NTNU

Norwegian University of
Science and Technology

Creatively Evolving Cooperative Behaviour with the 'NeuroEvolution of Augmenting Topologies' Algorithm

Einar Hov

Andreas Høgetveit Weisethaunet

Master of Science in Informatics
Submission date: June 2018
Supervisor: Björn Gambäck, IDI
Co-supervisor: Marinos Koutsomichalis, IDI

Norwegian University of Science and Technology
Department of Computer Science

Einar Hov & Andreas Høgetveit Weisethaunet

Creatively Evolving Cooperative Behaviour with the 'NeuroEvolution of Augmenting Topologies' Algorithm

Master's Thesis in Informatics, Spring 2018

Data and Artificial Intelligence Group
Department of Computer Science
Faculty of Information Technology and Electrical Engineering
Norwegian University of Science and Technology

Abstract

This thesis combines creativity and cooperative behaviour, and aims to investigate if a machine learning algorithm called 'NeuroEvolution of Augmenting Topologies' (NEAT) can evolve cooperative behaviour in a creative process. Such a process may be used to generate cooperative behaviours in creative systems—such as video games or other kinds of simulation software.

This research was mainly a design, implementation and experiment research. Inspired by previous work performed in the research field of computational creativity, a definition and evaluation criteria for creativity and cooperative behaviours were formulated. A system was designed and implemented to simulate and evolve the behaviour of multiple interacting agents. Experiments were conducted using this system. The cooperation of the generated artefacts and the creativity of the system were evaluated.

Experiments were run on four different environments. In two of the environments no cooperation was found. In one environment cooperation was found, but the results were inconclusive whether the behaviour emerged through a creative process. In the last environment the algorithm evolved behaviour that satisfied our definition of cooperation and which was evolved through a process that satisfied our criteria for creativity.

Sammendrag

Denne oppgaven kombinerer kreativitet og samarbeidsadferd, og tar sikte på å undersøke om en maskinlæringsalgoritme kalt NeuroEvolution of Augmenting Topologies (NEAT) kan utvikle samarbeidsadferd i en kreativ prosess. En slik prosess kan brukes til å generere samarbeidsadferd i kreative systemer—slik som videospill eller andre simuleringsprogramvarer.

Denne forskningen var hovedsakelig basert på design og implementasjon av et system, og eksperimentering med dette systemet. Inspirert av tidligere arbeid utført i forskningsfeltet computational creativity, ble en definisjon og evalueringskriterier for kreativitet og samarbeidsadferd formulert. Et system ble utformet og implementert for å simulere og utvikle adferdene til flere samarbeidende agenter. Samarbeidet mellom de genererte adferdene og systemets kreativitet ble evaluert.

Eksperimenter ble kjørt på fire forskjellige miljøer. I to av miljøene ble det ikke funnet noe samarbeid. I ett miljø ble det funnet samarbeid, men ut fra resultatene kunne det ikke konkluderes om oppførselen hadde oppstått gjennom en kreativ prosess. I det siste miljøet utviklet algoritmen atferd som tilfredsstilte vår definisjon av samarbeid, og som ble utviklet gjennom en prosess som tilfredsstilte kriteriene våre for kreativitet.

Preface

This is a Master's Thesis in Informatics with specialisation in Artificial Intelligence at the Norwegian University of Science and Technology (NTNU). The thesis was authored by two students in cooperation.

We would like to thank our supervisor Björn Gambäck and our co-supervisor Marinos Koutsomichalis for all their great help and feedback while we were working on the thesis.

Einar Hov & Andreas Høgetveit Weisethaunet

Trondheim, 1st June 2018

Contents

1. Introduction **1**
1.1. Background and Motivation . . . 1
1.2. Goals and Research Questions . . . 2
1.3. Research Method . . . 3
1.4. Contributions . . . 4
1.5. Thesis Structure . . . 4

2. Background Theory **5**
2.1. Machine Learning . . . 5
2.2. Evolutionary Algorithms . . . 5
2.2.1. The Algorithm . . . 6
2.2.2. Representations . . . 7
2.2.3. Population . . . 8
2.2.4. Genetic Operators . . . 8
2.2.5. Fitness Function . . . 8
2.2.6. Termination Criteria . . . 9
2.3. Artificial Neural Networks . . . 9
2.3.1. Neuron Categories . . . 9
2.3.2. Activation Function . . . 10
2.3.3. Network Structures . . . 10
2.3.4. Network Training . . . 11
2.4. NEAT . . . 12
2.4.1. Minimal Structures . . . 12
2.4.2. Mutation Function . . . 13
2.4.3. Crossover Function . . . 14
2.4.4. Speciation . . . 15

3. Related Work **17**
3.1. Computational Creativity . . . 17
3.2. Definition of Creativity . . . 18
3.3. Evaluating Creative Systems . . . 19
3.4. Systems in Computational Creativity . . . 20
3.4.1. CreBe . . . 21

3.5. Scriptbots 21

4. Architecture **23**
4.1. Overview 23
4.2. Simulation Life Cycle 25
4.3. Agents 26
4.3.1. Sensors 26
4.3.2. Actuators 28
4.4. Training Methods 29
4.5. Data Collection 29
4.5.1. Control and Visualisation Windows 30
4.5.2. Saving, Loading and Brain Visualisation 31
4.6. Libraries 34
4.6.1. MultiNEAT 34
4.6.2. Box2D 34
4.7. Minimum Requirements 34

5. Experiments and Results **35**
5.1. Cooperative Behaviours 35
5.2. Test Plan 36
5.2.1. Test Procedure 36
5.3. Test Phases 37
5.3.1. Preliminary Testing 38
5.3.2. First Set of Experiments 38
5.3.3. Second Set of Experiments 39
5.3.4. Third Set of Experiments 40
5.4. Food Environment 40
5.4.1. Results of the Food Environment 42
5.5. Food Chain Environment 42
5.5.1. Results of the Food Chain Environment 43
5.6. Evasion Environment 45
5.6.1. Individual Fitness Results 46
Prey 46
Predators 48
5.6.2. Shared Fitness Results 48
Prey 48
Predator 49
5.7. Door Environment 49
5.7.1. Individual Fitness Results 50
Behaviour A 50
Behaviour B 53

5.7.2. Shared Fitness Results . . . 54
Behaviour A . . . 55
Behaviour B . . . 55
5.8. NEAT Parameters . . . 58
5.8.1. Population Size . . . 59
5.8.2. Other Parameters . . . 59

6. Evaluation and Discussion **63**
6.1. Evaluation . . . 63
6.1.1. Food and Food Chain Environments . . . 63
6.1.2. Evasion Environment . . . 64
Prey Evaluation . . . 64
Predator Evaluation . . . 64
Environment Evaluation . . . 65
6.1.3. Door Environment . . . 65
Individual Fitness Evaluation . . . 65
Shared Fitness Evaluation . . . 66
Environment Evaluation . . . 66
6.1.4. NEAT Parameters . . . 66
6.2. Discussion . . . 67
6.2.1. Research Question 1 . . . 67
6.2.2. Research Question 2 . . . 68
6.3. Limitations . . . 69
6.3.1. Training Methods . . . 69
6.3.2. Architecture . . . 69

7. Conclusion and Future Work **71**
7.1. Contributions . . . 71
7.2. Future Work . . . 72

Bibliography **73**

A. Appendices **75**
A.1. Example Configuration . . . 75
A.2. Running Multiple Tests . . . 78

List of Figures

2.1. Example of a genotype and its phenotype. 7
2.2. A model of an artificial neuron from an ANN. 10
2.3. Two examples of simple ANN models. 11
2.4. NEAT mutation operator. 14
2.5. NEAT crossover operator. 15

4.1. Screen capture of the Evsim system. 23
4.2. Overview of the Evsim architecture. 24
4.3. Simulation life cycle in Evsim. 26
4.4. Screen capture of an agent observing another agent. 27
4.5. A possible use of the yell mechanic. 28
4.6. Screen capture of the control window of Evsim. 30
4.7. Screen capture of the visualisation window of Evsim. 32
4.8. Example of a brain evolved by Evsim. 33

5.1. Screen capture of the food environment. 41
5.2. Average score of the herbivores in the Food environment. 41
5.3. Screen capture of the Food Chain environment. 42
5.4. Average score of the herbivores in the Food chain environment. . . . 44
5.5. Average score of the predators in the Food chain environment. . . . 44
5.6. Example of the herbivore and predator dynamic. 45
5.7. Screen capture of the Evasion environment. 46
5.8. Screen capture of a behaviour that emerged in the Evasion environment. 47
5.9. Average score of the prey in the Evasion environment. 47
5.10. Screen capture of the Door environment. 49
5.11. Screen captures of a behaviour that emerged in the Door environment. 51
5.12. Count of agents that entered the goal in an experiment in the Door environment. 52
5.13. Screen capture of a poorly performing population in the Door environment. 53
5.14. Screen capture of another behaviour that emerged in the Door environment. 54

5.15. Count of agents that entered the goal in an experiment in the Door environment. 55
5.16. Screen captures of the waiting behaviour in the door environment. . 56
5.17. Screen capture of a behaviour in the Door environment. 57
5.18. Count of agents that entered the goal in the best individual in different experiments on the Door environment. 58
5.19. Average performance of agents that entered the goal in experiments testing different population sizes on the Door environment. 60
5.20. Count of agents that entered the goal in the best individuals when testing different NEAT parameters on the Door environment. 61

List of Tables

4.1. Agent sensors. 26
4.2. Agent actuators. 29

5.1. Interesting NEAT parameters. 38
5.2. Evsim parameters. 39
5.3. Configurations in the first set of experiments. 39
5.4. NEAT parameters and the values tested. 40

Glossary

ACC Association for Computational Creativity. 1, 17

AI Artificial Intelligence. 5, 17, 21

ANN Artificial Neural Network. 5, 9–12

Computational creativity A subfield of Artificial Intelligence. 1, 17–19

CreBe Creative Creature Behaviour. 21

CTRNN Continuous-time Recurrent Neural Network. 21

Evolutionary algorithm An optimisation algorithm inspired by biological evolution. 5, 6, 8, 12, 20, 21, 72

Evsim The evolution simulator created for this thesis. 23, 24, 28–31, 34–38, 69, 71, 75, 78

Genetic algorithm A variant of evolutionary algorithms. 12

GUI Graphical User Interface. 25

Historical marking Marking that tracks the lineage of innovations in NEAT. 13, 14

ICCC International Conferences on Computational Creativity. 1, 17, 18

MultiNEAT A software library implementing the NEAT algorithm with optional extensions. 25, 34

NEAT NeuroEvolution of Augmenting Topologies. 2–5, 12–16, 21, 24, 25, 29–31, 34, 37, 38, 40, 42, 43, 45, 50, 53, 54, 58, 59, 63–69, 71, 72

Neuroevolution The process of evolving neural networks with evolutionary algorithms. 12

Reinforcement learning A machine learning technique. 5

RNN Recurrent Neural Network. 11, 22

SPECS Standardised Procedure for Evaluating Creative Systems. 19–21

Supervised learning A machine learning technique. 5

TWEANN Topology and Weight Evolving Artificial Neural Network. 12

Unsupervised learning A machine learning technique. 5

1. Introduction

People have programmed computers to be creative systems for some time. Computers have, for instance, produced creative artefacts such as music or images—either independently through an algorithm, or as tools helping human artists. Computers have also been used to evolve or learn behaviours, for instance to teach robots to walk. While these two topics have been worked with a lot, there has not been much research done that explicitly combines creativity and behaviour. This thesis focuses on the intersection of creativity and behaviour. It specifically examines the evolution of cooperative behaviour in a creative system.

1.1. Background and Motivation

Computational creativity is a growing research field in artificial intelligence that focuses on creativity in computational systems. It has an increasing number of workshops and conferences (Jordanous, 2012a), such as the yearly international conferences called the International Conferences on Computational Creativity (ICCC). The ICCC is organised by the non-profit organisation called Association for Computational Creativity (ACC) which is working on the advancement of the field[1]. Among the main questions in the field of computational creativity is the question of what creativity is and how to make systems that are considered to be creative.

Why should we even bother to create creative systems? Creativity is claimed to be one of the key features that make us human and it is a feature that is highly valued in human society (Colton and Wiggins, 2012). By focusing on creativity, there have been built many systems that contribute value in creative fields, such as painting, video game design, poetry and mathematics (Colton and Wiggins, 2012). One of the long-time goals in computational creativity is to see creative software used in the community. The uses can range from people listening to music or playing games created by creative software, to other applications created by software that exceeds human creativity (Colton et al., 2015).

[1] `http://computationalcreativity.net/home/` (accessed 8.12.2017)

This thesis investigates whether an algorithm called NeuroEvolution of Augmenting Topologies (NEAT) can creatively evolve cooperative behaviour. This was investigated by evolving agents in multi-agent simulations, with the agents evaluated based on their ability to solve predetermined tasks. The interaction between agents of different species—with different tasks—was also of interest. Cooperative behaviour was chosen as the area of focus, as we expected that this sub-field could bring the most interesting behaviours. If NEAT is found to be capable of evolving such behaviour creatively, concepts behind the algorithm may be built upon to create creative software which can be used in, for example, video games or other kinds of simulations.

1.2. Goals and Research Questions

Goal *The project goal is to explore if the NEAT algorithm can be used to creatively evolve cooperative behaviour.*

RQ1 *How does NEAT perform in creatively evolving cooperative behaviour in multi-agent settings?*

Two types of cooperative behaviour will be investigated. We expected that more interesting behaviour could occur with the second type while the first type would be easier to achieve.

1 Can NEAT evolve cooperative behaviour that is potentially beneficial to all of the agents involved in the interaction?

In this behaviour, two or more agents interact in such a way that the interaction brings a potential for benefit for the agents, and is either beneficial for all of those involved or for none of them. An example would be a group of predators herding a prey towards each other in a setting where all the predators are rewarded when the prey is caught.

2 Can NEAT evolve cooperative behaviour that is potentially beneficial to some of the agents involved in the interaction?

In this behaviour, the agents interact in such a way that there is a potential for a reward for at least one of the participating agents. An example of such behaviour is the schooling behaviour of many species of fish, which amongst other things help the individual fish avoid predators.

RQ2 *To what degree do different parameters affect the ability of NEAT to creatively evolve cooperative behaviour?*

The NEAT algorithm has multiple parameters that may affect the obtainable results. Which of these parameters have a noticeable influence on the results, and what is the magnitude of their effects?

1.3. Research Method

This research was mainly a design, implement and experiment research. A system was designed and implemented which simulates multi-agent environments. Experiments were then run on this system to collect data needed to answer the research questions.

The goal of this thesis is to explore whether the NEAT algorithm can *creatively evolve* cooperative behaviour. In this thesis, 'creatively evolving' behaviour means that the evolved behaviour has to emerge from a process which is considered to be creative. The behaviour has to be novel and useful, and the process should be able to evaluate the usefulness of the behaviour and guide itself based on this evaluation. The process should arrive at new behavioural patterns that improve the fitness of the agents and refine and keep this behaviour for as long as it is useful.

The agents needed an environment that enabled them to be evolved based on their performances and interactions with other agents. To accommodate for the evaluation of the evolutionary process, the system had to have a way to present the agents' behaviours for observation. A few existing systems with relevant features were considered as potential frameworks for the experiments, but the cost of adapting any of them to this project's requirements was estimated to be higher than creating a solution from scratch. A new system was therefore implemented. The experiments run on the implemented system tested if cooperative behaviour can be evolved creatively through the NEAT process. NEAT's ability to evolve cooperative behaviour under different circumstances was tested with multiple environments and configurations of these.

The cooperative behaviour was evaluated based on the two types of cooperation defined under Research Question 1. The animations created by the system were inspected to see whether cooperative behaviour had emerged. Data of the agents' performances in solving their tasks were also used, as a guide to what parts of the resulting data from the experiments to inspect and to draw conclusions about properties of the cooperation. The agent's motives—fitness function—were also considered. The agents' fitness functions are important to determine to which one of the two types of cooperative behaviour the behaviour belongs. The observed

behaviours were also described.

The NEAT algorithm has multiple parameters that affect how the algorithm runs. To answer Research Question 2, some of the parameters we believed would have the most impact on the emergence of creativity were selected for experimentation.

1.4. Contributions

This thesis' maincontribution is an evaluation of NEAT's ability to creatively evolve cooperative behaviour in a multi-agent system. The implemented system which was used to conduct the experiments is also a contribution to the field of computational creativity.

The system code is open source and can be found at `https://github.com/einhov/evsim`.

1.5. Thesis Structure

Chapter 2 introduces the background literature needed to understand the rest of the thesis.

Chapter 3 describes the field of computational creativity. It also presents some creative systems and some systems with focus on evolving behaviour.

Chapter 4 shows the architecture of the system which was developed and used to conduct the experiments.

Chapter 5 describes the experiments and reports the results. It also shows the test plan which was followed.

Chapter 6 evaluates and discusses the results from the experiments.

Chapter 7 contains the conclusion and future work section.

2. Background Theory

This chapter introduces the background literature needed to understand the rest of this thesis. First it explains machine learning and evolutionary algorithms. Then it continues by explaining Artificial Neural Networks (ANNs). Finally, it describes an algorithm called NeuroEvolution of Augmenting Topologies (NEAT) that combines an evolutionary algorithm to evolve and optimise ANNs.

2.1. Machine Learning

Machine learning is a technique in Artificial Intelligence (AI) which enables computer programs to learn from and identify patterns in data without being programmed specifically for the particular data set. This allows computers to learn and adapt to new circumstances and environments. There are three main types of machine learning differentiated by how much feedback is given back to the system during the learning process. The types are *unsupervised learning*, *supervised learning* and *reinforcement learning*.

In unsupervised learning the agent identifies patterns in data without any feedback from the environment. Most often this is done by clustering the input examples. In reinforcement learning the agent receives occasional feedback from the environment, for example whether the agent has won or lost a board game. The agent will then use the feedback to tune its behaviour based on whether the feedback was positive or negative. In supervised learning every input to the agent has a known correct output. The agent receives as feedback this correct output every time it processes an input and tunes its behaviour towards the known correct output (Russell and Norvig, 2010).

2.2. Evolutionary Algorithms

Evolutionary algorithms is a class of algorithms designed to create solutions for optimisation problems (Eiben and Smith, 2015). Drawing inspiration from evolu-

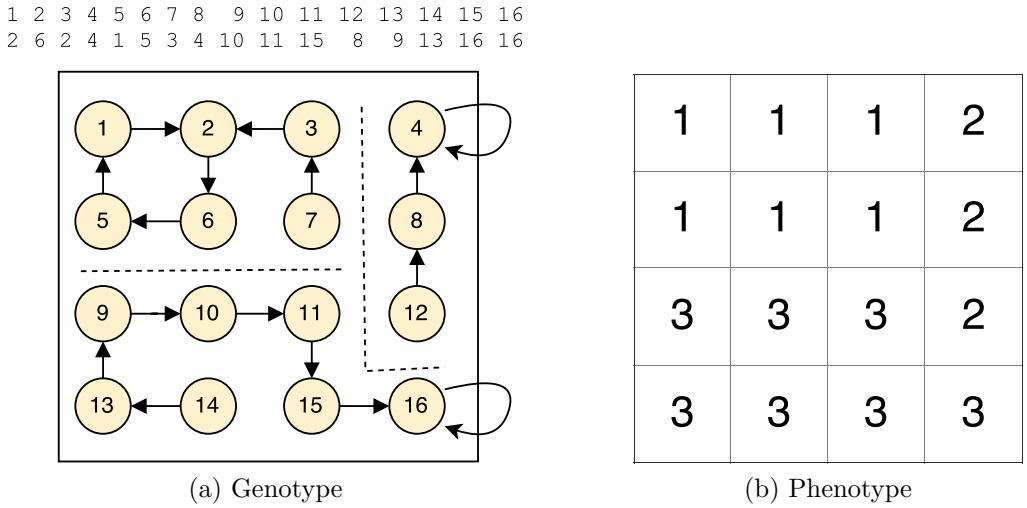
tionary processes observed in nature, evolutionary algorithms maintain a dynamic *population* of candidate solutions for the problem to be solved. The changes to the population are guided over time in such a way that the candidate solutions—one hopes—generally move towards improved solutions. To guide the changes, a quality measure of each candidate solution is typically procured by means of a *fitness function* over the evaluation metrics for the given problem. Changes to the population is done by the application of *genetic operators*—*selection*-, *crossover*- and *mutation* operators of various behaviours—to the population. The exact operators are chosen for the particular problem at hand, but are typically designed to retain structures from good candidate solutions while exploring the search space by combinations of and mutations on the solutions. The operators act upon representations of the candidate solutions—these representations can map to solutions in various degrees of directness.

2.2.1. The Algorithm

The general structure of the algorithm is shown as pseudo-code in Algorithm 1. The first step of the algorithm—line 2 of the pseudo-code—initialises the population with random individuals often to a fixed size. The next step evaluates the individuals in the population. The first step in the loop is the selection of parents. The two steps in lines 6 and 7 form the population of the next generation by optionally applying crossover- and mutation operators to the selected parents. The new individuals are then evaluated and the loop repeats until the chosen termination condition is satisfied.

Algorithm 1 Evolutionary Algorithm

```
1: procedure EA
2:     initialise population
3:     evaluate population
4:     repeat
5:         select parents for new population
6:         crossover selected parents (optional)
7:         mutate the children (optional)
8:         evaluate the new population
9:     until termination condition is satisfied
10: end procedure
```



(a) Genotype (b) Phenotype

Figure 2.1.: Example of genotype[1] and phenotype from an image segmentation problem. The genotype in (a) is a graph in which each node has exactly one edge to a geometrically neighbouring node. The numbers on top encode the edges of the graph. Below is a visual representation of this graph. The graph is composed of disjoint connected subgraphs. Each of these subgraphs represent one segment. The genotype decodes to a bitmap of segment numbers, visualised in (b).

2.2.2. Representations

The genetic operators derive new solutions by making changes to existing solutions. Many problems do not have solutions that lend themselves to a model in which a solution can be easily changed by the genetic operators into new useful and valid solutions. For this reason, alternative representation models more suitable for transformation are employed. These alternative representations are called *genotypes* and the solutions they represent are called *phenotypes*. Solutions have a mapping from genotypic to phenotypic space. Genetic operators operate on the genotypes. The genotypes are decoded to phenotypes for fitness evaluation. An example of a genotype and its phenotype is illustrated in Figure 2.1.

[1]Figure 2.1a has been taken from an assignment delivery authored by Christian Duvholt and Einar Hov as part of the course 'Bio-Inspired Artificial Intelligence' (IT3708) at NTNU during the spring of 2017.

2.2.3. Population

All the individuals in one generation of the algorithm form a *population* of candidate solutions. The population is a pool for parent- and survivor selection when making new generations. It is desirable to maintain diversity—making sure individuals do not get too similar—in the population to properly search the solution space and avoid having the algorithm settle on local optima. This may be a factor to consider when designing the genetic operators. Parameters to the algorithm may also need to be tuned for each instance of the problem to ensure diversity.

2.2.4. Genetic Operators

In the breeding of a new population, various genetic operators are applied to the individuals of the current population—notably *mutation-*, *crossover-* and *selection* operators. The operators fill different roles and variants of evolutionary algorithm may choose which operators to use. The mechanism of each operator is dependent on the problem to be solved and the chosen representation.

The selection operator is used to select parents for reproduction. In selecting parents, the operator will generally have a preference for the fitter individuals. However, the selection is usually stochastic so that even less fit individuals have a chance to be selected.

The mutation operator produces an offspring from a single selected parent. It introduces stochastic changes in the genotype of the parent to produce an offspring slightly different from the parent. The operator is usually employed to introduce variance in the population.

Inspired by sexual reproduction in nature, the crossover operator—also known as the recombination operator—produces an offspring from the genetic material of two or more parents. The idea is that by combining material from different parents useful features from both parents may be collected into one individual, spreading good innovations into the population.

2.2.5. Fitness Function

Evaluation of individuals is done with a fitness function. The fitness function implements a problem specific measure of performance with the individual's phenotype as input. The calculated fitness is used for parent selection, guiding the search towards more and more fit solutions.

2.2.6. Termination Criteria

The question of when to terminate the algorithm depends on the problem to be solved. If an optimal fitness for the problem is known or one is able to set a desired targeted fitness, the algorithm can be made to run until this fitness has been reached by one or more individuals in the population. Depending on the problem and how the genetic operators have been designed, the algorithm can stagnate—that is, fail to create improving solutions—for example due to a lack of diversity in the population or increasing difficulty of generating better solutions. In this case the algorithm can be made to terminate if the rate of improvement drops below a threshold. If the budgeted resources for the search is known, these can be used as the termination criteria—for example running the algorithm for a set amount of time or generations. Finally, the user of the algorithm can choose when to terminate the search interactively.

2.3. Artificial Neural Networks

Artificial Neural Networks (ANNs) are a family of systems inspired by neural networks in biological brains. An ANN consists of a number of interconnected nodes named artificial neurons. These nodes are analogous to neurons in biological brains. Directed connections between neurons form inputs and outputs to the neurons. In one activation of the ANN, each artificial neuron calculates an output based on its inputs and an activation function. Each input has a weight which says how much the input contributes to the neuron's activation (Russell and Norvig, 2010). As neurons are activated throughout the network, a signal is propagated through it.

2.3.1. Neuron Categories

The neurons can be separated into three categories based on their locations in the network—namely input-, hidden- and output neurons. The input neurons receive external data as input to the ANN—this can for example be sensor data. The output neurons collect the result of activating the network. Hidden neurons are those that are neither input nor output nodes. These neurons allow the network to implement more complex functions than with only input and output nodes. Input neurons output the raw input data. Hidden and output nodes calculate their outputs with their activation functions. A model of an artificial neuron is visualised in Figure 2.2.

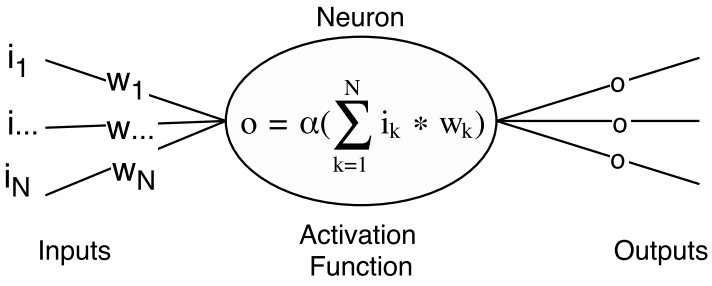


Figure 2.2.: A model of an artificial neuron from an ANN. i_n are input values to the displayed neuron, originating from the outputs of other neurons. w_n are weights of the input connections. α is the activation function of the neuron. It calculates the neuron's output o.

2.3.2. Activation Function

The activation function is used by artificial neurons to calculate their output values. All the input values to the neuron—that is, the outputs of the neurons that have their outputs connected to this neuron—are reduced into a weighted sum, which is input to the activation function. The type of activation function to use will depend on what one is trying to achieve with the network (Russell and Norvig, 2010). Examples of useful activation functions are the identity function, binary step and various sigmoids.

2.3.3. Network Structures

The structure of ANNs vary depending on the nature of the problem they are trying to solve. A simple form of ANNs are called *perceptrons*. The perceptron is an ANN with no hidden nodes and one output node with a binary threshold activation function. Perceptrons can solve any linearly separable classification problem, but if the classification is not linearly separable the network requires a more complex structure with hidden nodes. For many problems one employs multiple layers of hidden nodes, where each layer of hidden nodes is connected to the next one. If all neurons of each layer are connected to all of the neurons in the next layer, the network is said to be a fully connected network.

There are two distinct ways the data can flow—as seen in Figure 2.3. In the first way the data flows in only one direction from the inputs through the hidden nodes and to the output nodes without any cycles. This is called a *feed-forward network*. In the other way cycles are allowed. This means that data can flow

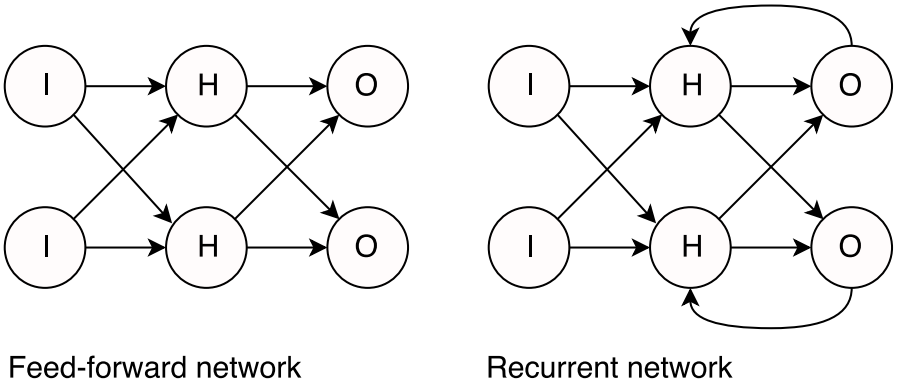


Figure 2.3.: Two examples of simple ANN models. Both networks have one input layer (I), one hidden layer (H) and one output layer (O).

backwards into earlier parts of the network by connecting outputs from a layer deeper in the network to inputs in earlier layers. This kind of network is called a Recurrent Neural Network (RNN). An important difference between feed-forward networks and RNNs is that the recurrent connections in RNNs can maintain states between activations of the network—a memory from previous activations—while a feed-forward network has no way of storing any such state (Russell and Norvig, 2010).

2.3.4. Network Training

Before an ANN produces sensible results it has to be trained. The training consists of manipulating the weights of the network's connections to maximise the network's performance at a task—for example, to classify images into categories with as few errors as possible. There are various techniques for training the networks.

One of the most popular training methods is a supervised learning method—see Section 2.1—called *backpropagation*. With this method a series of inputs from a data set with known correct outputs are sent through the network. After the output has been produced by the network, it is compared to the known correct output and an error is calculated. This error is then propagated backwards through the network and the network's weights are adjusted slightly such that the error is decreased. The intention is that through repeated application of this method the errors in the outputs from the network approach zero (Russell and Norvig, 2010).

If a correct output is not known, or there is no single correct output, one has

to look to other techniques to train the network. One possibility is to train the network with evolutionary algorithms—evolutionary algorithms are described in Section 2.2—using genetic operators to change the weights and possibly structure of the network (Stanley and Miikkulainen, 2002). This is useful when it is impossible or hard to tell a network's performance in a single activation.

2.4. NEAT

NeuroEvolution of Augmenting Topologies (NEAT) is an algorithm that uses *neuroevolution* to evolve ANNs. Neuroevolution algorithms use evolutionary algorithms to evolve the structure and parameters of neural networks, but search through the problem space instead of estimating the utilities of different actions. This makes neuroevolution effective in continuous and high-dimensional state spaces (Stanley and Miikkulainen, 2002).

NEAT is part of a subset of the neuroevolution algorithms called Topology and Weight Evolving Artificial Neural Networks (TWEANNs). As the name suggests, algorithms in this family evolve both the topology—the structure—of the network and the weights of the network's neurons (Stanley and Miikkulainen, 2002). The evolutionary algorithm used in NEAT is of the *genetic algorithm* variant, using both mutation and crossover operators. New generations are formed by first removing the least fit individuals from the population and then the entire population is replaced with the offspring of the remainders. The specifics of the genetic algorithm as used in NEAT is presented later in this section.

2.4.1. Minimal Structures

In many TWEANNs the topologies of the individuals in the initial population are random in order to begin with a diverse population. When starting out with random topologies, it may be hard for the algorithm to prune the topologies to minimal solutions as there may be more complex topologies with better fitness that will dominate. This may lead to extraneous nodes and connections. One workaround is to encourage the algorithm to remove nodes by taking the size of the network into account when designing the fitness function, giving a negative reward for large networks. However, this fitness function can be hard to tune. Furthermore, the modification to the fitness function can lead the search away from the main intention of the fitness function.

NEAT instead starts out with a minimal topology with no hidden nodes and all

inputs connected to all outputs. A minimal structure is preferable as it keeps the search space as small as possible thus keeping the performance of the algorithm as high as possible (Stanley and Miikkulainen, 2002). The structures are further kept as small as possible as newly evolved structures only get transferred to future generations if the individuals carrying them perform well enough. This keeps the structures as minimal as possible throughout the whole training. At the same time this makes it hard for new structures to be innovated, as a new structure may not be useful immediately, but grow useful some generations later. NEAT attempts to solve this by dividing the population into *species* as explained in Section 2.4.4 (Stanley and Miikkulainen, 2002).

2.4.2. Mutation Function

The NEAT algorithm has two structural mutation functions as shown in Figure 2.4. A mutation either inserts a node on a connection between two nodes or adds connection between two nodes. NEAT always grows its structure and does not have mutation functions that remove nodes or connections (Stanley and Miikkulainen, 2002). In addition, the figure illustrates the structure of the phenotype—the tree structure—and its associated genotype—the lists below the phenotypes—that are used for the topology in NEAT.

The genotype consists of a list of genes representing connections in the network. Each gene consists of a reference to the two nodes of the connection, as well as the weight of the connection and an innovation number. The innovation number is a global number for each gene and each globally new gene is allocated a new innovation number by taking the globally highest number and incrementing it by one. As a result the same gene has the same innovation number in all the genotypes it exists in (Stanley and Miikkulainen, 2002). The innovation number is used as a *historical marking* that identifies the ancestors of each gene. Figure 2.4 shows one example each of a node and a connection being added to a network. When adding a new node, the old connection from Node 1 to Node 4 is replaced by the two new connections from Node 1 to Node 5 and Node 5 to Node 4. This results in the old gene being disabled, and the two new genes being added at the end of the genotype. When adding a new connection the new gene is simply added at the end.

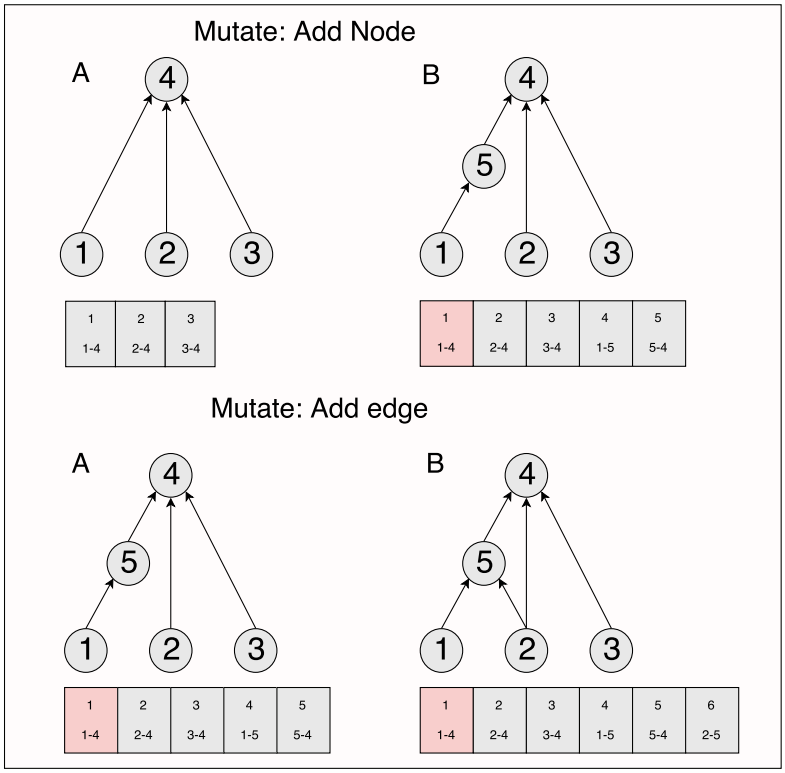


Figure 2.4.: NEAT mutation operator. Inspiration taken from Figure 3 in (Stanley and Miikkulainen, 2002). The red nodes represent disabled nodes.

2.4.3. Crossover Function

In addition to mutating new nodes and connections, NEAT has the ability to create an offspring by merging two parents. As NEAT operates on individuals with diverse topologies, it has to have a way to match parts of the individuals' topologies to identify the corresponding genes in the parents for merging. NEAT accomplishes this by using the historical markings of the genes. If two individuals have the same set of genes, the two individuals must have the same structure. This holds even if the individuals have taken a different sequence of genetic operations to arrive at the current topology. When two individuals are matched, their genes are distributed into three different categories, namely matching genes, disjoint genes and excess genes. The genes are matching if the same gene is present in each of the parents. The remaining genes are categorised as disjoint or excess based on whether they occur within or outside the range of the other parent's innovation numbers (Stanley and Miikkulainen, 2002). An example of matching two parents is shown in Figure 2.5.

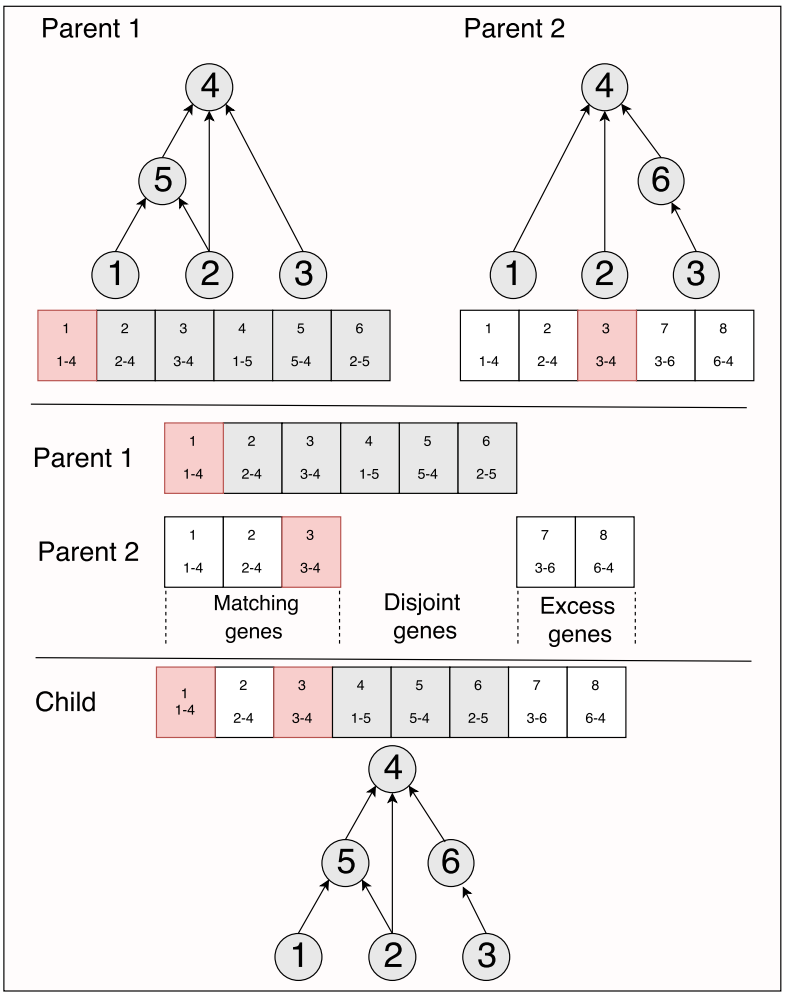


Figure 2.5.: NEAT crossover operator. Inspiration taken from Figure 4 in (Stanley and Miikkulainen, 2002). Assuming equally fit parents.

Any matching genes in the parents are passed on to the offspring with the weight randomly chosen from either of the parents. In case of disjoint and excess genes, the offspring receives the genes of the most fit parent. When both of the parents are equally fit, the offspring inherits the disjoint and excess genes randomly (Stanley and Miikkulainen, 2002). This is the case in Figure 2.5.

2.4.4. Speciation

To maintain diversity in the population and to protect new structures, NEAT uses a technique called *speciation*. Speciation is a partitioning of the population

into species based on similarity. In NEAT the similarity is computed from the matching-, disjoint- and excess genes, as well as the genes' weights, between two individuals. The idea is that similar individuals also behave similarly, so that the species can occupy different niches of the search space.

To prevent one or a few species from completely dominating the population, the individual's fitness is not directly used when the algorithm is to choose parents for reproduction. Instead, it uses a modified fitness that takes into account the size of the species the individual belongs to and punishes individuals in larger species. This is called *explicit fitness sharing*. This helps new structures in species with fewer individuals survive long enough to adjust and become useful (Stanley and Miikkulainen, 2002).

NEAT uses speciation by primarily comparing individuals in the same species. This is done by modifying the individuals' fitness value based on how many individuals there are in the species. This gives individuals with newly formed structure time to adjust as they will come in species with fewer individuals resulting in a higher fitness compared to the more established species.

3. Related Work

This chapter presents some works related to subjects of this thesis. First, the research field of computational creativity is presented. Then some views on the definition of creativity from established authors are given and a few methods to evaluate creative systems are shown. Three projects in the field of computational creativity are covered. Finally, a system called 'Scriptbots' that evolves behaviours in simulated agents is described.

3.1. Computational Creativity

The field of computational creativity is a subfield of AI research. One central difference between mainstream AI research and computational creativity is that AI research is mainly in the problem solving paradigm, while computational creativity research is mainly in the artefacts creation paradigm (Colton and Wiggins, 2012). Computational creativity research is usually based on developing and working with systems that create ideas and artefacts in historically creative domains, such as poetry, story telling, music composition, mathematics, science, video games, industrial and graphic design (Colton and Wiggins, 2012).

Computational creativity is in growth with an increasing number of workshops and conferences (Jordanous, 2012a). Computational creativity has had yearly international conferences since 2004 called the International Conferences on Computational Creativity (ICCC). ICCC has been organised by the Association for Computational Creativity (ACC) since 2010. ACC is a not-for-profit organisation working on the advancement of computational creativity as a discipline and technology[1].

ACC has defined the goal of computational creativity as using a computer to simulate, model or replicate creativity in order to achieve one of three objectives. The first is to construct a program capable of human-level creativity. The second is to better understand human creativity through the work done in the field. The

[1] `http://computationalcreativity.net/home/` (accessed 8.12.2017)

last point is to create software that is not necessarily creative on its own, but can enhance human creativity.

3.2. Definition of Creativity

In the preface to the proceedings of ICCC 2017, a definition of the field of computational creativity is formulated. It reads: 'the art, science, philosophy and engineering of computational systems which, by taking on particular responsibilities, exhibit behaviours that unbiased observers would deem to be creative' (Goel et al., 2017).

Exactly what creativity is has proven to be a difficult question. Different science fields have put in much effort to define creativity. In the field of computational creativity several researchers, for instance Margaret Boden (1998) and Anna Jordanous (2012b), have tried to formulate a definition.

Margaret Boden is an author and a central figure in computational creativity with a number of published books and papers (Colton et al., 2009). She writes (Boden, 1998) that creativity is a fundamental feature of human intelligence. Boden further claims that creative programs can, among other things, help cognitive science by assisting psychologists towards an understanding of creativity in humans, and states that for an idea to be creative it needs to be surprising, valuable and novel. An idea can be novel in two different ways. Firstly, it can be novel for the agent itself. This creativity is called *P-creativity* with the P standing for *psychological*. Alternatively, it can be a novel idea for both the agents and its peers. This is called *H-creativity* with the H standing for *historical*. According to Boden, computational creativity systems should try to achieve P-creativity. Occasionally—if the models are good enough—H-creativity can occur.

Boden (1998) distinguishes three types of creativity. The first is *combinational* creativity that combines familiar ideas to create something new. An example of combinational creative artefacts are analogies. The second and third type are similar to each other. They are *exploratory* and *transformational* creativity. Exploratory creativity explores conceptual spaces and generates novel ideas throughout the search, such as creating new music by playing the guitar. Transformational creativity can in addition transform one or more dimensions of the conceptual space. For example, an agent can decide to also use the guitar as a drum. The guitar was originally designed to be a string instrument. By transforming the conceptual space of the instrument the agent found a novel way to make music, expanding the capabilities of the instrument.

Anna Jordanous (2012a) wrote about 14 key components of creativity. These components were derived from academic literature about human and computational creativity. This was done by extracting words from the literature that appeared more frequently than expected in particular topics. The result after extracting and clustering the frequent words were 14 key components—building blocks—of creativity that she argues together form a clearer understanding of the concept of creativity. Additionally it makes the concept of creativity more tractable and easier to grasp as it is separated into parts. This separation also helps when attempting to evaluate creativity, as the separation gives more detailed information of what creativity is (Jordanous, 2012a). One of the 14 components is originality. This component says that originality and novelty is important in creativity. This can be finding new associations between concepts, or producing results which are surprising and unexpected. Another component is Value. The value component says that the contribution must be useful and bring some kind of value that is relevant in the domain.

3.3. Evaluating Creative Systems

Evaluation of the creative systems has become increasingly important in the field of computational creativity (Jordanous, 2014). The evaluation is important to identify and track strong and weak points of a system. It can help researchers see where progress is made and to point out how the systems can be improved (Jordanous, 2012a). Several evaluation methods have been developed, such as *Ritchie's empirical criteria* (Ritchie, 2007), the *creative tripod model* (Colton, 2008) and the Standardised Procedure for Evaluating Creative Systems (SPECS) (Jordanous, 2012b).

Ritchie's empirical criteria Ritchie's empirical criteria are based around the evaluation of the artefacts created by the creative systems and not the process of creating the artefact. He has created an evaluation framework that focuses on the artefact's typicality and quality. Typicality is a measure of how typical the artefact is within its domain. Quality is a measure of how high the quality the artefact is considering its domain (Ritchie, 2007).

The creative tripod The creative tripod model takes the creative process in consideration in addition to the artefacts created by the creative system. For a system to be deemed creative by the creative tripod method, the system needs to have three qualities. These qualities are skill, appreciation and imagination. Skill

is needed for the human or computer to be able to create anything at all in the domain. Appreciation is necessary to create something which has value. Imagination is important to create something which is novel, as without imagination the system may at best imitate other artefacts (Colton, 2008).

SPECS SPECS is separated into three steps (Jordanous, 2012a). The first step consists of two parts. The first part is to formulate a universal definition of creativity independent of the domain. The second part is to find the aspects of the definition found in part one that are most important in the domain in which the system operates in. The second step is to derive evaluation criteria from the aspects found in step one and to clearly state them as testable standards. The third—and final—step is to test the system based on the standards developed in step two. SPECS leaves it up to the researchers to weight the test results according to how important the different aspects are.

3.4. Systems in Computational Creativity

A well known project in the field of Computational Creativity is the 'Painting Fool' by Simon Colton[2]. The Painting Fool is both a computer program and a painter. Works made by the program have been shown in both online and physical galleries. The program has been used in multiple research works, where Colton and other researches have tried to identify how to create creative software (Colton et al., 2011; Pease and Colton, 2011; Colton, 2009, 2008). Another well known project in this field is 'GenJam'. GenJam is a program using genetic algorithms—genetic algorithms are variants of evolutionary algorithms, see Section 2.2—to improvise jazz music (Biles, 1994). The system has been described in books (Bentley and Corne, 2002; Miranda and Al Biles, 2007) and has been used in research work (Biles, 1994, 2003). The two previous systems are interesting as they make creative artefacts comparable to art produced by humans. The systems have also been evaluated, and thus serve as examples of how creative systems can be evaluated. While there have been many systems like these two that focus on generating artefacts in traditional, human creative fields, less work has been done on the generation of behaviour.

[2] `http://www.thepaintingfool.com` (accessed 14.12.2017)

3.4.1. CreBe

Creative Creature Behaviour (CreBe) is a computationally creative system with a focus on the generation of behaviour. It is a simulation system described in the Master's Thesis 'Creative Behaviour in Evolving Agents' by Alvestad and Larsen (2017). In the thesis, Alvestad and Larsen examined agents that were evolved using various techniques—through the NEAT algorithm, through an evolutionary algorithm applied to parameters of a fuzzy logic system, and by training of Continuous-time Recurrent Neural Networks (CTRNNs). Additionally, the physical structures of the agents—the locations and numbers of sensors and actuators—were also evolved using an evolutionary algorithm. The evolution of behaviour was tested on both agents with predesigned structure and agents that had their structure evolved together with the behaviour.

Alvestad and Larsen (2017) developed agents in three environments. Two of them were single-agent environments and one was a multi-agent environment. The first single-agent environment consisted of the agent and food for the agent to collect. The second single-agent environment was similar to the first, but with poison introduced in the environment. The agent's goal was to collect food while avoiding poison. The poison looked like food, but could be discerned from food by its smell. The multi-agent environment consisted of the agent to be evolved together with one predesigned agent. The goal of the agent to be evolved was to kill the predesigned agent while suffering as little damage as possible. The predesigned agent behaved in a fixed, stochastic way. The CreBe project's focus was restricted to the evolution of a single agent at a time. Their environments also had at most two simulated agents, and then in an adversarial setting.

SPECS—described in Section 3.3—was employed to evaluate the creative outcome of the system in Alvestad and Larsen (2017). They found that their system exhibited limited capabilities of producing creativity and argued that the agents performed problem solving rather than exploring novel behaviour (Alvestad and Larsen, 2017).

3.5. Scriptbots

There exist a few open source systems that simulate agents in a multi-agent setting using AI techniques. One of them is a system called 'Scriptbots' (Karpathy and Link, 2011). Scriptbots is a simulation system with multiple agents whose goals are to eat food and to avoid being eaten by other agents.

The idea in Scriptbots is to evolve agents that attempt to survive as long as possible in a given environment and to have interesting agent behaviour emerge in the process. The environment consists of plant food and the agents. The agents have sensors to detect objects in the environment and actuators to execute actions. The agents need to eat regularly in order to survive. At the same time they have to avoid getting harmed and eaten by other agents. In the newest version of Scriptbots at the time of writing this thesis—Scriptbots v4—there is no clear classification of herbivore and carnivore. Instead the different agents can both eat meat—that is other agents—and plant food with various levels of tolerance to plant food and meat. To generate new offspring, Scriptbots uses mutation and crossover functions. These functions are described in Section 2.2.4. Scriptbots uses a modified version of Recurrent Neural Networks (RNNs)—RNNs are explained in Section 2.3.3—as brains in the simulated agents, one for each individual. The Scriptbots system is interesting because it evolves the behaviour of many agents in simulated multi-agent environments. Unfortunately, it has not been evaluated in academic literature.

4. Architecture

As part of the work on this thesis, a system was implemented which simulates and evolves the behaviour of agents. The name of this system is 'Evsim'. This chapter describes the architecture of the simulation system. It first gives an overview of the system, followed by more detailed descriptions of its essential parts.

4.1. Overview

Evsim simulates and evolves one or more species in various environments. A screen capture of the system can be seen in Figure 4.1. A species is a population of agents. The agents have a set—predetermined by the type of agent in question—of sensory inputs from the environment and actuators with which they may affect

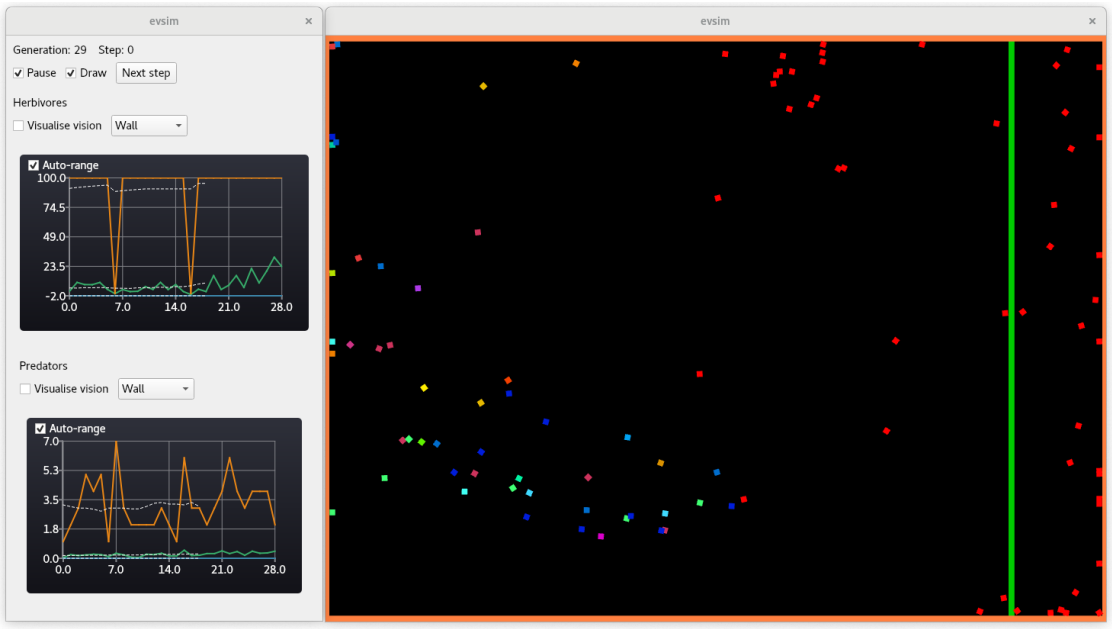


Figure 4.1.: Screen capture of the Evsim system showing the configuration window and the simulation window.

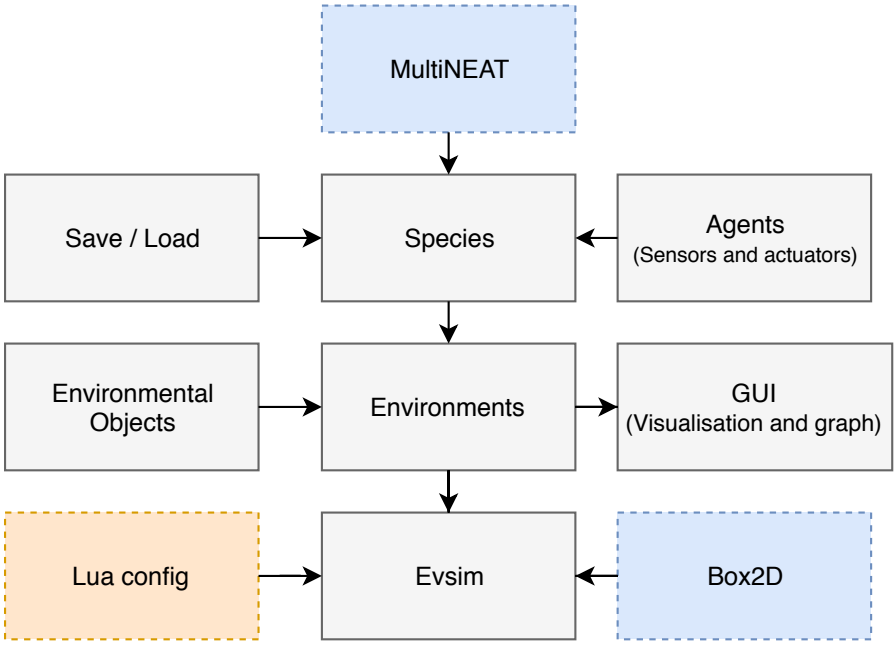


Figure 4.2.: Overview of Evsim. The blue boxes with dashed borders are external libraries used by Evsim. The yellow box with with dashed borders represents configuration files that are fed as input to the simulator. The grey boxes are the simulator itself.

the environment. Each Evsim species is trained as a population in the NEAT algorithm. Figure 4.2 shows an overview of Evsim, including the most important parts of the system.

The Evsim box controls the simulation. When the program is started, it is responsible for bringing up the simulation with the user's chosen configuration. Evsim accepts command-line arguments when started. The first argument is the path to a configuration script written in the Lua programming language.[1] The rest of the arguments are passed through to the configuration script, which can then implement arbitrary logic in Lua to initialise global variables which Evsim will extract for configuration. Parameters configurable from this script are general simulation parameters—such as max generations to simulate and what environment to use—as well as parameters changing properties of the environments and their contained agents. Configuration parameters that are not set in the script are assigned default values by Evsim. For a complete example configuration script see the Appendix, Section A.1. The flexibility of the configuration system facilitates automatisation of the execution of experiments with different parameters. As an

[1] `https://www.lua.org/about.html` (accessed 12.05.2018)

example, Section A.2 of the Appendix contains a Lua configuration script which takes additional command-line arguments, and a Python script which uses this configuration script to execute 78 experiments with different parameters—running eight instances concurrently.

The environment defines the simulated world and the physical properties of its agents. The environment contains one or more species of agents, and may also contain further environmental objects that may have interactions with the agents. Movement and collision detection in the simulation is computed by the Box2D physics simulation library (Catto, 2015). The species in an environment contain the environment's agents. Each species is trained by the NEAT algorithm using the MultiNEAT library (Catto, 2015). All species have functionality to save their population to files every generation and to load its population from files. Finally, the simulator has a Graphical User Interface (GUI) which lets the operator inspect experiments in progress. The GUI consists of a visualisation window which renders the environment, and a control panel with fitness graphs and a few visualisation parameters.

4.2. Simulation Life Cycle

The simulation life cycle is illustrated in Figure 4.3. Each execution of Evsim is one experiment. When starting the environment, its constituents are initialised based on the parameters chosen by the operator. After the initialisation, the main simulation loop is executed until its terminating criteria have been met—these can be configured as the maximum number of generations or at the operator's request. The main simulation loop iterates over ticks, steps and generations. The tick is the smallest temporal unit in Evsim. Each tick the physics simulation is progressed to its next state. Afterwards, every agent's neural networks are activated with sensory input, and the resulting outputs are used with their actuators. A step is a series of ticks forming one episode of the environment's simulation. The states of all objects and agents in the environment are reset before every step. A generation is a series of steps, and each generation is one epoch in the training of the species. The agents are given fitness values based on their performances in the steps of the generation, and then, at the end of the generation, the NEAT algorithm produces a new population to use for the subsequent generation.

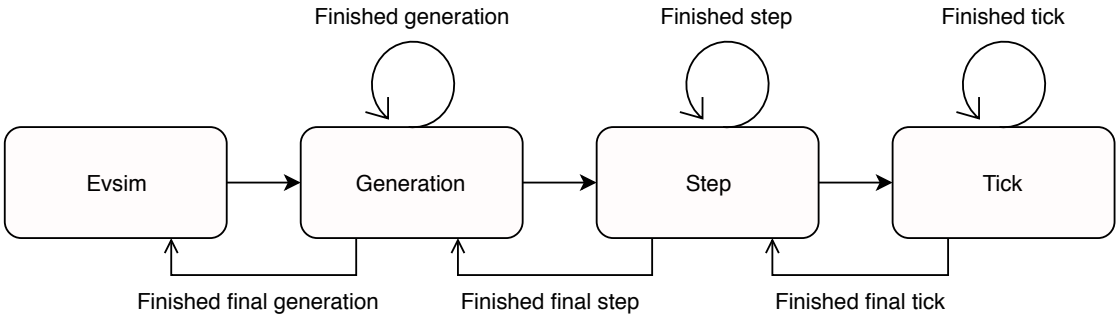


Figure 4.3.: Simulation life cycle in Evsim.

Sensor	Description	Number of inputs
Velocity	Linear- and angular velocities of the agent	2
Vision	Vision of various types and segment counts	$segments \times types$
Position	X- and Y-position of the agent in the environment	2
Angle	Angle of agent in the environment	1
Hear yell	Whether a yell is detected	1
Yell centre	Vector from agent to the yell's centre	2
On button	Whether the agent is on top of a button	1

Table 4.1.: Agent sensors.

4.3. Agents

The agents in the environment are rewarded based on how well they accomplish their predefined goals—for instance, to collect food. To help them accomplish their goals, they are equipped with sensors and actuators. An agent's sensors collect information about the environment. Based on this information the agent can choose each tick how to utilise its actuators to reach its goals. The brains of the agents are neural networks. Sensory data is fed as input to the network on activation, and the outputs control how the agent's actuators behave for the duration of that tick. An actuator can, for example, apply a linear force to the agent. Tables 4.1 and 4.2 list the sensors and actuators—respectively—that are used across the agents in the implemented environments.

4.3.1. Sensors

Table 4.1 lists all the sensors that are implemented in agents in Evsim. The velocity and vision sensors are used in all the species; the remaining sensors are only used

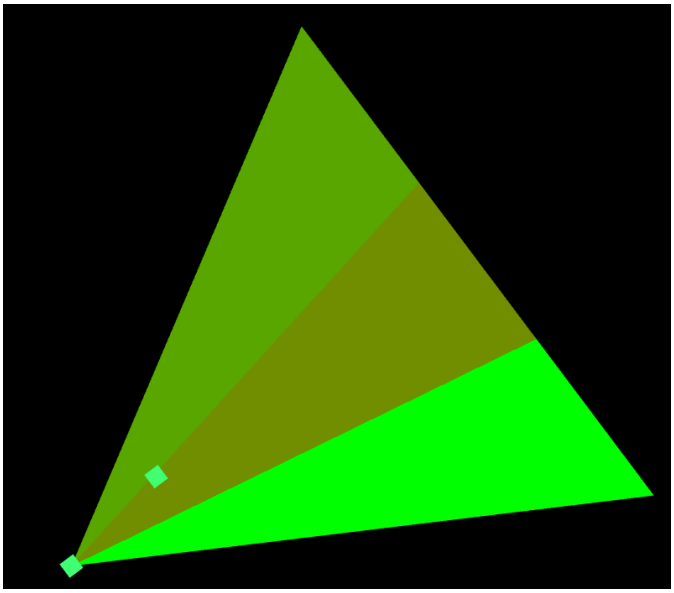


Figure 4.4.: Screen capture of an agent observing another agent.

in some of them. The two velocity inputs consist of the linear velocity of the agent along its forward vector and the angular velocity around its centre. These are provided to help the agent achieve better control over its movement. The X- and Y-position inputs are absolute coordinates in the environment's space and can be provided in cases where the location of the agent could be useful information for solving the goal. Likewise, the angle input is also an absolute angle in the environment's frame of reference.

The vision inputs let the agents observe their surroundings. An agent has one vision field for each type of object it is to differentiate. The vision field is separated into a number of segments, and each segment corresponds to one real-numbered input to the agent's neural network. The magnitude at which an object inside the vision field contributes to a vision segment depends on the distance of its centre from the observing agent and its location within the field. A segment is affected if the centre of the observed object is within the segment or in the closest half of neighbouring segments, and the magnitude falls off the further the object is from the centre of the segment. This way the detection of objects becomes continuous over the vision field. An example of the visualisation of an agent's vision is shown in Figure 4.4. This agent has a vision with five segments. An agent is located inside segment three of the vision field, contributing the inputs for segment two and three. Low values on the segments are visualised as a green colour, which blends into red for high values.

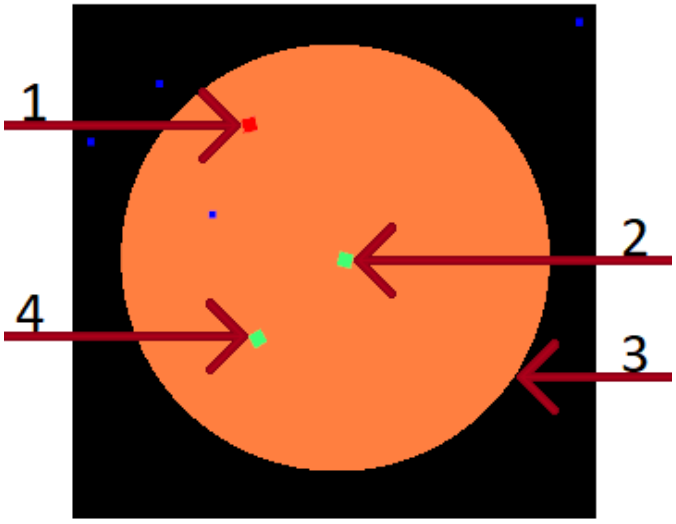


Figure 4.5.: A possible use of the yell mechanic. A dangerous object (1) is observed by an agent (2). The agent gives a yell signal, visualised as a disc (3). A different agent (4) is within range of the yell and receives the signal and becomes aware of the presence of the threat.

The yell mechanic enables agents to perform primitive communication. Yell signals are implemented as circular areas in the environment where agents inside the circle receive the signal. The hear yell input is set to 1 if the agent is inside a yell or 0 otherwise. The yell centre input is a vector from the yell to the agent that heard the yell. Figure 4.5 shows an example of how the yell signal looks in the visualisation and explains a possible use of the mechanic. The *on button* sensor is set to 1 if the agent is on top of a button or else 0. It is used for one environment where agents need to stay on top of a button.

4.3.2. Actuators

Compared to the sensors, there are fewer actuators. Table 4.2 shows all the actuators implemented in Evsim. The force actuator is used by every species. The force actuator is the agents' method of moving around in the environment. The two outputs for the force actuator are linear and angular forces to apply to the agent for the tick. The yell actuator creates a yell object in the environment centred at the agent's location. As the outputs from the neural network are continuous values, the yell actuator is programmed to activate if its output signal is below a certain threshold.

Actuator	Description	# of outputs
Force	Linear and angular forces to apply to the agent	2
Yell	Spawn a yell object if output is above threshold	1

Table 4.2.: Agent actuators.

4.4. Training Methods

Species in Evsim can be trained with two different training methods named *individual fitness* and *shared fitness*. In the former method, each agent has a score of its own calculated at the end of a simulation step. In the latter, all agents of the same species share the same score at the end of the step. Two methods were implemented because they were expected to yield different types of behaviour.

Individual fitness When agents of a species are trained with individual fitness, the entire species is simulated at the same time with one individual from the NEAT population represented by one agent in the environment. Each of the agents is given a score based on how well they accomplish their goal in the environment. For instance, if the agents' goal is to collect as much food as possible during a step, they will receive a score based on many pieces of food they collected. The NEAT algorithm will use the agents' scores to discriminate them in the parent selection when evolving the next population.

Shared fitness In the shared fitness training method, one individual of the species is evaluated at the time. The agents in the simulation representing the species are instantiated from the one individual to be evaluated and work together to obtain a shared score at the end of the simulation step. In other words, the agents are identical clones that all work towards the same goal. The NEAT algorithm then discriminates its individuals based on how their simulated population of agents performed.

4.5. Data Collection

As Evsim was created to answer the research questions of this thesis, it was essential to implement functionality for collecting data produced by the simulations. The performance of the NEAT individuals are recorded at the end of each generation. This information is presented to the operator of the simulation in a control

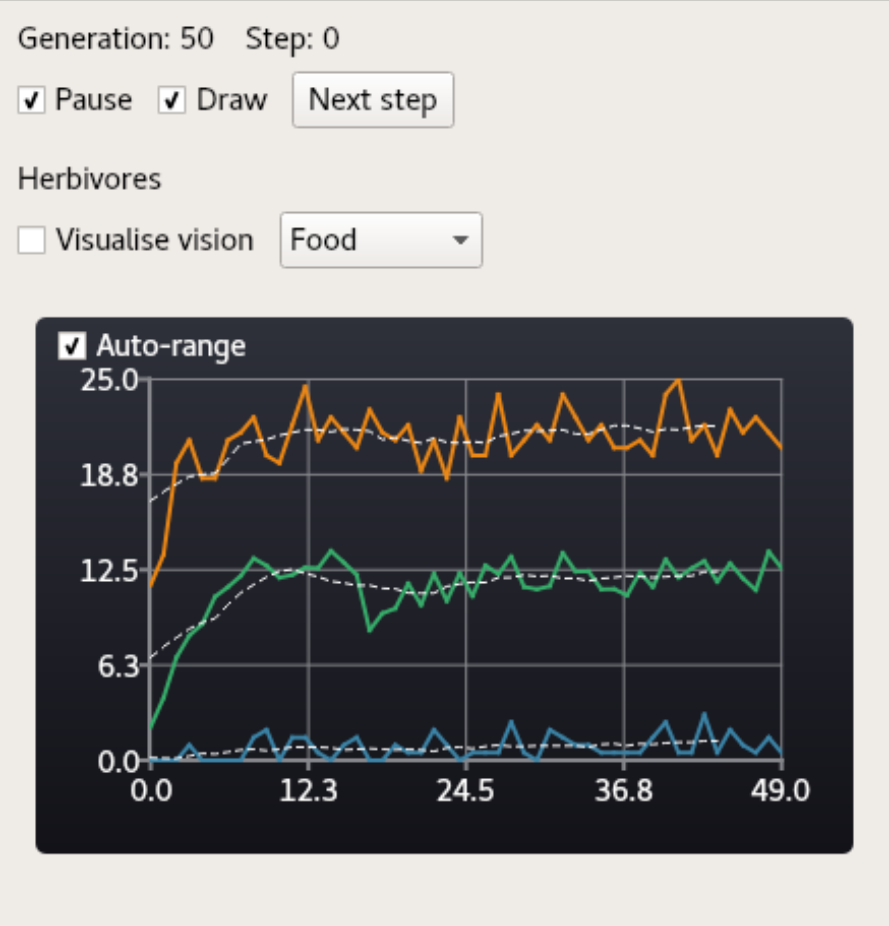


Figure 4.6.: Screen capture of the control window of Evsim.

window during training. If configured with an output directory, Evsim will also write the data it produces to files. Evsim will also store the NEAT populations to files every generation, and these can be loaded in again in later executions of the simulator. It is also possible to configure the simulator so that it does not perform training on a species. This configuration can be used to observe already trained populations from stored files. Lastly, the simulator can store the topology and weights of an agent's brain to a file.

4.5.1. Control and Visualisation Windows

The control window lets the operator of Evsim get an overview of how the species perform during an experiment, as well as control various aspects of the visualisation of the simulation. Figure 4.6 shows an example of the control window in

action. In the figure, an experiment is running with an environment containing one species named 'Herbivores'. At the top of the window, some information about the simulation state is displayed alongside controls for drawing the simulation in the visualisation window, pausing the simulation or moving to other steps.

Each species gets to contribute its own controls and displays to the window. For every species, the operator can control whether to visualise the agents' visions in the visualisation window and which type of object the vision visualisation should show. Underneath this is a graph plotting the worst, best and average performances of individuals in each generation—in orange, green and blue respectively. The x-axis is the generation number and the y-axis is the performance. Dashed lines show the centred moving average of the three data series, with a configurable window size.

Figure 4.7 shows the visualisation window of an environment with agents tasked to consume food. The large, variously coloured squares are agents of the one species in the environment. The agents' colours indicates which species in the NEAT algorithm they belong to—that is, agents with the same colour are similar to each other in behaviour. The smaller blue squares are stationary bits of food waiting to be consumed. The visualisation window can also draw other objects such as walls and goal posts, and sensors such as agents' vision fields or yell areas.

4.5.2. Saving, Loading and Brain Visualisation

Evsim's save and load functionality lets the simulation store all the data produced during training so that it can be loaded at a later time, whether that be to resume training from a loaded population or to run the simulation without training to inspect the behaviour of the generation. The performance of the individuals is also logged so that they may be analysed with other tools after the experiment has finished.

When inspecting an experiment with the visualisation window, it is possible to click on an agent to dump the neural network of its brain to a file. The brain dump is written as a graph specification in the DOT language[2] and can be visualised with any tool capable of drawing graphs from this specification, such as those from the Graphviz project.[3] An example of a visualised brain evolved in Evsim can be seen in Figure 4.8.

[2] `https://www.graphviz.org/doc/info/lang.html` (accessed 11.05.2018)
[3] `https://www.graphviz.org/` (accessed 11.05.2018)

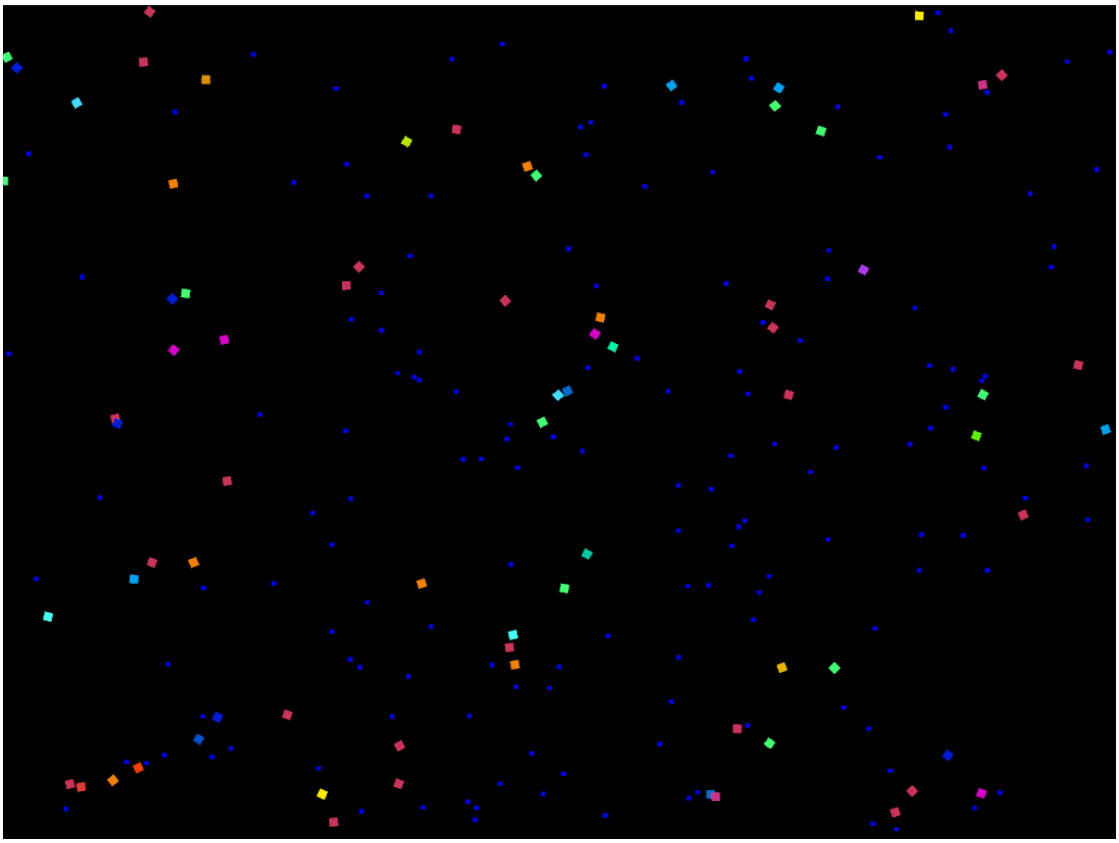


Figure 4.7.: Screen capture of the visualisation window of Evsim.

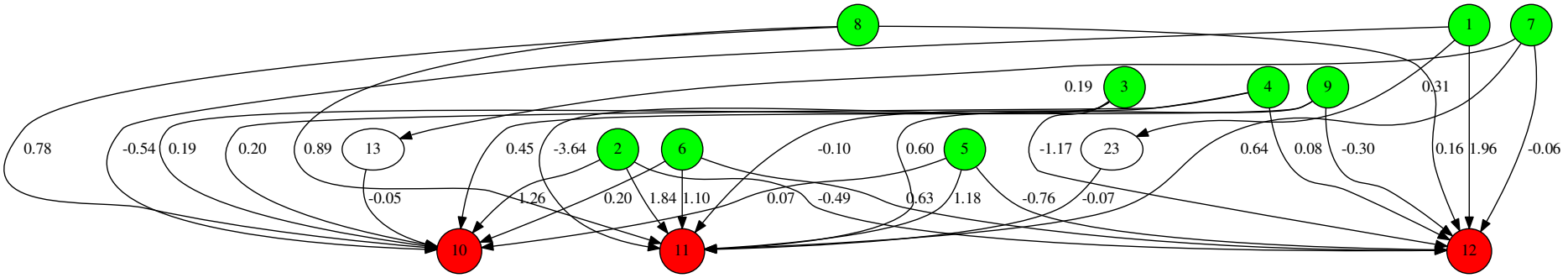


Figure 4.8.: Example of a brain evolved by Evsim. The green nodes are input nodes, the white are hidden nodes while the red nodes are output nodes. A line represents an edge that connects two nodes. The decimal number on each edge represents the weight value of the connection between the two connected nodes.

4.6. Libraries

Evsim builds upon various open-source libraries for things such as visualisation, UI, configuration, and to implement the simulation itself. Especially important are two libraries used to implement the simulation, namely an implementation of the NEAT algorithm named 'MultiNEAT' and the rigid body physics engine named 'Box2D'.

4.6.1. MultiNEAT

MultiNEAT (Chervenski, 2012) is a software library written in C++ which implements the NEAT algorithm. In addition to traditional NEAT, the library also provides variations such as HyperNEAT, rtNEAT and the use of novelty search. The implementation exposes many adjustable parameters. MultiNEAT has been used in both commercial and research contexts. The code is available under the GNU Lesser General Public License version 3 (GNU LGPLv3).

4.6.2. Box2D

Box2D (Catto, 2015) is used to simulate the physical environment in Evsim. Box2D is a two-dimensional rigid body simulator written in C++, calculating contacts and forces on geometric objects. The code is available under the zlib license. In Evsim Box2D is used to simulate the movement of objects, and to detect contacts used for activating sensors or interactions between agents and objects in the environment.

4.7. Minimum Requirements

Evsim will run on any modern machine with support for OpenGL version 3.3. It was implemented and tested on Linux systems, but may build fine for other systems with no or small modifications. Evsim is CPU intensive and if multiple experiments are scheduled to, run it is beneficial to have a modern multi-core processor to run multiple experiments in parallel. If the storing of data is enabled, Evsim requires some free disk space to store the files. The disk space needed and the computational time it takes to run an experiment depend highly on the environment and configuration of the experiment. As an example, the uncompressed data generated from the first set of our experiments—described in Section 5.3.2—was 7.1GiB in size, and the experiments crunched with up to 8 instances in parallel for approximately 194 hours on an Intel Core i7-4790 CPU before completion.

5. Experiments and Results

This chapter describes the experiments done as part of this thesis, how they were conducted, and the results that were obtained. In the start of the chapter, the different cooperative behaviours we imagined could evolve in the different experiments are presented. Then the test plan that was followed to obtain the result for this thesis is described. Afterwards, the test phases of this project are shown. Four different environments were implemented for Evsim, namely the Food, Food Chain, Evasion and Door environments. These environments are described in turn. The results from experiments on each environment are provided directly following the environment's description.

5.1. Cooperative Behaviours

Already before conducting the experiments, we had imagined a few possible behaviours that could emerge in the different environments. The following paragraphs describe three of these behaviours that were especially considered.

Swarm In this thesis, swarm behaviour means that agents of the same species move together and react to the positions and movements of other agents. This type of behaviour is inspired by nature where groups of individuals use proximity to other individuals as a way to reduce the risk of being caught by predators, either by reducing the chance of each individual to be targeted by a predator or by confusing predators who become unsure about which of its preys to chase. It is also inspired by predator behaviour were they use swarming patterns to, for instance, create 'nets' that the prey will not be able to get through without getting caught.

Warn behaviour Warn behaviour is a behaviour where the agents use their yell actuator—described in Section 4.3—to signals other agents in their species about nearby predators. The results were inspected for situations where an agent spotted a danger and reacted by issuing a yell signal as a warning to other agents nearby making them react appropriately to the danger. The inspiration behind expecting this type of behaviour is that similar mechanisms have been observed among animals in nature. For instance, Seyfarth et al. (1980) described experiments performed with the alarm calls of vervet monkeys. They showed that the vervet monkeys not only used

alarm calls to signal each other about nearby threats and reacted to the signals, but that the alarm calls also carried information classifying the threat.

Stay on button behaviour One environment in Evsim includes a button that one or more agent must stand on to open a door. When the door is open, the other agents can walk through it and reach a, goal. An obvious behaviour that might occur is that agents try to stay on the button and thus holding the door open for other agents. This behaviour could be a good tactic to get agents to the goal.

5.2. Test Plan

The test plan—shown in List 5.1—was partitioned into multiple steps forming a test procedure. The purpose of the test plan was to have detailed step-by-step instructions guiding the procedure of running the experiments and analysing the results. It is reproduced here to show how the data used to discuss and draw conclusions about the experiments' performances were obtained. In addition, it can be used as a guide for other researchers seeking to reproduce the experiments.

5.2.1. Test Procedure

The testing started—Step 1 in List 5.1—with choosing an environment and a set of configuration parameters to test. The different environments are described later in this chapter. The environment was simulated with training for a predefined number of generations, while data was collected for each generation.

Step 2 checked whether cooperative behaviour had occurred during the training run. If cooperative behaviour was observed, three questions were answered. The first question asked what kind of behaviour had emerged. The environments were designed with preconceived ideas of behaviours that could emerge in them. During testing especially these behaviours were looked for, but other cooperative behaviours that were found were also investigated. Answering the last two questions put the behaviour in context of the two types of cooperative behaviour introduced in the research question one.

Step 3 inspected the behaviour and its evolution in the context of the system as a creative process, as described in Section 1.3. It consisted of four sub-steps. The first step assessed the behaviour's usefulness. To evaluate whether the behaviour was useful or not, individuals exhibiting the behaviour were compared to individuals that did not have the behaviour. If the individuals with the behaviour were found to be consistently performing better than those without, then that can be taken as an indication of the behaviour being useful to the agents' performance. The process' ability to keep the useful behaviour and refining it over time was tested in the two following steps. The generations where the behaviour emerged and—if applicable—what generation it vanished were found. The last step was used to evaluate how novel the behaviour is in

1. Choose an experiment and let the algorithm run for a predefined number of generations.
2. See if cooperative behaviour has occurred.
   a) What kind of behaviour has emerged?
   b) Is it beneficial to all or none?
   c) Is it beneficial to some?
3. See if the behaviour evolves through a creative process.
   a) Is the behaviour useful?
      - Look at previous generations to see if:
        i. The individuals with good fitness have the behaviour
        ii. The individuals with bad fitness have the behaviour
      - If the individuals that have the behaviour are consistently better than the generations without the behaviour, then the behaviour is useful.
   b) Can the algorithm keep the behaviour?
      - See if the behaviour persists in other generations of the run.
   c) Can the process refine the behavioural patterns over time?
      - Check if the behaviour has improved since the first time it appeared. Improvement means that the fitness increases due to refinement of the behaviour.
   d) Is it novel in the context of the system?
      - Look at how often the behaviour occurs in multiple runs.

List 5.1.: Test procedure.

the context of the system, and looked at how often the behaviour occurred in multiple runs.

5.3. Test Phases

This section explains the preliminary test phase and the three experimental phases in this project. The preliminary testing was done to find potential parameter values for Evsim and parameters for MultiNEAT which could be interesting to investigate further in later experiments. Each of the three following test phases each had one set of experiments. The first test phase was done to see whether NEAT could creatively evolve cooperative behaviour and to find better configuration parameters for any eventual further testing. The second phase was used to experiment further, based on the results from the first experiment, on environments if needed. The third phase was used to investigate how different NEAT parameters could affect the algorithm's ability to creatively evolve cooperative behaviours. The experiments in the three main test phases are referred to as the first, second and third set of experiments for the rest of the thesis.

Parameter	Description	Default
Population size	Amount of individuals in a NEAT species	N/A
Dynamic compatibility	Whether the algorithm should attempt to keep the amount of NEAT species within the min. and max. species range	True
Min. number of species	Minimum amount of species for dynamic compatibility	3
Max. number of species	Maximum amount of species for dynamic compatibility	20
Crossover rate	Chance of an offspring being the result of a crossover	0.7
Overall mutation rate	Chance of mutating offspring if it was not a result of crossover	0.25
Elite fraction	Fraction of the best individuals to be directly copied over to the new generation	0.01
Old age penalty	Factor multiplied into fitness of agents in old species	0.5
Compatibility threshold	Threshold in the compatibility calculation during speciation	5.0

Table 5.1.: Interesting NEAT parameters.

5.3.1. Preliminary Testing

Table 5.1 shows the different MultiNEAT variables that were found to be the most interesting after the preliminary testing. The table includes MultiNEAT's standard values which were the ones used in all of the experiments unless other values are explicitly mentioned. Table 5.2 shows the different Evsim variables and the values expected to work best as a base when running the experiments. If nothing else is stated explicitly, these values were used.

Additionally, an estimate of how many generations the environments needed to train for before stagnating was made, and was initially set to 500. After a number of generations, no significant improvements were made to the fitness of the agents. Additionally, training for very many generations caused the brains to grow very large and complex without any noticeable corresponding change in the agents' performance. This is undesirable because as the brain becomes bigger it becomes harder for NEAT to perform optimisation on the increasingly complex topology. Larger brains also cause a bigger computational complexity for each iteration of the algorithm, increasing the time it takes to train one generation.

5.3.2. First Set of Experiments

The purpose of the first set of experiments—consisting of the experiments defined in Table 5.3—was to see if NEAT could creatively evolve cooperative behaviour. In addi-

Parameter	Description	Default
Training method	Per-species individual or shared fitness	N/A
Vision length	Length of the vision field	120
Vision field of view	Field of view of the vision in degrees	45
Simulation size	Amount of agents to be simulated for the species when using shared fitness	N/A
Yell radius	Radius of the yell circle	30

Table 5.2.: Evsim parameters.

Environment	Species 1	Fitness 1	Population size	Simulation size	Species 2	Population size	Number of experiments
Food	Herbivore	Individual	25,50,75				3
	Herbivore	Shared	10,50,100	2,10,25			9
Food chain	Herbivore	Individual	25,50,75		Predator	25,50,75	9
	Herbivore	Shared	10,50[a],100[b]	25,50,75	Predator	50	9
	Predator	Shared	10,50[c],100[d]	10,25,50	Herbivore	50	9
Evasion	Prey	Individual	25,50,75		Predator	25,50,75	9
	Prey	Shared	10,50,100[e]	25,50,75	Predator	50	9
	Predator	Shared	10,50,100	10,25,50	Prey	50	9
Door	Agents	Individual	25,50,75				3
	Agents	Shared	10,50,100	2,10,25			9
Total							78

Table 5.3.: Configurations in the first set of experiments. Species 2 is always trained with individual fitness[1].

tion, it was run to find better configurations for further testing if needed. Experiments were run with different population sizes and simulation sizes in each environment to find the optimal values in each case. A low population and simulation size is useful, as it reduces the computation time required to finish the experiments. An intended outcome of running the set of experiments was also to find how many generations were needed in the different situations to train the agents. In the first set of experiments, the system was configured to run for 500 generations in each of the experiments—as suggested in the preliminary testing. The first set of experiments was run two times.

5.3.3. Second Set of Experiments

A second set of experiments was initiated in the different environments in which the first experiments gave promising results. Here some environments and configurations were changed based on the gathered knowledge from the first set of experiments.

[1]Experiments that needed over 100 hours of computation time were cancelled. In (a) the run with simulation size 75 was cancelled. In (b) the runs with simulation sizes 50 and 75 were cancelled. All the runs in (c) and (d) were cancelled. In (e) the run with simulation size 75 was cancelled.

Parameter	Values tested
Population size	5, 10, 15, 20
Dynamic compatibility	False
Min number of species	2, 20
Max number of species	3, 50
Crossover rate	0.0, 1.0
Overall mutation rate	0, 0.05, 0.5
Elite fraction	0.5
Old age penalty	0.0, 1.0
Compatibility threshold	1.0, 10.0

Table 5.4.: NEAT parameters and the values tested.

5.3.4. Third Set of Experiments

Due to time constraints, experiments to test different NEAT parameters were only run on one environment. The Door environment—described later in this chapter—was chosen, as it was the environment that showed the most interesting results from the first set of experiments. In addition, we believed that it would be easier to analyse the performance of the agents in this environment, than in the other environments.

Table 5.4 lists the different parameters and values that were tested—in addition to the default values lists in Table 5.1.

5.4. Food Environment

The food environment—shown in Figure 5.1—consists of one species with a population of agents—hereby called herbivores. In addition to the herbivores, there is also food scattered around in the simulated world. The goal of the herbivores is to consume as much food as possible during a simulation step. When a food element is consumed, it reappears at a random position inside the simulation area. The herbivores have the minimal set of inputs and actuators as explained in Section 4.3. The herbivores can see other agents in their species and they can see food pieces.

When the herbivores are trained with individual fitness, they receive one point for each piece of food they consume in a step. When the steps in a generation are finished, the agents' fitness scores are calculated as the average of the scores they received in the steps. When trained with shared fitness, the fitness of each NEAT individual is the total number of food pieces that are collected by its simulated agents in the step divided by how many simulated agents there were.

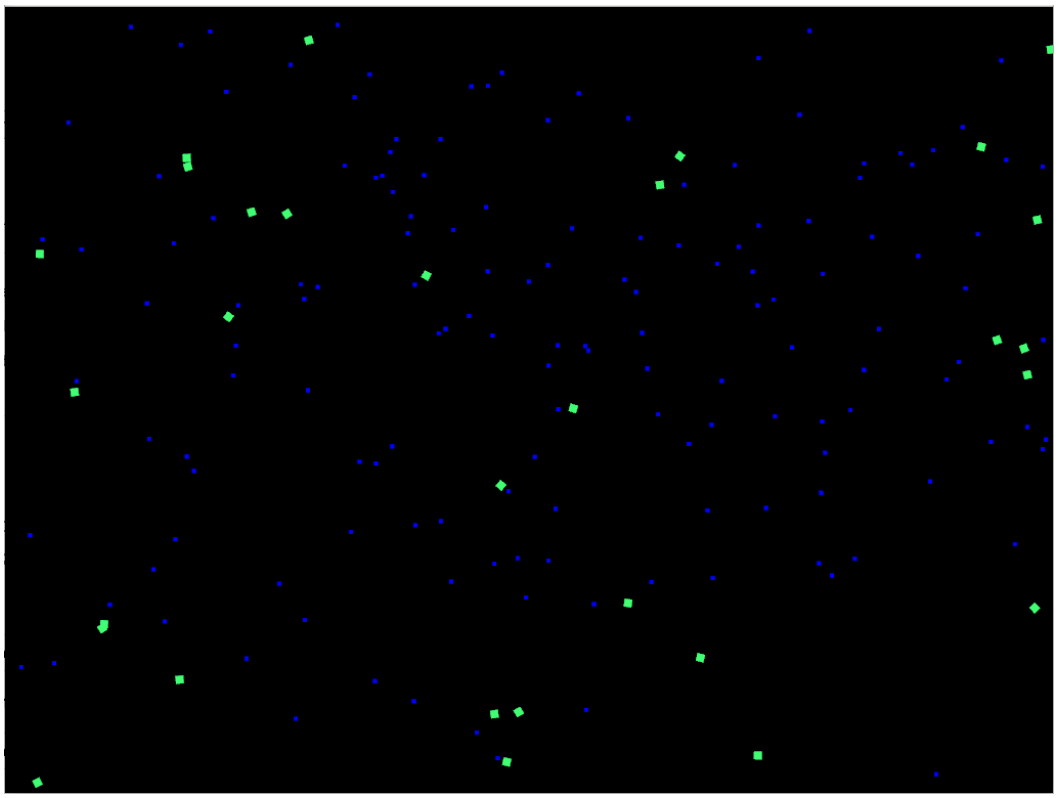


Figure 5.1.: Screen capture of the food environment. The green squares are herbivores while the blue squares are food.

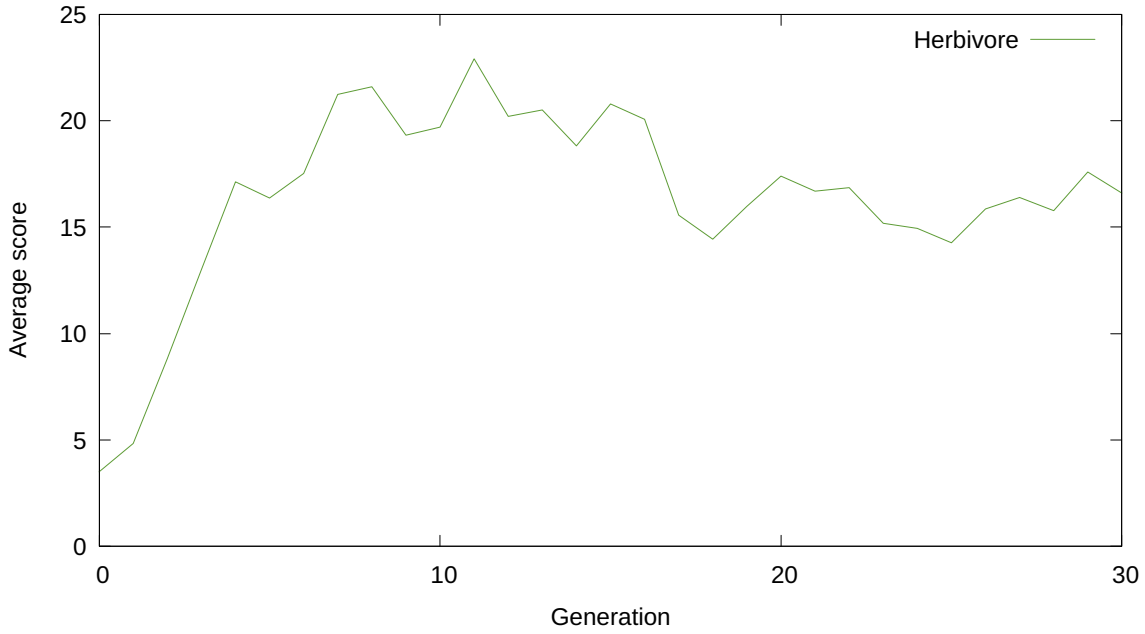


Figure 5.2.: Average score of the herbivores in the Food environment.

5.4.1. Results of the Food Environment

In the Food environment the herbivores learnt behaviours that enabled them to collect food. This was accomplished with both shared and individual fitness experiments. Figure 5.2 is an example of how the average fitness of the agents changed over the first 30 generations in an experiment with individual fitness. The agents moved around in the simulated area while scanning for food. When agents detected a piece of food, the agent turned towards the food and consumed it. There were no signs of cooperative behaviour in this environment.

5.5. Food Chain Environment

The Food Chain environment—shown in Figure 5.3—consists of two species, herbivores and predators. Each of the species has its own NEAT population and its own goals. The goal of the herbivores is to eat food—like the ones in the Food environment—and to avoid being eaten by the predators. The goal of the predators is to catch and consume as many herbivores as possible. When a predator has caught a prey,

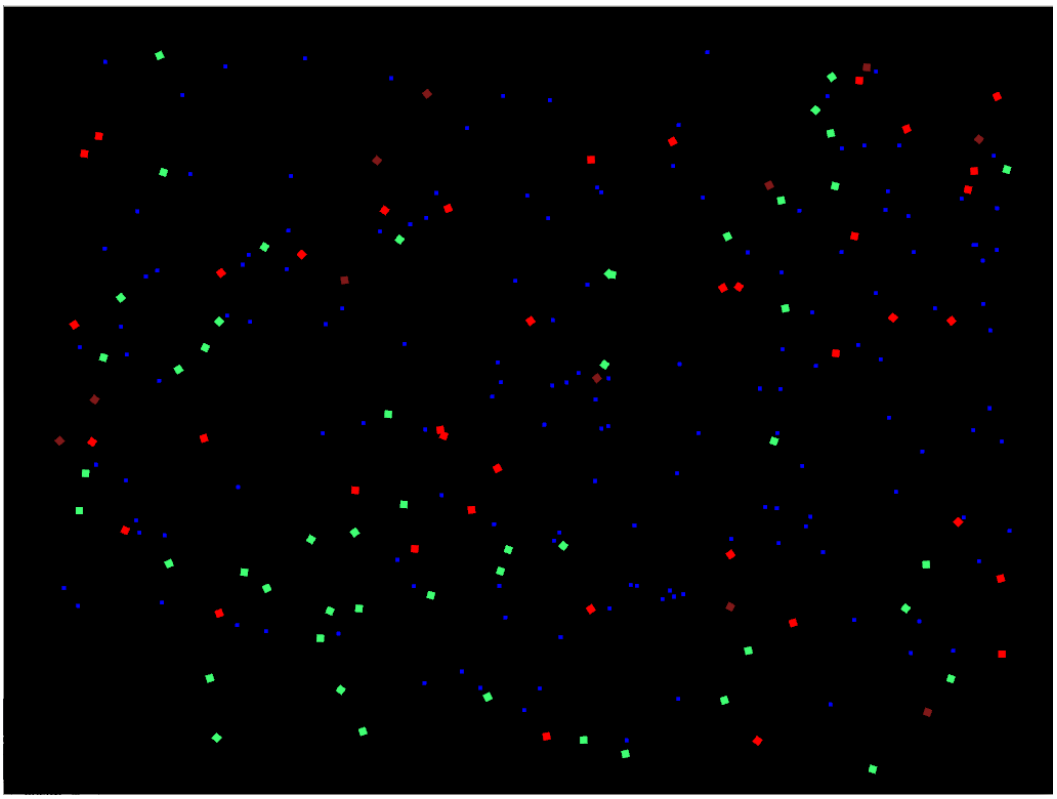


Figure 5.3.: Screen capture of the Food Chain environment. The green squares are herbivores, the blue squares are food while the red squares are predators. The squares with a dark red colour are predators that have just eaten a prey and are temporary disabled.

it is disabled for a short amount of time while it consumes the prey. The herbivores and predators have the default sensors and actuators, except that the herbivores have gained the ability to emit and detect yell signals. Both the herbivores and predators can see both other herbivores and predators.

When the herbivores are trained with individual fitness, they receive one point for each piece of food they consume and for every time they are consumed by a predator they are penalised by removing two points from their score. At the end of a generation, their fitness is calculated as the average score for all the steps. When trained with shared fitness, the fitness of each NEAT individual is the total number of food pieces collected by its simulated agents in the step, minus two points for individual consumed by predators and divided by the number of simulated agents of the species.

When the predators are trained with individual fitness, they receive one point for each herbivore they consume in a step. The agents' fitnesses are then calculated as the average of the scores they received in the different steps. When they are trained with shared fitness, the fitness of each NEAT individual is the total number of herbivores that are consumed by its simulated agents in the step, divided by the number of simulated agents.

5.5.1. Results of the Food Chain Environment

As in the Food environment, the herbivores in the Food Chain environment learnt behaviours to collect food. The predators learnt behaviours to chase and consume herbivores they detected. Figure 5.4 shows the average score of herbivores over the first 30 generations of an experiment with individual fitness. Figure 5.5 shows the average score for the predators in the same run. Figure 5.6 is screen capture of a herbivore moving towards a piece of food while being chased by a predator. Additionally, the herbivores learnt to avoid predators when they detected them. The herbivores and predators learnt these behaviours in experiments with both individual and shared fitness. The results of the first set of tests showed no indication of the agents using the yell actuators in either of the training methods. There were no noticeable patterns in how the yell actuator was used. Agents would often just yell constantly and as rapidly as possible. As in the Food environment, the results from the first set of experiments did not show any signs of other cooperative behaviours either.

In an attempt to make the agents learn to use the yell mechanic, a second set of experiments was conducted. The simulation area was very crowded in the first set of experiments, especially in the configurations with many agents simulated at once. To let the agents breathe a little, the size of the simulation area was increased by a factor of 5 in both dimensions for the second set of experiments. To compensate for the increased environment, the yell signal radii were also increased. To eliminate the food collection as possible interference in the agents learning to use the yell mechanic, the second set of experiments also tested configurations without food. In other words, a number of combinations of different yell signal radii and availability of food were tested

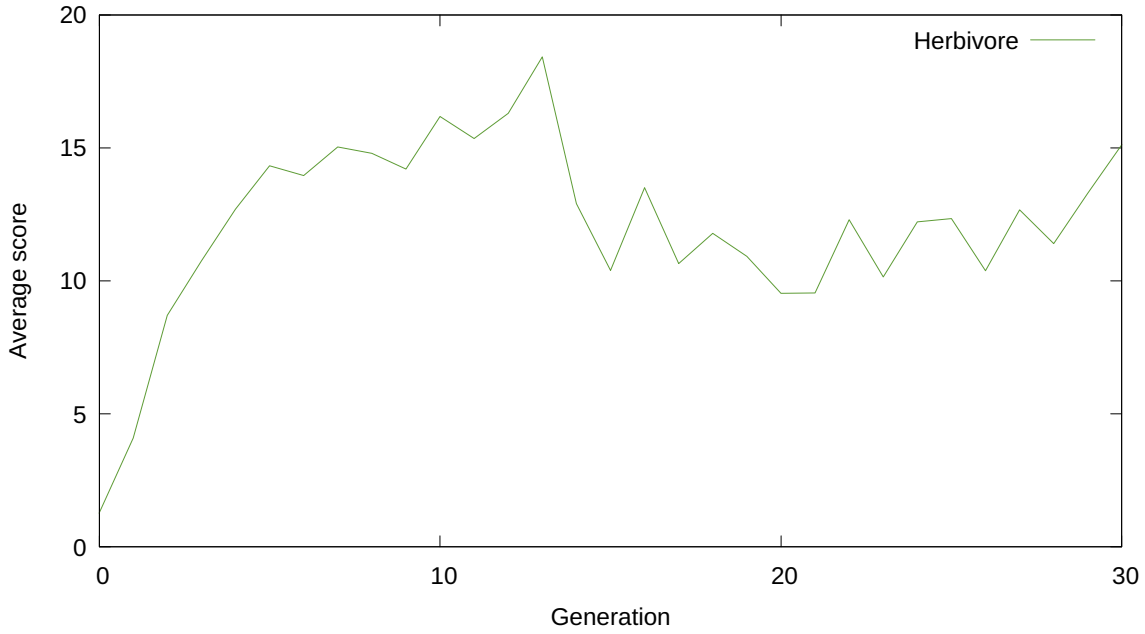


Figure 5.4.: Average score of the herbivores in the Food chain environment.

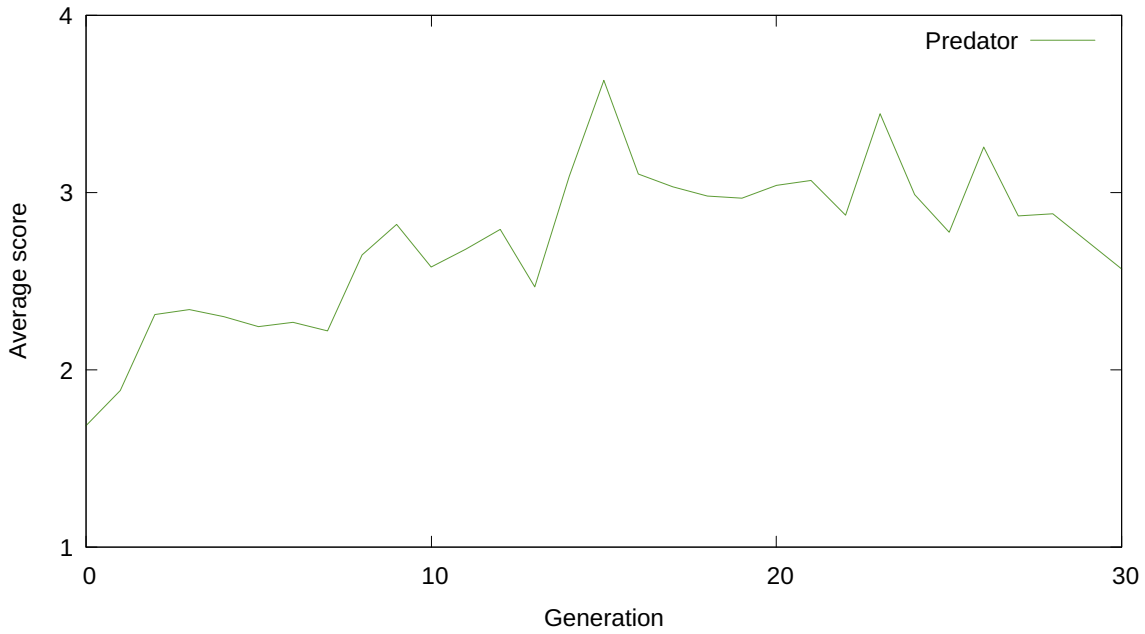


Figure 5.5.: Average score of the predators in the Food chain environment.

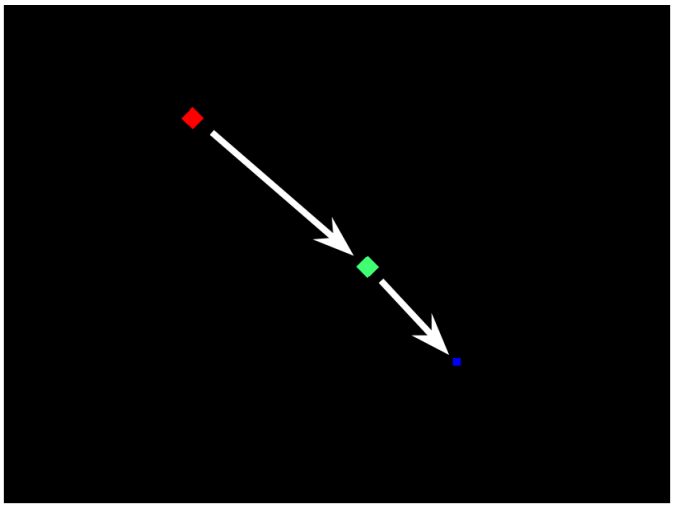


Figure 5.6.: Example of the herbivore and predator dynamic. The light blue herbivore is chasing the blue food piece while the red predator is chasing the herbivore.

in an enlarged environment. Despite these adjustments, no cooperative behaviours were identified in the second set of experiments neither.

5.6. Evasion Environment

In the Evasion environment there are two species. The first species—the prey—start on the left side of the simulated area and need to get to the goal line on the right side in order to be rewarded and despawned to safety. Predators start on the right side of the simulation area and are rewarded for consuming the prey. The prey also receive a small penalty if eaten to further disincentivise their capture. In addition to the default inputs and actuators, the prey and the predators have their current positions as inputs. Both the prey and predators can see other prey and predators. A screen capture of the Evasion environment can be seen in Figure 5.7.

When the prey are trained with individual fitness, they receive 100 points if they reach the goal and two points are removed if they are consumed by a predator. Their fitnesses are then calculated as the average score of the steps they were simulated in. When they are trained with shared fitness, the fitness of each NEAT individual is the total number of its simulated agents that reached the goal times 100, minus two points for each clone which was consumed by the predators, and divided by the number of simulated agents. When the predators are trained with individual fitness, they receive one point for each prey they consume in a step. Their fitnesses are their average scores for the steps. When they are trained with shared fitness, the fitness of each NEAT individual is the total number of prey that are consumed by its simulated agents, divided by the number of simulated agents.

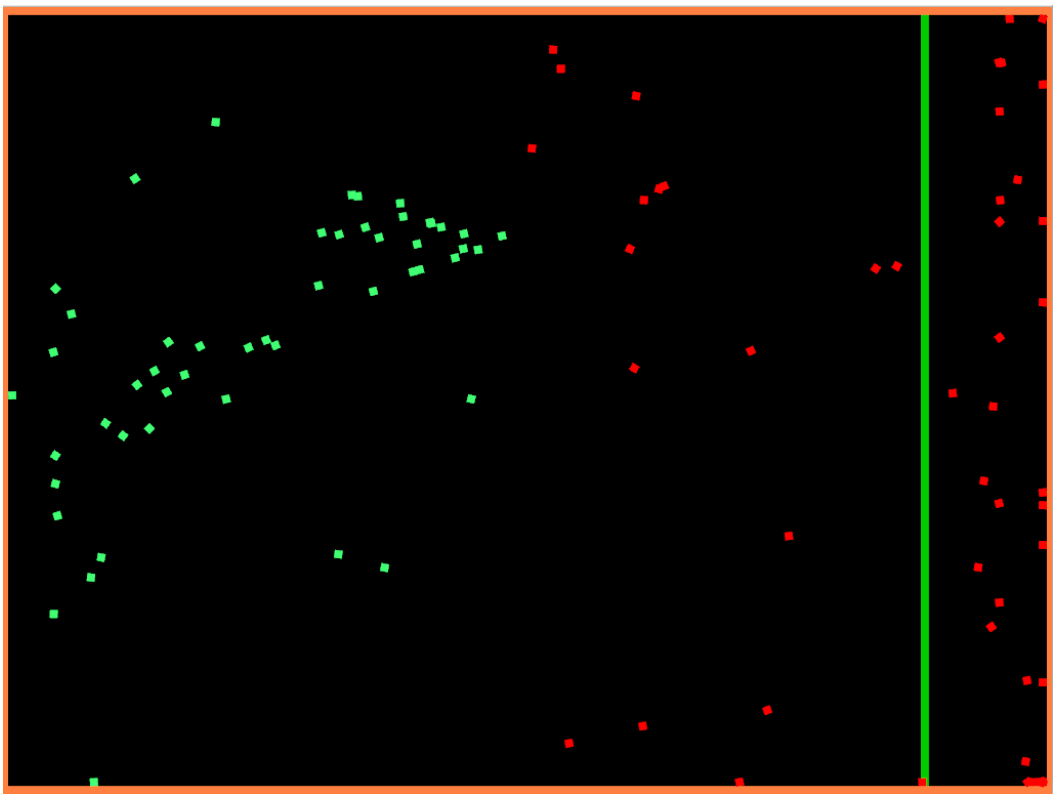


Figure 5.7.: Screen capture of the Evasion environment. The green agents are prey while the red agents are predators. The green line is the goal the prey are trying to reach.

5.6.1. Individual Fitness Results

This section presents the results found in the Evasion environment when the prey and predators were trained with individual fitness.

Prey

In most of the experiments in the first set of experiments behaviours were found where the prey moved together from the left to the right side of the board along one or both of the walls. Figure 5.8 shows an example of this type of behaviour. To test whether the prey were following each other or merely taking similar paths whilst guided by the wall, multiple generations of differing scores in a well performing experiment were simulated both normally and with the sensory inputs of the prey used to see other prey cleared to zero—effectively making prey invisible to other prey. These simulations were run for 1000 steps to record average performance values. Figure 5.9 shows the average score from the different generations in the experiment which was used for this testing. In generations where the score was low—for example in generations 2 and 212—the average scores were not significantly affected by removing the ability of prey to see each other.

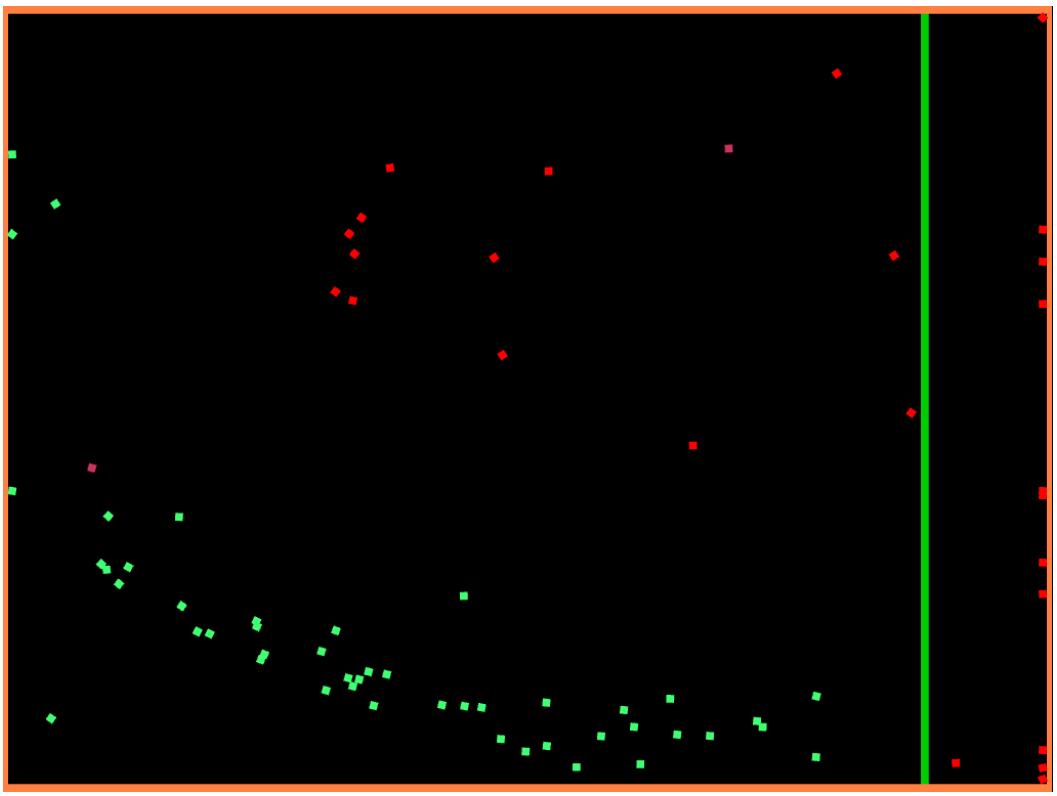


Figure 5.8.: Screen capture of a behaviour that emerged from an experiment with individual fitness on the Evasion environment. The prey followed each other and the wall at the bottom to get to the goal.

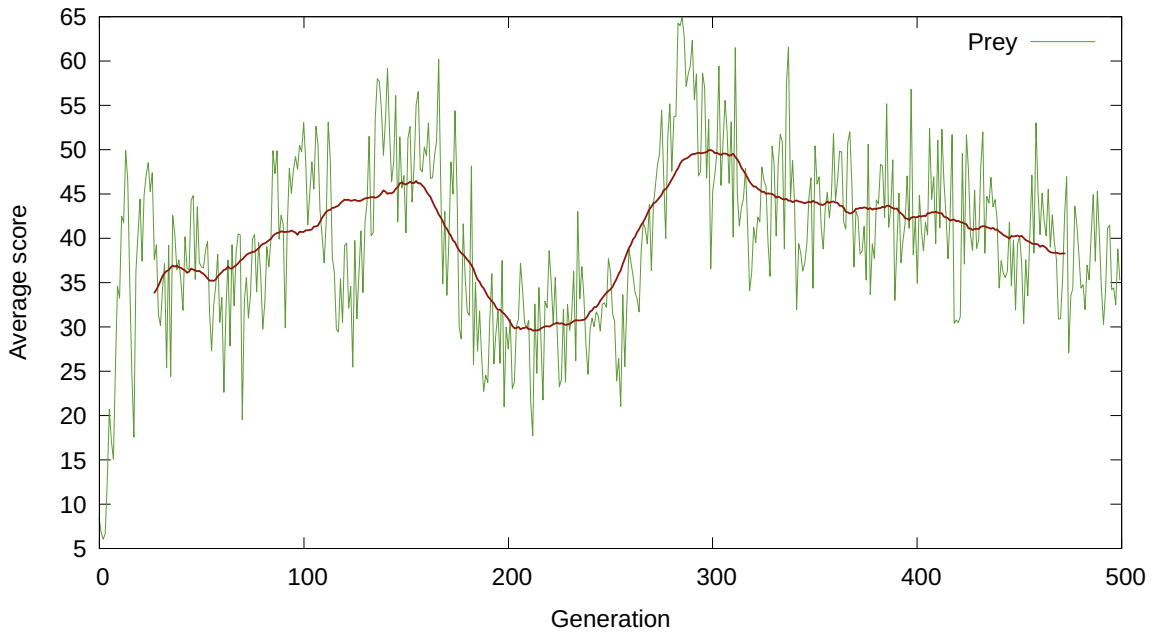


Figure 5.9.: Average score for prey in the Evasion environment with population size 50. The predators have a population size of 25. The red line represents a centred moving average with window size 55.

In the generations with high average, there were big differences. For example, simulating Generation 285, the prey with normal sight had an average score of around 65 and the prey with changed vision had an average score of around 39. By visually inspecting the two simulations it was also clear that prey who normally went with the other prey towards the right side ceased to do so if their vision of other prey was removed.

Predators

In the experiments examined from the first test set, behaviours were found where the predators followed and consumed the prey. No obvious cooperative behaviours were found by visual inspection. To see whether the predators' vision of other predators was beneficial to them, similar testing as was done for the prey was performed. One of the well performing experiments was closely examined. The testing was done on multiple generations of differing scores. The prey were evaluated with both normal vision and where their vision of other predators were removed. The predators with good fitness performed a bit worse when they were unable to see other predators. In one of the experiments, the performance of the predators was about 22 percent lower when their ability to see other predators was removed than with normal vision. No clear pattern was found in the simulations to explain this difference.

5.6.2. Shared Fitness Results

This section presents the results found in the Evasion environment when either the prey or the predators were trained with shared fitness.

Prey

When the prey was trained with shared fitness they evolved similar behaviours as when they were trained with individual fitness. Similar testing was done and the sensory inputs of the prey that was used to see other prey was cleared. One well performing experiment in the first test set was examined closely. In the generations with good fitness the prey usually performed significantly worse when they were unable to see other prey. Among others, the behaviour of the best individual of the best generation in the experiment was lost when the prey lost their ability to see other prey. The prey stood still for the most part and none of them could get to the goal. Another test was then run to see if the agents from that individual used the wall to guide themselves at all. In this test, the prey could see other prey, but they could not see the wall. Here multiple agents went to the goal, but a bit fewer than when the prey could see the wall.

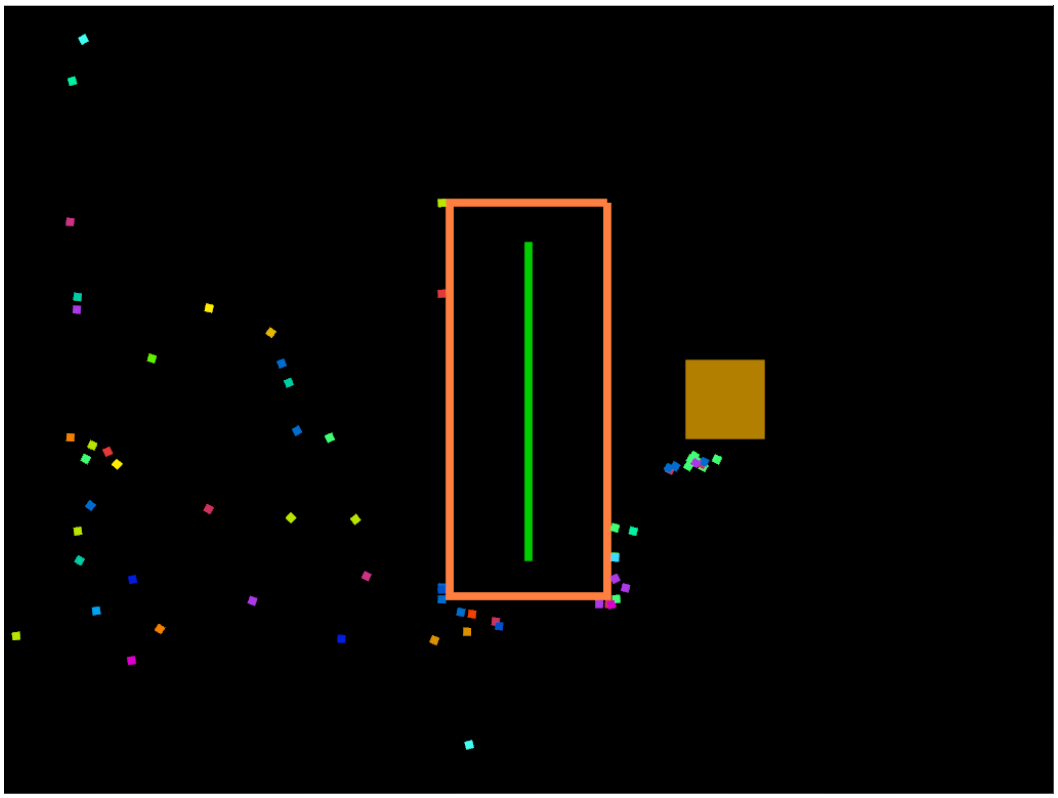


Figure 5.10.: Screen capture of the Door environment. The green line is the goal, the orange tiles are the door, and the brown square is the button.

Predator

Similar behaviours were found when the predators were trained with shared fitness as when they were trained with individual fitness. They were evaluated in a similar matter and the results showed that the fitness of well performing predators was reduced when their ability to see other predators was removed. In one experiment, the average score of the best individual was reduced by 40 percent. No clear pattern was found in the simulations which could explain this difference.

5.7. Door Environment

The Door environment consists of one species, a door, a goal and a button. An illustration of the environment is shown in Figure 5.10. The goal of the agents is to get to the goal enclosed by door tiles. The agents have to open the door and hold it open before they will be able to move to the goal. When the button is kept pressed, the door will disappear and agents may enter the enclosed area. If all agents leave the button, the button is no longer pressed and the door will reappear. This makes it impossible for a lone agent to reach the goal. Some form of cooperation is therefore a necessity for any agent to achieve a score above zero in the environment. In addition

to the default inputs, the agents have their current angle as input, and an input which tells them whether they are on the button or not. The agents can see the door, the goal, the button and each other.

When the agents are trained with individual fitness, they receive 100 points if they reach the goal in a step and they get an additional score for staying on the button and help other agents move through the door and enter the goal. This score is calculated by giving each agent that is currently on the button points when other agents are entering the goal. An agent on the button receives a score equal to the number of agents that entered the goal in a tick times 100 divided by the number of agents that were on the button at the time. The agents' fitness are calculated based on an average of the scores they received in the different steps. When they are trained with shared fitness, the fitness of each NEAT individual is the total number of clones that entered the goal in the step. If for example the simulation size is set to ten, then the optimal score for one NEAT individual would be nine as a single agent would not be able to get through the door without having another agent staying on the button.

5.7.1. Individual Fitness Results

Throughout all the individual fitness experiments of the Door environment, most of the agents were trained to move in one direction around the door, following the sides of the door. Some agents touched the button on the way, removing the door and prompting other agents to enter into the goal. The touching of the button happened with various levels of intent. Some crossed the button on their natural path around the door, others steered through it once they saw it, and in some cases the agents would even attempt to stay on the button instead of going further along the side of the door. The cooperation observed has potential benefit to each of the interactants. The door opener may not receive any reward from the cooperation if an agent which it lets past the door enters the goal after the opener has left the button. Two behaviours from different experiments were examined more closely.

Behaviour A

This behaviour is illustrated in chronological order in the four subfigures of Figure 5.11. First, most of the agents went above the door and goal. Upon turning the right-hand corner, some of these agents went intently towards the button, while most of them went down between the button and the door. With the door opened by agents on the button, the agents that went between the button and the wall all entered the goal, followed by most of the agents who initially went to the button. One agent made an effort to stay on the button until the end of the round. Some stragglers who remained on the left side of the door also took advantage of the opened door to reach the goal. In the end, the only agents who did not reach the goal were the agent on the button and a small amount of stragglers who were not attracted to the goal.

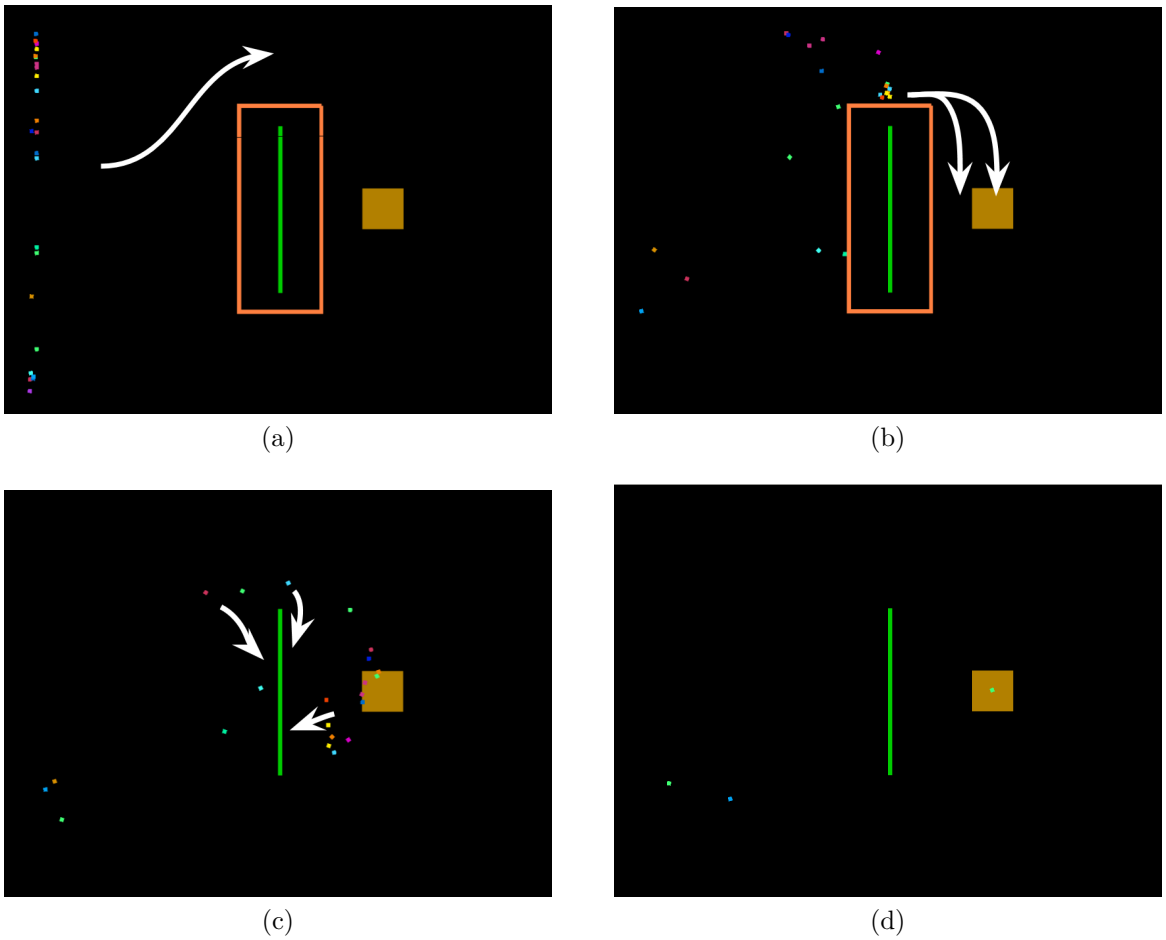
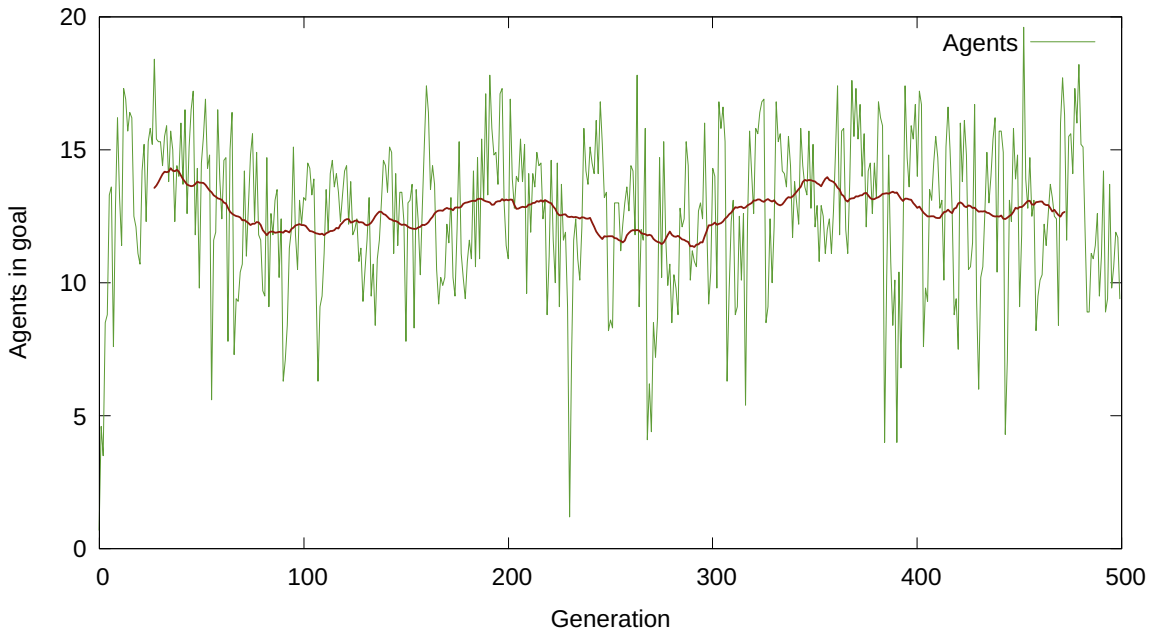


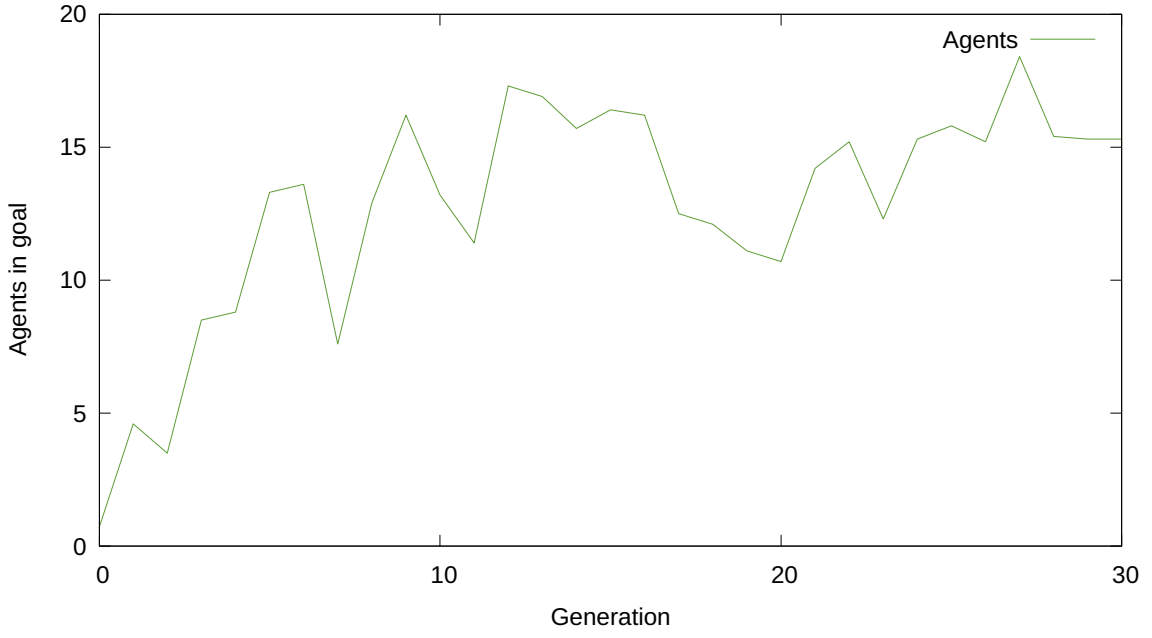
Figure 5.11.: Screen captures of a behaviour that emerged in an experiment with individual fitness on the Door environment. The agents managed to open the door and reach the goal.

The graphs in Figure 5.12 plot the average amount of agents which made it to the goal for each generation of this experiment. Looking at generations with high and low scores, we see that the overall behavioural pattern of the majority of the population going around the door was still there. One area in which they differed was in the amount of agents that attempted to reach the goal, and whether there were agents who attempted to keep the door open by staying on the button. In the underperforming generations, either too many agents stayed on the button or no one went to it at all. No generation in the experiment produced a collective behaviour which enabled the population to obtain optimal scores in all its evaluation steps.

From 5.12b we see that the agents reached the overall average performance of the experiment in about five generations of training. While the smoothed performance in 5.11a stayed relatively flat, local variations were jittery. Wider dips in score took amounts of generations similar to the initial training to bounce back to the average. Throughout the entire experiment, the behaviour in the well performing generations looked similar, despite the dips and subsequent climbs in performance.



(a) Generations 0-500



(b) Generations 0-30

Figure 5.12.: Count of agents that entered the goal in one experiment with population size 25 trained with individual fitness. The theoretical maximum is 24. The red line in (a) represents a centred moving average with window size 55.

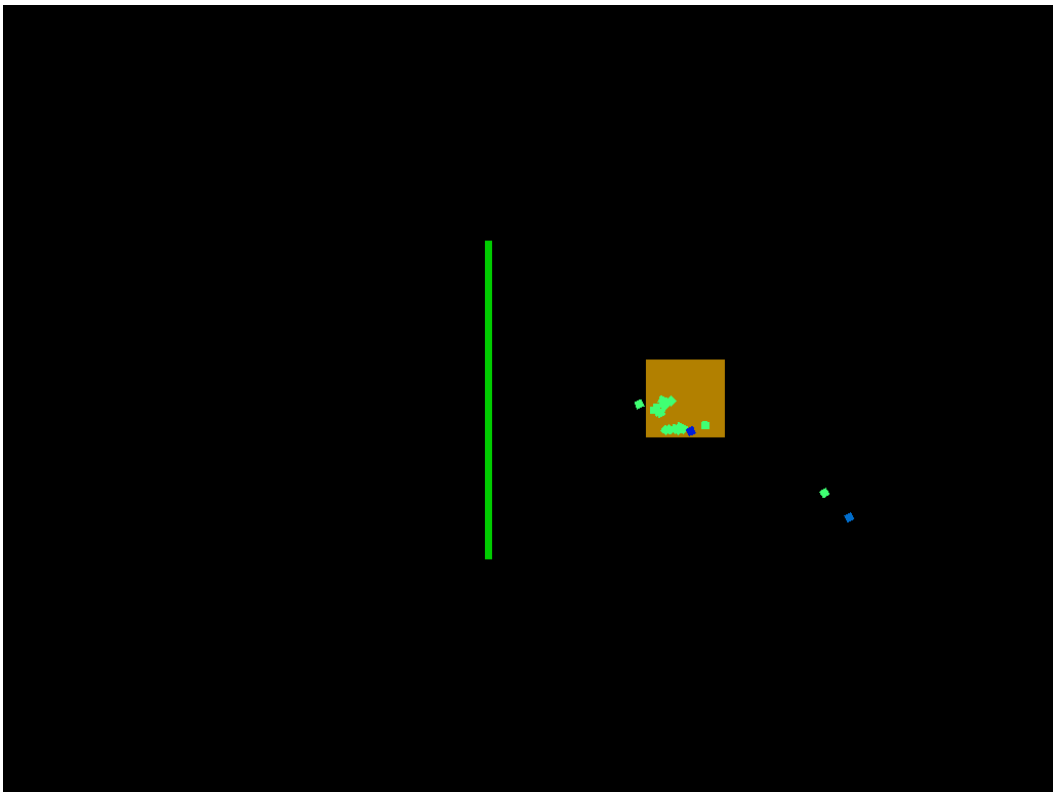


Figure 5.13.: Screen capture of a poorly performing population in the Door environment. The agent were trained with individual fitness with a population size 25. The population is from generation 230. Nearly the entire population attempted to stay on the button, leaving no one to enter the goal.

Generation 230 of the experiment was a particularly dramatic dip. The reason for its low performance can be seen in Figure 5.13. In the population of this generation, nearly all the individuals belonged to a NEAT species with agents of which all attempted to stay on the button. This left no agents that attempted to reach the goal.

Behaviour B

Figure 5.14 illustrates another behaviour which emerged in the Door environment. This behaviour emerged in an experiment with a higher population size than in Behaviour A. The behaviour consisted of the agents going around the door in spiral paths of which many crossed the button. Some agents had learned to intentionally turn towards and stay within the button by moving in a circle on top of it. The rest of the agents rushed through the button and further into the goal. In the figure, the green agents attempted to stay on the button, while the pink agents rushed through it in order to reach the goal.

As in the experiment described in Behaviour A, variations of the emerged behaviour in

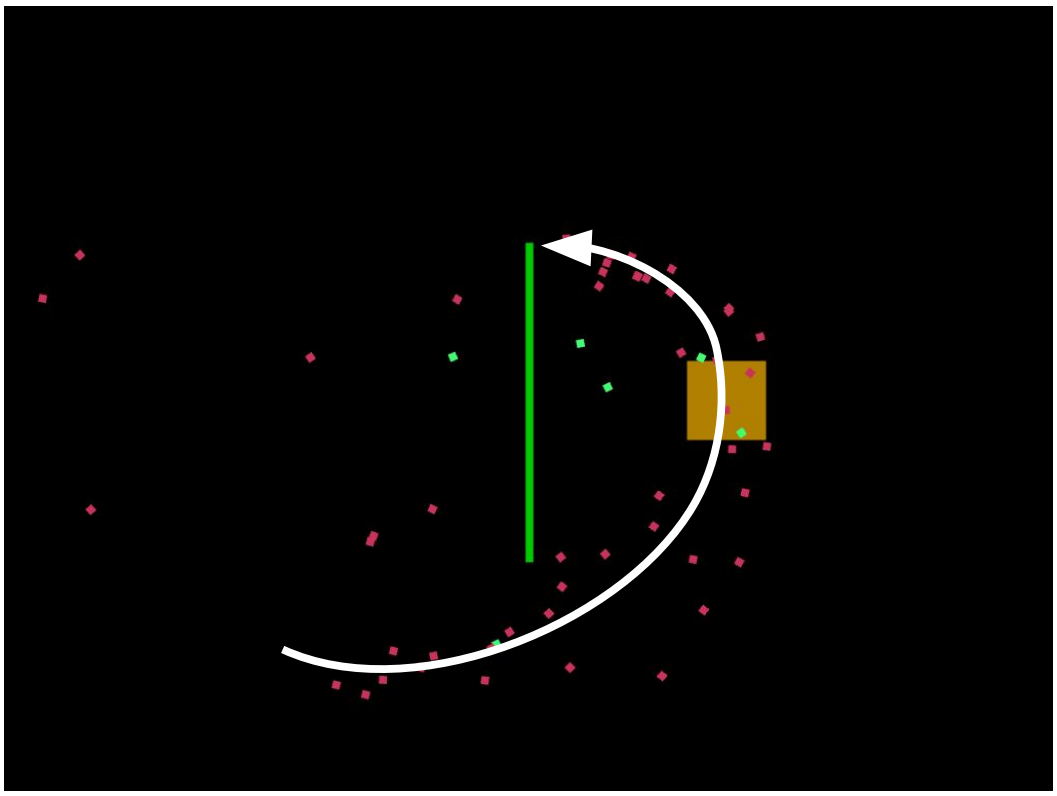


Figure 5.14.: Screen capture of another behaviour that emerged from experiments running with individual fitness on the Door environment. The agents opened the door and went to the goal.

this experiment were consistently employed by the populations of the well performing generations. The amount of agents that made it into the goal on average for each generation of the experiment is plotted in Figure 5.15. Generations with noticeably higher scores had a few of the agents that attempted to stay on the button and the rest of the agents all went into the goal. The dips in the average score were generations where too many agents either did not intently move towards the goal, or where no agents attempted to keep the door open. In the lowest dips, the populations consisted of unusually many NEAT species, where most of the agents showed no sign of behaving coherently.

5.7.2. Shared Fitness Results

As in the results from individual fitness, most of the useful behaviours obtained with shared fitness training are variations of behaviours where the agents traversed around the door, onto the button and kept the door open. Different behaviours emerged in the experiments where the agents were trained with shared fitness.

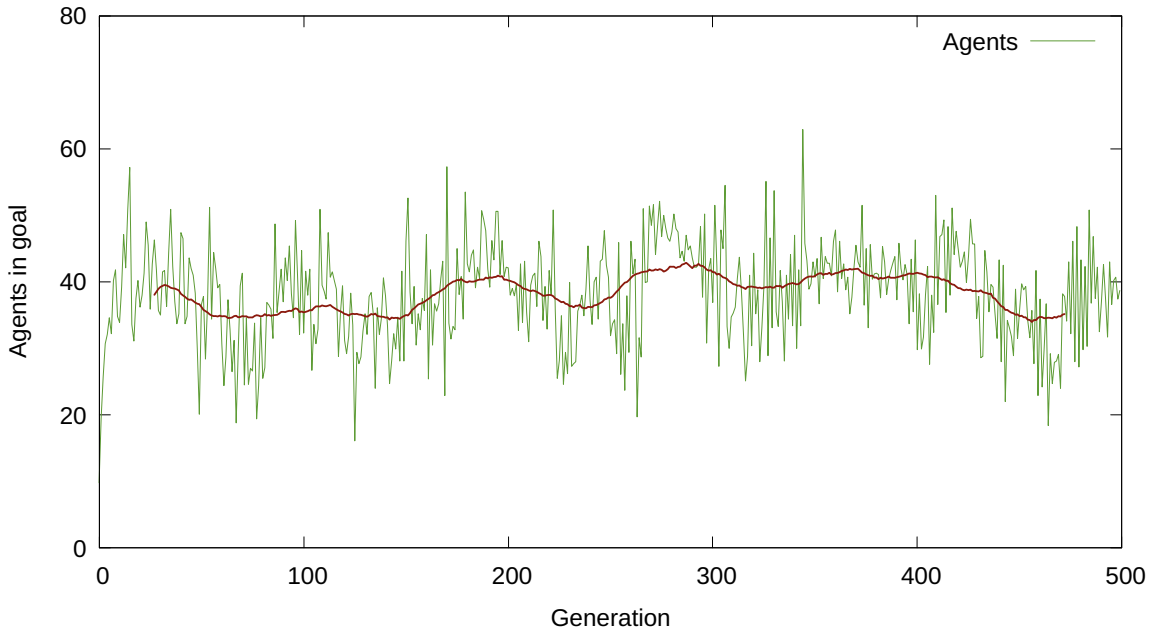


Figure 5.15.: How many agents which entered the goal in one experiment with population size 75 trained with individual fitness. The theoretical maximum is 74. The red line in represents a centred moving average with window size 55.

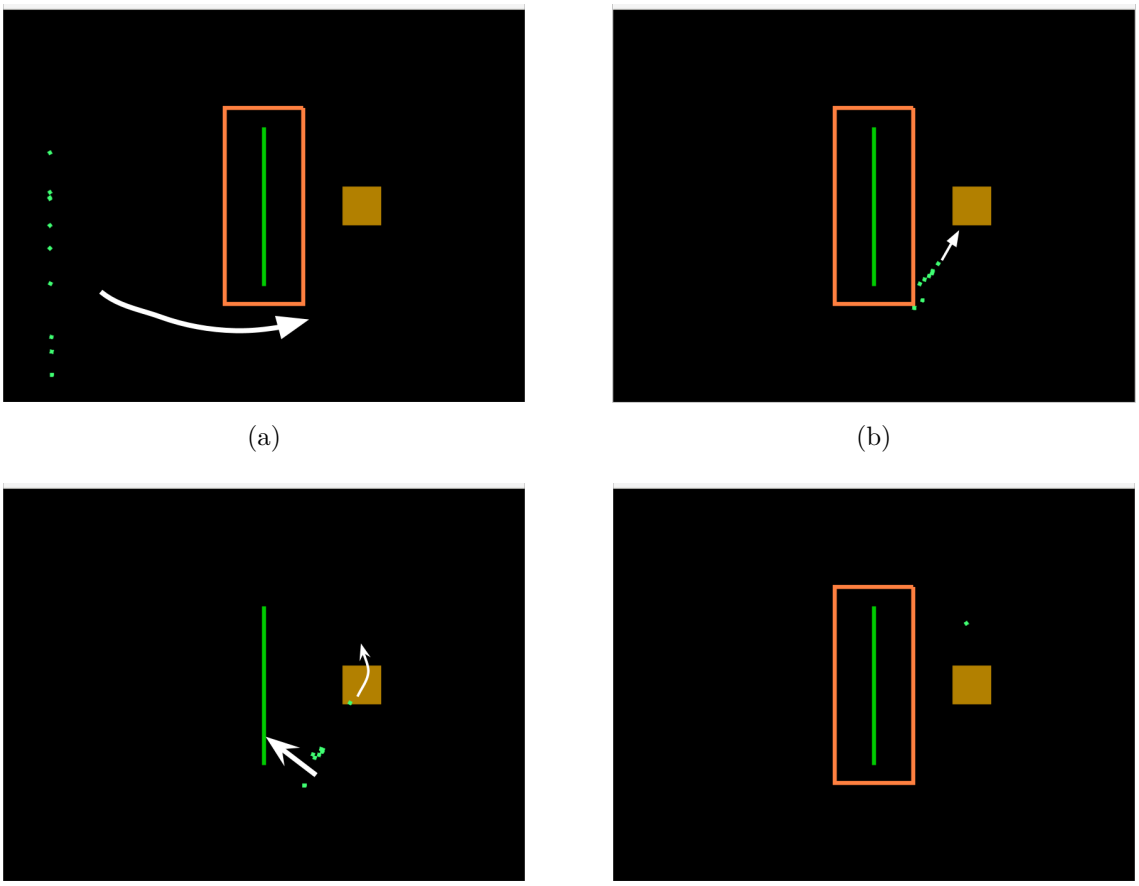
Behaviour A

In many of the experiments, variations of a behaviour occurred where the agents would in specific situations stop and wait due to other agents. Figure 5.16 shows an example of this behaviour where the waiting was used to the agents' benefit. In the example, the agents first moved as a group past the lower side of the door. Upon reaching the corner of the door, the agents' vision fields contained both the button and other agents. This combination of visible objects caused all the agents to halt, except for the leading agent which did not have any agents in its vision field. The leading agent then went directly for the button. When the leading agent reached the button—causing the door to open—the rest of the agents entered the goal.

Waiting behaviours appeared early in the runs, but then mostly in detrimental ways. Cases where agents completely halted when detecting some objects were seen in badly performing individuals throughout the entire experiments. Once beneficial waiting behaviours occurred in an experiment, they appeared in well performing individuals for the remainder of the experiment.

Behaviour B

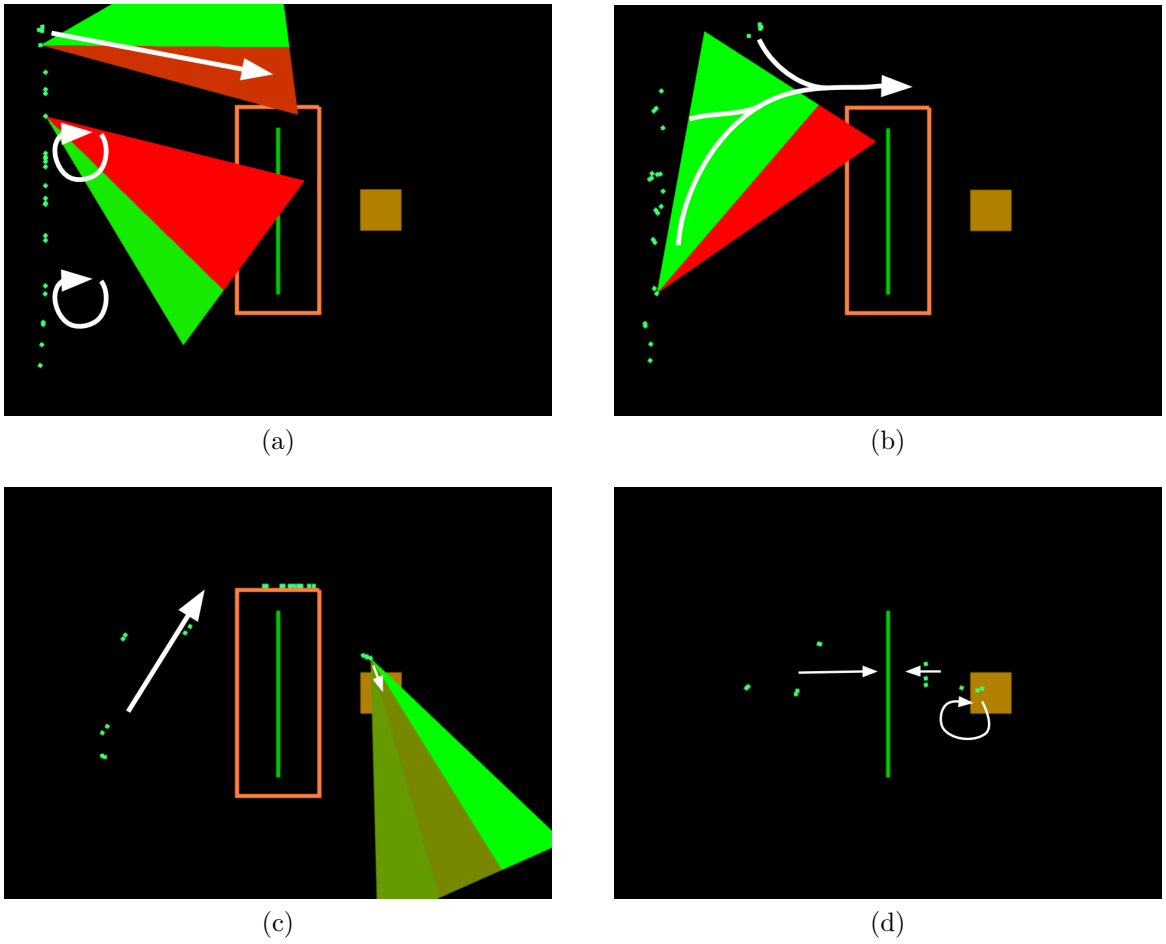
Another behaviour which emerged in an experiment with shared fitness is shown in Figure 5.17. In this behaviour, agents followed the door clockwise when they saw it



(a) (b)

(c) (d)

Figure 5.16.: Screen captures of the waiting behaviour in the door environment. The agents were trained with shared fitness. Between (a) and (b), all the agents moved to the lower right corner of the door and stopped. In (b), one of the agents moved to the button, while the others waited. Then in (c), the agent which went to the button moved through it, and all the other agents went into the goal. Afterwards—in (d)—only one agent was left in the environment. The agents finished the task optimally.



(a) (b) (c) (d)

Figure 5.17.: Screen capture of a behaviour in the Door environment, emerged during training with shared fitness. In (a), the topmost agents saw the door in the right side of their vision fields and started to move towards it. Meanwhile, the lower agents circled around due to seeing the door in the middle and left parts of their vision. In (b), more of the agents had the door in the right part of their vision from circling and they started moving along the wall as well. Eventually in (c), the first agents to move along the door saw and went towards the button. At this point, the last agents on the left side had also gotten close enough to start moving along the door. As the agents by the button went over the button, all the agents who could see the goal went towards it. In (d) the agents were either moving towards the goal, or circling between the door and the button. Eventually only a single agent remained by the button, as the rest had successfully managed to get to the goal.

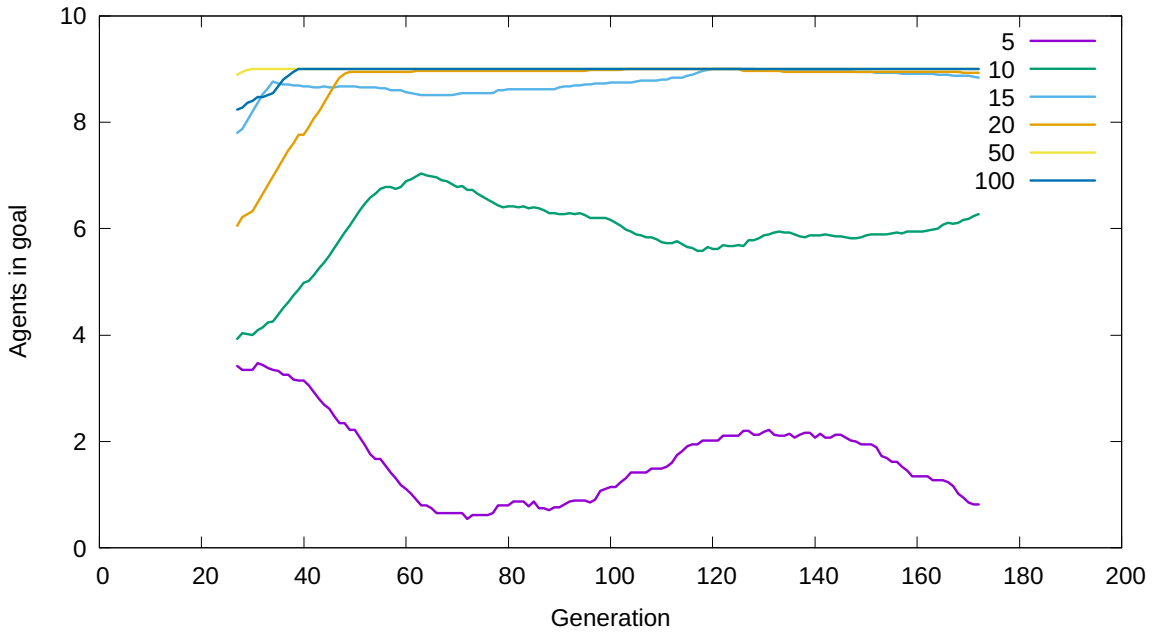


Figure 5.18.: Count of agents that entered the goal in the best individual in different experiments on the Door environment. The graph plots centred moving averages of the best individual of each generation from the experiments. The different lines represent runs with different population sizes. A window size of 55 is used.

on their rightmost vision fields. When they saw the button, they moved towards its centre and through it. If they either did not see anything at all or a part of the door was in their other vision fields, they instead moved in a tiny clockwise circular motion. This circular motion seemed to be very beneficial when multiple agents were on the button. As they left the button, they saw and moved towards the goal, but as the last one left the button the door reappeared. The agents then saw the door in all their vision fields, prompting them to turn right in the circular motion, returning them to the button. The rotating agents quickly became out of sync, letting some of them get stuck on or get through the door. In the end, only a single agent was left to circle around between the button and the wall. In the individuals where this behaviour was most refined, there were not a single episode where the simulated population failed to perform optimally.

5.8. NEAT Parameters

This section contains the results from testing different NEAT parameters on the Door environment. An overview of the parameters tested can be found in Table 5.4.

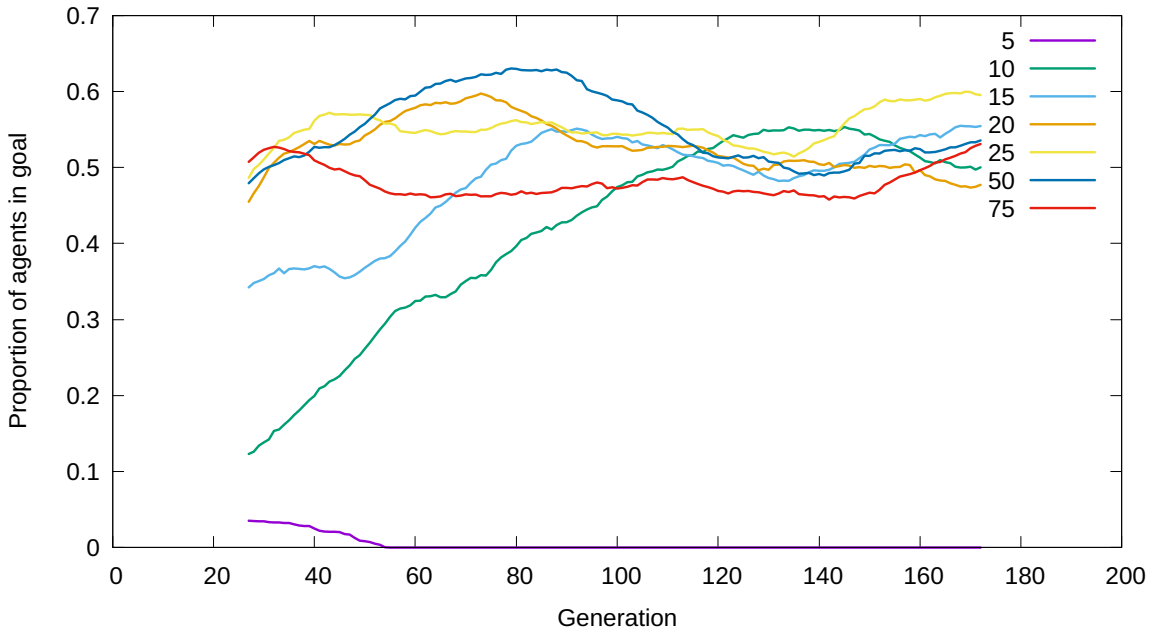
5.8.1. Population Size

Figure 5.18 shows experiments run on the Door environment testing different population sizes. The graph shows how many agents that reached the goal in the best individual in the generations from the different experiments. The agents were trained with shared fitness in all the experiments and the simulation size was set to ten. The optimal score for an NEAT individual was therefore nine. The results showed that in the experiments where the population size was high, the algorithm learnt an optimal behaviour fast and was able to keep the behaviour. With population sizes 50 and 100, the score became optimal around generation 40 and stayed optimal for the remaining generations. With lower population sizes the results were more unstable, and did not stabilise at an optimal score. With population sizes ten and five, the averaged best scores were considerably lower and never became optimal.

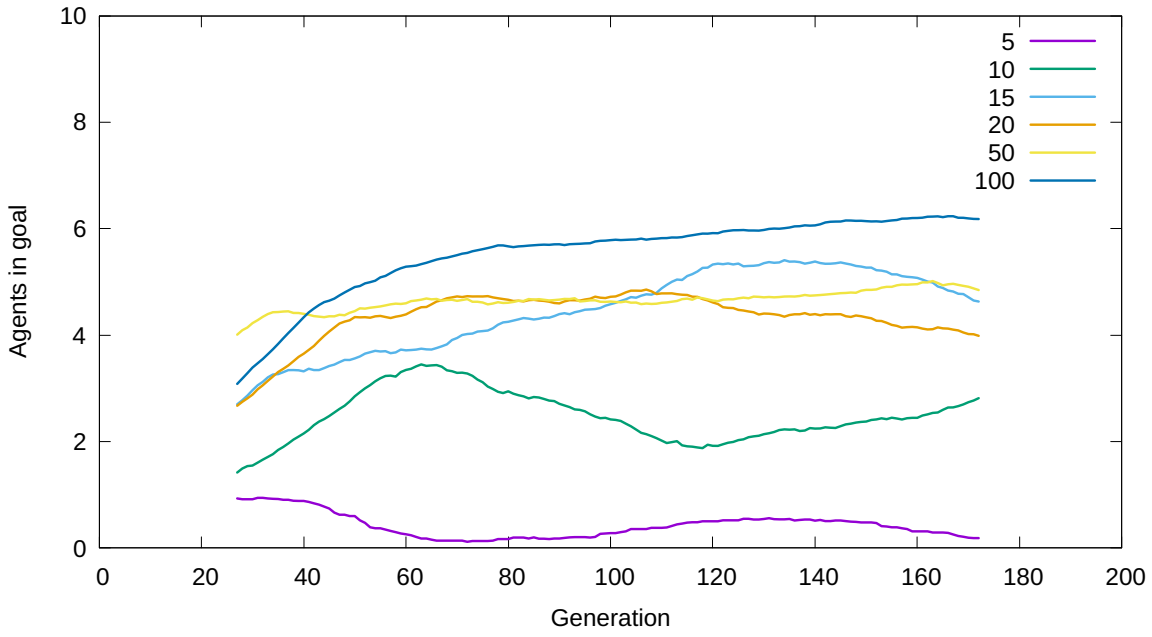
Figure 5.19 contains two graphs that shows the average of how many individuals that reached the goal in each generation of experiments with different population sizes. The graph in Subfigure 5.19a represents experiments where the agents were trained with individual fitness. Here the graph shows a centred moving average of the proportion of agents that reached the goal in the different generations. The graph in Subfigure 5.19b represents experiments where the agents were trained with shared fitness. Here the graph shows a centred moving average of the average number of individuals that reached the goal in the different generations. The shared fitness graph represents the same experiments that were used to produce Figure 5.18. In the experiment where the agents were trained with individual fitness and a population size of five, the proportion of agents that reached the goal was low from the start and dropped to zero after around generation 50. In the experiments with higher population sizes, the proportion averaged at around 0.45 to 0.6 at generation 172. The shared fitness graph shows that the average number of agents that reached the goal in experiment with population size 5 stays low the whole run and scored between 0 and 1. The experiment with a population size of 10 performed a bit better, and the average scores were between 1.6 and 3 throughout the run, and in generation 172 the score was around 2.7. The rest of the experiments with population sizes 15, 20, 50 and 100 the average scores were between 4 and 6 in generation 172.

5.8.2. Other Parameters

Figure 5.20 shows results based on experiments run on the Door environment with different NEAT parameters. The agents were trained with shared fitness. In these experiments, the population size was set to 50. In each of the experiments one NEAT parameter was changed while the rest of the parameters were set to default values. The graph shows the centred moving average of how many agents that reached the goal in the best individual in the generation in the different experiments. The results show that the score became almost optimal for the all the different experiments within generation 14. There was a minor reduction in score in the experiment where the minimum number of species was set to 2, and a minor reduction in the experiment



(a) Individual fitness.



(b) Shared fitness with simulation size 10.

Figure 5.19.: Average performance of agents that entered the goal in experiments testing different population sizes on the Door environment. The different lines represent experiments with different population sizes. The graphs have been smoothed with a centred moving average with a window size of 55.

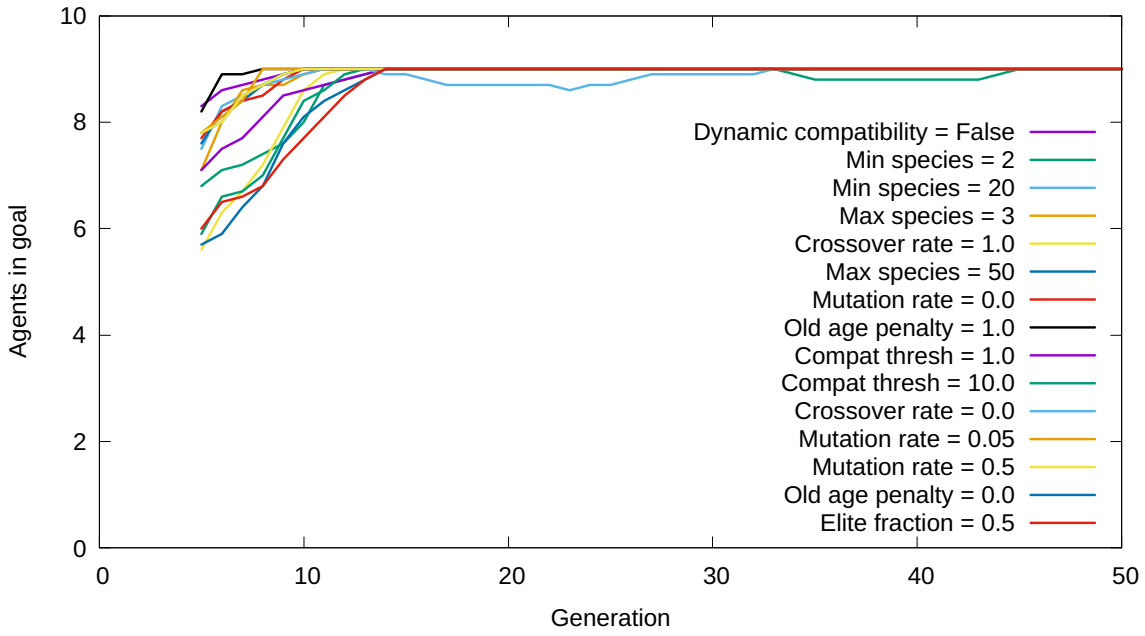


Figure 5.20.: Count of agents that entered the goal in the best individuals when testing different NEAT parameters on the Door environment. Agents are trained with shared fitness with simulation size 10. The graphs have been smoothed with a centred moving average with a window size of 55.

where the crossover rate was set to 0.0. From generation 45 and until the end of the runs—centred moving average values up to generation 195—all of the experiments had optimal scores in each generation.

6. Evaluation and Discussion

This chapter evaluates and discusses the experiments from the previous chapter. In the evaluation section, the results of each environment are evaluated. The discussion section draws on the evaluations to address the two research questions of this thesis. At the end of the chapter, there is a section about limitations found during this project.

6.1. Evaluation

Experiments were run on four environments. The results of the experiments were gathered according to the test plan described in Chapter 5. Each environment is evaluated separately. After all the environments have been evaluated, the results from NEAT parameter testing are evaluated.

6.1.1. Food and Food Chain Environments

In the Food environment, the agents learnt to collect food. No cooperation was expected in this environment, as no cooperative behaviours were imagined that could make the agents better at the task in the environment. As expected, no cooperative behaviours were identified in the experiments. It is possible that the agents may have tried to spread out in the environment more easily to cover more of the potential areas food may spawn, but this would be hard to see from the simulation.

In the Food Chain environment, no useful patterns were observed in their usage of the yell mechanic, neither when training with individual nor shared fitness. From the start, it was unlikely that this behaviour would emerge in experiments with individual fitness training, as the prey which produced the yell would not be directly rewarded by alerting other nearby prey. On the other hand, prey with shared fitness would be directly rewarded if the yell signal helped other prey avoid being caught by predators. The reward may, however, have been too small of an incentive for useful behaviour to emerge, swamped by the effect of other random variables in the environment. The yell mechanic as implemented may also have had too many individual behaviours that needed to be combined to actually get anything useful at all. No cooperative behaviours were found in this environment either.

6.1.2. Evasion Environment

In the Evasion environment, the algorithm produced more interesting behaviours than in the Food and Food Chain environments. Similar behaviours evolved both when the agents were trained with individual and with shared fitness.

Prey Evaluation

The prey evolved behaviour where they used their vision of other prey and of the walls to guide them across the simulation area to the goal. The results showed that—both with individual and shared fitness—the performance of the prey decreased when their vision of other prey was removed. As an extreme example, the prey from a well performing NEAT individual trained with shared fitness lost their ability to move to the goal—and stood almost always entirely still with only occasional movement—when their visions of other prey was removed. This shows that the prey evolved a useful cooperative behaviour where they interacted with each other through vision. The behaviour can be seen as a type of swarm behaviour as described in Section 5.1.

It is unclear whether the prey need to see each other to perform at the level they did in the experiments. The decrease in performance from blinding them from each other happened after they had been trained with the possibility to see each other. It is possible that the prey could have performed just as well if they could not see other prey during training and instead had to rely more on the wall for guidance. Due to a lack of time, no testing was done to explore this possibility.

It is also difficult to say whether the behaviour was refined after its initial introduction. The prey and predators evolved at the same time in an adversarial manner, making it difficult to tell if fluctuations in fitness over time was due to improved behaviour in one species or a decrease in performance by the other. What we can see is that the process kept the behaviour in the population throughout the generations, and it appeared in the well performing individuals in the examined experiments.

Predator Evaluation

The predators learnt to chase and consume prey. No cooperative behavioural patterns were identified. While the predators also showed a reduction in performance when their ability to see other predators was removed, no clear pattern in the simulations was found to explain this difference. Thus it is unclear whether the reduction in performance is due to the removal of any cooperative behaviour related to the predators seeing each other.

Environment Evaluation

While the adversarial training of the prey and predators in this environment was an interesting dynamic, it made it challenging to associate changes in behaviour with changes in the species' performances. Training the species alone against a fixed adversary would have made analysing their process more attainable. There was also an issue of agents—both predators and prey—immediately getting stuck on the walls. The walls should probably have been placed further away from the spawning positions of the agents, so that the ones who starting facing the wall were less likely to immediately crash into it.

6.1.3. Door Environment

In the door environment, the algorithm managed to produce well performing populations. Using the amount of agents that managed to enter the goal as the metric of success, the shared fitness experiments found optimal solutions and the individual fitness experiments found solutions in which at least half of the agents or more entered the goal. Both the training methods produced behaviours that used cooperation to solve the environment's problem.

Individual Fitness Evaluation

The cooperative behaviours which emerged in the individual fitness experiments were of the type of cooperation which has an independent possibility for reward for each interactant. Agents opening the door had a potential for being rewarded if they remained on the button until other agents entered the goal. Other agents had a potential for being rewarded by entering the goal if they got past the door while another agent was holding it open.

The cooperative behaviours described in Section 5.7.1 are clearly useful—without behaviour leading the agents to the button no agent would be able to get any reward at all. The behaviours appeared early in the run and was kept with variations throughout the runs. Behavioural patterns were refined during the run. For instance, how well the agents managed to stay on top of the button varied. The performance of the agents and the population as a whole was dependant on the amount of agents in the population that exhibited specific behaviours. For example, some badly performing generations—such as the one mentioned as an example in 5.7.1—still had the behaviours present in individuals, but too many or too few of its individuals had specific behaviours required to solve the environment. That this happens is logical. The NEAT algorithm does not attempt to optimise every individual of a given population, it tries to create some individuals who are better while also exploring novel structures. In a few cases this led to useful behaviours disappearing—or at least near disappearing—from the population, so that they had to be regained.

Shared Fitness Evaluation

As the agents shared their fitness, the cooperative behaviours with the shared fitness training method were all of the type in which either all or none of the interactants in the cooperation were rewarded.

As in the results from individual fitness training, good individuals quickly appeared in the experiments with shared fitness. Early in and throughout the generations of the experiments, there were behaviours similar to the well performing cooperative behaviours, but without being useful. For instance, agents with waiting behaviour that ended up waiting for each other and deadlocking. In the very early generations of the experiments, most of the behaviours where the agents reacted to each other had this behaviour, and the useful ones evolved from them. In contrast to the experiments in this environment with individual fitness, the experiments with shared fitness did not lose useful cooperative behaviours once they had evolved. With the shared fitness training method, the behaviours of the simulated agents are stored in individual NEAT individuals, so it is less likely that good behaviours will disappear. The algorithm can also optimise the behaviour of the entire set of simulated agents. Thus, it avoided issues of different portions of dependent behaviours affecting the performance of the agents. The good solutions of every experiment all had variations of a small set of behaviours that appeared early in the experiments. In other words, when looking at multiple experiments with this configuration, the algorithm wasn't produce as novel artefacts as we expected. However, it did produce a few quite different behaviours, so it is capable of some novelty.

Environment Evaluation

The environment gave interesting results with many useful cooperative behaviours. Especially with shared fitness, the algorithm found very good solutions which optimally solved the environment. The processes from which the behaviours emerged fulfil most of the criteria for creativity. However, the individual fitness training method had some issues with losing behaviours when different behaviours needed to work together for the agents to solve the goal. With shared fitness, the experiments converged on populations containing optimal agents rather quickly. This probably means that the environment was too easily solved by the shared fitness method. Given time, it should have been explored whether the algorithm could produce more novel results if tested on a more difficult environment in which it would need to take longer to find optimal behaviours.

6.1.4. NEAT Parameters

Multiple experiments were conducted to test how the different parameters effected NEAT's ability to creatively evolve cooperative behaviours. The results showed that population size was the only parameter that made a significant difference. When

the population size was too small, NEAT's ability to evolve cooperative behaviour was reduced, and the algorithm was not able to stabilise or even find an optimal behaviour. Such results were expected, as NEAT uses its population for exploring and keeping different solution candidates. If the population is too small, then NEAT will have a reduced capability to explore the problem space. It will have a reduced ability to keep both good solutions candidates in the populations while still keeping the diversity in the population. If good individuals in the population are lost, then NEAT may lose useful behaviours. If the diversity is too low then NEAT can get stuck in local optima.

When running the experiments on different population sizes, we found that the data produced by the agents trained with shared fitness were the most useful. Based on this result, the other experiments were only run with agents trained with shared fitness. The results from the experiments done with different population sizes also showed that population size 50 was the lowest of the population sizes tested where the algorithm was still able to keep an optimal behaviour once one had emerged. The population size was therefore set to 50 in these experiments. The results from the experiments checking the various parameters showed no significant differences in NEAT's ability to creatively evolve cooperative behaviours. In all the experiments the algorithm evolved an optimal quickly and was able to keep the behaviour. The NEAT parameters were tested one at a time, but the values of the parameters were set to extreme values. For example, one of the configurations was with the crossover rate set to zero, effectively disabling crossover in the algorithm and causing the algorithm to explore exclusively through mutations. The results showing no significant differences in these experiments—except with different population sizes—indicated that the the Door environment with shared fitness was too simplistic—making it too easy for the algorithm to find optimal behaviours—-for this testing.

6.2. Discussion

The goal of this thesis was to explore if the NEAT algorithm could be used to creatively evolve cooperative behaviour. It was split up into two research questions.

6.2.1. Research Question 1

RQ1 *How does NEAT perform in creatively evolving cooperative behaviour in simple multi-agent settings?*

1 Can NEAT evolve cooperative behaviour that is beneficial to all or none of the agents involved in the interaction?

2 Can NEAT evolve cooperative behaviour that is potentially beneficial to some of the agents involved in the interaction?

To answer Research Question 1, two things had to be evaluated. The first thing that needed be evaluated was whether the artefacts produced by the algorithm—that is, the behaviour of the agents—had cooperative behaviour. The second thing was whether the process which made the artefacts with cooperative behaviour had the properties required for it to be a creative process.

The definition of a creative process in this thesis states that the behaviour has to be novel and useful, and that the process should be able to evaluate the usefulness of the behaviour and guide itself based on this evaluation. The process should arrive at new behavioural patterns that improve the fitness of the agents, and refine and keep this behaviour for as long as it is useful. This definition is inspired by works by both Boden and Jordanous. Their source material is described in Section 3.2. Boden (1998) states that for an idea to be creative it needs to be surprising, valuable and novel. Jordanous (2012b) lists similar properties, including the value and originality components amongst her 14 key components of creativity.

The most fruitful environment in terms of positive results was the Door environment. Cooperative behaviours of both types were found, and they were clearly useful to the agents. We also see that the process had kept and improved on the behaviours over a longer period of time. The behaviours may also be seen as novel, as the NEAT algorithm produced them from an initially minimal network. However, similar behaviours did appear in most of the experiments with similar configurations. Based on this, the emergence of the behaviours could be considered less novel when looking at the algorithm over multiple experiments. Otherwise, the behaviours and their emergence satisfy the criteria for both cooperation and the creative process.

The other environments gave more modest results. In the Food and Food Chain environments, no cooperative patterns were identified at all. In the Evasion environment, cooperative behaviours were observed among the prey. While the algorithm kept these useful behaviours over a long time, whether it refined the behaviours is inconclusive. The behaviours were also less novel over multiple experiments, as small variations on the same behaviour emerged to dominate the better performing solutions.

The results show that NEAT has been creative in evolving some of the behaviours during our experiments. However, the system was unable to produce artefacts that could extract usefulness from the yell mechanic. This indicates that the yell mechanic required too complex behaviours before starting to be useful. The more aspects there are of the agents' behaviours that need to work together for the whole behaviour to be rewarding, the less likely it is that the evolutionary process will evolve all these aspects. It is clear that there are limitations to NEAT's use in this context.

6.2.2. Research Question 2

RQ2 *To what degree do different parameters affect the ability of NEAT to creatively evolve cooperative behaviour?*

The results showed that there was a correlation between population size and NEAT's

ability to creatively evolve cooperative behaviour. The results from the other parameters, however, showed no correlation in the ability of NEAT to creatively evolve cooperative behaviour. This is likely due to the Door environment being too simplistic, making it too easy for the algorithm to find optimal behaviours. Based on the results, the NEAT algorithm ought to be able to solve more complex problems in environments with tasks similar to the Door environment.

6.3. Limitations

This section presents the limitations that were found during the work of this thesis.

6.3.1. Training Methods

Both the training methods implemented in Evsim have limitations. The first training method implemented was based on the individual fitness of the simulated agents. This worked well for evaluating the performance of each agent in solving independent tasks. However, this could not be used to train the agents based on the performance of the entire simulated population, as the NEAT algorithm uses the differences between the agents' fitness to rank the individuals in the population for reproduction. The method with training based on a shared fitness value was implemented to overcome this issue. This worked, but had the limitation that every simulated agent in a candidate solution had identical brains and thus exhibited the same behaviour with identical inputs. There is no method in the system as of now which can be used to evolve candidate solutions with distinct agents trained with a shared fitness value.

6.3.2. Architecture

In Evsim, currently a lot of the code in the environments and their contained species is duplicated. As a result, changing something common to multiple environments or species requires the same modification in many places. Adding new environments will exacerbate this issue. If the system is to be extended, refactoring ought to be done to extract the duplicated code into shared components to be composed into environments and species.

The agents' vision sensors are written to detect points. This works fine for small objects such as other agents or the food particles, but has issues with other types of objects. In the currently implemented environments, the walls consist of many segments of shorter walls to let the agents properly see them. This worked, but made the collision detection calculations more computationally intensive than they ought to be. There is also an issue with the button in the Door environment. When an agent is inside the button and facing away from the centre of the button, it will not see the button at all.

7. Conclusion and Future Work

This thesis explored whether the NEAT algorithm could creatively evolve cooperative behaviour. To this end, a simulation system named Evsim was developed. Experiments were run with different configurations on four environments implemented in the system, namely the Food, Food Chain, Evasion and Door environments. The results showed that NEAT was unable to evolve any cooperative behaviour in the Food and Food Chain environments. In the Evasion environment, NEAT was able to evolve a cooperative behaviour, but there were insufficient results to conclude that the process from which it emerged satisfied all the criteria necessary for being considered a creative process. In the Door environment, NEAT was able to evolve several cooperative behaviours creatively. Of all the NEAT parameters tested in the experiments, only the population size had a significant effect on NEAT's ability to creatively evolve cooperative behaviour. It is likely that the environment used to test the parameters was too simplistic for properly evaluating the parameters.

7.1. Contributions

This thesis contributes to the field of computational creativity by providing an evaluation of NEAT's ability to creatively evolve cooperative behaviour in a multi-agent system.

Another contribution to the field is the Evsim simulation system that was used to conduct the experiments. The system can be extended to support additional algorithms and environments. This can be useful to other researchers who want to perform similar experiments with other algorithms or further experiments with the NEAT algorithm. The system's code is open source and can be found at `https://github.com/einhov/evsim`.

Both the individual and cooperative behaviours that have evolved during the experiments may be useful to game developers as examples of what kind of behaviours the NEAT algorithm can produce. If they find that they have environments with about the same level of complexity in their games, then the NEAT algorithm may be worth looking into. The cooperative behaviours are, however, quite simple, so it would be beneficial to do further experiments in more advanced environments to see if NEAT can produce more advanced behaviours.

7.2. Future Work

As mentioned in Section 6.1.2, the agents in the Evasion environment used their ability to see other agents of its own species to decide on their actions. While the performance of the agents decreased when they were blinded to agents of their own species, it is not clear whether they wouldn't perform at the same level as the non-blinded agents if they weren't trained with the ability to see other agents of their species. It would be interesting to do more experiments in the Evasion environment to determine this. The species should be trained with and without seeing their own species. The resulting behaviours could then be compared to check whether they perform as well as the species trained while seeing each other—blinded after the fact or not. It would also have been beneficial to reduce the number of random variables in the environment, as it was difficult to draw conclusions from the visualisation and the data produced due to the randomness. One thing to try would be to only train one species at the time against a static, pre-trained species.

A few of the agents' brains evolved in the experiments conducted were visualised and examined. The visualisations showed that the edges connected to and from most of the hidden nodes in the brains had weight values close to zero. This indicates that the brains have a low benefit from the hidden nodes, and that similar acting brains could potentially have been created using a minimal network. As the evolving topology of the neural networks is a central part of the NEAT algorithm, other algorithms may perform as well or better on the experiments of this thesis. Future work could try evolving the weights of a minimal networks with fixed topology with other evolutionary algorithms.

It would also be interesting to add more complex environments. One example is a variation of the Door environment where the agents would need to stand on multiple buttons and go through multiple doors to get to the goal. Another interesting variation could be that the agents would need to push the buttons in a certain sequence to open a door to get to the goal. If more complex environments have been developed then it would be interesting to rerun the experiments with different NEAT parameters to further investigate how they affect NEAT's ability to creatively evolve cooperative behaviour.

Bibliography

Bjørnar Walle Alvestad and Endre Larsen. Creative Behaviour in Evolving Agents. Master's thesis, Norwegian University of Science and Technology (NTNU), Trondheim, Norway, 2017.

Peter J. Bentley and David W. Corne, editors. *Creative Evolutionary Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

John A Biles. GenJam: A genetic algorithm for generating jazz solos. In *International Computer Music Conference (ICMC)*, volume 94, pages 131–137, 1994.

John A Biles. GenJam in perspective: a tentative taxonomy for GA music and art systems. *Leonardo*, 36(1):43–45, 2003.

Margaret A. Boden. Creativity and artificial intelligence. *Artificial Intelligence*, 103 (1):347–356, 1998.

Erin Catto. Box2D, 2015. URL `http://box2d.org/`. [Online; accessed 12-May-2018].

Peter Chervenski. MultiNEAT, 2012. URL `http://multineat.com/`. [Online; accessed 20-April-2018].

Simon Colton. Creativity versus the perception of creativity in computational systems. In *AAAI spring symposium: creative intelligent systems*, volume 8, 2008.

Simon Colton. Seven catchy phrases for computational creativity research. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

Simon Colton and Geraint A. Wiggins. Computational creativity: The final frontier? In *Proceedings of the 20th European Conference on Artificial Intelligence*, pages 21–26. IOS Press, 2012.

Simon Colton, Ramon López de Mántaras, and Oliviero Stock. Computational creativity: Coming of age. *AI Magazine*, 30(3):11, 2009.

Simon Colton, John William Charnley, and Alison Pease. Computational creativity theory: The FACE and IDEA descriptive models. In *Proceedings of the 2nd International Conference on Computational Creativity*, pages 90–95, 2011.

Simon Colton, Alison Pease, Joseph Corneli, Michael Cook, Rose Hepworth, and Dan Ventura. Stakeholder groups in computational creativity research and practice. In Tarek R. Besold, Marco Schorlemmer, and Alan Smaill, editors, *Computational Creativity Research: Towards Creative Machines*, chapter 1, pages 3–36. Atlantis Press, Paris, 2015.

Agoston E Eiben and Jim E Smith. *What is an evolutionary algorithm?* Springer, 2015.

Ashok Goel, Anna Jordanous, and Alison Pease. Preface. In *Proceedings of the Eighth International Conference on Computational Creativity*. ACC, 2017.

Anna Jordanous. A standardised procedure for evaluating creative systems: Computational creativity evaluation based on what it is to be creative. *Cognitive Computation*, 4(3):246–279, 2012a.

Anna Jordanous. *Evaluating computational creativity: a standardised procedure for evaluating creative systems and its application*. PhD thesis, University of Sussex, 2012b.

Anna Jordanous. Stepping back to progress forwards: Setting standards for meta-evaluation of computational creativity. In *Proceedings of the Fifth International Conference on Computational Creativity*, pages 129–136, 2014.

Andrej Karpathy and Casey Link. Scriptbots, 2011. URL `https://sites.google.com/site/scriptbotsevo/home`. [Online; accessed 6-December-2017].

Eduardo Reck Miranda and John Al Biles. *Evolutionary computer music*. Springer, 2007.

Alison Pease and Simon Colton. On impact and evaluation in computational creativity: A discussion of the Turing test and an alternative proposal. In *Proceedings of the AISB symposium on AI and Philosophy*, 2011.

Graeme Ritchie. Some empirical criteria for attributing creativity to a computer program. *Minds and Machines*, 17(1):67–99, 2007.

Stuart Russell and Peter Norvig. *Artificial Intelligence A Modern Approach Third Edition*. Prentice Hall, 2010.

Robert M. Seyfarth, Dorothy L. Cheney, and Peter Marler. Monkey responses to three different alarm calls: Evidence of predator classification and semantic communication. *Science*, 210(4471):801–803, 1980.

Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

A. Appendices

A.1. Example Configuration

Evsim takes in multiple optional arguments where the first one is a Lua configuration file. When Evsim is initiated, the configuration file is handled as a Lua script that runs and configures the system. The following configuration file includes all the parameters that can be changed by the configuration. The rest of the arguments that are given to Evsim are passed to the configuration script. In this configuration the second argument is used to choose between the environments. Two of the environment's names are different in the thesis and in Evsim. The Food Chain environment in the thesis is called `multi_food` in Evsim, and the Evasion environment is called `multi_move`.

Evsim can be started with the following configuration file by issuing the command:

./evsim config-file-path environment-name

Where environment-name can be `food`, `multi_food`, `multi_move` or `door`.

```
-- Example configuration file

-- Contains (or should contain) an exhaustive listing of parameters

max_generations = 50

neat_params = {
  dynamic_compatibility = true,
  min_species = 3,
  max_species = 20,
  compat_thresh = 5.0,
  crossover_rate = 0.7,
  mutation_rate = 0.25,
  tournament_size = 4,
  elite_fraction = 0.01,
  old_age_penalty = 0.5,

  MutateNeuronActivationTypeProb = 0.0,

  ActivationFunction_SignedSigmoid_Prob = 0.0,
  ActivationFunction_UnsignedSigmoid_Prob = 1.0,
  ActivationFunction_Tanh_Prob = 0.0,
  ActivationFunction_TanhCubic_Prob = 0.0,
  ActivationFunction_SignedStep_Prob = 0.0,
  ActivationFunction_UnsignedStep_Prob = 0.0,
  ActivationFunction_SignedGauss_Prob = 0.0,
  ActivationFunction_UnsignedGauss_Prob = 0.0,
  ActivationFunction_Abs_Prob = 0.0,
  ActivationFunction_SignedSine_Prob = 0.0,
  ActivationFunction_UnsignedSine_Prob = 0.0,
```

```
  ActivationFunction_Linear_Prob = 0.0,
  ActivationFunction_Relu_Prob = 0.0,
  ActivationFunction_Softplus_Prob = 0.0
}

food = {
  name = "food",
  food_count = 150,
  steps_per_generation = 10,
  ticks_per_step = 60 * 15,

  herbivores = {
    population_size = 100,
    training_model = "normal", -- "normal", "normal_none", "shared", "shared_none"
    thrust = 1000.0,
    torque = 45.0,
    shared_fitness_simulate_count = 5,
    save = nil, -- path to directory for storing the population every generation
    initial_population = nil, -- path to file with initial population
    avg_window = 21, -- size of window used for moving average in plot

    neat_params = neat_params
  }
}

multi_food = {
  name = "multi_food",
  food_count = 150,
  steps_per_generation = 50,
  ticks_per_step = 60 * 15,

  herbivores = {
    population_size = 100,
    training_model = "normal", -- "normal", "normal_none", "shared", "shared_none"
    thrust = 1000.0,
    torque = 45.0,
    yell_delay = 30,
    shared_fitness_simulate_count = 5,
    save = nil, -- path to directory for storing the population every generation
    initial_population = nil, -- path to file with initial population
    avg_window = 21, -- size of window used for moving average in plot

    neat_params = neat_params
  },

  predators = {
    population_size = 100,
    training_model = "normal", -- "normal", "normal_none", "shared", "shared_none"
    thrust = 1000.0,
    torque = 45.0,
    eat_delay = 60, -- < 0: once, == 0: no_delay, > 0: delay
    shared_fitness_simulate_count = 5,
    save = nil, -- path to directory for storing the population every generation
    initial_population = nil, -- path to file with initial population
    avg_window = 21, -- size of window used for moving average in plot

    neat_params = neat_params
  }
}

multi_move = {
  name = "multi_move",
  steps_per_generation = 10,
  ticks_per_step = 60 * 15,

  herbivores = {
    population_size = 50,
    training_model = "normal", -- "normal", "normal_none", "shared", "shared_none"
```

```
    thrust = 1000.0,
    torque = 45.0,
    shared_fitness_simulate_count = 5,
    save = nil, -- path to directory for storing the population every generation
    initial_population = nil, -- path to file with initial population
    avg_window = 21, -- size of window used for moving average in plot

    neat_params = neat_params
  },

  predators = {
    population_size = 50,
    training_model = "normal", -- "normal", "normal_none", "shared", "shared_none"
    thrust = 1000.0,
    torque = 45.0,
    eat_delay = 60, -- < 0: once, == 0: no_delay, > 0: delay
    shared_fitness_simulate_count = 5,
    save = nil, -- path to directory for storing the population every generation
    initial_population = nil, -- path to file with initial population
    avg_window = 21, -- size of window used for moving average in plot

    neat_params = neat_params
  }
}

door = {
  name = "door",
  steps_per_generation = 1,
  ticks_per_step = 60 * 15,

  herbivores = {
    population_size = 50,
    training_model = "normal", -- "normal", "normal_none", "shared", "shared_none"
    thrust = 1000.0,
    torque = 45.0,
    shared_fitness_simulate_count = 5,
    save = nil, -- path to directory for storing the population every generation
    initial_population = nil, -- path to file with initial population
    avg_window = 21, -- size of window used for moving average in plot

    neat_params = neat_params
  }
}

environments = {
  food = food,
  multi_food = multi_food,
  multi_move = multi_move,
  door = door
}

physics = {
  linear_damping = 10.0,
  angular_damping = 10.0
}

sensors = {
  length = 45.0,
  fov = 60.0
}

environment = environments[arg[2]] or food
```

A.2. Running Multiple Tests

The following Python script is an example for scheduling multiple experiments in Evsim. This particular script was used to run the experiments in the first test set. Up to eight instances of Evsim were run simultaneously.

```
import multiprocessing as mp
import subprocess as sp
import os
import time
import parse
import signal

from tqdm import tqdm

UPDATE = 1
DONE = 2
NEW = 3

queue = mp.Queue()

def execute(args):
  with open('overnight/{0}-{2}-{4} {3}-{5}-{1}.timing'.format(*args), "w+") as
      ↪ timing_file:
    current = mp.current_process()
    queue.put((current.name, NEW, "{0}-{2}-{4} {3}-{5}-{1}".format(*args)))

    start = time.time()
    timing_file.write(str(start) + '\n')

    proc = sp.Popen(
      [   './evsim',
        '../overnight.lua',
        str(args[0]),
        str(args[1]),
        str(args[2]),
        str(args[3]),
        str(args[4]),
        str(args[5])
      ],
      stdout=sp.PIPE,
      stderr=sp.PIPE
    )

    for line in proc.stderr:
      result = parse.parse('Generation: {}', line.decode("utf-8"))
      if(result):
        queue.put((current.name, UPDATE, int(result[0])))

    end = time.time()
    timing_file.write(str(end) + '\n')
    timing_file.write(str(end - start) + '\n')

    queue.put((current.name, DONE))

runs = []

# individual fitness door and food runs
runs.extend([
  (env, 0, "normal", pop_size, "none", 0)
  for env in ["door", "food"]
  for pop_size in [25, 50, 75]
])

# shared fitness door and food runs
runs.extend([
```

```
  (env, sim_count, "shared", pop_size, "none", 0)
  for env in ["door", "food"]
  for pop_size in [10, 50, 100]
  for sim_count in [2,10,25]
])

# individual fitness move and multi_food runs
runs.extend([
  (env, 0, "normal", pop_size_0, "normal", pop_size_1)
  for env in ["multi_food", "multi_move"]
  for pop_size_0 in [25, 50, 75]
  for pop_size_1 in [25, 50, 75]
])

# shared fitness move and multi_food runs

# prey
runs.extend([
  (env, sim_count, "shared", pop_size_0, "normal", 50)
  for env in ["multi_food", "multi_move"]
  for pop_size_0 in [10, 50, 100]
  for sim_count in [25, 50, 75]
])

# predators
runs.extend([
  (env, sim_count, "normal", 50, "shared", pop_size_1)
  for env in ["multi_food", "multi_move"]
  for pop_size_1 in [10, 50, 100]
  for sim_count in [10, 25, 50]
])

if __name__ == '__main__':
  #runs = runs[:1]
  bars = {'main': tqdm(total=len(runs), ncols=120, position=0, desc="Runs: ")}

  def handler_sigwinch(num, frame):
    print("\033c")
    for _,bar in bars.items():
      bar.refresh()
  signal.signal(signal.SIGWINCH, handler_sigwinch)

  with mp.Pool(8) as pool:
    results = []

    for run in runs:
      results.append(pool.apply_async(execute, (run,)))

    remaining = len(runs)
    while remaining > 0:
      msg = queue.get()
      if msg[1] == DONE:
        bars['main'].update()
        remaining -= 1
      elif msg[1] == UPDATE:
        bars[msg[0]].update()
      elif msg[1] == NEW:
        if msg[0] in bars:
          bars[msg[0]].close()
        position = parse.parse('ForkPoolWorker-{}', msg[0])
        bars[msg[0]] = tqdm(total=500, ncols=120, position=int(position[0]), desc=msg
            ↪ [2])
        signal.signal(signal.SIGWINCH, handler_sigwinch)

    for res in results:
      res.wait()

  for _,bar in bars.items():
```

```
    bar.close()
```

The python file calls the following lua script.

```
local env = arg[2]
local shared_fitness_simulate_count = tonumber(arg[3])

local species0_training_model = arg[4]
local species0_population_size = tonumber(arg[5])
local species1_training_model = arg[6]
local species1_population_size = tonumber(arg[7])

local storage = string.format(
  "overnight/%s-%s-%s %d-%d-%d",
  env,
  species0_training_model,
  species1_training_model,
  species0_population_size,
  species1_population_size,
  shared_fitness_simulate_count
)

max_generations = 500

environment = {
  name = env,
  steps_per_generation = 10,
  ticks_per_step = 60 * 30,

  herbivores = {
    training_model = species0_training_model,
    train = true,
    population_size = species0_population_size,
    shared_fitness_simulate_count = shared_fitness_simulate_count,
    save = storage .. "/h"
  },

  predators = {
    training_model = species1_training_model,
    train = true,
    population_size = species1_population_size,
    shared_fitness_simulate_count = shared_fitness_simulate_count,
    save = storage .. "/p"
  }
}

if species0_training_model == "shared" then
  environment.steps_per_generation = species0_population_size
elseif species1_training_model == "shared" then
  environment.steps_per_generation = species1_population_size
end

sensors = {
  length = 120,
  fov = 45
}
```