocols	Description
Send22C1	One send command only. Agents send C1 command.
Message Command(s)	C1
Message Data	$if agent <= FOV\ then$ send dir to prey $else$ send default dir
Send22C1orC2	Agents send either C1 or C2 command.
Message Command(s)	C1 or C2
Message Data	$if agent <= FOV\ then$ send dir to prey with C2 $else$ send default dir with C1
Send22PvA	Agents send either C1 or C2 command.
Message Command(s)	C1 or C2
Message Data	$if agent <= FOV\ then$ send dir to prey with C2 $else$ send dir to sending agent with C1

C2 commands, along with simple environment data, are sent individually from one agent to another through a message passing communication channel. An agent learns to associate a meaning to each command through evolving branches of its GP tree. Each agent has 1, 2, or 3 child branches (command trees) depending on the protocol. The first command, *C0*, is evaluated when the agent has no messages in its message buffer. The second command, *C1*, is evaluated when the agent has at least one *C1* message. The third command, *C2*, is evaluated when the agent has at least one *C2* message.

6.1.3 Communication Protocols

A communication protocol defines the method by which a sending agent sends a message to a receiving agent's message buffer. Three different protocols are examined, Send22C1, Send22C1orC2, and Send22PvA with descriptions listed in Table 6.1. Message passing for Send22 is the same as described previously (**Send22**: Two teams of two agents). Agent 0, "Blinky" (red ghost), and Agent 1, "Pinky" (pink ghost), form one team and Agent 2, "Clyde" (orange ghost), and Agent 3, "Inky" (blue ghost), form the other team. Each agent sends only to its partner.

Message types are C1, C1orC2, and PvA. Depending on the protocol, agents can send only one of the following types of command: 1) only C1 commands, and 2) either a C1 or C2 command (depending if an agent is within FOV of prey).

In the *Send22C1* protocol, only one send message is used to send a message with the C1 command. Similar to earlier experiments, if the agent is within FOV of the prey, the message data sent to the receiving agent contains the direction from the receiving agent to the prey, otherwise, it contains a default value of (4000, 4000).

In the *Send22C1orC2* protocol, one send message command is used to send either a C1 command or a C2 command. If the agent is within FOV of the prey, the message data sent to the receiving agent contains the direction from the receiving agent to the prey with the C2 command, otherwise, the message data contains a default value of (4000, 4000) with the C1 command.

The *Send22PvA* protocol functions the same way as the *Send22C1orC2* protocol except for the message data that is sent. If the agent is within FOV of the prey, the message data sent to the receiving agent contains the direction from the receiving agent to the prey with the C2 command, otherwise, the message data contains the direction from the receiving agent to the sending agent with the C1 command.

6.2 Experiment Details

Each experiment uses the same settings for the GP parameters as seen in Section 5.2 and are repeated in Table 6.2. The fitness function, GP language, testing and training methods are described below. In this experiment the prey moves in a random pattern (Up, Down, Left or Right) on the grid. Both agents and prey move only one step per time cycle. Each communication protocol is tested individually (see Table 6.1).

6.2.1 GP Language

This experiment uses the strongly typed language shown in Table 6.3.

Terminal Set

The terminal set is a scaled down version of the one used in Section 5.2, and is defined in Table 6.4. Movement commands (Up, Down, Left, Right) are sent directly to the agent as a result of the evaluation of its command tree. The language is typed such that only 1 movement is sent per evaluation. The direction vectors, North, South, West and East, contain the unit vector of each direction. The Goal terminal gives

Table 6.2: GP Parameters

GP Parameter	Value
Initial Tree Method	Koza's *Ramped half-and-half* [30], [22]
Min-Max Tree size (ramp)	7-12
Population size	1000
Generations	125
Selection	Tournament, size = 4
Crossover	90%
Mutation	10%
Runs per experiment	20

Table 6.3: Strongly Type Language

ROOT	$::= (SIM, SIM, SIM, SIM)$
SIM	$::= CommandTree(EXPR, EXPR, EXPR)$
EXPR	$::= Left \mid Right \mid Up \mid Down \mid$
	$::= MoveForward(EXPR) \mid$
	$::= MoveInDir(NIL, NIL, NIL, NIL) \mid$
	$::= IfGrtEql(NIL, NIL, EXPR, EXPR) \mid$
	$::= IfDist(NIL, NIL, EXPR, EXPR) \mid$
	$::= IfDot(NIL, NIL, EXPR, EXPR) \mid$
	$::= Send(EXPR) \mid$
NIL	$::= Goal \mid AgentDir \mid$
	$::= Add(NIL, NIL) \mid$
	$::= Sub(NIL, NIL) \mid$
	$::= Rotate90(NIL, NIL) \mid$
	$::= Reverse(NIL, NIL) \mid$
	$::= North \mid South \mid East \mid West \mid$
	$::= LRM$

the direction to the agent only if the agent is within field of view (FOV) of the prey otherwise, it gives a default direction of (4000, 4000). AgentDir holds the current direction of the agent. The Last Received Message (LRM) terminal node holds data for the last message removed from an agent's message buffer. Before evaluation of

Table 6.4: Terminal Set

Name	Description
Up,Down,Left,Right	move commands: $\uparrow, \downarrow, \leftarrow, \rightarrow$
North,South,West,East	$(0,1), (0,-1), (-1,0), (1,0)$
Goal	direction from prey to agent
	if agent is within FOV
AgentDir	the direction the agent is currently facing
LRM	Last Received Message
MaxDist	returns a large distance value (4000)

its tree, an agent checks its message buffer. If it contains messages, the first message is removed and its data is set to the LRM variable to be used for that evaluation cycle. If there are no messages in its buffer, the LRM node is set to the default vector (4000, 4000). The MaxDist node holds a constant value of 4000 representing a large distance value.

Function Set

The function set, seen in Table 6.5, is primarily the same as the one found in Section 5.2. Functions include mathematical operations on two dimensional vectors, logical operations and message sending commands. Message sending commands are listed as *SendC1, SendC1orC2,and SendPvA*. MoveForward is a movement command which takes an expression as input. It results in the agent moving one step in its current direction only if it can move in that direction. If the agent can't move in its current direction then the MoveForward evaluates its input expression. MoveInDir is also a movement command which takes in four vectors as input. It results in the agent moving one step in the direction of the first available cell as defined by the input vectors.

6.2.2 Training and Testing Methods

Similar to previous experiments, the training and testing of a GP individual consists of cycles and episodes. The training and testing methods in this experiment are for the most part the same as in Section 5.2.2. Training and testing begin with the ghosts and Ms. Pac-Man starting in their own areas on the grid. Figure 6.2 shows the ghosts and Ms. Pac-Man in their starting positions. In this figure, Agent 0 is the

Table 6.5: Function Set

Function	Description
Root	returns the evaluated value of the entire tree
SimAgent	returns the evaluated value of one agent
Add	vector addition
Sub	vector subtraction
Rotate90	rotates vector by 90 degrees
Reverse	multiplies vector by -1
IfGrtEql	compares the length of two vectors
IfDot	calculates the dot product of two vectors
IfDist	checks if agent is withing FOV of prey
MoveInDir	Moves in direction of the first available spot as defined by four input vectors
MoveForward	moves one step in agent's direction if it can else evaluates input expression
SendC1, SendC1orC2, SendPvA	(see Communication Protocols in Table 6.1)

red ghost"Blinky", Agent 1 is the pink ghost "Pinky", Agent 2 is the orange ghost "Clyde" and Agent 3 is the blue ghost "Inky". All ghosts start in the same area (lair) in the center of the grid while Ms. Pac-Man's starting position is just below the lair. In one cycle, all four agents evaluate their command tree once in sequence, one after the other. Each evaluation results in one movement of the agent, where one movement equates to one pixel on the grid. Instead of 30 cycles (as seen in other experiments), an episode is complete after 200 cycles (i.e. 1 episode = 200 cycles). In training, each GP individual is given 10 episodes and the positions of the agents/prey are reset to the original starting position after each episode is complete. The test run uses the GP individual with the best fitness in training. This GP individual is tested with 30 episodes and positions of the agents/prey are reset to the original starting position after each episode is complete.

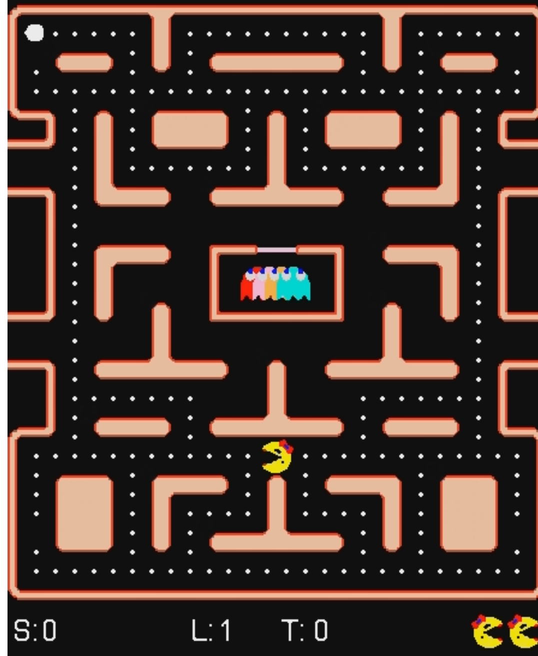


Figure 6.2: Agents' Starting Positions in Ms. Pac-Man
Agent 0 "Blinky" (red ghost), Agent 1 "Pinky" (pink ghost)
Agent 2 "Clyde" (orange ghost), Agent 3 "Inky" (blue ghost)

6.2.3 Fitness Function

The fitness function is the same as the one used in Section 5.2 with some small differences. The total fitness value is measured by accumulating sums from cycles to episodes. In each cycle of an episode, the sum of two fitness values is used as a fitness measure for that cycle. The total sum of each cycle fitness is used as the episode fitness. Finally, the total sum of each episode fitness score is used as the final fitness value for each individual GP.

The first fitness value calculated for each cycle is the sum of each of the agent's distance to the prey, where each pixel on the grid represents 1 unit of distance. Equation (6.1) shows the calculation of each agent's distance to the prey where A is the agent's position and P is the prey's position.

$$fAgentDist(A) = \sqrt{(A.x - P.x)^2 + (A.y - P.y)^2} \tag{6.1}$$

The second fitness value used in the cycle fitness measure is the total base penalty value for the cycle. This value represents a penalty value (25 points) for each predator agent. In any given cycle, each agent that is not within FOV of the prey is penalized by adding 25 points towards the cycle fitness sum. Thus, if all four agents are in FOV

of the prey, the total base penalty value for that cycle would be zero. However, if all agents are not in FOV of the prey then the total base penalty value for the cycle would be 100. Equation (6.2) shows how the total base penalty value is calculated in each cycle where A is the agent's position on the grid.

$$fBasePenalty(A) = \begin{cases} 0 & \text{if } fAgentDist(A) <= 16, \\ 25 & \text{otherwise.} \end{cases} \tag{6.2}$$

Equation (6.3) shows the total distance fitness calculation used in training. Here, $APos_i$ represents the location of $Agent_i$, where $i = 0...3$, m represents the number of cycles and q is the number of episodes. We set q to 10 in training and to 30 in testing in our experiments.

$$TotFitness = \sum_{k=1}^{q} \sum_{j=1}^{m} \sum_{i=0}^{3} \Big(fAgentDist(APos_i) + fBasePenalty(APos_i) \Big) \tag{6.3}$$

Similar to Equation (6.3), the test run fitness measures the average distance of all the episodes as seen in Equation (6.4).

$$AveDist = \frac{TotFitness}{q} \tag{6.4}$$

6.3 Results

In this section the performance results of the communication protocols Send22C1, Send22C1orC2, and Send22PvA are shown. Using random movement for Ms. Pac-Man for each protocol, the training results are first displayed in Figure 6.3 followed by the testing results in Table 6.6.

6.3.1 Statistical Analysis

In Figure 6.3 it is seen that for training, the average of best fitness for Send22C1 (dark orange line) outperforms the other best fitness averages (SendC1orC2 (yellow line) and Send22PvA(green line)) across all generations. Send22C1orC2 and Send22PvA best fitness values are fairly equal. Interestingly, in the mean fitness values, Send22C1 (dark blue line) and SendC1orC2 (grey line) are very similar throughout all the generations outperforming Send22PvA(light blue line).

The test results are shown in Table 6.6. It is seen that Send22C1orC2 has the

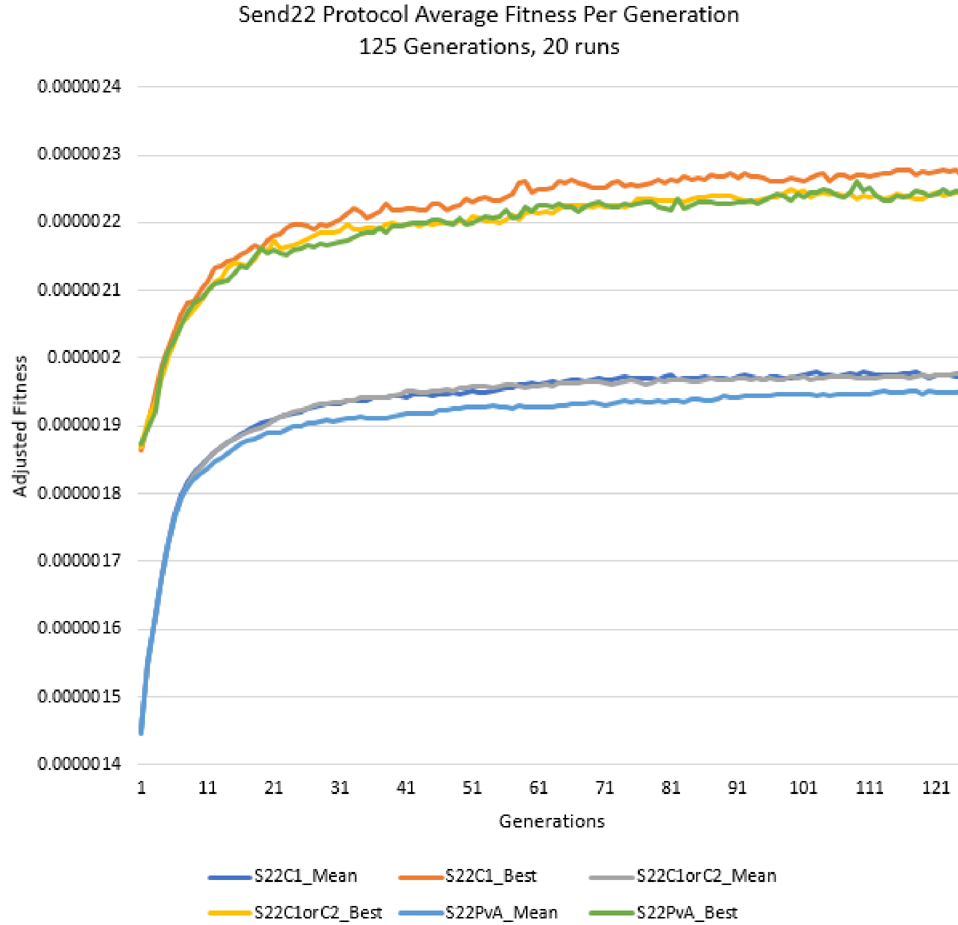


Figure 6.3: Send22 - Training for C1, C1orC2, PvA

Table 6.6: Send22 Fitness Summary (20 Test Runs)

Communication Type	Min Fitness	Average Fitness	Max Fitness
Send22C1	48507	53747	57149
Send22C1orC2	50808	53047	55865
Send22PvA	49037	53925	58480

Fitness is a minimization function. See Equation (6.4).

lowest average fitness value of 53047 with Send22C1 finishing second with a score of 53747 and Send22PvA finishing third with an average fitness value of 53925. Thus, Send22C1orC2 outperforms both Send22C1 and Send22PvA.

To verify the significance of the test results the One-Way ANOVA test (using Minitab [37]) with a 95% confidence interval is used. The ANOVA test uses a two-tailed T-test with 3 factors, where each factor represents one communication protocol including the final test fitness for the 20 test runs. The ANOVA test results in the $P - Value > \alpha$ (see Appendix A) indicating that there is not a significant difference

Table 6.7: Tukey Comparisons for Send22 Protocols

Communication Protocol	Group Letter	Average Fitness
Send22PvA	A	53925
Send22C1	A	53747
Send22C1orC2	A	53047

Protocols that do not share the same letter are significantly different

in the test results for Send22C1, Send22C1orC2, and Send22PvC shown in Table 6.6. The Tukey method [38] for multiple comparisons results are shown in Table 6.7. In this table all protocols share the same group letter indicating that their range of difference of mean does contain a zero. Protocols labelled with the same letter groups are considered to not be significantly different [39]. Thus, although the test results show that SendC1orC2 is the best performer with the lowest fitness average, it is found that there is not a significant difference among the results and the three protocols perform equally as well.

6.3.2 Qualitative Analysis

This section discusses the qualitative aspects such as GP tree structure, message sending patterns and playback of test runs in order to understand how the ghost agents in each of the protocols, Send22C1, Send22C1orC2, and Send22PvA learn to achieve their goal of finding and following the prey. The top performers in each of the protocols produce larger tree structures compared to the ones in previous chapters. This makes it more difficult to understand the structure of each of the C0, C1, and C2 command branches. Reflections on the data are made by examining the playback of the test runs, specific sections of command branches, and portions of the agents' message buffers.

The playback of the test runs show that, unlike the experiments in the Pursuit Domain, there is no stand out protocol that allows the ghost agents to follow Ms. Pac-Man. Each protocol allows agents to follow Ms. Pac-Man some of the time (in specific moments within their 200 cycle time allowance) but not all of the time. A typical scenario is that some ghosts find the prey and follow the prey for a specific time period and then lose the prey. A possible reason for this might be the fact that the Ms. Pac-Man environment has collisions. Thus, in order to follow Ms. Pac-Man, ghosts must learn to move around obstacles. Discussions in the following sections

focus on situations where one or more ghosts are within proximity of Ms. Pac-Man.

Similar to Section 5.3.5, this analysis reveals all protocols create sub-trees that allow sending agents to give meaning to the commands they send. An emergent sending behaviour found in each of the protocols reveals there is one ghost agent (of 4 ghosts) that only sends commands to its partner when it is in FOV of the prey.

Emergent Sending Patterns and Behaviours

Message sending patterns for top performers in each of the protocols are seen in Table 6.8. This table shows that each protocol has at least one sending pattern in which there is an agent that sends messages to its partner only if it is in FOV of the prey. In order to do this an agent's sub-tree uses the IfDist expression to check its proximity to the prey. If it is in FOV then the agent sends a message. Examples of this for each protocol are seen in Figure 6.4.

Figure 6.4 shows that for the Send22C1 protocol, Agent 2 (Clyde) sends a message only if it is in FOV of Ms. Pac-Man using the IfDist expression in its C0 branch. The sub-tree for Agent 0 (Blinky) in the Send22C1orC2 protocol shows that Blinky uses the IfDist expression to check if it is within view of Ms. Pac-Man. If it is in view 3 messages are sent to Agent 1 (Pinky). The Send22PvA protocol demonstrates the decision making for Agent 3 (Inky). Again, it is seen that Inky uses the IfDist expression to check its proximity to Ms. Pac-Man, if it is within range it sends a message to Agent 2 (Clyde).

The top performers of these protocols all have one agent that decides when to send a message. This is especially interesting for the Send22C1orC2 and Send22PvA protocols since they are programmed with the ability to send either a C1 command or a C2 command depending on the sending agent's proximity to the prey. The agents, sending only when in FOV of Ms. Pac-Man, ensure that the message will be sent with the C2 command along with useful directional information to Ms. Pac-Man. The same decision making behaviour for the Send22C1orC2 protocol was also seen earlier in Section 5.3.5.

Figures 6.5 to 6.7 show typical behaviours revealed in the playback runs for the top 3 runs in each protocol. Figure 6.5 shows that the Send22C1 ghost behaviour is such that the ghosts follow each other in a line in search of Ms. Pac-Man. Typical ghost behaviour for the Send22C1orC2 protocol is seen in Figure 6.6. Here, 2 of the 4 ghosts (Inky (blue) and Clyde(orange)) wait in the same spot for Ms. Pac-Man to enter their FOV, while 2 other ghosts (Blinky (red) and Pinky (pink)) search for Ms. Pac-Man. Finally, Figure 6.7 shows that the Send22PvA common behaviour is such

Table 6.8: Sending Patterns for Send22C1, Send22C1orC2, and Send22PvA

Send22C1 Run 14

Sender		*Receiver*	*Description*
A0			*(never sends)*
A1	→	*A0*	*(frequently sends)*
A2	→	*A3*	*(sends message only when in FOV of Ms. Pac-Man)*
A3	→		*(rarely sends)*

Send22C1orC2 Run 7

Sender		*Receiver*	*Description*
A0	→	*A1*	*(sends 3 messages only when in FOV of Ms. Pac-Man)*
A1			*(never sends)*
A2	→	*A3*	*(frequently sends)*
A3	→	*A2*	*(frequently sends)*

Send22PvA Run 16

Sender		*Receiver*	*Description*
A0	→	*A1*	*(frequently sends)*
A1	→	*A0*	*(frequently sends)*
A2			*(never sends)*
A3	→	*A2*	*(sends message only when in FOV of Ms. Pac-Man)*

that 2 ghosts (Blinky (red) and Pinky (pink)) wait in different spots, while 2 other ghosts (Inky (blue) and Clyde(orange)) search for Ms. Pac-Man.

The reason for this typical behaviour for each of the protocols may be the fact that although Ms. Pac-Man chooses random movements in both training and testing, the starting position of Ms. Pac-Man is always the same at the beginning of each episode. This is because in the Ms. Pac-Man game, Ms. Pac-Man always starts at the same position. Future work should include training runs that have Ms. Pac-Man starting at different positions at the beginning of each episode (similar to the Pursuit Domain experiments).

Evidence of Communication

The previous section showed that the Send22C1, Send22C1orC2, and Send22PvA protocols can produce sub-trees for agents which allow them to make decisions on when (and what command) to send. This section analyzes the playback for one of

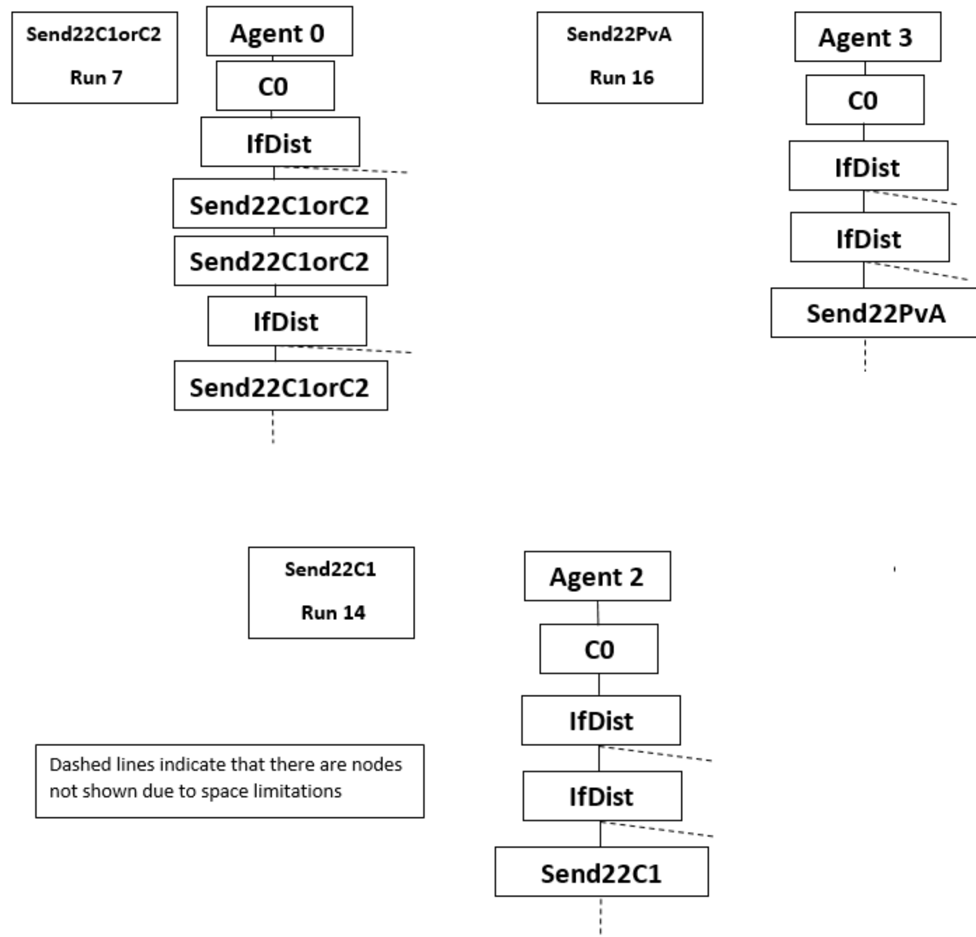


Figure 6.4: Decision Sub-trees for Send22C1, Send22C1orC2, and Send22PvA

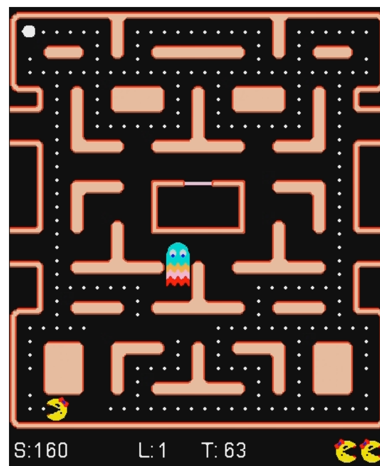


Figure 6.5: Send22C1 Typical Behaviour
Ghosts follow each other in a line in search of Ms. Pac-Man.

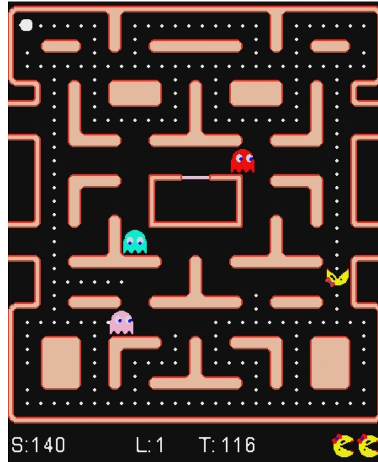


Figure 6.6: Send22C1orC2 Typical Behaviour
Inky (blue) and Clyde (orange) wait in the same spot for Ms. Pac-Man, while Blinky (red) and Pinky (pink) search for Ms. Pac-Man.

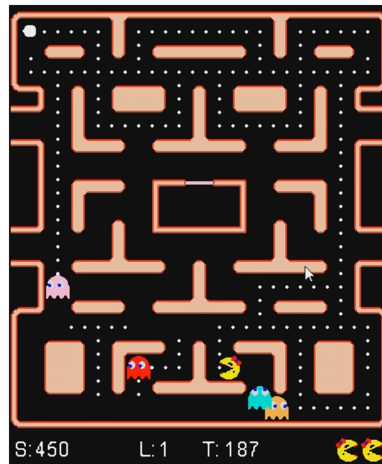


Figure 6.7: Send22PvA Typical Behaviour
Blinky (red) and Pinky (pink) wait in different spots for Ms. Pac-Man, while Inky (blue) and Clyde (orange) search for Ms. Pac-Man.

the top test runs (Run 7) for the Send22C1orC2 protocol. Looking at portions of the GP structure, images from the test run at specific cycles, and corresponding sections of the message buffers it is seen that in certain cycles, agents communicate in order to help each other follow the prey.

As shown in Table 6.9, Agent 0 (Blinky) in test Run 7 for the Send22C1orC2 protocol sends 3 messages (with the C2 command) to Agent 1 (Pinky) when it is in FOV of the prey. Pinky does not send messages to Blinky at any time. Thus, Blinky always evaluates its C0 command branch and Pinky evaluates its C2 branch when it receives the C2 commands.

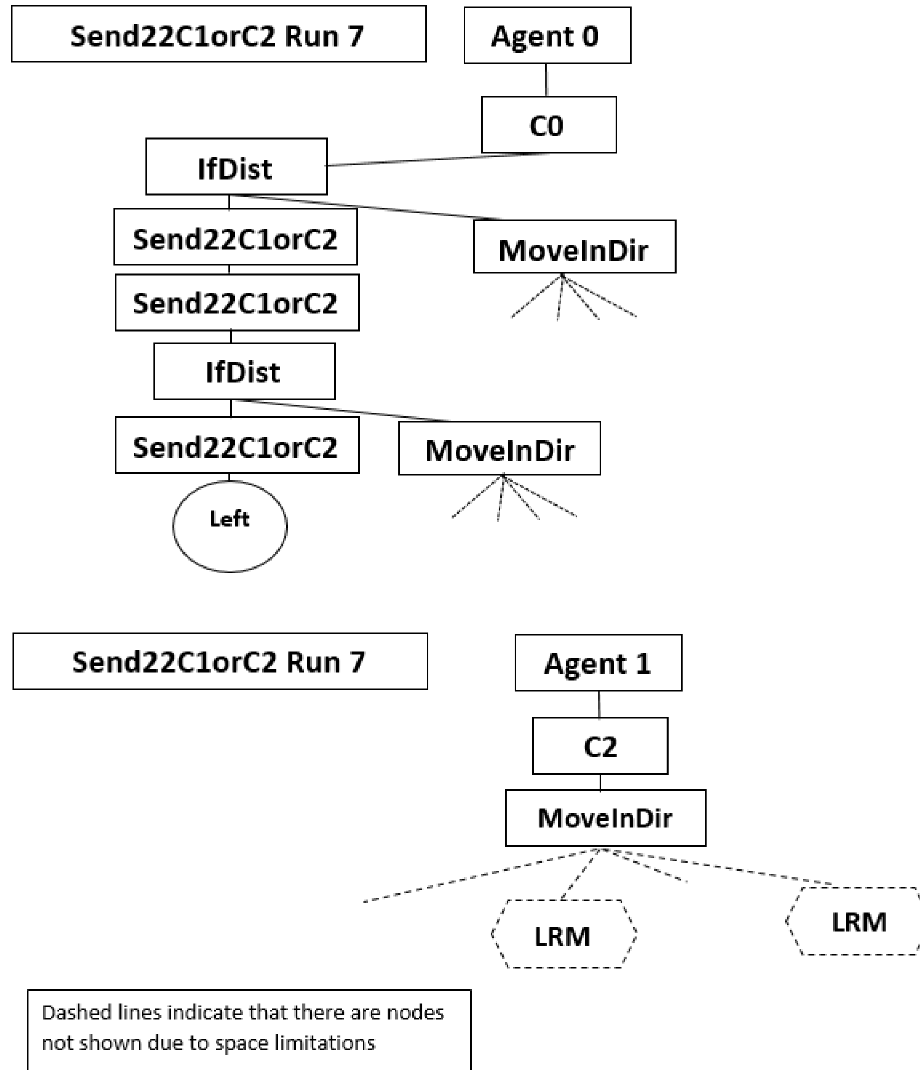


Figure 6.8: Send22C1orC2 Agent 0 & 1 C0 & C2 Branches
Ghost Agent 0 (Blinky) sends 3 messages
to Agent 1 (Pinky) only when in view of Ms. Pac-Man

Figure 6.8 shows the C0 branch for Blinky and the C2 branch for Pinky. The C0 branch for Blinky is used to send 3 messages with LRM data containing relational directional information to Ms. Pac-Man. The C2 branch for Pinky shows that the LRM data is used in two input vectors for the MoveInDir expression. The dotted lines in the diagram indicate that there are more nodes in the tree that could not be included due to space limitations. Although, it is not clear how the MoveInDir uses the LRM node (because the trees are large and difficult to interpret), it is clear that the LRM node is used in movement expressions for the second and fourth input vectors for the MoveInDir expression. Although this is not as straight forward as the

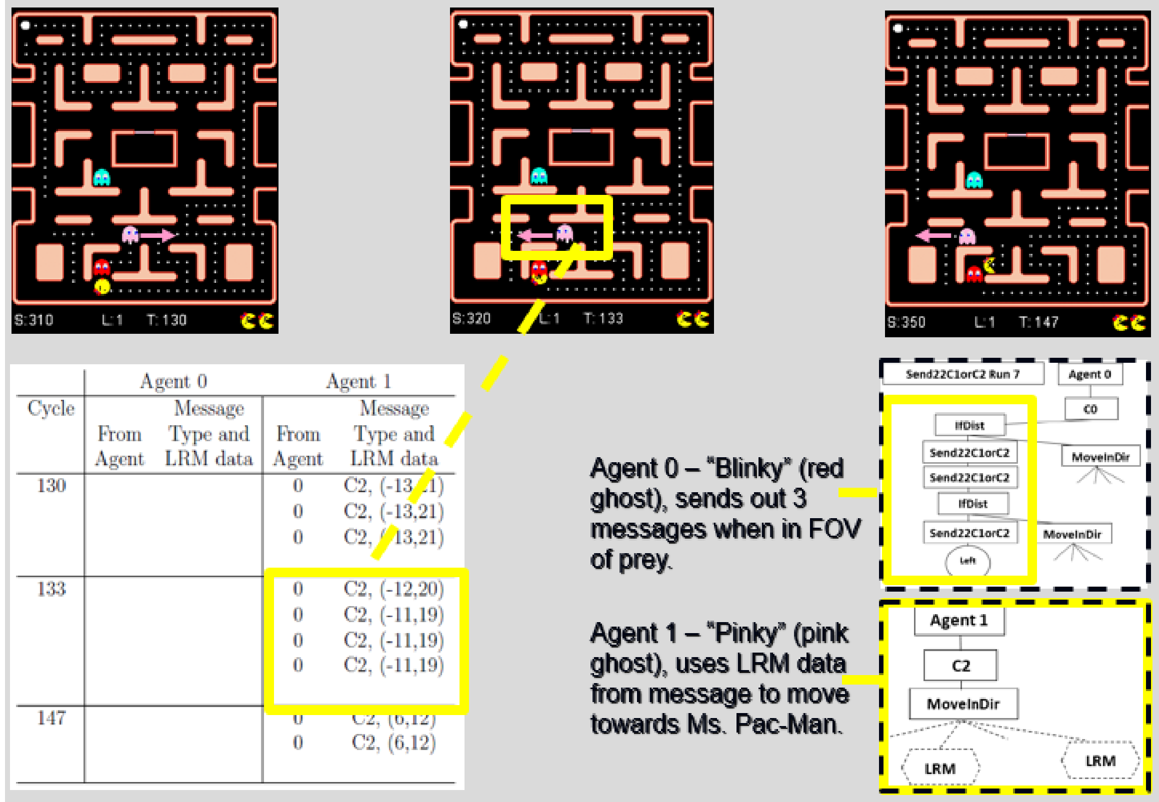


Figure 6.9: Send22C1orC2 Run 7, **Cycles 130 - 147**

MoveInDir(LRM) expression that we have seen in the previous chapter, it does show that this sub-tree is indirectly associating the use of the LRM data to the C2 branch (as was communicated by the sending ghost, Blinky).

A specific example of this communication is seen in images of the playback test run for Run 7. Figure 6.9 shows Cycles 130 to 147. The corresponding cycles in the message buffers for the agents are seen in Table 6.9. Here it is seen that in Cycle 130 Blinky (red ghost) is in view of Ms. Pac-Man and Pinky (pink ghost) has 3 C2 messages in its message buffer. The playback images show that Pinky moves towards the right, bringing it farther (in the horizontal direction) from Ms. Pac-Man. But in Cycle 133 and Cycle 147 it is seen that Pinky begins to move to the left to get closer (in the horizontal direction) to Ms. Pac-Man.

The message buffer contents from Cycles 130 to 147 show that Pinky has C2 messages with LRM containing directional data to Ms. Pac-Man. It is not quite clear if Pinky is using the LRM data to move left but it is clear that C2 is causing the change in direction from right to left. By Cycle 147, Ms. Pac-Man moves farther to the right and in later cycles moves out of view from Blinky. Thus, although the ghosts are not able to follow Ms. Pac-Man for a long period, this shows that there

	Agent 0		Agent 1	
Cycle	From Agent	Message Type and LRM data	From Agent	Message Type and LRM data
130			0	C2, (-13,21)
			0	C2, (-13,21)
			0	C2, (-13,21)
133			0	C2, (-12,20)
			0	C2, (-11,19)
			0	C2, (-11,19)
			0	C2, (-11,19)
147			0	C2, (6,12)
			0	C2, (6,12)

Table 6.9: Send22C1orC2 Test Run 7 Message Buffer
Agent 0 and 1 Message Buffer Contents. (Cycles 130, 133 & 147)

are specific times in the 200 cycles that Blinky and Pinky are communicating to help each other track the prey.

One issue that makes the Ms. Pac-Man environment more difficult than the Pursuit Domain environment is the fact that the ghost agents have to deal with collisions. In the Ms. Pac-Man environment even if the ghosts have access to the prey's information, they have the added problem of needing to move around obstacles in order to get to the prey. If there is a collision between a Ghost and Ms. Pac-Man then the Ghost may have to first move away from Ms. Pac-Man in order to move around the collision. The current GP Language and fitness function do not account for this and will need to be enhanced in future work.

6.3.3 Summary of Results

The results show that the GP language and protocols are promising in the Pursuit Domain environment but, could use further refinement in the Ms. Pac-Man game environment. This analysis reveals that all protocols performed equally as well. Each protocol creates sub-trees that allow sending agents to decide when to send a command. An emergent sending behaviour reveals there is one ghost agent (of 4 ghosts) that only sends commands to its partner when it is in FOV of the prey. The ability to make decisions on when to send a message is also found as an emergent behaviour

in the results in Section 5.3.5. Also, waiting to send messages until the prey is in view is found in the results in Section 4.3.2, in which evolved agents reveal a guard and reinforcement behaviour commonly found in stealth games such as MGS [36].

The GP Language does not allow the predator agents to follow the prey as well as they do in the Pursuit Domain. However, it is seen that communication (in small segments of the 200 cycle time period) influence an agent's direction bringing it closer to the prey. In order to improve the results, future work should focus on enhancing the GP Language and fitness functions so that they better account for collisions. For example, the current fitness function is not able to distinguish the difference in distances in which two predator agents are equal distant from the prey but in one case the predator agent has a wall between itself and the prey. One way to make the distances between these two agents distinguishable is to use the Manhattan distance (instead of the Euclidean distance) as seen in [40]. Also, in this experiment, Ms. Pac-Man always starts at the same position (as per game rules). Future work should include training runs in which Ms. Pac-Man starts at random starting positions (in the top, middle and bottom portions of the grid).

Chapter 7

Conclusion

7.1 Summary of Main Results

Using a variety of different communication protocols, this thesis continues the research of using GP in a multi-agent environment. The goals of this research are to investigate how genetic programs can influence predator agents' in associating meaning to commands in a predator-prey scenario and to investigate how well agents communicate in order to learn the behaviour of tracking prey. Fitness is measured in how closely agents can follow the prey.

This work is divided into three distinct parts. Chapter 4 looks at many different communication protocols to see which types are better able to allow predators agents to track the prey using communication in the Pursuit Domain environment [19]. Chapter 5 enhances the GP language used in Chapter 4 in search of a better solution to experiments in which the prey moves randomly. Finally, Chapter 6 uses the GP language from Chapter 5 and tests how well it performs in a game environment, the Ms. Pac-Man simulator [20].

The results show that most tests evolve competent agents that can associate meaning to commands and use message data in order to find and follow the prey. Many of the communication protocols do not produce significant differences in fitness scores or perceived behaviours. Generalized, powerful agents are not commonly seen. However, some experiments do regularly evolve interesting behaviours that show high-levels of coordination among agents such as the emergence of a synchronized message sending pattern seen in Chapter 4. Highly specialized evolved agents are shown to be the most effective and are revealed in the guard and reinforcement behaviours of Chapter 4, and the "sender" and "receiver" behaviours of Chapters 5 and 6. In addition, it is shown that training is important and choosing the movement type for the prey

significantly influences the performance of evolved agents in test runs.

A synchronized alternating message sending pattern emerges among predator agents in Chapter 4. Experiments involving two types of prey movement, *Prey Linear Movement* and *Prey Random Movement*, show that agents (regardless of the communication protocol) learn to send messages on alternating cycles. This pattern allows agents to evaluate each command branch (*C0* or *C1*) every other frame, resulting in some agents using a staircase pattern of movement in order to find the prey. This emergent behaviour is found to be more effective (allowing agents to find and follow the prey) in the *Prey Linear Movement* experiment than in the *Prey Random Movement* experiment. It is found that an enhanced GP language is needed to solve the *Prey Random Movement* problem. An additional interesting outcome reveals that evolved predator agents' behaviour in the best result resembles the scripted behaviour of guard and reinforcements that can be found in popular stealth video games (e.g. *Metal Gear Solid (MGS)* [36]).

An enhanced GP language is used to improve agent performance in Chapter 5. This chapter is divided into two parts. The first part focuses on the top performing protocols, SendAll and Send22, of Chapter 4. The GP language, fitness measure and message types are augmented. The goal of this experiment is to see if predator agents can learn the meaning of commands in order to track the prey as it moves randomly across the grid environment. The results in the first part of this experiment show an emergent behaviour such that one agent is designated as the "sending agent" and all other agents are designated as "receiving" agents. It is found that in most cases receiving agents are able to associate a meaning to commands received from (and determined by) the sending agent. This allows most agents to achieve their goal of tracking the random moving prey quite successfully.

The second part of Chapter 5 attempts to answer questions such as: How does training predator agents with a random moving prey affect test runs with a linear moving prey or vice versa? Does the movement of the prey in training influence the test results in which the prey moves in a different movement pattern? These questions are answered as the results show that the types of training chosen for a problem are important. For example, there are significant differences and improvement of performance in test runs when the test runs only include the type of prey movement that is used in training. For example, when using a random moving prey in training runs, test runs which include only a random moving prey outperform test runs which include only a linear moving prey or both (switching from linear to random moving prey). This is true for training using linear prey movement as well. However, the

results show that all combinations of having the prey switch from linear movement to random movement in training and testing performed equally as well.

The Ms. Pac-Man game environment [20] in Chapter 6 is used to further test the GP language defined in Chapter 5. The goal of this chapter is to see if predator agents, using a similar GP language as in earlier experiments, can learn to find and follow the prey as it moves randomly in this new environment. A new protocol type, *Send22PvA* along with the top protocols, *Send22C1* and *Send22C1orC2*, from Chapter 5 are used in this solution. Outcomes of this study reveal that possibly due to collisions in the environment, agent evolution does not allow the predator agents to learn to follow the prey quite as well as they do in the Pursuit Domain simulator [19]. Similar to Chapter 5, a successful result in this study shows an emergent message sending pattern such that "sending" agents are able to make decisions about when and what type of command to send.

7.2 Future Work

Future work should include more tests using variants of the Pursuit Domain environment [19]. For example, Iba [6] found that communication is not required (and is a burden due to the overhead cost of communication) when agents are close together and communication is most effective when agents are farther apart. Thus, future experiments should include comparing the affects of using the communication protocols in different grid sizes (40x40, 50x50 or 100x100 etc...) while increasing the number of cycle and episodes in training/testing.

In Chapter 4, the evolved guard and reinforcement behaviour could be beneficial to future work specifically related to game production. For example, an *agent behaviour* design tool could train predators to respond to scenarios using different states. That is, an agent could learn to associate one command to a specific state of behaviour such as tracking or retreating. Similar work was done by Kadlec [16] using the Unreal Tournament 2004 (UT) [17]. Adding an interactive component to the tool, designers could interactively train game agents by selecting preferred behaviour during evolution and test trained behaviours in actual game scenarios.

Currently, there is a strong interest in research to use GP and other learning algorithms to aid in the development of commercial games [3]. For example, most game AI existing in commercial games is based on written scripts created at significant expense by game programmers [41]. An interesting alternative to scripting is the use of computational intelligence methods in which an agent's behaviour can be evolved

through a learning algorithm instead of being scripted by a programmer.

This has interesting advantages as stated by Lucas and Kendall [41]: evolving the scripts can be financially attractive as it saves expensive programming time, evolving behaviours may create novel approaches (which may have been overlooked by programmers) that game players may find interesting, and evolved agents are very good at finding and exploiting loopholes in the game. Loopholes allow a player to find a fast way to win a particular level of the game or even the game itself. The ability to detect loopholes is notable because removing them in a game is a very important part of the game development life cycle for which developers spend many hours. More challenges and opportunities for future research and the use of learning algorithms in games are found in the Dagstuhl Report by Lucas et al. [3].

Tests conducted in Chapter 6 are preliminary, and there is much room for future work using communicating predator agents (ghosts) in the Ms. Pac-Man simulator [20]. For example, although the game begins with Ms. Pac-Man starting at the same position, it would be interesting to see if more robust GPs are created when Ms. Pac-Man is started at random positions in the envrinoment during training episodes.

The primary goal for this chapter is to see how well the GP language from Chapter 5 performs in a completely new environment. As a result, only minor changes are made to the GP language. Future work should enhance the language so that it is better able to handle collisions. For example, the Euclidean distance in the fitness function is not able to distinguish the difference in distances in which two predator agents are equal distant from the prey but in one case the predator agent has a collision between itself and the prey. One way to account for this type of scenario is to use the Manhattan distance as seen in [40]. In addition, it would be interesting to add more aspects from the Ms. Pac-Man game to the fitness measure to see how agent evolution (using communication protocols) reacts to different game elements such as different game modes (chase and retreat), or including ghosts which track Ms. Pac-Man and at the same time try to minimize her score (i.e. reducing the number of pills she consumes).

In conclusion, this research adds to previous work showing that GP-evolved emergent behaviour can be used to help agents learn to associate meaning to commands and to interpret message data. With this approach, predator agents are evolved to learn to find and track prey.

Bibliography

[1] L. Pannait and S. Luke, "Cooperative Multi-Agent Learning: The State of the Art," *Autonomous Agents and Multi-Agent Systems*, vol. 11, no. 3, pp. 387–434, 2005.

[2] J. Reverte, F. Gallego, R. Satorre, and F. Llorens, *Cooperation Strategies for Pursuit Games: From a Greedy to an Evolutive Approach.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 806–815.

[3] S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius, "Artificial and computational intelligence in games (dagstuhl seminar 12191)," *Dagstuhl Reports*, vol. 2, no. 5, pp. 43–70, 2012.

[4] T. Haynes, S. Sen, D. Schoenefeld, and R. Wainwright, "Evolving multiagent coordination strategies with genetic programming," Proc. Artificial Intelligence, Tech. Rep., 1995.

[5] J. Denzinger and M. Fuchs, "Experiments in learning prototypical situations for variants of the pursuit game." in *Proceedings on the International Conference of Multi-Agent Systems(ICMAS-1996)*, 1996, pp. 48–55.

[6] H. Iba, "Evolutionary learning of communicating agents," *Journal of Information Sciences*, vol. 108, no. 1-4, pp. 181–205, 1998.

[7] J. Kam-Chuen and C. Giles, "Talking Helps: Evolving communicating agents for the predator-prey pursuit problem," *Artificial Life*, vol. 6, no. 3, pp. 237–254, 2000.

[8] H. Yanco and L. Stein, *An adaptive communication protocol for cooperating mobile robots*, H. R. Meyer, JA and S. Wilson, Eds. Cambridge MA: The MIT Press, 1993.

[9] J. Kuo, H. Yu, K. F.-R. Liu, and F. Lee, "Multiagent cooperative learning strategies for pursuit-evasion games," *Mathematical Problems in Engineering*, vol. 2015, pp. 1–13, 2015.

[10] I. Tanev, M. Brzozowski, and K. Shimohara, "Evolution, generality and robustness of emerged surrounding behaviour in continuous predators-prey pursuit problem," *Genetic Programming and Evolvable Machines*, vol. 6, no. 3, pp. 301–318, 2005.

[11] B. Zhang and D. Cho, "Evolving complex group behaviors using genetic programming with fitness switching," *Artificial Life and Robotics*, vol. 4, no. 2, pp. 103–108, 2000.

[12] S. Luke, C. Hohm, J. Farris, G. Jackson, and J. Hendler, "Co-evolving soccer softbot team coordination with genetic programming," in *Proceedings of the First International Workshop on RoboCup, IJCAI-97*, Nagoya, Japan, 1997.

[13] J.Y.Kuo, F. Huang, S. Ma, and Y. Fangjiang, "Applying hybrid learning approach to robocup's strategy," *The Journal of Systems and Software*, vol. 86, no. 7, pp. 1993–1944, 2013.

[14] A. Cardona, J. Togelius, and M. Nelson, "Competitive coevolution in Ms. Pac-Man," in *2013 IEEE Congress on Evolutionary Computation (CEC)*, Cancun, Mexico, 2013, pp. 1403–1410.

[15] A. Alhejali and S. Lucas, "Using a training camp with genetic programming to evolve Ms. Pac-Man agents," in *2011 IEEE Conference on Computational Intelligence in Games (CIG)*, Seoul, Korea, 2011, pp. 118–125.

[16] R. Kadlec, "The Paradox of Overfitting," Master's thesis, Charels University, Prague, Czech, 2008.

[17] (2004) Unreal Tournament. [Online]. Available: https://www.epicgames.com/unrealtournament/ (Last accessed: 29-Nov-2016).

[18] G. Grossi and B. Ross, "Evolved communication strategies and emerged behaviour of multi-agents in pursuit domain," in *2017 IEEE Conference on Computational Intelligence in Games (CIG)*, New York City, USA, 2017.

[19] J. R. Kok and N. Vlassis, "The pursuit domain package," Informatics Institute, University of Amsterdam, The Netherlands, Tech. Rep. IAS-UVA-03-03, Aug. 2003.

[20] (2011) Ms. Pac-Man Competition. [Online]. Available: http://dces.essex.ac.uk/staff/sml/pacman/PacManContest.html (Last accessed: 2-Aug-2016).

[21] R. Eberhart and S. Y., *Computational Intelligence: Concepts to Implementations*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[22] R. Poli, W. Langdon, and N. McPhee, *A Field Guide to Genetic Programming*. United Kingdom: Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008, [Online]. Available: http://www.gp-field-guide.org.uk (Last accessed: 9-June-2017).

[23] D. J. Montana, "Strongly typed genetic programming," *Evol. Comput.*, vol. 3, no. 2, pp. 199–230, Jun. 1995.

[24] Y. Demazeau, F. Zambonelli, J. Rodríguez, and J. Pérez, *Advances in Practical Applications of Heterogeneous Multi-Agent Systems - The PAAMS Collection: 12th International Conference, PAAMS 2014, Salamanca, Spain, June 4-6, 2014. Proceedings*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014.

[25] R. Singh, A. Singh, and S. Mukherjee, "A critical investigation of agent interaction protocols in multiagent systems," *International Journal of Advancements in Technology*, vol. 5, no. 2, pp. 72–81, 2014.

[26] S. Barrett, P. Stone, and S. Kraus, "Empirical evaluation of ad hoc teamwork in the pursuit domain," in *Proc. of 11th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, May 2011.

[27] H. Muñoz-Avila, C. Bauckhage, M. Bida, C. B. Congdon, and G. Kendall, "Learning and Game AI," in *Artificial and Computational Intelligence in Games*, ser. Dagstuhl Follow-Ups, S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius, Eds. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, vol. 6, pp. 33–43.

[28] G. Tesauro, "Temporal difference learning and TD-Gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[29] S. Lee-Urban, M. Vasta, and H. Muñoz-Avila, "Retaliate: Learning winning policies in first-person shooter games." in *Seventeenth Innovative Applications of Artificial Intelligence Conference (IAAI-07)*, 2007.

[30] (2010) ECJ 22 A Java-based Evolutionary Computation Research System. [Online]. Available: http://cs.gmu.edu/ eclab/projects/ecj/ (Last accessed: 16-Sept-2015).

[31] (2015) MARS Release. [Online]. Available: https://eclipse.org/mars/ (Last accessed: 16-Sept-2015).

[32] J. Schrum and R. Miikkulainen, "Evolving multimodal behavior with modular neural networks in Ms. Pac-Man," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2014)*, Vancouver, BC, Canada, July 2014, pp. 325–332.

[33] A. Alhejali and S. Lucas, "Evolving diverse Ms. Pac-Man playing agents using genetic programming," in *2010 UK Workshop on Computational Intelligence (UKCI)*, Sept 2010, pp. 1–6.

[34] ——, "Using genetic programming to evolve heuristics for a monte carlo tree search Ms. Pac-Man agents," in *2013 IEEE Conference on Computational Intelligence in Games (CIG)*, Niagara Falls, ON, 2013, pp. 1–8.

[35] T. Haynes, K. Lau, and S. Sen, "Learning cases to compliment rules for conflict resolution in multiagent systems." in *Working Notes for the AAAI Symposium on Adaptation, Co-evolution and Learning in Multiagent Systems*, S.Sen, Ed., Standford University, CA, 1996, pp. 51–56.

[36] (2004) Metal Gear Solid The Twin Snakes. [Online]. Available: http://www.konami.jp/gs/game/mgs_tts/ (Last accessed: 18-Jan-2017).

[37] (2017) Minitab 17 Support. [Online]. Available: http://support.minitab.com/en-us/minitab/17/ (Last accessed: 25-May-2017).

[38] (2017) What is Tukey's Test and Honest Significant Difference? [Online]. Available: http://www.statisticshowto.com/tukey-test-honest-significant-difference/ (Last accessed: 25-May-2017).

[39] (2017) What is Tukey's method for multiple comparisons? [Online]. Available: http://support.minitab.com/en-us/minitab/17/topic-library/modeling-statistics/anova/multiple-comparisons/what-is-tukey-s-method/ (Last accessed: 25-May-2017).

[40] I. Andersone and A. Nikitenko, *Reliable Multi-robot Map Merging of Inaccurate Maps.* Cham: Springer International Publishing, 2014, pp. 13–24.

[41] S. Lucas and G. Kendall, "Evolutionary computation and games," *Computational Intelligence Magazine,IEEE*, vol. 1, no. 1, pp. 10–18, 2006.

Appendix A

Additional Experimental Analysis

A.1 ANOVA Hypothesis

Table A.1 shows the ANOVA hypothesis used for all ANOVA tests in this research.

Table A.1: ANOVA Hypothesis

Hypothesis	Description
Null hypothesis	All means are equal
Alternative hypothesis	At least one mean is different
Significance level	$\alpha = 0.05$

A.2 Evolved Communication Protocols: ANOVA Results

The following shows the ANOVA results for the *Communication Protocols* experiment described in Chapter 4.

Table A.2: Prey Linear Movement ANOVA results

Analysis of Variance					
Source	DF	Adj SS	Adj MS	F-Value	P-Value
Factor	6	20618	3436	2.38	0.032
Error	133	192147	1445		
Total	139	212765			

$P - Value < \alpha$ *indicating that there is a significant difference in the test results*

Table A.3: Prey Random Movement ANOVA results

Analysis of Variance					
Source	DF	Adj SS	Adj MS	F-Value	P-Value
Factor	6	4811	801.8	0.90	0.494
Error	133	117961	886.9		
Total	139	122772			

$P - Value > \alpha$ indicating that there is not a significant difference in the test results

A.3 Learning the Meaning of Commands Part A: ANOVA Results

The following shows the ANOVA results for the first part of the *Learning the Meaning of Commands* experiment described in Chapter 5.

Table A.4: Send22 Protocol Types ANOVA results

Analysis of Variance					
Source	DF	Adj SS	Adj MS	F-Value	P-Value
Factor	2	60951	30475	3.27	0.045
Error	57	531399	9323		
Total	59	592350			

$P - Value < \alpha$ indicating that there is a significant difference in the test results

Table A.5: SendAll Protocol Types ANOVA results

Analysis of Variance					
Source	DF	Adj SS	Adj MS	F-Value	P-Value
Factor	2	132365	66183	8.90	0.000
Error	57	424055	7440		
Total	59	556421			

$P - Value < \alpha$ indicating that there is a significant difference in the test results

A.4 Learning the Meaning of Commands Part B: ANOVA results

The following shows the ANOVA results for the second part of the *Learning the Meaning of Commands* experiment described in Chapter 5.

Table A.6: Training: Prey Linear Movement Types ANOVA results

Analysis of Variance					
Source	DF	Adj SS	Adj MS	F-Value	P-Value
Factor	2	946936	189387	24.25	0.000
Error	114	890444	7811		
Total	119	1837380			

$P - Value < \alpha$ *indicating that there is a significant difference in the test results*

Table A.7: Training: Prey Random Movement Types ANOVA results

Analysis of Variance					
Source	DF	Adj SS	Adj MS	F-Value	P-Value
Factor	5	1814579	362916	30.89	0.000
Error	114	1339138	11747		
Total	119	3153717			

$P - Value < \alpha$ *indicating that there is a significant difference in the test results*

Table A.8: Training: Prey Linear/Random Movement Types ANOVA results

Analysis of Variance					
Source	DF	Adj SS	Adj MS	F-Value	P-Value
Factor	5	17202	3440	0.26	0.932
Error	114	1485765	13033		
Total	119	1502967			

$P - Value > \alpha$ *indicating that there is not a significant difference in the test results*

Table A.9: Testing: Prey Linear Movement Types ANOVA results

Analysis of Variance					
Source	DF	Adj SS	Adj MS	F-Value	P-Value
Factor	5	2096509	419302	30.47	0.000
Error	114	1568859	13762		
Total	119	3665368			

$P - Value < \alpha$ *indicating that there is a significant difference in the test results*

Table A.10: Testing: Prey Random Movement Types ANOVA results

Analysis of Variance					
Source	DF	Adj SS	Adj MS	F-Value	P-Value
Factor	5	782784	156557	19.25	0.000
Error	114	927339	8135		
Total	119	1710123			

$P - Value < \alpha$ *indicating that there is a significant difference in the test results*

Table A.11: Testing: Prey Linear/Random Movement Types ANOVA results

Analysis of Variance					
Source	DF	Adj SS	Adj MS	F-Value	P-Value
Factor	5	48118	9642	0.9	0.484
Error	114	1219150	10694		
Total	119	1267268			

$P - Value > \alpha$ *indicating that there is not a significant difference in the test results*

A.5 Agent Evolution in Ms. Pac-Man Environment: ANOVA Results

The following shows the ANOVA results for the *Agent Evolution in Ms. Pac-Man Environment* experiment described in Chapter 6.

Table A.12: PacMan Testing: Send22 ANOVA results

Analysis of Variance					
Source	DF	Adj SS	Adj MS	F-Value	P-Value
Factor	2	8627340	4313670	1.27	0.290
Error	57	194337300	3409426		
Total	59	202964640			

$P - Value > \alpha$ *indicating that there is not a significant difference in the test results*