THE EFFECT OF MUTATION ON EXPLORATIVE & EXPLOITATIVE BEHAVIORS WITH RT-NEAT

Master thesis for Khoa Pham

Institute of Information and Media Science

University of Bergen

Spring 2017

Key words: rtNeat, exploration, exploitation, EDA, Mutation

Abstract

This thesis aims to explore how different factors can affect the search performance of evolutionary algorithms. Additionally how applying different mutation techniques changes the overall search performance of rtNEAT. This thesis demonstrates how mutation affects exploration and exploitation when optimizing for a 3-input XOR gate as well as optimizing agent movements in a real time environment.

This thesis is also provided as a guideline in the development of an evolutionary algorithm, particularly the implementation of rtNEAT algorithm, and a simple game environment in Python.

Acknowledgements

Firstly, I would like to thank my mentor, Bjørnar Tessem for being of great help and guided me throughout the work of my thesis.

Secondly I would like to thank my friend who had encouraged me to finish this thesis even when I did not believed in myself.

Table of Contents

1 Introduction 6

1.1 Introduction 6

1.2 Motivation 6

1.3 Research Questions 7

1.4 Research Method 8

1.5 Contributions 8

1.6 Thesis structure 9

2 Background 9

2.1 Artificial Neural Network 9

2.2 The Perceptron and Multi Layered Perceptron (MLP) 10

2.3 Supervised learning and Backpropagation 14

2.4 Reinforcement Learning 15

2.5 Population Based Optimization Algorithms 18

2.6 Genetic Algorithms 22

2.7 Neuro Evolution 25

2.8 Neuro Evolution of Augmenting Topologies (NEAT) 28

2.9 Real Time NEAT (rtNEAT) 33

2.10 Exploration vs Exploitation 35

2.11 Estimation of Distribution Algorithms (EDA) 40

2.12 Experimenting With A.I in Game Environments 41

3 Related Studies 42

3.1 NERO 43

3.2 NEAT and XOR Function 43

3.3 Differential Evolution and EDA 44

4 Implementation 44

4.1 Implementation of rtNEAT – The Metaheuristic Search 45

4.2 Implementation of rtNEAT – The Neural Network 47

4.3 Implementing the real-time Game Environment 49

4.4 Implementation of EDA Mutator 50

4.5 The NEAT Mutator 52

4.6 Verifying rtNEAT – The Evolution of XOR 53
4.7 Implementation of the Experiments 55
4.8 Issues 56
5 Experiments 57
5.1 XOR 57
5.1.1 RtNEAT only 58
5.1.2 RtNEAT with explorative search 58
5.1.3 RtNEAT with exploitative search 59
5.1.4 RtNEAT with exploitative & explorative search 60
5.1.5 RtNEAT with EDA exploitative search 60
5.2 Game 60
5.2.1 RtNEAT only 61
5.2.2 RtNEAT with exploitative & explorative search 62
5.2.3 RtNEAT with exploitative EDA 62
6 Results 62
6.1 XOR 62
6.1.1 RtNEAT only 63
6.1.2 RtNEAT with explorative search 64
6.1.3 RtNEAT with exploitative search 66
6.1.4 RtNEAT with exploitative & explorative search 66
6.1.5 RtNEAT with EDA exploitative search 67
6.2 Game 68
6.2.1 RtNEAT only 68
6.2.2 RtNEAT with exploitative & explorative search 69
6.2.3 RtNEAT with exploitative EDA 70
7 Discussion, Conclusion and Future Work 71
7.1 Discussion 71
7.2 Conclusion 73
7.3 Future Work 74
References 75

1 Introduction

1.1 Introduction

Artificial intelligence is an emerging field and is rapidly becoming one of the most popular and debatable topics that keeps popping up in the news. Since A.I can be applied to solve a wide range of problems across all field of studies, questions have raised to challenge further development of A.I, such as if A.I is capable of achieving human intelligence or whether it will eventually take all of our jobs.

Whether an A.I can achieve human intelligence is still one of several questions that needs to be answered. Yet the quest to answer this kind of question have been prompted by many researchers by the proposal of different general machine learning algorithms, in hope to improve the ability of computers to solve different problems on a general level, something we as human are very good at.

Recently a team of researchers from Google Deep Mind have created an A.I named AlphaGo [1] capable of winning against world champion in the classical board game Go, a game that was believed to be one of the most challenging problems for machines to learn. Yet with machine learning, particularly reinforcement learning they managed to achieve this goal. But even when AlphaGo can be excellent at playing Go, that is all it can do, no indication of general intelligence can be demonstrated by the system.

To push the boundaries further, researchers (including Google Deep Mind) have developed techniques in an attempt to allow computers to play several computers games in the same manner as a human would, believing that by beating those games, an indication of general A.I can be promised [2][3].

This thesis attempts to explore the challenge of teaching machines to play computer games by using techniques in neuroevolution, which is capable of learning reinforcement learning tasks. Furthermore, this will hopefully inspire more researchers to explore the same direction.

1.2 Motivation

The original motivation which led to the work behind this thesis is the promising idea of teaching computers to play video games on a general and professional level. As many games in the commercial market today implements simplistic A.I models which often serve their purpose very well, but for real time competitive games such

as in e-sport[1] games, simplistic A.I tends to fail or display mechanical and predictive behaviors when playing against human players. For this reason along with the belief that learning A.I is the holy-grail for developing challenging and interesting game agents with human-like behavior, interest grew towards the direction of neuroevolution, especially using a technique called rtNEAT which had been demonstrated to work for gaming environments, specifically for a game named NERO [4]. Yet no technique is perfect, therefore another motivation was to attempt to modify rtNEAT to allow for better performance.

1.3 Research Questions

As further reading and preliminary development were carried out, the original motivation did no longer define the research questions of interest in this thesis. One of the main reasons was the discovery of how different factors in different evolutionary algorithms can play a big role in how well they can solve certain tasks. This led to more reading towards the direction of how to tune different algorithmic parameters in order to generally optimize for any problem.

The result of preliminary work laid ground for the research questions in this thesis:

1. **How do different elements of learning algorithms, particularly different mutation techniques combined with rtNEAT influence the search behavior?**

2. **What are the common issues in tuning algorithmic parameters for balance between explorative and exploitative search?**

3. **How well does rtNEAT perform particularly in a game environment when applying different mutation techniques?**

Additionally this thesis also aims to provide a detailed theoretical description of different techniques and methods in using neural network to solve reinforcement learning tasks, as well as how implementation for generic learning algorithms may be carried out, especially in the context of neuroevolution in game environments. This thesis and the work behind it can therefore be considered as a general guideline on how to research and implement neuroevolution algorithms as well as other generic optimization techniques.

[1] https://en.wikipedia.org/wiki/ESports

1.4 Research Method

The research method used in this thesis will be a combination of two methods:

1. **By proof of concept and experiments** – One reason for using proof of concept is that it produces artifacts that can be used to run different experiments in order to strengthen the answer of research questions. Another reason is because the produced artifacts may lay ground for future work.

   The artifacts that are produced as a proof of concept in this thesis is a computer program used to run experiments to demonstrate the different effects of relevant learning algorithms, specifically neuroevolution algorithms.

2. **By literature review** – Literature review lay the foundation for the work in this thesis as well as providing a guideline in the research direction. This is crucial in research as it would allow for a rigorous connection between research questions and the research work. In case of research question 2, which is more of a higher level question that requires a combination of actual research work and previous studies to answer.

1.5 Contributions

Besides the answers to the research questions of this thesis, the following additional contributions can be also considered:

1. A general guideline (the thesis itself) in the developmental process of neuroevolution techniques with hints on issues and drawbacks during development.
2. A pure rtNEAT along with an EDA Mutator implemented in Python to contribute to the NEAT Users Group (where different users have implemented their own versions), as well as the general research community as a whole.
3. A simple visual 2d game framework that can be used to test different learning algorithms.

1.6 Thesis structure

This thesis is divided into 7 chapters, where the purpose of each chapter is outlined below:

1. **Introduction** – Introduces to the work of this thesis.
2. **Background** – This chapters aims to provide readers with general knowledge behind the work of this thesis.
3. **Related Studies** – This is a short chapter discussing different studies that are similar to the study presented here.
4. **Implementation** – This chapter aims to provide a detailed description of how the different implementation stages were done.
5. **Experiments** – This chapter describes the different experiment setups.
6. **Results** – This chapter presents the results from the experiments in chapter 5.
7. **Discussion, conclusion and future work** – This is the final chapter discussing and summarizing the work of this thesis, as well as what may be the next steps in future work.

The reason this thesis is structured as described above is to provide readers with a gradual construction of background knowledge before diving deeper into the problems. This would allow readers to better understand each step as well as the purpose of the work presented in this thesis.

2 Background

This section presents related background theories and techniques used in this thesis.

2.1 Artificial Neural Network

Artificial neural network (ANN) is a computational network structure inspired by the biological neural network similar to those in the human brain [5].

ANNs typically function by propagating information (usually from sensory inputs) through the network via nodes called Artificial Neurons (neurons) and connections known as Artificial Synapses (synapses). Nodes and Connections are therefore the fundamental building blocks of ANNs.

In order for an ANN to carry out meaningful computations, information is transferred between neurons via synapses by a cascade of activation functions and connection weights from an initial state of input values to a final state as output values. Typically values accumulated at any layers of nodes are multiplied by their

corresponding outgoing connection weights before being added as input values to the next layer of nodes, at which these input values are summed before being applied an activation function for further propagation [6][7].

Signals fired and sent through synapses in an ANN can be either inhibitory for negative signals or excitatory for positive signals, which in turn will directly affect the activity of neurons down the paths in the network.

The mechanism described above is well known and is commonly used in most implementations of ANN. Despite this there are also other models of ANNs where the process of activating and transferring information signals gets more complicated to imitate the actual processes in the biological brain. Such a different model of ANN is for example the spiking neural network, where signals are transferred in spikes of action potentials and can compute more complex functions than the traditional ANN model [8][9].

In this thesis, the more commonly used ANN model using a non-linear activation function (i.e. the sigmoid function) emitting a single action potential signal will be used instead of a more complex one such as the spiking model.

Next we will discuss in more detail how values are computed within the simplest form of ANN, namely The Perceptron and its multi layered successor known as Multi Layered Perceptron (MLP).

2.2 The Perceptron and Multi Layered Perceptron (MLP)

The Perceptron is a name given to the simplest form of ANN invented by Frank Rosenblatt [10], which gained popularity among researchers and practitioners in the earlier days of neural computing. The Perceptron is an ANN consisting of a single neuron at which all the input synapses in the network are connected.

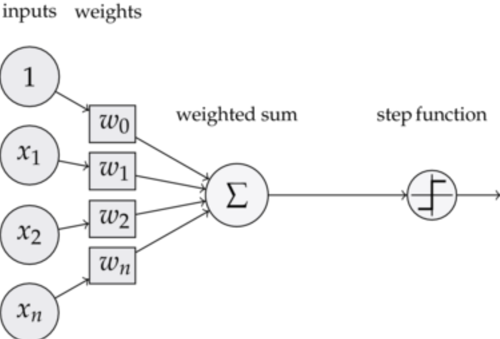

Fig1. The Perceptron has a single neuron which sums all of its inputs before going through a step function. Note that there is a constant input of 1 connected through w_0, this is the bias connection.

The type of neuron used in the Perceptron (and generally in any ANN) can vary and depending on the problem space, but one of the most commonly used neuron models that have originally been proposed to work with the Perceptron is the McCulloch Pitt's model [11].

The McCulloch Pitt's model describes a type of neuron known as "all-or-none", meaning that once excited above an activation threshold the neuron will emit a fixed signal regardless of the strength of the incoming stimuli. If the neuron does not receive enough stimulation to be excited, no signals will be fired until the activation threshold is reached. In addition the McCulloch Pitt's model is also a binary model [12], this means that fired signals are constant and can therefore only represent either 1 or 0 (fired and not fired).

In recent years different neuron models have been proposed and used where the behavior of the neuron becomes more complex as different firing patterns and activation functions have been used (fig2) [5][6]. Some of the most common activation functions that have been widely adopted and also used in this thesis are the logistic sigmoid and the hyperbolic tangent functions.

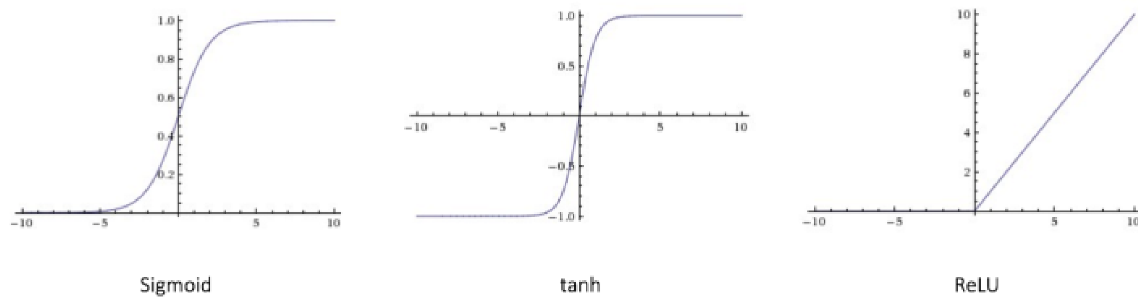

Fig2. Sigmoid, hyperbolic tangent and the linear rectifier activation functions.

Generally most implementations of ANN (including the Perceptron) computes the network signals in similar way; the output signal for any given neuron y is given by

$$y = \theta\left(\sum_{i=1}^{n} w_i x_i\right) \tag{1}$$

where θ is the activation function, n the number of incoming connections, w_i denotes the weight of incoming connection i and x_i is the incoming signal at connection i.

For the McCulloch Pitt's model, θ is considered a unit step function that steps at x = 0 defined by

$$\theta(x) = \begin{cases} 1 \; if \; x \geq 0 \\ 0 \; if \; x < 0 \end{cases} \tag{2}$$

As mentioned earlier, this thesis will instead use the more commonly used activation functions, namely the logistic sigmoid:

$$\theta(x) = \frac{1}{(1 + e^{-\beta x})} \tag{3}$$

where β is the slope parameter, and the hyperbolic tangent:

$$\theta(x) = \tanh(x) = \frac{2}{(1 + e^{-2x})} - 1 \tag{4}$$

An important difference between the logistic sigmoid and the hyperbolic tangent function is that the logistic sigmoid has an output range between 0 and 1 while the hyperbolic tangent outputs in the range between -1 and 1, this is essential for calculations that requires negative values (i.e. velocity vector in a coordinate system).

It has become a common practice to introduce so called bias connections in ANN (fig1). These bias connections are usually connected to an input source that constantly emits 1 and then multiplied with the bias connection weight. Without the bias connections it is almost impossible (if not very difficult via complicated circuitry) to move the activation function along the X-axis. These bias connections give an ANN a very powerful property that makes it a better universal approximator.

Consider the activation function $\theta(x)$ in equation (3), if we are to move the activation function along the X-axis we must be able to express $x+a$, this can be done by introducing an independent bias input a from all other actual inputs x at each neuron, i.e. for the logistic sigmoid we get

$$\theta(x + a) = \frac{1}{(1 + e^{-\beta(x+a)})} \tag{5}$$

Perceptron learning is the concept of iteratively adjusting the connection weights in the Perceptron until a desired output is computed. In order to adjust the weights the Perceptron must be provided training examples of input patterns and expected output values. This form of weight training belongs to the class of supervised learning algorithm where an error-correction rule is used to correct the connection weights [5], this algorithm is also known as the Perceptron learning algorithm:

1. Initialize random weights for all connections (including bias)
2. Feed an input pattern of $(x_1, x_2, \ldots, x_n)$ and evaluate the output value y

3. Update each weight according to

$$w_i(t+1) = w_i(t) + \eta(d-y)x_i \tag{6}$$

where η is the learning rate and t is the iteration step.

Many ANN learning algorithms use the notion of learning rate to train the network gradually over many iterations. The concept of learning rate is therefore crucial for most learning algorithms, particularly for supervised learning algorithms such as the backpropagation algorithm. We will discuss the backpropagation algorithm in more detail later but let's first discuss the limitations of the single layered Perceptron and the introduction of the multi layered Perceptron.

The multi layered Perceptron (MLP) is an ANN that introduces multiple layers of neurons between the input and the output of the originally proposed single layer Perceptron discussed above. These extra neurons and layers are called hidden neurons and hidden layers respectively (see fig3 for illustration).

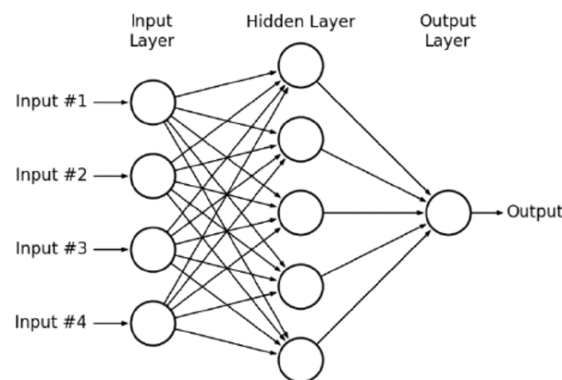

Fig3. MLP with a single hidden layer of hidden nodes.

With the introduction of extra hidden layers, the MLP had overcame one of the biggest limitations in the Perceptron - the capability of computing only linear separable problems [13] - which made it possible for the MLP to compute one of the most fundamental logic XOR gate [14], something which the conventional Perceptron (using monotonic activation functions) could not be trained to do. It is possible to train a perceptron with a single node to compute the XOR if a non-monotonic activation function is used, such as the Gaussian function [15].

Over the years, usage of more hidden layers have been shown to improve performance of many challenging problems [16].

ANN structures with hidden layers and nodes are also known today as deep neural networks and is the beating heart in the emerging and popular field of deep learning.

Next we will discuss one of the most popular supervised learning algorithms for deep neural networks, the backpropagation algorithm.

2.3 Supervised learning and Backpropagation

Supervised learning in the field of machine learning is a category of learning algorithms that require data sets of training examples of inputs and expected outputs. These algorithms typically works by gradually adjusting an internal data structure (e.g. an ANN) by comparing expected output with what is computed from the internal structure. This section will only focus on the notion of using supervised learning algorithms to train artificial neural networks, hence when referring to supervised learning we mean exactly those that train neural networks instead of other supervised learning techniques like regression or support vector machines (SVM).

Other interesting categories of learning algorithms that will be discussed in later sections of this thesis are; reinforcement learning, evolutionary algorithms, population based optimization, genetic algorithms and estimation of distribution (EDA) algorithms.

Supervised learning can be very well suited for pattern recognition and classification tasks where large datasets of input and output examples are available [17][18][19], but lacks the ability to adapt to change and cannot explore for new solutions. This is due to a limitation where neural network based supervised learning algorithms are usually entirely bounded by the quality and quantity of the training datasets used.

Let us now take a look at a well-known algorithm for training neural networks in supervised learning.

Backpropagation is a type of supervised learning algorithm that can be used to efficiently train deep neural networks (ANN with hidden nodes or layers). It works by first forward propagating network inputs through the network until output is obtained. It then compares the obtained output with the expected output from the training set to calculate an error gradient. Finally it backward propagates and adjust the network weight accordingly.

The backpropagation algorithm can be briefly described as follows

1. **Initialize random weights to the network**
2. **Feed input pattern $(x_1, x_2, \dots, x_n)$ and apply forward propagation until output is obtained.**

3. **Compare the obtained output with the expected output and calculate the error.**
4. **Backward propagate the error and calculate the gradient to adjust the weights for each layer.**
5. **Repeat from step 2 until the network output converges.**

The gradient that is discussed in step 4 above is the derivatives of the error in respect to the weights. For this reason, backpropagation is a gradient descent algorithm. To understand more of the mathematical backbone of backpropagation please consult these articles [13][6].

Since backpropagation is a type of gradient descent algorithm, it has the same limitation of getting stuck in local minima. The reason for this is that the gradient descent algorithm is designed to iteratively follow a gradient until convergence is reached. This means that if following this gradient leads to a local optimum, there is no way for the algorithm backtrack and find another optimum in the landscape. Nonetheless, multiple techniques has been proposed to address this limitation [20][21][22].

To summarize this section; supervised learning (in respect to training ANN) is a paradigm of learning algorithms that adjust ANN weights by comparing input and output examples with what is produced from the network until desired behavior is reached (typically by convergence). Backpropagation is a supervised learning algorithm that is well suited for several tasks but has a limitation of getting stuck in local minima.

In the next section we will look into another category of learning algorithm that is adaptive and capable of naturally avoiding local minima by design.

2.4 Reinforcement Learning

Reinforcement learning is a branch of machine learning that learns by evaluative feedback instead of instructive feedback like supervised learning.

The difference between evaluative feedback and instructive feedback learning is that in evaluative feedback learning, feedback is only provided to the learning system by evaluating on how well the system is performing regarding the environments and its states. Usually the evaluation is designed as a form of reward and value function that rewards the system depending on the set of actions taken by the system.

Instructive feedback learning is instead learning by giving exact information on how a learning system should behave, which often requires pre-existing knowledge of the problem domain. In the case of supervised learning it is for example crucial to

provide training data that depicts how the system should behave according to an input and output dataset.

Reinforcement learning is considered to be a more natural way to how human and animals learn. The basic idea is to learn through interaction with the environment by taking actions and receiving feedbacks for those actions, then over time figure out which set of actions gives the most optimal performance.

Consider a scenario where we learned how to not put our fingers into the fire. We would for example learn by first putting our fingers into a fire by chance, then observe that there is a sensation of pain, after which we would update our internal knowledge of the world to consider that fires are hot and is painful to touch. This will in turn make it less likely for us to touch the fire, as we most likely evaluated the feedback to be not so rewarding.

The scenario above describes how reinforcement learning typically learns the environment its associated actions and states. Below I will discuss some key concepts of reinforcement learning. For more detail on how reinforcement learning and associative algorithms works, consult a book by Richard S. Sutton and Andrew G. Barto [23].

In reinforcement learning some key elements that distinguishes it from other types of learning. These elements are the **policy**, **reward function**, **value function** and **model**. According to Richard S. Sutton and Andrew G. Barto [23] the policy is the most important aspect of a reinforcement learning agent because it defines the behavior of the agent, the other elements are only there to serve at improving the policy by maximizing both the short and the long term rewards for the agent.

As mentioned, the **policy** defines the actions of an agent at each state, from which it also defines the overall behavior of an agent in an environment. For example in the scenario of experiencing pain when putting finger into fire, the policy is updated after which the sensation of pain was perceived, this updated policy will make it less likely for the harmful action to be taken again. As with other learning algorithms, this policy update usually happens over several iterations based on some form of learning rate.

The policy also defines the ability of an agent to balance between exploitative and explorative search. This balance is very crucial for an agent to be able to maximize its reward in term of its long term goal, because too much exploitation will lead to finding only sub optimal solutions, while too much exploration may never lead to the most optimal state as the agent will keep exploring indefinitely.

It is important to understand that states and actions of an agent does not need to be a high-level representation, but can be as low-level as raw sensory inputs and motorized actuators of a robot [23].

The **reward function** is the first evaluative feedback function that will tell an agent if an immediate action in a specific state provides good or bad rewards. This function helps the agent to navigate the search locally to find short term rewards, the analogy to supervised learning is that the reward function can be considered as the gradient towards a local optimum.

The **value function** is similar to the reward function, but instead of providing short term feedback to the system, the value function provides an estimate of how rewarding it will be in the long run if an agent was to take a certain action for a given state. This function serves as a heuristic and helps the agent to update policies that can help the agent to reach optimal solutions or goals. The value function is important because it allows the agent to select an action that may look not so rewarding on the short term, but in the long run will be more rewarding because it may lead to consecutive high rewarding states.

The **model** element serves to represent the environment, it is important and beneficial for planning tasks. Some reinforcement learning algorithms – such as the Q-Learning algorithm – are model-free, meaning they do not utilize a model of the environment to improve their performance while learning. Other types of reinforcement learning algorithms instead rely on the model of the environment in order to learn efficiently [24].

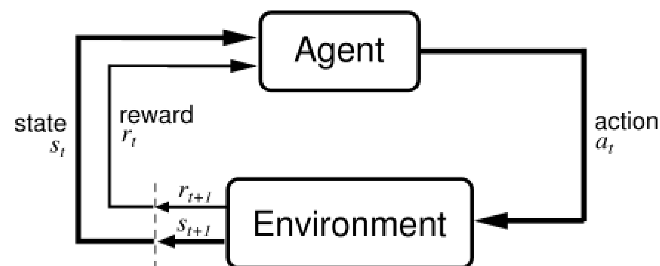

Fig4. Diagram showing the cycle of interaction between an agent and the environment in reinforcement learning.

Because this thesis focuses on the aspects of training and evolving neural networks, it is therefore essential to discuss how reinforcement learning algorithms may incorporate neural networks in learning. One of such algorithm that has been shown to be performing well in learning to play Atari games is a version of the Q-Learning [25] algorithm called Deep Q-Network [2][26]. But without diving in too deep, we will only discuss the basics of the Q-Learning algorithm and how to expand this idea to benefit from deep neural networks in reinforcement learning.

The **Q-Learning** algorithm in its simplistic form is presented below

1. Initialize the **Q(s, a)** for all state-action pairs, typically to 0
2. Observe the current state **s**
3. Select an action **a** that gives most utility based on and execute it
4. Receive the immediate reward **r(s, a)**
5. Observe the new state **s'**
6. Update **Q(s, a)** according to the Q-Learning update rule:

$$Q(s, a) = r(s, a) + \gamma * max_a{}'Q(s', a') \quad (7)$$

7. Set **s = s',** go to step 2

The most important step in Q-Learning is step 6, where the Q-Value is updated after which an action has been taken, then the implication of the Q-Learning algorithm is that given enough time it will eventually be able to reach convergence and derive an optimal policy [27]. This algorithm is also called an off-policy algorithm, because it updates the utility of a state-action based on the assumption of a greedy algorithm (maximizing for utility), while the actual policy that is to be derived is not a greedy algorithm, but rather try to find an overall optimal solution.

The problem to Q-Learning arises when the state-action space become too large, such as that of in a dynamic game world where the number of states and actions are practically unbounded, in this case storing all the updated **Q(s, a)** values will no longer be feasible. To address this problem one approach is to encode the Q-value function in a neural network where the inputs are the state and action while the output is the utility of the given state and action.

To incorporate neural network in encoding the Q-value function properly is by itself a challenging task, this thesis will not discuss in detail how this is done as there are several adaptations of this idea [28][2][26][29][30].

2.5 Population Based Optimization Algorithms

This section will step a bit away from specifically talking about learning algorithms to give a brief introduction on population based optimization algorithms, from which lay the foundation for some important algorithms used in this thesis. Furthermore we will discuss how population based optimization algorithms may be directly related to the learning algorithms used in this thesis and how it can be associated with other learning algorithms such as supervised and reinforcement learning.

Population based optimization algorithms are in this thesis referring to algorithms that take the advantage of a multi-agent environment to optimize multiple solutions across a population of agents. These algorithms are mostly derived from a group of algorithms called nature-inspired algorithms [31], from which they can further be divided into subsets of swam intelligence (SI), bio-inspired (BI), physics and chemistry based and a set of other but still nature inspired (i.e. based on social interaction models), for a comprehensive list of different algorithms see Iztok Fister Jr. et al. [31].

One advantage in using population based algorithm is that they provide a way to optimize problems when the landscape in which optimal solutions can be found are hard to define precisely. A second advantage is the ability to search multiple solutions at once by encoding possible solutions in a population of agents, this allows for implementation efficiency when implemented in computing systems that are capable of simulating agents in parallel (i.e. using computer graphics processing unit) [32].

According to Yang [33] swarm intelligence algorithms can be represented by the following inductive expression:

$$(x_1,\ x_2, \dots, x_n)^{t+1} = A\big((x_1^t, x_2^t, \dots, x_n^t); (p_1, p_2, \dots, p_k); (\epsilon_1, \epsilon_2, \dots, \epsilon_k)\big)(x_1,\ x_2, \dots, x_n)^t \tag{8}$$

where $A(\cdot)$ is an algorithm that takes three sets of parameters; a set of solutions $(x_1^t, x_2^t, \dots, x_n^t)$ at step *t*, a set of algorithm dependent parameters $(p_1, p_2, \dots, p_k)$ and a set of random variables $(\epsilon_1, \epsilon_2, \dots, \epsilon_k)$. The implication is that $A(\cdot)$ calculates an improvement from a population of existing solutions $(x_1,\ x_2, \dots, x_n)^t$ and generate new improved set of solutions $(x_1,\ x_2, \dots, x_n)^{t+1}$.

Since swarm intelligence as described by Yang reflects a population of potential solutions and can therefore be considered as population based, hence this thesis will assume equation (8) to be adequate when discussing population based algorithms.

Equation (8) is an inductive expression which implies that population based algorithms are applied iteratively until a criteria is met such as when sufficient solutions are found.

As mentioned above population based algorithms must be provided algorithm dependent parameters, as these parameters are crucial in determining the overall behavior of an algorithm as well as how will it performs. Because of this one of the most challenging problems in using population based algorithm is to find a set of parameters that is a sweet spot for such an algorithm, as this can be very difficult according to discussion by Yang [33].

The challenge of finding such a sweet spot in finding the correct set of parameters will later be discussed in more detail on how each parameter can affect the result of the experiments in this thesis.

Population based algorithms (as well as most nature inspired algorithms) belongs to a class of search algorithms called **population metaheuristics**, which are population based algorithms that does not directly search for heuristics for a specific problem, but rather search for heuristics on a higher level that are not dependent on any specific problem [34]. For this reason metaheuristic algorithms are extremely adaptive and can be applied to many optimization problems.

An example of a heuristic search algorithm that is problem dependent is for example the A* search algorithm [35], where the algorithm tries to minimize a cost function $f(n) = g(n) + h(n)$, from which the term $h(n)$ is a heuristic (estimated distance) guiding the search towards the target. A heuristic like this is specific for the A* algorithm as well as it can only be applied to problems that can be mapped to shortest (least cost) path search.

On the other hand a **metaheuristic** search algorithm will not be bounded to a specific problem (such as to find shortest paths), but will instead search in the solution space using a form of evaluative function to evaluate how well a solution (or a set of solutions) performs. Because of the lack of problem dependent heuristics (which behaves as a guide), the only way to generate new and potential better solutions are to introduce random variables to create or modify existing solutions by random for evaluation.

It is also important to mention that the only parts in a metaheuristic search algorithm that may be problem dependent is the evaluative function and the representation (encoding) of solutions, similarly as described by Darrell Whitley for genetic algorithms [36].

In order to prevent any metaheuristic search algorithm to only generate useful solutions instead of a total random set of solutions, it is important to control these random variables by a set of parameters, such that existing solutions can be improved progressively over time. These are the algorithmic dependent parameters mentioned above.

The element of randomness also makes it possible to generate new and potentially better solutions than by just improving on existing solutions which can lead to a local optimum. What this implies is that metaheuristic search algorithms usually have the two elements of exploitation and exploration.

An analogy for this is to imagine a situation where an animal tries to locate a good food source; it can do this by either searching randomly at nearby locations or to travel to faraway lands. If the animal was to only search at nearby locations then it may be able to find a food source, but if this food source is good and can last for a long time is not known. The only way for this animal to know is to search farther

away as there can exist potential better food sources out there. What this animal will need then is to balance between searching nearby (local) and searching far away (global) so that it can find good food sources as well as not wander itself to death.

The elements of local search and global search is often referred to as the ability to exploit and explore, and controlling the balance between these 2 elements in metaheuristic search can be very difficult [33].

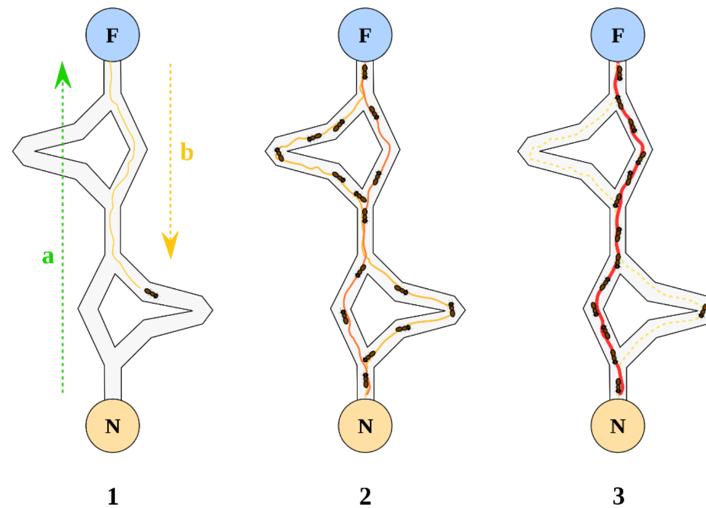

Fig5. Illustration of Ant Colony Optimization simulating how population of ants can over time to find the shortest path.

Population based algorithms take the advantage of metaheuristic search as well as speeding up the search by implementing not only a single solution but multiple solutions called a population of solutions. Several population based algorithms also implement a mechanic where solutions within the population can interact and share information, such as the firefly algorithm [37], ant colony optimization (fig5) [38] and genetic algorithm [36]. This information sharing mechanism is used to combine existing solutions with each other in order to generate new and potentially better solutions. Without this mechanism each solution will just behaves as if they were independent single solution search and may cause overlapping search, which in turn leads to inefficiency as each solution does not inform other solutions of what solutions already exist.

The mechanism of information sharing is analogue to how each individual in a society may inform and share knowledge between each other so that the total knowledge of the entire society improves as a whole.

To summarize this section, we have looked at the fundamental concepts of population based optimization algorithms, which in turn are mostly inspired from nature. These algorithms can be used over a wide range of problems, but challenge

the user in setting a set of algorithm dependent parameters where parts are to find a balance between exploitation and exploration.

Next we will discuss genetic algorithm, which is also considered as a nature inspired algorithm, but since the concepts in genetic algorithms are essential to this work, it deserves a dedicated section.

2.6 Genetic Algorithms

Genetic algorithms are population based search algorithms inspired by evolution theories, natural selection and genetics [39]. As any population based algorithms, genetic algorithms also have some of the same advantages and challenges such as being capable of evolving multiple solutions, perform localized and global search (exploitation vs exploration), but also inherit the challenges of parameter setting and balancing between exploitation and exploration.

In the case of optimization for genetic algorithms, solutions are often called to be evolving through generations to optimize for better solutions. This is analogue to biological evolution where organisms evolve through generations to adapt and become better at surviving in their environments.

Previously when discussing population based algorithms, two important elements were mentioned that usually are problem dependent; the evaluation function and the encoding of solutions. In genetic algorithms evaluation functions are canonically called fitness functions, these functions are to evaluate how fit a solution (or a population of solutions) is in each generation, while the encoding of solutions are called genomes [40] or chromosomes [41], and behaves just like how biological genomes in organisms encodes their genetic traits.

Genomes in genetic algorithms are usually represented by a string or a sequence of information (conventionally a sequence of bits), and can be modified by mutation and crossover operators. Let us take a look at some important elements in genetic algorithms, for more detailed description please see [36][39][42][41].

Encoding of solutions can be done in several representations, one of such representation is by representing a solution in a sequence or string of bits effectively forming a binary sequence.

Using binary sequence is analogue to how data structures are stored within a computer, and since any piece of data are naturally stored in binary format it does not require any complex conversion or transformation other than casting the data representing a solution into bits, an operator that is often natively implemented in most modern programming languages.

A problem in using raw binary data in such a naïve way as described above is that such a representation cannot be easily manipulated by the way of how genetic algorithms modify (mutate) and recombine solutions (crossover). Any binary sequence representing any meaningful piece of data within a computer must follow certain structure for the type of data it represents (e.g. a data object, integer or string), it is therefore difficult to slice bits and pieces from one binary sequence and merge with another without corrupting the underlying data structure.

A concrete example is for any given a binary sequence representing a text string, it is not possible to cut, slice or change bits within the sequence at any arbitrary location and still get a meaningful text string in return. The reason for this is because any literal symbol in a computer is usually represented by a "byte" (8 bits). To make meaningful manipulation of text strings, groups of 8 bits must be consider as a single smallest unit in the sequence.

As a conclusion, when choosing an encoding for solutions that would allow genetic mutation and crossover, it is important to design the representation carefully in such a way that it is possible to modify and cut **genes** in the genome without corrupting the underlying representation. For example in the algorithms used in this thesis, genes are represented as a list of data objects representing neural network connections and nodes.

Mutation is the process at which genomes of solutions are modified, usually by modifying the genes by some random about. The number and probability of genes that are selected to be modified can be set by the parameter of the algorithm.

Mutation is in fact the mechanism in genetic algorithms that allows both local and global search leading to support the capability of optimizing and finding new solutions. It is therefore crucial to implement a mutation operator that allows for balance between exploitation and exploration. One can for example use a non-uniform distribution such as the Lévy-flight such as used in the Cuckoo Search algorithm [43] for random mutation to control the distribution between of local and global search (small and big jumps).

By randomizing genes in a genome is only one of several ways to mutate, as it is usually up to the design of individual algorithms that decides what kind of manipulations are possible for modifying genomes. The main idea of mutation is still clear, it is used to modify genomes in hope for finding better solutions.

Crossover in genetic algorithm is based on the idea of biological counterpart where organisms crosses their genes when creating offspring. This mechanism allows for preserving genetic traits which in turn may preserve genetic traits of promising solutions, just like how fit biological organisms preserves their genes by having their offspring inheriting their genes.

Since crossover only preserves genes from parent genomes, this further implies that the main searching mechanism in genetic algorithms are by mutation to both

optimize and find new solutions and by crossover to preserve and to combine existing solutions in hope for better ones. Supported by the **building block hypothesis,** crossover also behaves as a guide for genetic algorithms in searching for better solution more efficiently instead of trying every single combination using only the mutation operator.

As with the mutation operator, crossover can also be implemented in several ways, some of the most common are **k-point crossover** and **uniform crossover** [42]. For different problems and applications, different crossover operators may be needed to allow for meaningful recombination of genes. What this means is that for genetic algorithms, the implementation of mutation and crossover operators may be problem dependent for certain types of problems, which also usually depends on the encoding method used.

The **building block hypothesis** mentioned earlier refers to the a hypothesis supported by the schema theorem [44] that small genes (low-order schema) which can provide to increase in fitness have a higher chance in surviving through crossover and mutation and can therefore be recombined with other small fit genes in order to construct even more better set of genes. But the effect of whether this hypothesis holds for every generation with complex genomes is still debatable and hard to prove [44][45]. Nonetheless genetic algorithm is still one of the most popular optimization algorithms used today with several adaptations, one of such an adaptation is the NEAT algorithm that will later be discussed in another section.

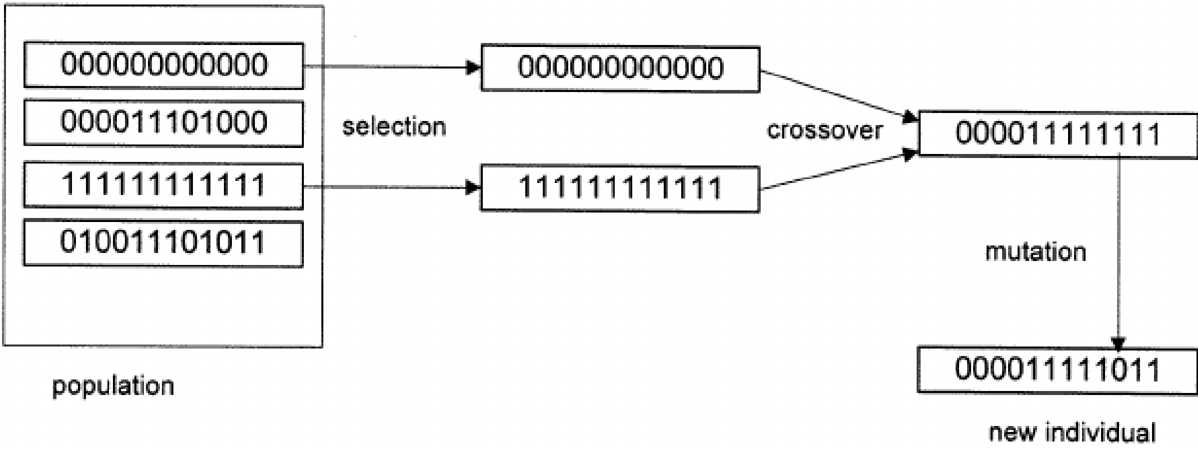

Fig6. Diagram of genetic algorithm showing the steps of selection, crossover and mutation.

The genetic algorithm can generally be summarized as follow:

1. **Initialization** – The first step is to create an initial population of solutions, this is usually done by creating solutions randomly over the solution space, but specific knowledge about the solution space can also be used to initialize better solutions to speed up the search.

2. **Evaluation** – Evaluation step is where solutions are evaluated for fitness using a fitness function. This information is important for the next steps.
3. **Selection** – At this step, solutions are to be selected to create offsprings for the next generation. How solutions are selected can vary, but the idea is to somehow select solutions in such a way that good traits of fit solutions are can be preserved (i.e. the block hypothesis). It is also important to not select only the top solutions, because some of the other genes of the less fit solutions may contribute to create better solutions later, preserving variety also needs to be considered.
4. **Recombination** – This is the crossover step, solutions selected from previous step can now be recombined with each other to create new offsprings. The number of parent solutions used to recombine offspring solutions can vary depending on the adaptation of the genetic algorithm, but commonly two parents are selected to create new offsprings.
5. **Mutation** – As discussed previously, the mutation operator plays an important role in both optimization and also searching for new solutions. This step is therefore dedicated to mutating offspring solutions created from previous step. In order to not destroy good genes from parent solutions, it's important to consider how intensive (frequency and amount) mutation should occur on each offspring solution. If an offspring always gets too much mutation, then the chances that it will converge to an optimal solution is rather scarce. The effect of this is as if pure random walk is used to find optimal solutions, something which destroys the purpose of the crossover operator.
6. **Replacement** – Once offsprings have been created with through crossover and mutation, the entire population should be replaced with the newly created offsprings. This steps epochs the population into a new generation. Not all adaptations of genetic algorithms would replace an entire population with a new generation of offsprings, some adaptions would gradually replace solutions within a population, this would for example allows for real time evolution [4].
7. Go to step 2 if termination condition is not met, otherwise terminate.

2.7 Neuro Evolution

Neuroevolution is a machine learning technique that utilizes evolutionary algorithms (e.g. genetic algorithms) to evolve artificial neural networks. The idea is to utilize the power and flexibility (as discussed earlier) of evolutionary algorithms to optimize for optimal neural network weights and structures. Neuroevolution are commonly used to solve reinforcement learning tasks [46]. Even though it lacks the evaluation of direct interactions between agents and environment as required by typical

reinforcement learning algorithms, but with carefully designed fitness functions neuroevolution can perform as well [3] as reinforcement learning techniques [2].

In principle, any metaheuristic optimization algorithms can be used to optimize neural networks (as long as appropriate search operators are implemented), but neuroevolution is mostly associated with evolutionary algorithms, this association is intuitive since both neural networks and evolutionary algorithms takes inspiration from nature. Nevertheless it is worth to keep in mind that other optimization techniques (or a combination of techniques) can be used to evolve neural networks, e.g. [47].

To put this into perspective, this thesis considers neuroevolution to be made up of two components:

1. A metaheuristic (usually population based such as GA) optimization algorithm, used to evolve a population of neural networks (solutions).
2. Neural networks and related encoding and manipulation operators (e.g. mutation and crossover) used by the optimization algorithm.

Besides the two distinct components mentioned above, problem dependent fitness functions for different optimization scenarios are required as well.

Notice that the above interpretation of neuroevolution is quite ambiguous as it does not narrow on any specific algorithms, because the purpose is to attempt to simplify the understanding of the modular components in neuroevolution on a macroscopic level.

Encoding of neural networks is about representing the structure of a neural network in such a way that it is possible to apply manipulation operator while maintaining the functionality of neural networks as a whole. For example fixed neural network structures can be encodes as a vector of weights [48], and evolution of each vector (network) is driven by randomly mutating the weight values within the vector. A population of weight vectors will be evaluated in each generation using a fitness function to assign corresponding fitness to each network.

According to a review by Yao [49], some more complex encoding schemes can be used to encode not only the weights of but also the topology and transfer functions of a network, other techniques to encode the transfer functions to evolve heterogeneous networks had also been studied [50]. What is to be encoded for a neural network usually depends on the objective of the application or experiment, but for some problems it is worth considering whether to use indirect or direct encoding scheme.

Direct encoding is an encoding scheme where the relevant structures of a neural network can be directly mapped to the encoding and vice versa (isomorphic). The mapping assumes that whatever that is encoded are all that is needed to directly represent a network. For example a direct encoding of a fixed topology network may

be a fixed length vector of weight values corresponding to the different connections in the network, meaning that it is possible to directly translate a network between encoding and network structure consistently.

A problem with direct encoding is the length of the encoding string will grow in proportion to the number of encoded elements in a network. This is not a problem for small neural networks but can quickly become an issue when neural networks have the ability to grow bigger (i.e. changing topology). For certain task it is fundamentally required that the neural networks used must be big, e.g. consider the work by Matthew et al. [3] where raw screen pixels are fed into the inputs of a neural network that is to be evolved for playing Atari games. This is where indirect encoding start to show some promising properties.

Indirect encoding alleviates the issue with direct encoding where the encoded string can grow and become too big that can cause performance problems when mutation and crossover are applied. The idea to indirect encoding is that instead of representing a network structure as exact as possible, it is possible to be represented by a set of rules that can be used to construct a neural network. These rules can be used to generate any relevant part of an ANN, i.e. the weights, topology and transfer functions.

For example the ES-HyperNEAT algorithm [51] indirectly encodes not just the connection weights but also the density and connections within a complex neural network called a substrate. This algorithm is based in HyperNEAT [52] and therefore evolves convolutional neural networks (CNNs) to generate structural patterns of large scale ANNs. The encoding of the CNNs utilizes direct encoding while the behavior of the CNNs indirectly encodes the actual ANN structures that are to be evaluated.

Once an encoding scheme has been decided then the next step is to design evolving mechanisms. These mechanisms are manipulation operators that takes a genome and apply modifications to it to create new genomes. Manipulation can be on the weights, topology, transfer functions or even other properties such as learning rules for dynamic neural plasticity [53][54]. In genetic algorithms for example, manipulation operators are the mutation and crossover operators that were previously discussed.

As evolution carries on and crossovers are applied, one of the questionable issues that arises with neuroevolution is what is known as the **competing convention problem** [55] also known as the **permutations problem** [56].

The competing convention problem refers to a problem that occurs when a naïve recombination operator is applied to genomes, e.g. by using single point crossover. This is because some recombination operators do not take into consideration the topological ordering of neural network structures. What this means is that e.g. the single point crossover operator assumes that each gene in a genome uniquely

contributes to the overall behavior of that genome, therefore by cutting a genome and combine with another should produce new genome that should inherit some of the properties its parents. But this is not the case for recombination of neural networks as different ordering of genes can exhibit the same behavior for distinct genomes, this makes it challenging to know how to cut and recombine genes so that offspring genomes will be corrupted.

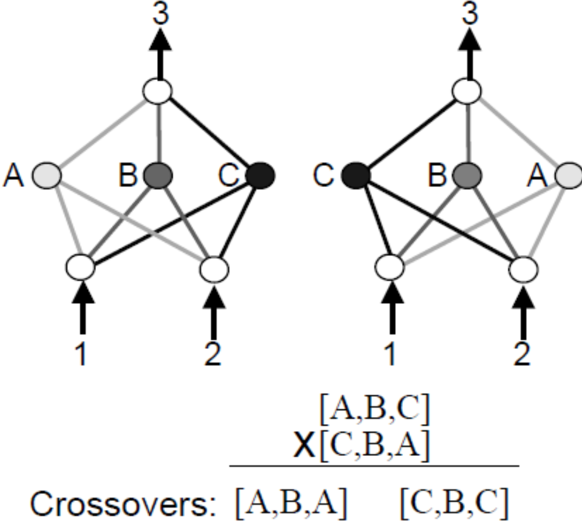

Fig7. Two genomes of exact same topological structure but breeds corrupted children using single point crossover operator.

Consider (fig7), two different genomes may have the exact same topological structure and weights, but when a simple crossover operator is used corrupted children are created with duplicated neuron genes A and C. A better recombination operator is therefore needed to combine neural network structures such that genomes with similar structures also preserves their structure when creating offsprings, otherwise this can lead to a performance impact when searching for solutions. The competing convention is addressed in the next section introducing the Neuro Evolution of Augmenting Topologies algorithm.

2.8 Neuro Evolution of Augmenting Topologies (NEAT)

This section will be dedicated to introducing the Neuro Evolution of Augmenting Topologies (NEAT) algorithm, most of the information presented here will be largely based on the work of Kenneth O. Stanley and Risto Miikkulainen [57], review their work for more detail. This section will only summarize key points of the NEAT algorithm.

NEAT is a neuroevolution technique which is implemented as a population based genetic algorithm. NEAT addresses challenges such as how to consistently evolving both the weight and topology of neural networks as well as dealing with the competing convention problem in neuroevolution. NEAT also utilizes the mechanism of speciation, also known as **niching** [58][59], to protect innovation and allow diversity in a population.

Encoding in NEAT is implemented using direct encoding, each genome contains is a list of connection and node objects called connection and node genes. Each connection gene encapsulates the incoming node, outgoing node, connection weight, an innovation number and a flag indicating whether that connection is enabled or disabled. Only enabled connections will be used when constructing neural networks. Node genes simply encodes if a node is an input, hidden or an output node.

The **innovation number** encoded within a connection gene is a unique number assigned to each new connection innovation in the population of genomes. Innovation number is globally tracked throughout evolution for all genomes so that genes representing the same topological structure gets the same innovation number. This concept is called **historical marking**, which historically marks all innovations uniquely to track all the distinct genes within the entire population.

Historical marking is essential to keep track of what genes are compatible with each other, as NEAT utilizes this marking to apply crossover consistently and avoid the problem of competing convention. Because historical marking makes it possible to identify exactly which genes are of the same innovation, crossover can now align genes properly and not produce faulty offsprings.

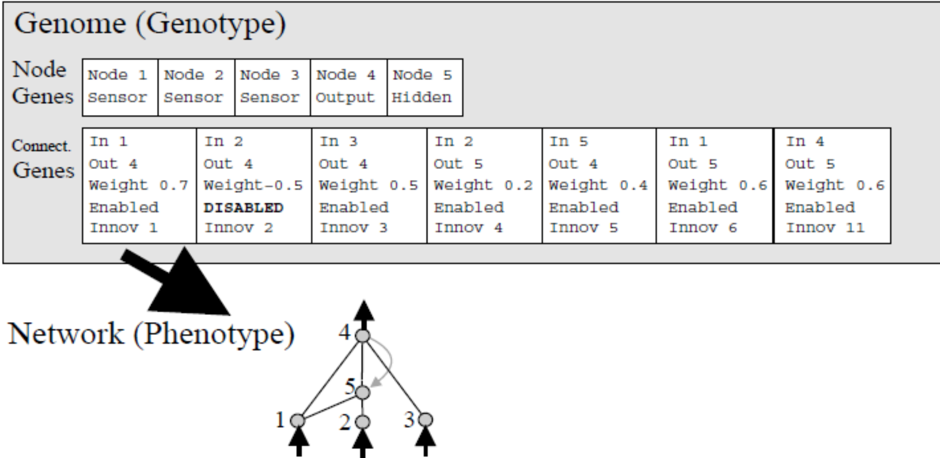

Fig8. Encoding of a genome in NEAT.

As mentioned, NEAT can evolve both the weights and topology of a neural network. Weight can be evolved by either assigning random weight or by perturbing the

existing weight by a small amount. The probability for randomizing and perturbing weight are usually controlled by global parameters.

What is special about NEAT is the ability to augment topology, what this means is that NEAT can add topological structure to an existing neural network to make it more complex over time. In addition NEAT can also disable existing connections which makes it possible to remove faulty connections.

The two main topological innovative mutations in NEAT are (fig9):

1. **Mutate add connection** – Adds a connection to existing unconnected nodes
2. **Mutate add node** – Adds a new node between an existing connection, this is done by first selecting an existing connection, disable it, add a new node, create connections to bridge the gap from the disabled connection. Connection weight of the disabled connection is maintained in one of the new connections while the remaining new connection gets a weight of 1, this is to minimize disruption to the functionality of the old connection.

Crossover is done by first aligning genes between two genomes using information from historical marking, offspring genes are then inherited depending on which parent is more fit. If both parents are equally fit then genes can be randomly inherited (fig10).

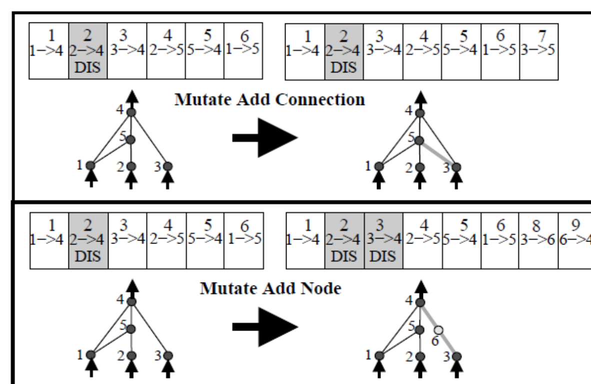

Fig9. Mutate Add Connection and Mutate Add Node.

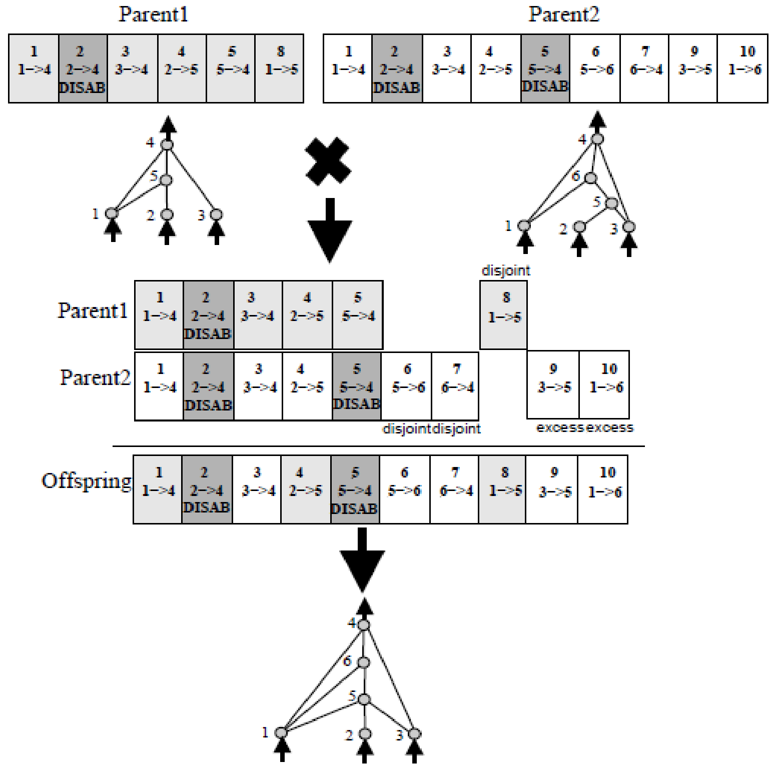

Fig10. Crossover of genomes utilizes historical marking to match and align genes before creating offsprings.

Because NEAT allows for topological innovations, newly created offsprings can suffer from fitness loss because of recent augmented structures that does not yet have time to optimize. Fitness loss can destroy innovation because a network with an important innovation can be removed from the population too early before it gets a chance to catch up with the rest. This is why NEAT utilizes the concept of speciation to protect innovation.

Speciation divides the population into species of similar topology, this is done by aligning and comparing genes with each other using historical marking. Genomes that are too different from each other will be put into different species, while similar genomes will be assigned to the same species. The distance (difference) between genomes are calculated using the formula in equation (9).

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W} \tag{9}$$

E and *D* are the number of excess and disjoint genes, *E*+*D* makes the total number of different genes, $\overline{W}$ is the average weight differences, the coefficients c_1, c_{21} and c_3 are constants adjusting the importance of each term, finally *N* is the number of genes in the largest genome. Distance δ is then compared with a threshold value δ_t, if it is within this threshold then both genomes will be put into the same species. As the number of species may grow over time, it is suggested to adjust δ_t to maintain a somewhat stable number of species throughout evolution.

Genomes that belong to the same species share their fitness, what this means is that every genome within the same species will have their raw fitness normalized by the number of genomes within that same species. This allows for genomes in small species (e.g. young species with innovative genomes) to have a chance to compete with genomes in larger species, because genomes in larger species will have their fitness diminished by the larger number of genomes in that species. The fitness sharing function for genome f_i' for genome i is specified as follow:

$$f_i' = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))} \tag{10}$$

where $\delta(i,j)$ is the distance between genome i and j, $sh(\cdot)$ is set to 1 if the distance $\delta(i,j)$ is within the threshold δ_t, otherwise it is set to 0, this means that $sh(\cdot)$ is the number of genomes in the same species. f_i is the raw fitness of genome i, and n is the number of genomes within the entire population.

Genomes in the population of NEAT are assigned to species according to the following steps [4]:

```
The Genome Loop:
  Take the next genome g from population P
  The Species Loop:
    If all species in S have been checked,
      create new species s_new and place g in it
    Else
      Get the next species s from S
      If g is compatible with s, add g to s
    If g has not been placed,
      continue the Species Loop
    Else exit the Species Loop
  If not all genomes in G have been placed,
    continue the Genome Loop
  Else exit the Genome Loop
```

Fig11. The genome loop that assigns genomes to species in NEAT.

The NEAT algorithm generally starts with a population of primitive genomes (i.e. genomes with only input and output nodes) and add more complex topology as evolution carries on. This allows for **dimensionality reduction**. What this means is that since topology are slowly added over time, the population will be able to search for solutions in a smaller dimension incrementally. This effectively reduces the dimensionality of the search which makes it possible for the algorithm to find compact solutions. Yet a problem may arise if the parameters selected for structural

mutations are too frequent, then NEAT may not be able to find compact solutions because it will not have enough time to optimize for the weight values (i.e. perform search in weight space). Because it is not always clear which parameters are suitable for topology mutation, synaptic pruning may be a solution to counter this problem [60][61].

Since the first proposal of the NEAT technique, several adaptations have been created that have shown great promises in using neuroevolution for solving different kind of tasks [62][63][3][64].

Next we will look into one of the adaptations of NEAT called rtNEAT, which is implemented in the work of this thesis.

2.9 Real Time NEAT (rtNEAT)

Real time NEAT (rtNEAT) is an adaptation of the original NEAT algorithm that allows for evolution in real time. It was originally developed and used in the neuroevolution video game NERO [4].

The algorithm was developed to demonstrate that it is possible to use neuroevolution in real time video game environments where team of agents can be trained and learn to solve different tasks in real time (online evolution).

The idea of rtNEAT is to evaluate and replace worst performing agents with offsprings one at a time instead of replacing an entire generation of agents with offsprings to produce the next, since the process of replacing the entire population of agents can be very costly and is not desirable in interactive environments such as video games. When replacing agent(s), it is important to replace the worst performing agent(s) based on their adjusted fitness, otherwise the effect of speciation will be destroyed.

Agents in rtNEAT are also assigned minimum time to stay alive, this allows newborn agents to have time to adapt and optimize to the environment. Without this minimum lifespan agents may get replaced and destroyed too quickly before they can prove for themselves. This is typically important in a dynamic environment (i.e. a game) as fitness of agents can only be evaluated over time instead of over discrete generations.

Because this thesis implements the rtNEAT algorithm, the steps for the main loop of the rtNEAT algorithm are presented below for a deeper understanding on how it works.

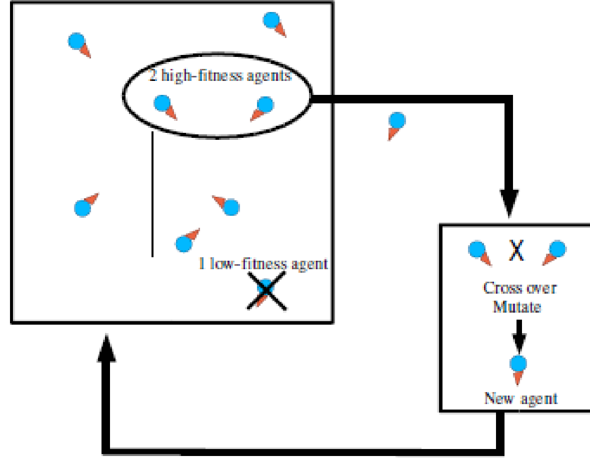

Fig12. Illustration of rtNEAT reproduction cycle.

The rtNEAT Algorithm:

1. Calculate the adjusted fitness f_i' for every individual i from the population using equation (10). This step prepares the fitness sharing for the next step to preserve speciation dynamics.
2. Remove the worst performing agent with lowest adjusted fitness. When removing agent, consideration on how long an agent has been alive must be accounted for. Because agents in rtNEAT and created and destroyed continuously, it is needed to assign to each agent a minimum timer in order to track which agent have been alive in sufficient amount of time for fitness evaluation. Agents that have not been alive long enough need a chance to be evaluated for fitness before they can be considered for removal. The minimum timer is a parameter m and can be set experimentally depending on the task, as some tasks requires more or less time for evaluation of fitness. Using parameter m, population size |P| and specifying the percentage I of the population that are ineligible for removal and can't be replaced by offsprings, it is possible define the number of ticks n between replacements as follow:

$$n = \frac{m}{|P|I} \tag{11}$$

3. Each species in rtNEAT is assigned an average fitness $\bar{F}$, at this step it is necessary to re-estimate $\bar{F}$ as this is used for selecting parent species in the next step to produce offspring. The average fitness $\bar{F}$ needs to be re-estimated at this step because an agent has been removed from the previous step.
4. At this step a parent species is selected to create an offspring. In the offline version NEAT, the number of offsprings created per species is proportional to the average fitness $\bar{F}$ of individual species. But since rtNEAT only create one

offspring at a time, parent species are therefore selected by a probability also proportional to the average fitness of each species:

$$Pr(S_k) = \frac{\overline{F_k}}{\overline{F_{tot}}} \tag{12}$$

where $\overline{F_k}$ is the average fitness of species k and $\overline{F_{tot}}$ is the total average fitness of all species in the population. Once a parent species is created, two individuals from this species are selected to combine and create a new offspring individual.

5. At this stage the newly created individual must be assigned a species. Intuitively the Genome Loop (fig11) could be run each time a new individual is born, but as this process can have an impact on the performance of rtNEAT it is not always necessary to completely reassign all individuals in the population. Depending on the task environment, it may just be enough to assign the newly created individual to the same species as its parents, and the Genome Loop can be run once every few replacements. This can save computing power as rtNEAT was designed for real time environments, meaning for a program to run at 30 frames a second there's only 0.03 seconds available for all the calculations.
6. Lastly is to connect the newborn neural network to an existing agent in the environment, this can for example be done by separating the implementation of the visible agent from the brain (neural network) by making them modular. As long as neural networks can be replaced for the same agent then it is perfectly compatible with how rtNEAT is intended to work, because one of the main goals of rtNEAT is to seamlessly integrate neuroevolution into interactive environments.

To summarize, rtNEAT is based largely on the original offline version NEAT with some modification to allow for real time (online) evolution. One of the main mechanism that allows for this is the modification of parent species selection using equation (12), this allows for the speciation dynamics which is one of the essential features of NEAT.

2.10 Exploration vs Exploitation

Because exploration vs exploitation plays an important role in the research questions of this thesis, it is therefore necessary to look at what importance exploration and exploitation have for evolutionary algorithms, as well as why it is an important topic in regard to general optimization.

This section will be largely based on discussions from a comprehensive survey regarding exploration and exploitation in evolutionary algorithms by Črepinšek et al. [65], as well as one of the earlier discussions on the topic by Eiben and Schippers [66].

In order to become familiar with the concept of exploration vs exploitation, let's begin by laying down some common understandings around the topic that will be assumed in this thesis. In fact, several general concepts around this topic will also be based on ideas and observations from the work by Mehlhorn et al. [67].

Exploration is commonly understood as the behavior of organism in searching for new areas and locations that may in the long run be rewarding. For example how the human race have historically explored the surface of the earth to discover new lands and eventually led to acquisition of both new knowledge and resources. Exploration is essential for survival, i.e. it allows for finding new habitable locations once resources in the current known locations have been depleted. Exploration may also looked at in the perspective of evolution where organisms have explored ways to adapt to existing environment through mutation. For instance there is a breed of trees called Sequoias that have found a way to survive forest fire by utilizing the heat to crack their cones to seed the earth. In addition to this the Sequoias also somehow adapted to the fact that fire would kill other competing tree breeds and their ashes would further fertilize Sequoias seeds. The adaptation of Sequoias are quite exceptional as fire is generally considered as purely destructive, but nature still have found its way through the ashes and flames to create new life.

So exploration seems to be generally about searching for new ways or places in order to survive, because survival is the single witness to successful evolution according to the general understanding of evolution; the fittest will live. But it is necessary to keep in mind that with pure exploration there would be no beneficial effect, because it means to constantly moving or changing which often comes with a cost, i.e. in the form of energy burned. This implies that in order to survive, there must also be some countering mechanism that would stop exploration to save and gather energy (perhaps for further exploration). This brings us to the idea of exploitation.

Exploitation is an opposite cornerstone to exploration [66], because exploitation is not about discovering new land or adapting new ways to survival, but is instead about utilizing existing discoveries to maximize potential rewards. For example humming birds might first explore to find a field of flowers, but once found, they would choose a specific flower patch and settle to feed. In other words, humming birds would exploit the potential of newly discovered sources of nutrient by sticking to a specific flower patch and would only switch once that nutrient potential is depleted. Another example is how men have discovered new continents and eventually settled on those. This is because discoveries of new continents have provided opportunities for better lives, and once these opportunities were known, it was naturally to exploit the potential they were estimated to benefit.

If the idea of exploitation and exploration seems to be countering each other, then how can they co-exist as well as why does it seems like this is a must for survival? Furthermore how can for example the behavior of exploration suddenly change to the behavior of exploitation as in the case of the humming birds?

Exploration vs exploitation discusses the notion of why two seemingly incompatible behaviors are important for survival of organisms but as well as effective search in optimization algorithms. Discussions regarding the benefits and challenges of having exploration and exploitation as features in optimization algorithms will be left for later. Let us first look at some of the issues that arise when exploration and exploitation are often perceived as two distinct and often mutually exclusive behaviors.

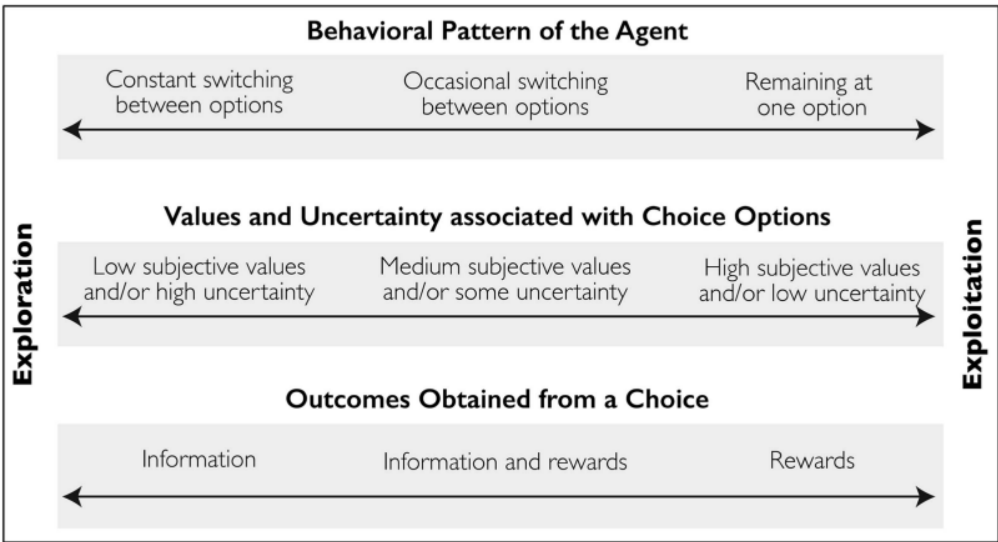

Fig13. Illustration showing behavioral pattern of exploration and exploitation as a continuum.

The idea of exploration and exploitation being mutually exclusive means that these two behaviors cannot be exhibited at the same time in decision making of organisms. In other words if an organism is exploring, it cannot exploit and vice versa. But Mehlhorn et al. [67] argues that exploration and exploitation can in fact be a continuum instead of a binary trade-off model (i.e. explore and exploit are mutually exclusive). Additionally behaviors can seem to be exploring in one dimension but might as well be an exploiting behavior in another dimension.

Viewing exploration and exploitation as a continuum gives room for modelling exploration and exploitation models where both can co-exist and have different kind of transitions in between.

Imagine for example a population of men that have never seen the value to gold. At first there may be a handful individuals who would accidently stumble upon gold

ingots. As these individuals bring home the metal, it would attract more and more people to explore for gold as they start to see the value in it. This can be interpreted as exploitation gradually become exploration in the perspective of the population as a whole (i.e. the population was exploiting whatever was valuable to them until they saw a metal of great value).

Exploration and exploitation can also be difficult to define because depending on the dimension of interpretation exploration can be exploitation and vice versa. For example a group of humming birds that explore a field of flowers can be looked at as an exploiting behavior if we consider the whole field of flower as a single source. Likewise on the level of individual flowers, when a bird is exploiting a single patch of flowers can also be looked at as exploring as they jump between individual flowers. This is referred to as **spatial scale** according to Mehlhorn et al. [67], because the dependency on scale of space defines the notion of exploitation and exploration. Time can also play a role in defining exploitation and exploration, in this case it is referred to as **temporal scale**. Generalizing this we can imagine that there may be several other scales that could affect the definition of exploration and exploitation behaviors.

Let us now move away from the general discussion of exploitation and exploration models and look at how this topic also influences the design of evolutionary algorithms.

Exploration vs exploitation in evolutionary algorithms have been discussed in several studies, but not many have attempted to lay down a common ground for researchers to navigate in the field, according to Črepinšek et al. [65]. For this reason they had put together a survey to discuss common issues, misconceptions and challenges regarding exploration and exploitation in evolutionary algorithms, from which this thesis will summarize key ideas.

Exploration and exploitation in neuroevolution have mostly been concerned with how to concentrate and diversify a population of solutions, in order to find a global optimum. This seemingly shows the need for controlling balance between exploration and exploitation in order to maintain a balanced global and local search in evolutionary algorithms. Supporting this, Črepinšek et al. [65] argues that more research is needed in order to understand more on how different factors in evolutionary algorithms may affect the ability of intelligent systems to explore and exploit:

- **Defining phases of exploration and exploitation.** As discussed previously, defining when exploration or exploitation occurs can be tricky as there seems to be no thin red line dividing them.

- **Which parts of evolutionary algorithms contribute to exploration and exploitation?** Since evolutionary algorithms consist of many parts that can contribute to the behavior of search, i.e. mutation, crossover and selection

operators can all contribute to what and how the search space is explored. At some level an operator appear to contribute to exploration, but at another level the exact same operator could also be contributing to exploitation.

- **How balance between exploration and exploitation can be achieved.** Several algorithms have control parameters, how do we know which parameters contribute to exploration and exploitation? Many parameters seems to be set by the user through trial and error, how can we decide a sweet spot that would achieve the balance between exploration and exploitation?

- **When to control the balance between exploration and exploitation.** As exploration phases to exploitation to find optimal peaks, how can we define when to phase between exploration and exploitation? Should exploration and exploitation occur simultaneously?

- **How to control the balance between exploration and exploitation.** Controlling the balance between exploration and exploitation means to be able to identify and control the moving parts of an evolutionary. For instance one technique may be to measure and control diversity as diversity is often regarded as a property that contributes to exploration. How can we identify those moving parts as well as controlling them in order to get the result we need?

- **How to measure exploration and exploitation.** Finally how can we measure if a system is exploring or exploiting, as this can be critical in maintaining the balance between these two phases.

The list above summarizes some of the elements that need to be considered when designing algorithms to achieve a balance between exploration and exploitation, which will hopefully help improving the performance of evolutionary algorithms.

This is a rather longer section discussing the difficulties in identifying and controlling explorative and exploitative behaviors of intelligent systems using i.e. evolutionary techniques. But this is clearly an important task to understand these challenges in order to design better techniques (e.g. metaheuristic algorithms) to solve reinforcement learning problems.

This thesis attempts to test some of the mentioned concepts regarding exploration vs exploitation. Particularly on how mutation parameters contribute to exploration and exploitation.

2.11 Estimation of Distribution Algorithms (EDA)

Because the work in this thesis implements a simplistic EDA, it is therefore supplementary to give a brief introduction to this type of stochastic population based algorithm known as Estimation of Distribution Algorithm (EDA), originally introduced by Mühlenbein and Paass [68].

EDAs are stochastic algorithms that works by generating new, hopefully better solutions based on a probabilistic model built from a set of promising solutions drawn from an existing population.

A drawback to EDA is perhaps the fact that new solutions are solely based on existing solutions which might lead to early convergence. One of several methods to solve this is for example by combining EDA with other algorithms [69][70] to expand the search bound in solution space, which in turn helps escaping local optima in large and deceptive fitness landscapes. In this thesis an adaptation of EDA will be combined with rtNEAT to test if it can speed up the process of finding optimal solutions.

The general EDA algorithm is presented below [69]:

1. P ⇐ Initialize the population
2. Evaluate the initial population
3. **while** *iter_number* ≤ *Max_iterations* **do**
4. P_s ⇐ Select the top s individuals from P
5. M ⇐ Estimate a new Model from P_s
6. P_n ⇐ Sample n individuals from M
7. Evaluate P_n
8. P ⇐ Select n individuals from $P \cup P_n$
9. *iter_number* = *iter_number* + 1
10. end while

Step 5 in the EDA algorithm is the most important part, as it builds a probabilistic model from top performing solutions. The way to build probabilistic models can vary greatly, and usually defines the type of EDA algorithm being used. For instance the Bayesian Optimization Algorithm (BOA) [71] is an EDA that builds a probabilistic model of Bayesian network.

This thesis implements a simplistic probability vector model similar to the one shown in (fig14), further details will be described in implementation section.

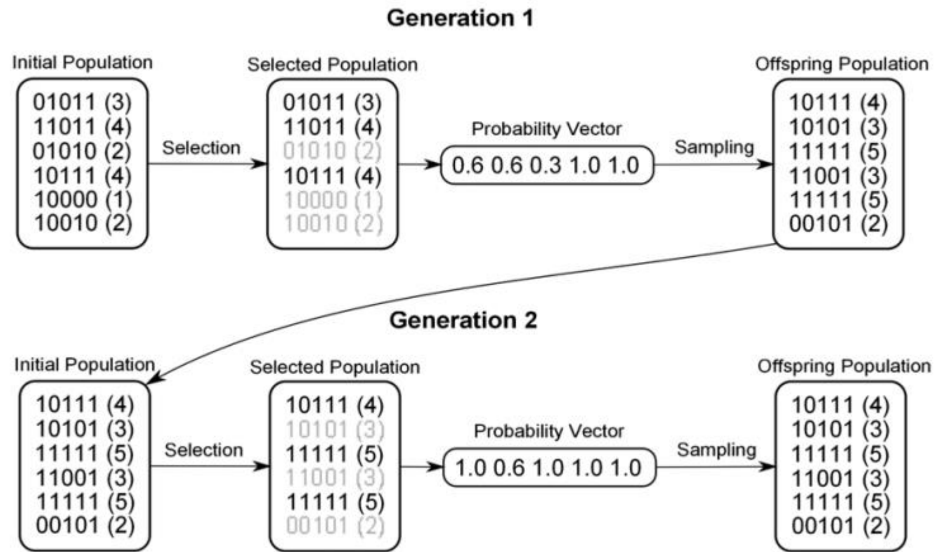

Fig14. Probabilistic model is sampled into a vector to represent the percentage of occurrences for each bit in sampled solutions. EDA then generates new solutions based on sampled vector model.

2.12 Experimenting With A.I in Game Environments

“Games are a great testing ground for developing smarter, more flexible algorithms that have the ability to tackle problems in ways similar to humans” [72], a statement from the minds behind AlphaGo [73], the deep learning system that had set an important milestone in A.I research, where a machine finally could win against some of the world’s best human Go players [73][1], a task that was believed to be one of the most challenging tasks for Artificial Intelligence.

Using A.I in video games have existed almost since the beginning of video game history, as the demand for computer controlled entities were necessary to create an interactive and fun environment for players. From primitive conditional based A.I to search & planning algorithms, and eventually integrates machine learning [74]. Because games have shown to be greatly flexible and can be modelled to reflect all kinds of tasks and problems. This is what that have made video games so popular in A.I research, simply because of the ability to model and simulate different scenarios.

For example, strategic decisions and planning have been studied using real time strategy games (RTS) to test different machine learning techniques [75], even with the usage of rtNEAT [76], which as discussed as an essential part of the work in this thesis. Other cases where games were used to test the ability of using reinforcement learning for learning to play at a human level using raw pixel inputs [2][77]. Even

popular games such as Angry Birds[2] and Minecraft[3] have been used as test beds for teaching A.I in games, see [78] and [79] respectively. Other specialized game environments have also been created specifically for A.I education as well as research [80].

As A.I research progresses in games, and as many good results have shown that machine learning techniques can be used to beat human in video game playing, a natural question which arises is that if those A.I models have reached human level intelligence? Certainly this is not the case because almost all A.I models are trained to excel really well in the tasks they are trained for and does not generalize too well, this is one of the motivations behind the General Video Game AI (GVGAI) framework [81].

GVGAI is a framework for designing gaming environments that is unbounded by quantity, i.e. using Video Game Description Language (VGDL) [82] to generate games dynamically for different A.I models to compete. This allows trained A.I models to face game environments that they have never seen before, allowing them to generalize over different unseen problems and tasks instead of being fixed to a set of games they have been trained for. With this framework Perez-liebana et al. hopes to help generalizing machine learning A.I models to equivalent to General Artificial Intelligence [81].

A central aspect to this thesis is regarding neuroevolution algorithms, which have also been extensively used to attempt to improve NPC behaviors, procedural content generation as well as other aspects of different kind of games [83]. In the same context of using A.I techniques to improve different game elements, it is worth taking a look at the work of Yannakakis et al. [84], where they have compared different computational intelligence techniques used in different aspects of video games.

Since using game environments have shown to be promising for A.I research, this thesis implements a simplistic real time game environment to run some of the experiments. This allows the possibility for future research of proposed techniques in this thesis to run more complex and interesting machine learning experiments.

3 Related Studies

As there are not many directly related studies, i.e. studies where rtNEAT is used in combination with other techniques to test for the effects of exploration and

[2] https://www.angrybirds.com/

[3] https://minecraft.net/

exploitation, this chapter will therefore only present some studies that are somewhat related and point in the same direction as the research done in this thesis.

3.1 NERO

NeuroEvolving Robotic Operatives (NERO), is a video game created by Stanley et al. [4] to demonstrate the ability of rtNEAT in evolving agents in a real-time game environment. For this purpose NERO was the first game to utilize rtNEAT as a core feature in its game mechanics.

NERO may as well be described as a game of its own style, where the main purpose of the game is to allow players to train robotic agents in several tasks. In fact the players interact with the game environment by designing different tasks for the agents to solve.

Agents in NERO starts out with no knowledge of the world and are gradually evolved and learn to solve different tasks. The agents are always equipped with a set of different sensors to perceive the environment around them.

This game demonstrated that it is possible to use neuroevolution to implement learning agents in an interactive game environment. This is a key idea that inspired made possible for the work in this thesis, by using rtNEAT to probabilistically evolve the population, real-time evolution was made possible.

3.2 NEAT and XOR Function

In the original paper of the original NEAT algorithm [57], Stanley and Miikkulainen evaluated their proposed algorithm using the XOR function. They argued that even though the XOR function is a simplistic function, but do well serves the purpose of testing their learning algorithm for evolving non-linear separable functions.

Another advantage in using XOR for evaluation is its simplicity, easy to implement and can test if the different mechanisms in NEAT were working as expected before evaluating for more complex functions.

This thesis leverages this idea and have as well used the XOR function for evaluation of rtNEAT's functionality.

3.3 Differential Evolution and EDA

In a study by Sun et al. [70], they had created a hybrid algorithm by combining DE and EDA algorithms called DE/EDA.

DE algorithms works by mutation and crossover similarly to genetic algorithms, with the only difference is that DE performs mutation and crossover by sampling from the existing population (usually 3 candidates) and calculates a differential between them in order to produce a trial candidate for evaluation. If the trial candidate shows improvement, it is accepted.

In DE/EDA, instead of performing mutation and crossover only using the calculated differential between sampled agents, EDA algorithm is used to build a probabilistic model from the population and contribute to creating new solutions. At the same time an adjustment coefficient is also introduced in the DE/EDA to allow adjustment of how much effect EDA would have during creation of new solutions.

The mechanism described above is closely related to how EDA is used with rtNEAT in the study of this thesis. Similarly the EDA Mutator implemented in this thesis also has an adjustable parameters to adjust how much influence the EDA algorithm should contribute to mutation.

4 Implementation

This chapter will be dedicated to the implementation progress of rtNEAT, as well as discussing some of the challenges and issues that arose during implementation that could directly and indirectly affect the overall behavior of the algorithms used, which in turn may affect the experiments and results.

All of the final implementations were done using the Python programming language. The reason Python was chosen was because it provides a simple workflow from coding to testing and debugging as compilation is not required.

C++ was also involved initially in an attempt to use Python purely for prototyping, while the main code would be implemented in C++ using the Unreal 4 game engine. The Unreal 4 game engine was chosen because of the promising workflow of using an existing game engine to ease the process of creating game environments. But this workflow of using Python and Unreal 4 eventually became a bottleneck, because Unreal 4 implements an adaptation of C++, which made porting Python code to Unreal 4 more difficult than planned.

For the reasons mentioned above, all the codes implemented in Unreal 4 was discarded and only Python was used to implement all the experiments discussed here in this thesis.

4.1 Implementation of rtNEAT – The Metaheuristic Search

As mentioned in section 2.7, this thesis divides neuroevolution algorithm into two components; metaheuristic search and neural network elements. Regarding the implementation of rtNEAT, the metaheuristic search component is first implemented and tested, then comes the addition of neural network elements (section 4.2).

The implementation of rtNEAT's metaheuristic search component was first identified to be a set of different key elements:

1. Data structure for species.
2. Data structure for population.
3. Selection mechanism, including speciation.

Note that historical marking, which is a key feature of rtNEAT does not belong to the metaheuristic component, because it is only used to match neural network genes when applying crossover of neural network genomes.

In order to test if the metaheuristic search algorithm works, simple implementation of the agent class was created. These agents simply represent 2d vectors in Euclidean space along with mutation and crossover operators. Mutation operator simply modified the vector components randomly, while crossover operator would blend then 2d vectors from two distinct agents to create an offspring.

Testing also required implementation of a simple problem to test if the code could optimize well, therefore a simple fitness function was created to evaluate the population of vector agents based on their distance from a predefined circle radius of 10; the closer the more fitness. As a result the metaheuristic search successfully optimized towards this problem (fig15).

A simple mechanism to test for the effect of exploration vs exploitation was also implemented. This was done by allowing the agents to mutate more (take bigger jumps) when they are on low fitness, while agents with high fitness would jump shorter distances, allowing for exploitation (fig16).

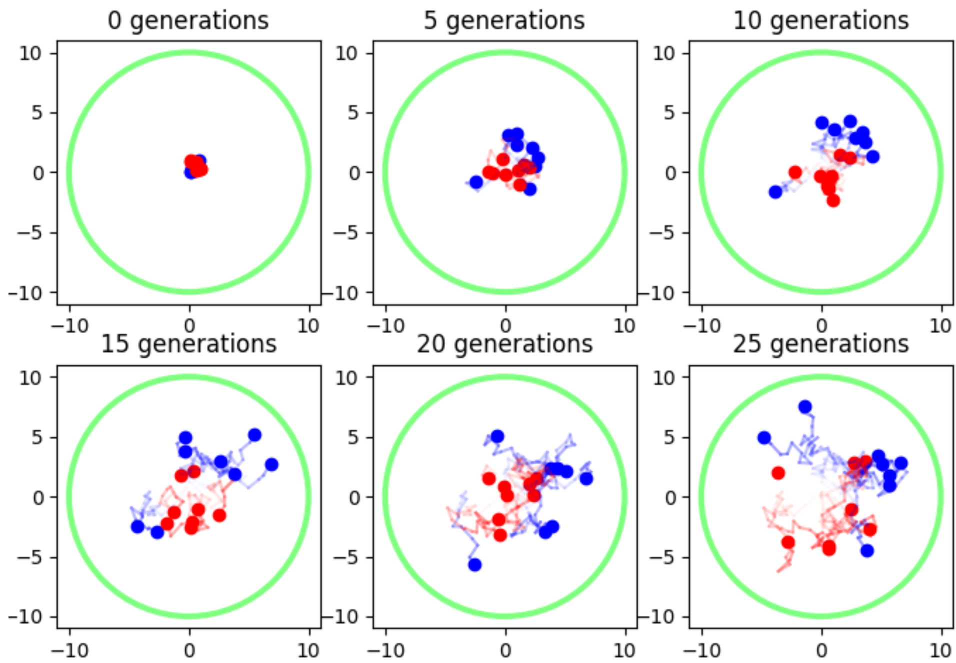

Fig15. Graphical output showing how the metaheuristic search improved agent positions over generations. Blue agents are the fittest, while red shows the worst agents.

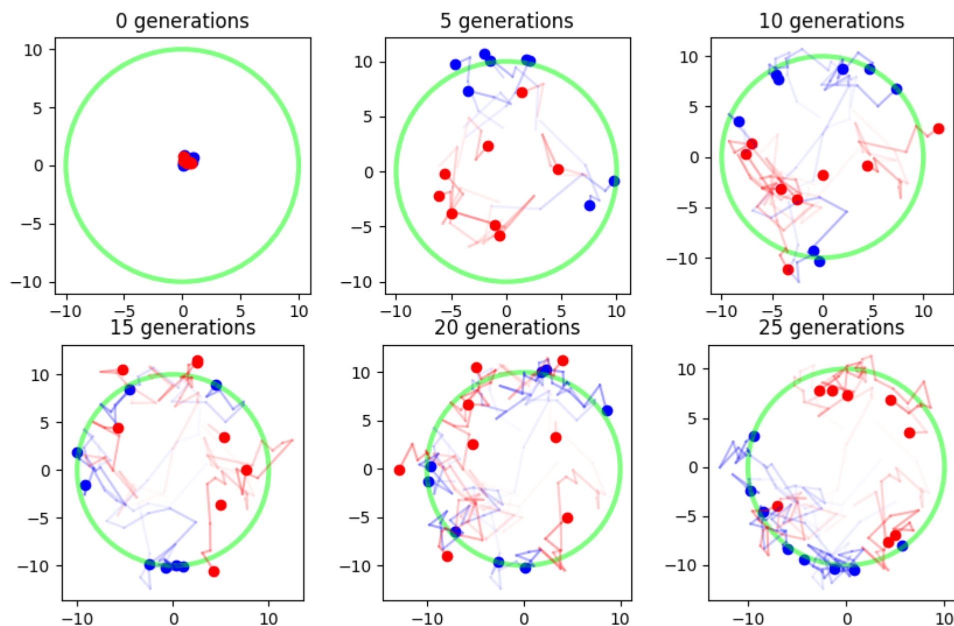

Fig16. Agents were able to explore and find the circle much faster as well as sticking to the edge (exploit) once found.

4.2 Implementation of rtNEAT – The Neural Network

At its core, a neural network is simply a graph of nodes and edges, therefore implementing any neural network data structure is similar to implementing graph data structures.

A graph consist of a list of nodes and edges, in the case of a neural network they are referred to as list of neurons and connections respectively. A graph of neurons and connections is called a genome.

Neurons in the implementation of rtNEAT in this thesis contain 3 lists of incoming, outgoing and blocked nodes, as well as a type string. Specifically for any given neuron **N**, the following data members are maintained:

1. **Incoming nodes** - A list of node indices pointing to neurons that have connections to **N**.
2. **Outgoing nodes** – A list of node indices pointing to neurons that **N** is connected to.
3. **Blocked nodes** – A list of node indices that **N** considers as blocked, this list is used to prevent cyclic paths as well as other connection prevention mechanisms, such as preventing output neurons to have outgoing connections.
4. **Type** – A string indicating what type of neuron **N** is; Input, Output or Hidden.

Connections simply contain neural network and rtNEAT required properties such as weight, enabled/disabled state, incoming and outgoing nodes.

A **Genome** encapsulates neurons and connections as well as important class methods for mutation and crossover operators. Genomes can be set to allow either recursive or feed forward networks. The implementation of forcing feed-forward graph structures in genomes have also taken much of development time and extensive debugging throughout development, as small mistakes have escalated to deeper and hard to spot bugs during genome mutation and crossover.

One key difference between the implementation of rtNEAt in this thesis and traditional NEAT is the possibility to physically remove a connections (not just disabling connections as in traditional rtNEAT). The implication of this is that by just removing connections, the algorithm can create new connections with opposite direction, this can be important for feed-forward networks as connections are removed and reconnected, new interesting structures can emerge.

```
Nodes:              [1, 2, 3]
Unconnected Pairs:  {}
Connections:        [(1, 2), (1, 3), (3, 2)]

After n1 is removed
Nodes:              [2, 3]
Unconnected Pairs:  {}
Connections:        [(3, 2)]

After new node is added
Nodes:              [1, 2, 3]
Unconnected Pairs:  {(1, 3): 246, (2, 1): 236, (1, 2): 251, (3, 1): 267}
Connections:        [(3, 2)]
```

Fig17. Output to demonstrate sampling of unconnected pairs. Picking 1000 samples of random unconnected pairs resulted in a uniform distribution across all possible pairs (1, 3), (2, 1), (1, 2) and (3, 1), where each pair was sampled 246, 236, 251 and 267 times respectively.

Comparing to the original C++ rtNEAT[4] implementation by Kenneth O. Stanley himself as well a NEAT implementation in python (neat-python[5]), the rtNEAT implemented in this thesis also has a second difference when it comes to the mutate-add-connection operator. The implemented rtNEAT in this thesis guarantees to always find a pair of unconnected nodes (uniformly distributed) and add a connection between them (fig17). It will only fail if the graph network is fully connected and provides no remaining free pair of nodes to connect, in contrast to neat-python and C++ rtNEAT where this kind of mutation will cancel if the randomly selected pair give rise to invalid connection (i.e. cycles in a feed-forward network). The implication of this is the effect mutation probability parameters have for C++ rtNEAT and neat-python can vary. Since adding connection can fail, it means that the mutation probability parameters do not precisely define the exact probability of mutation.

This differences above were discovered accidently when the source codes of mentioned implementations were looked into and compared, as the rtNEAT implementation within this thesis was initially designed purely by the description from the original articles [40][4] instead of porting existing codes.

[4] http://nn.cs.utexas.edu/keyword?rtneat

[5] https://github.com/CodeReclaimers/neat-python/

4.3 Implementing the real-time Game Environment

The implementation of the real-time game environment was done using the python Arcade[6] library. Arcade is simplistic 2d library intended for developers to provide a simplified workflow in 2d game creation.

The implementation of a simple game environment resulted in the following set of main code classes:

1. **NEATGame** – The core class for running the game environment. It defines the game bound as well as update methods for periodically updating entities and events within the environment.

2. **GameEntity** – Base class for visual entities within the game environment. An entity have position and velocity, and is as well bounded by the game bound, which performs simple collision check to prevent game entities to fall outside the screen. Even though entities can also have default colors and render radius defined, it is possible to override rendering and update methods to allow custom visual and behavior within the environment.

3. **NEATEntity** – This class extends from the GameEntity class to allow for interaction between NEAT agents and the game environment. The functionality of this class serves the purpose of encapsulating methods for perceiving environmental inputs as well as acting on neural network outputs to actions from an associated NEAT agent (NEATAgent). This is basically the physical representation of a NEAT agent within the environment.

4. **NEATAgent** – This class represents the brain of NEAT entities within environment. This class is also directly represents agents within species of the rtNEAT population. As agents evolve and are replaced with offsprings, associated entities will also get their brains (NEATAgent objects) replaced. By replacing brains instead of the entities themselves will make it look like as if game entities evolve their behavior over time instead of being replaced with new physical entities. This is important feature to allow for designing seamless interactive game environments where computer controlled entities are made to behave as continuously living agents.

Objectives represents the fitness function of rtNEAT in the environment. They are designed from extending the GameEntity class, this allow them to have different visualizations as well as interactions with the environment and other entities. For example to design a fitness function to evaluate and assign high fitness to agents

[6] https://pythonhosted.org/arcade/

within a rectangular frame, an objective is created with overridden update and rendering method to assign fitness as well as drawing the rectangular frames (fig18).

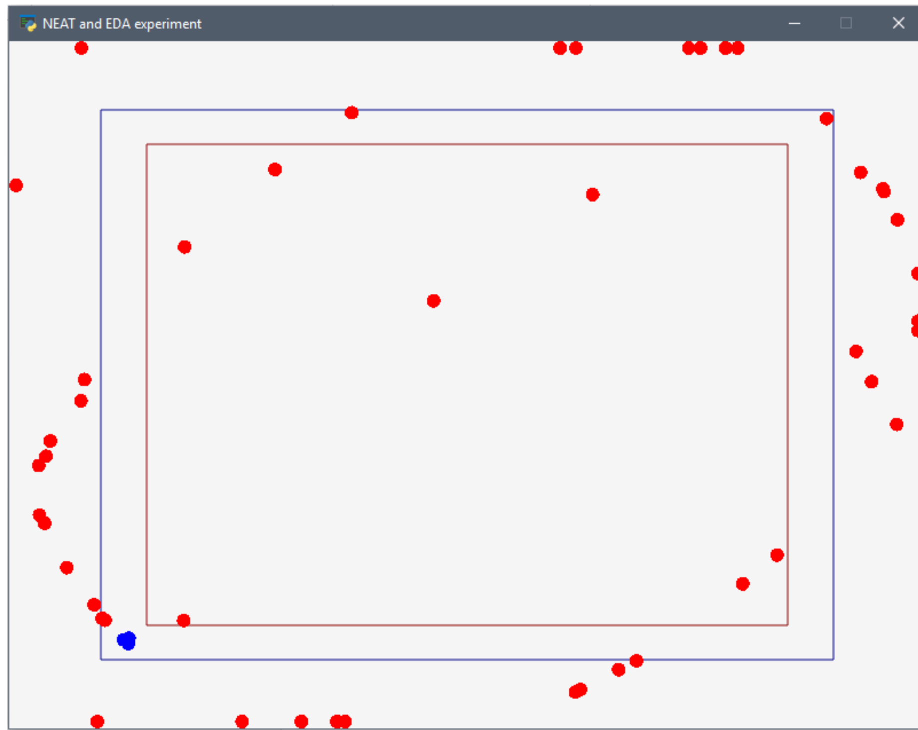

Fig18. Agents being evolved using rtNEAT to optimize for a fitness landscape of a rectangular frame. Blue agents are within the frame, red agents are outside. This simulation eventually made the agents circle in a somewhat rectangular pattern, though not fully optimal.

4.4 Implementation of EDA Mutator

To test the effect of using estimated distribution models from EDA algorithms, a simple approach was designed to allow for mutation in the same manner as how EDA builds probabilistic models.

The implemented EDA Mutator is no mean a fully stand-alone EDA algorithm, because an EDA algorithm would evolve its own population through generations like how evolutionary algorithms would. This is why it is called the EDA Mutator, as evolution is carried out by the implemented rtNEAT algorithm, EDA Mutator would contribute in mutating genomes based on built probabilistic models. The EDA Mutator was also implemented modular, meaning it can be turned on or off for different experiments.

How the EDA Mutator works mainly by using 2 class methods:

1. **BuildModel(genomes, reset=True)** – This method builds the probabilistic model of connections from a list of genomes (i.e. from top performing genomes). This model holds the probabilities of the existence of connection genes, the weight means and weight standard deviation. There is also a reset flag that can be used to reset model building or continue from previously built model, which allows for incremental model building.

2. **MutateGenome(genome)** – After a model has been built, the EDA Mutator can be used to mutate individual genomes, by mutating their genes using the built probabilistic model. This method takes a single genome as argument. Mutation is also controlled by a global parameter called **power**, which controls how much influence EDA mutation should affect mutated genomes, with 0 as no influence as all while 1 is considered fully influential.

The implemented rtNEAT algorithm has the functionality to set which mutator object is to be active at any given time, with a default set to NEAT Mutator. When needed, EDA Mutator can be assigned for EDA based mutations.

It is important to illustrate how the **power** parameter influences the normal distributions used for mutation of genomes, in order to understand the mechanism behind EDA mutation. Since an example is worth a thousand words, see (fig19).

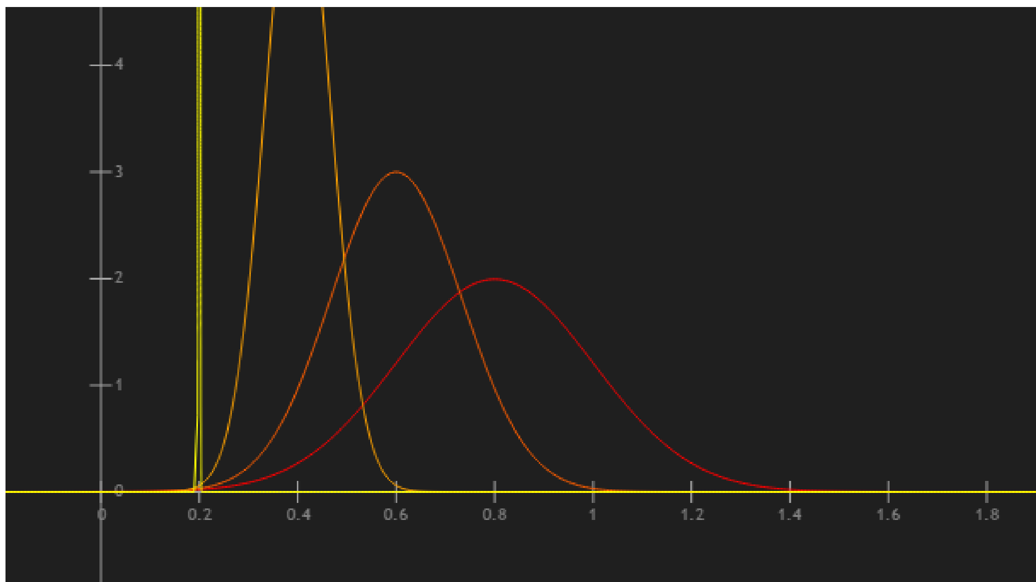

Fig19. Given a connection with weight 0.2, a probabilistic model where $\mu = 0.8, \sigma = 0.2$, the *power* parameter will interpolate the probability density function between the yellow and red distributions. At 0 *power* the weight will be exactly 0.2 with $\sigma = 0$, the yellow distribution is only an exaggerated illustration of this scenario for the purpose of visibility.

Lastly, the EDA Mutator described in this section is no mean the only way to implement mutation with EDA. This implementation only demonstrates the possibility to combine parts of one algorithm with another algorithm, particularly EDA and rtNEAT algorithms.

4.5 The NEAT Mutator

As mentioned in previous section, the rtNEAT algorithm implemented in this thesis is defaulted to use standard NEAT/rtNEAT mutation operators, originally described here [40]. But in the implementation presented here, there are some elements that differs from the standard NEAT. This can have implications for how evolution might affect the experiment results. Different NEAT based adaptions found on the internet may differ from each other as well (some can be found on the NEAT users page[7]), something which is expected as different developers implemented their own interpretation of the originally described algorithm. Nonetheless, the main core elements of the original algorithm is often kept intact.

Mutation, crossover and historical marking are controlled by the NEAT Mutator class in the current implementation of rtNEAT, therefore any adaptation to these elements also affects how the current implementation handles these rtNEAT mechanisms.

Key adaptation that differs from the original NEAT/rtNEAT mechanisms are:

1. **Historical Marking** – Historical marking plays an essential role in rtNEAT as well as all NEAT based adaptations, because it lay ground for consistent crossovers where genes need to be aligned for structural matching. The basic concept of historical marking is to assign new unique structural mutation a unique identification number. In the current implementation of rtNEAT, the NEAT Mutator also controls assignment of historical marking, but this was later realized to be unnecessary, as the implementation of the Genome class already supports unique identification of structural mutations.

   Each time a connection is created between two nodes, a hash table is updated. This hash table keeps track of which pair of nodes are associated with which connection objects. Initially this was only intended for easy retrieval of connection objects by only providing node indices as arguments. But this was adopted to be used for gene matching as well. Since each structural innovation between the same pair of nodes across genomes would give the same hash key in the form of a python tuple object (i.e. node1→node2 = (1, 2)), while new unique structural innovation would give distinct keys, this can be translated

[7] https://www.cs.ucf.edu/~kstanley/neat.html

to have the same effect as maintaining historical marking across the population.

2. **Mutate Remove Connection** – Another adaptation to the NEAT mechanism the connection removal mutation. In original NEAT a connection could mutate to become disabled or enabled (but is directionally maintained). In this implementation, connections disabled and enabled state can still be mutated, but additionally they can also be physically removed, which allows for re-creation of new connections with different connectional directions. This is rather important in mutating a feed-forward enabled genomes, otherwise it would be impossible to re-create a connections of opposite directions by just enabling and disabling connections, as the cyclic prevention mechanism would kick in and prevent this. This is not as critical in a recurrent network, as connections in both directions can be created between any pair of nodes, and by disabling and enabling those would allow for the same effect of flipping connection directions.

The NEAT Mutator also takes in most of the rtNEAT related parameters such as mutation probabilities. Those are shown in the figure below (fig20):

```
mutate_params = {
    'AddConnectionProb':0.2,    # probability to add a new connection between random unconnected pair
    'RemoveConnectionProb': 0.1,# probability to remove a random connection
    'AddNodeProb': 0.05,        # probability to mutate add new node
    'RemoveNodeProb': 0.0,      # probability when to remove a random node satisfying threshold below
    'RemoveNodeThreshold': 0,   # remove when a node the the number of connections in/out are equal to or below threshold
    'MutateStateProb': 0.0,     # probability for disabled/enabled state of random connection
    'MutateWeightProb': 0.1,    # probability to mutate a random connection weight
    'WeightBlendFactor': 0.7,   # blend factor weight when crossover, 1.0 means take only weight from fit parent,
                                # 0.0 means take from the other less fit parent

    'MutateWeightMean': 0.0,    # mean to use in the distribution when mutating random weight
    'MutateWeightDev': 5.0,     # deviation to use in the distribution when mutating random weight
    'PerturbWeightDev': 0.1,    # deviation to use when perturbing weight
    'PerturbWeightProb': 0.9    # probability to perturb a random connection weight
}
```

Fig20. NEAT Mutator parameters, largely affects the overall behavior of rtNEAT.

4.6 Verifying rtNEAT – The Evolution of XOR

Once rtNEAT had been implemented, the next natural step is to verify if everything was working as intended. One of the most common experiments to run for this kind of test is to optimize for the XOR logic function. A traditional reason to using the XOR is because as mentioned in section 2.2, the XOR function had shown to be one of the initial challenge to perceptron learning, therefore several learning algorithms have tested their performance by learning the XOR function, including NEAT. According to the NEAT User Group website[8], it is also mentioned that when

[8] https://www.cs.ucf.edu/~kstanley/neat.html

implementing NEAT, many would prefer to first test with the XOR function to ensure functionality of the implementation.

The XOR verification test was executed twice with two different settings; the first was to have agents starting out with only 2 input nodes, 1 output node and no connections, the second was with fully connected connections assigned random weights. The test evolution was running over 3000 reproductions in both cases. According to how rtNEAT works, each reproduction would create one offspring with a probability for it to be mutated. In Other words rtNEAT does not mutate any other agents in the population except for when reproducing. The result of the XOR test can be seen in (fig21).

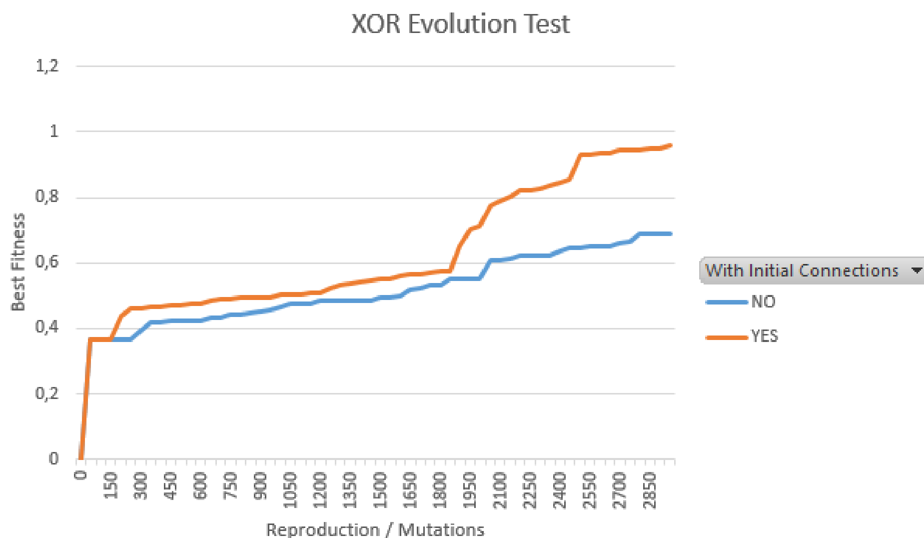

Fig21. XOR evolution with two different settings, this shows that by assigning random connections to agents initially would improve the performance of search (higher fitness). Nonetheless, it is important to remember that mutation parameters do play a big role in providing this result, using other parameters the result may be entirely different.

As (fig21) shows that agents that started with initial connections almost gained the maximum possible fitness of 1 over 3000 reproductions. Despite the differences of performance between the two cases, in both cases evolution was able to find the non-linearity of the XOR function. The output of the best performing agents from both cases are shown in (fig22):

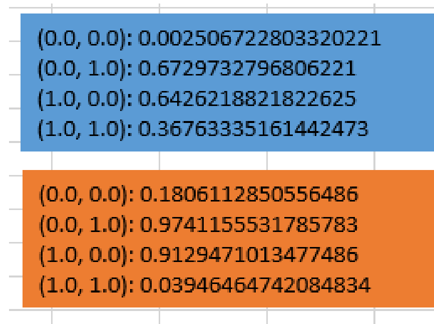

Fig22. Final output of best performing agents from both test cases, both achieved non-linearity of the XOR function.

4.7 Implementation of the Experiments

The experiments presented in this thesis are a mixture of XOR function experiments and Game Environment based experiments. The reason for using simplistic XOR function to run parts of the experiments is because it is quick to setup and can be used as a preliminary to larger experiments.

Because the development time was set back by the initial development in Unreal 4 engine, which forced a full re-implementation (several components were re-used) in python, this had also constrained the available time to design for more complex experiments. Despite this, using the XOR function in combination with the simplistic game environment can still show interesting results and lay the foundation for future work using rtNEAT.

Logging data for analysis is an important factor in designing experiments. In the following experiments of this thesis, data is recorded in two ways:

1. **Console output** – All development in python during the project have been based on using the Pycharm IDE[9], this had allowed for ease in coding and testing. As a habit of any developer, the console output is often used for printing out all kinds of program related data. In the case of this thesis, much of the console logged data could directly be used for statistics, such as fitness performance, mutations and neural network structures can all be printed in the console.

2. **Using python-dill** – Dill is a python library that allows for storing and loading any data object during runtime, this makes it possible to store any state of objects during evolution, i.e. storing an entire population every **X** number of mutations and later load for analysis.

[9] https://www.jetbrains.com/pycharm/

4.8 Issues

This section will briefly discuss some of the several issues and drawbacks that occurring during development that may have an effect on the experiments, which hopefully would be helpful to anyone who would try to implement and experiment with rtNEAT.

Metaheuristic Search - As previously described, the implementation of rtNEAT is divided in two parts. The development of metaheuristic search component did not meet many obstacles. But during the process, key mechanism such as speciation needed to be designed carefully, as this mechanism relies on the functionality of agents to be comparable with each other (for compatibility). Hence the implementation of comparability of agents needed to be done carefully, and is as well problem dependent based on the type of problem the agents represents. For example in the case of vector agents, their relative distances are measured for compatibility, while for neural networks, gene distance as described in NEAT is used.

Genomes & Neural Network – The implementation of neural network genomes were the most challenging due to the fact that they needed to be able to evolve purely feed-forward networks. One problem lead to the next, suddenly the problem was no longer related to neuroevolution, but rather graph theory related; how to procedurally generate arbitrarily non-cyclic networks. This took much of development time as small mistakes creeped up during evolution of complex structures, and made it difficult to track for the cause. The key idea here is to always test exhaustively the fundamental mechanics whenever something is changed in the implementation.

Developing Games - Often times it may be tempting to use existing libraries or game engines to develop the game environment that the experiments is going to take place, as this would simplify the process of development. But sometimes the actual libraries and engines themselves are in fact the crucial bottleneck to experimental design if preliminary knowledge was not known. Limitations of engines and libraries may only show themselves in later stages of development as the technological boundaries of these engines and libraries are reached. In the case of the Unreal 4 engine, the technological boundary was their custom adaption of C++ which broke several implementations and had to be implemented differently.

Performance - Moving away from Unreal 4 engine made it possible to speed up the process of development of rtNEAT and related codes. But when everything was up and running, another issue appeared, which also limited the scope of possible experiments in this thesis. Namely performance issues caused by the python interpreter. As most of the implemented codes were implemented naively using python classes and objects, which helped to speed up development time, but these python objects when nested deeply and without care can cause huge performance

impact during execution. This is one of the main culprits that prevents the experiments to scale. This can perhaps be avoided by first acquiring deep knowledge in the platforms available for development, but then again any project is time constrained, and is therefore a difficult trade-off task.

5 Experiments

The experiments were setup to be divided into XOR and Game related experiments, each demonstrate different aspects of how mutation affects exploration and exploitation in rtNEAT as well as in combination with EDA.

5.1 XOR

To make things more interesting, the XOR logic function used for the experiments presented here are extended to be a 3-input XOR function. 3 way input XOR functions can have different truth tables depending on the application, one is by chaining up 2 XOR logic gates, another is by assigning 1 only when one of the inputs equals to 1. The latter XOR function is used in the experiments presented here, with the following truth table (table1):

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

Table1. 3-input exclusive or truth table.

In addition, all agents in the population will start with all input nodes connected to the output node, this is to allow for a small speed up in evolution. The size of population is set to 50 with the following mutation parameters (found experimentally):

AddConnectionProb	0,2
RemoveConnectionProb	0,1
AddNodeProb	0,05
RemoveNodeProb	0
RemoveNodeThreshold	0
MutateStateProb	0
MutateWeightProb	0,6
WeightBlendFactor	0,7

MutateWeightMean	0
MutateWeightDev	5
PerturbWeightDev	0,1
PerturbWeightProb	0,9

Table2. Mutation parameters for 3-input XOR.

The activation function used is a logistic sigmoid function as shown in equation (3) where $\beta = 1$. Evaluation of evolutions (number of evaluations vary per experiment) will be limited to 5000 reproductions per evolution, this is equivalent to 100 generations (5000/50=100) per evolution. Number of species is also set to target 5 species, and the algorithm will attempt to maintain this number of species whenever possible. The coefficient c_3 from equation (9) is also experimentally set to be 0.1.

The fitness function to evaluate the 3-input XOR function is defined as

$$Fitness = e^{-(expected-output)^2} \tag{13}$$

The reason for using an exponential fitness function is to avoid negative values, because the selection mechanism in genetic algorithms don't behave well with negative values when proportional selected is used. Because NEAT/rtNEAT creates offsprings based on the proportion of species fitness (equation 12), it is therefore needed to convert negative fitness values into a positive range.

5.1.1 RtNEAT only

This experiment is designed to serve as the baseline for the comparison of other experiments. A secondary purpose of this experiment is also to show how sudden innovative mutations may boost the entire population in search for better solutions.

This experiment evaluates 3 complete evolutions of 8000 reproductions each. The reason for using 8000 reproductions instead of 5000, is only to demonstrate that when letting evolution run for long enough time, it will be able to converge to the optimal solution if the fitness landscape is not deceptive (including several optima), which it is for the case of 3-input XOR.

5.1.2 RtNEAT with explorative search

This experiment attempts to test the effect of how increasing the number of mutations effects the ability of population to find better solutions, as well as testing a secondary effect to see what happens when too large portion of the population is mutated.

This experiment will be evaluated 3 times, with the standard 5000 steps of reproduction. Additional at each step of reproduction a portion of the worst performing agents in the population is at the same time mutated. For the 3 evaluations, the portion from which the population will be mutated are 10%, 20% and 40% of the worst performing agents. Agent performance is measured using adjusted fitness to protect innovation in the same manner as the speciation mechanism.

5.1.3 RtNEAT with exploitative search

This experiment is similar to the one in 5.1.2, but instead of mutating for exploration, attempts to mutate for exploitation is tested. To test this, smaller mutation parameters were chosen as well as only applying mutation to well performing agents.

Agents will still be selected using their adjusted fitness, but here another approach is used for selection instead of selecting an increment % of the population from top performing agents. Three different ranges of the population will be selected for mutation: [0%-10%], [10%-30%] and [20%-60%]. What these ranges mean is that the population of agents will be ordered by adjusted fitness, then they will be selected from the given % range. For example a selection of [10%-30%] means the top 10% of agents will not be selected, but from 10% and onward to 30%, giving 20% agents of the entire population will be selected. This kind of selection tests how exploitative search for different ranges of agents would affect the overall performance of the population. Furthermore this kind of selection would also be indicative if mutating well performing agents would disrupt good solutions created from the Building Block Hypothesis**.**

Additional, to allow for exploitative search, the extra mutations will have their mutation parameters modified according to (table 3) below, while offspring mutations will remain the same as in (table 2):

AddNodeProb	0
AddConnectionProb	0,02
RemoveConnectionProb	0,01
MutateWeightProb	0,05
PerturbWeightDev	0,1
PerturbWeightProb	0,9

Table3. Exploitative mutation parameters. These parameters are mainly focused on perturbing weight values with small probabilities to mutate connections.

5.1.4 RtNEAT with exploitative & explorative search

In this experiment, attempts to boost the performance of rtNEAT search using both explorative and exploitative search is carried out. At the same time to see if combining explorative and exploitative mutations helps rtNEAT in finding optimal solutions faster. The mutation parameters in this experiment will be taken from the two previous experiments; 10%, 20% and 40% for explorative search combined with [0%-10%], [10%-30%] and [20%-60%] for exploitative search respectively.

5.1.5 RtNEAT with EDA exploitative search

Similar to 5.1.4, this experiment attempts to test exploitative search, but instead of using straight forward random mutations for exploitation, the EDA mutator will be used to test if using probabilistic model would result in better exploitative performance.

Here EDA will be used to build probabilistic models of the 20% of best performing agents and mutate using the same range defined in 5.1.3: [0%-10%], [10%-30%] and [20%-60%].

5.2 Game

This section is dedicated to experiments running in the game environment. Because running experiments in the game environment is demanding on performance, these experiments should not be considered exhaustive but rather indicative of what can be attempted.

The mutation parameters used for the experiments in the game environment is identical to the ones used for the 3-input XOR experiments. Dynamic threshold is still maintained to target 5 species. Time in between reproductions were set to target 25% of the population to be ineligible according to equation (11). Additionally all agents have a minimum lifespan of 3.0 seconds, this is to allow them enough time to explore the environment before being replaced.

In all experiments agents will initially spawn randomly in a cluster at the bottom left corner of the game window, and the objective is to move inside a circle placed a little off top to the right of the center of the screen (fig23).

Agents in the game environment are designed to perceive a normalized 2d vector of their position in the environment, additionally they also have the ability to walk around by manipulating a 2d velocity vector. This is a simplistic setup to allow

agents to see and learn their position in the world and adjust their movement accordingly in order to locate objectives.

When it comes to the fitness function defined agents to search for the circle objective, the fitness function is formulated as

$$Fitness = \begin{cases} 100/d & if\ d > R \\ 100 & if\ d \leq R \end{cases} \quad (14)$$

Where d is the distance between an agent and the circle, and R is the radius of the circle. This is to allow agents outside the circle to "feel" the fitness gradient while still rewarding agents inside the circle with equal amount. This has the effect that agents can stay distributed inside the circle radius instead of clustering at the circle center.

Another point to note regarding the experiments presented here is that there are no limit on how many reproductions/mutations evolution can run, each run will run continuously until the program is forced to stop. Therefore, all of the experiments here were left to run until deemed by the observer (user) that no further improvements were possible.

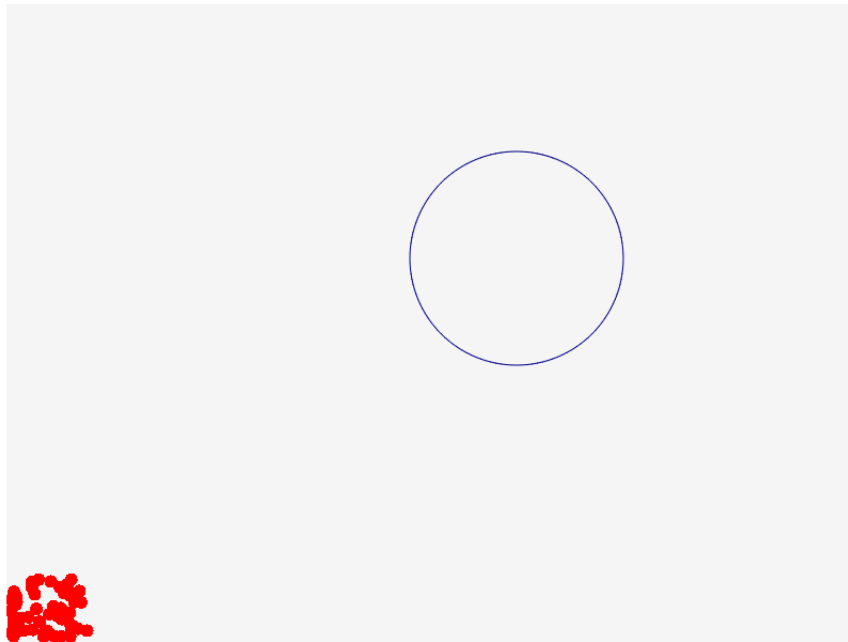

Fig23. Game environment with agents at the bottom left corner.

5.2.1 RtNEAT only

Similarly to the XOR experiments, a pure rtNEAT experiment without any additional mutations was carried out to form a base line. Furthermore the experiments

presented here will also have 3 evaluations for different scenarios. For the case of this experiment, agents are to search and stay within the circle shown in (fig23). The circle has a default radius of 100 and will shrink to 60 and then 20 over 3 evaluations. This is to test how effective rtNEAT is in finding small objectives.

For the remaining experiments the circle is kept at a radius of 100 units, this is because in the other experiments different mutation parameters are tested and compared, which requires a base line objective for evaluation.

5.2.2 RtNEAT with exploitative & explorative search

This experiment is designed similarly to the XOR experiment, a population of agents will be mutated using the same setup and parameters as in the XOR experiment. The goal here is to test the effect of different mutation parameters in a real time game environment.

5.2.3 RtNEAT with exploitative EDA

This experiment is again setup to be using the same parameters as in the case of the related XOR experiment. This section only serves to maintain the logical relationship to the sections in chapter 6.

6 Results

The results from the experiments in chapter 5 will be presented here, divided into sections and subsections in the same logical structure. Discussions related the individual experimental results will also be presented. The overall discussion of the entire research will be presented in chapter 7.

6.1 XOR

Experiment results for the 3-input XOR gate experiments and related discussions will be presented below.

6.1.1 RtNEAT only

This experiment shows (fig24) that the implemented rtNEAT (despite the differences with standard rtNEAT) managed to evolve the 3-input XOR function successfully. Through all 3 evaluations the algorithm managed to continuously optimize and find better and better solutions. It is also important to notice how evolution in rtNEAT may take big jumps from time to time in fitness optimization. This is because whenever innovative mutations occur it will be protected by speciation and slowly contribute to the growth of the entire population as a whole. Throughout all 3 evaluations a number of 5 species were maintained consistently.

The final behavior of best performing agents can also be seen in (tables 4-6), showing that all evaluations could in fact managed to evolve agents to calculate the 3-input XOR and is not just a measurement of fitness.

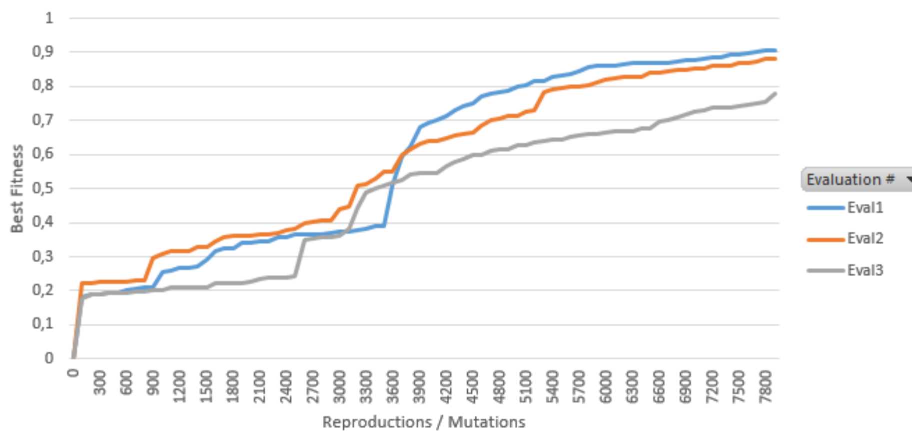

Fig24. XOR – rtNEAT only evaluations.

0.0	0.0	0.0	0,175257
0.0	0.0	1.0	0,862461
0.0	1.0	0.0	0,870095
0.0	1.0	1.0	0,056489
1.0	0.0	0.0	0,902221
1.0	0.0	1.0	0,074932
1.0	1.0	0.0	0,095245
1.0	1.0	1.0	0,018051

Table4. XOR – rtNEAT only, evaluation 1.

0.0	0.0	0.0	0,191184
0.0	0.0	1.0	0,899743
0.0	1.0	0.0	0,783509

0.0	1.0	1.0	0,141059
1.0	0.0	0.0	0,97132
1.0	0.0	1.0	0,002227
1.0	1.0	0.0	0,029106
1.0	1.0	1.0	0,001205

Table5. XOR – rtNEAT only, evaluation 2.

0.0	0.0	0.0	0,256503
0.0	0.0	1.0	0,943524
0.0	1.0	0.0	0,704761
0.0	1.0	1.0	0,002973
1.0	0.0	0.0	0,80122
1.0	0.0	1.0	0,010479
1.0	1.0	0.0	0,026444
1.0	1.0	1.0	0,002993

Table6. XOR – rtNEAT only, evaluation 3.

6.1.2 RtNEAT with explorative search

Computational performance during evaluation 2 of this experiment was crippling near the end of its evolution, which caused long computing time in between mutations. For this reason, adjustments to mutation parameters (table 7) were needed in order to run evaluation 3. The main reason for this was because once a larger portion of the population was mutated at the same time, the probability for genomes to grow larger increased over the entire population. As a result the fittest genome in evaluation 2 had acquired 29 nodes (neurons), causing a large impact on the computational performance during mutation and crossover. To put this in perspective, evaluation 1 evolved the best performing genome with only 9 nodes, while the smallest genome evolved contains only 5 nodes from experiment 6.1.1.

The adjustment of mutation parameters were to balance mutation probabilities so that the genomes would not grow too large too quickly. This adjustment could of course have an impact on the experimental result which cannot be directly compared to the results from evaluation 1 and 2. But this may as well be a demonstration of how parameters may depend on the problem scenario, may it be as a result of computational performance slowdown or other factors. Furthermore fast growing genomes would also increase the search space exponentially, which is another factor that impacts performance.

Once new parameters were assigned to evaluation 3, the computational performance improved as well as indication of improved optimization performance in regard to

population fitness (fig25). The resulting genome evolved 15 nodes, substantially less than from evaluation 2.

AddNodeProb	0,01
AddConnectionProb	0,1
RemoveConnectionProb	0,06
MutateWeightProb	0,6
MutateWeightDev	5

Table7. Adjustments towards smaller mutation rates were applied to a subset of original mutation parameters (table 2) for evaluation 3 in order to prevent genomes from growing too large too quickly.

The results from this experiment seems to indicate that when allowing more agents to mutate, would help speed up search and in turn boost the overall fitness gain of the population. Another indication is that when mutating a small portion of the worst performing agents seems to be more effective than when larger portions were mutated. This could be explained by the fact that when too many agents are mutated, the population can no longer maintain important building blocks and therefore slow down progress.

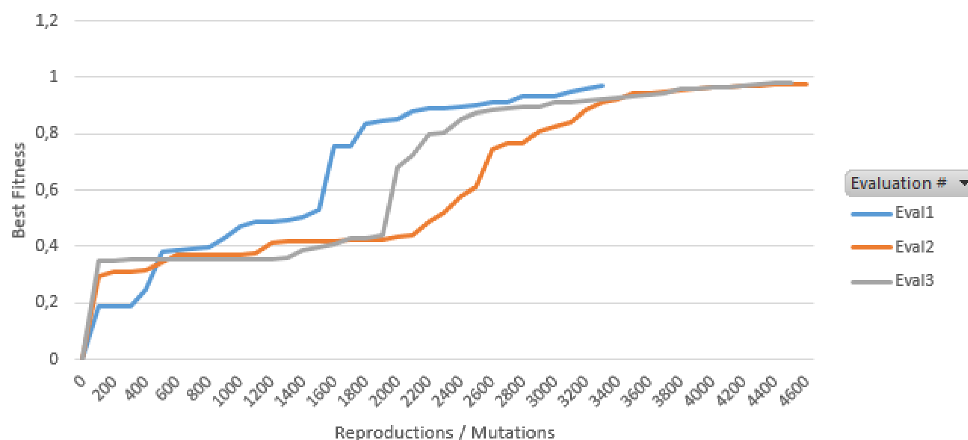

Fig25. Evaluations of rtNEAT explorative search. Notice how all 3 evaluations stopped before the limit of 5000 reproductions. This is due to the fact that evolution is set to automatically stop searching once it finds a solution with a fitness of 0.98 or above.

6.1.3 RtNEAT with exploitative search

The results from this experiment (fig26) shows that there is a difference in regarding to what portion of the population is mutated. The evaluations 1, 2 and 3 refers to mutation range [0%-10%], [10%-30%] and [20%-60%] respectively.

A possible explanation to getting this result is that when mutating in the range of top performing agents, genetic building blocks are more often disrupted and cannot maintain the good innovations acquired. If this is the case then the result shown here also strengthen the building block hypothesis for genetic algorithms. Yet exhaustive experimentation is still needed in order to acquire enough data to reach statistical significance.

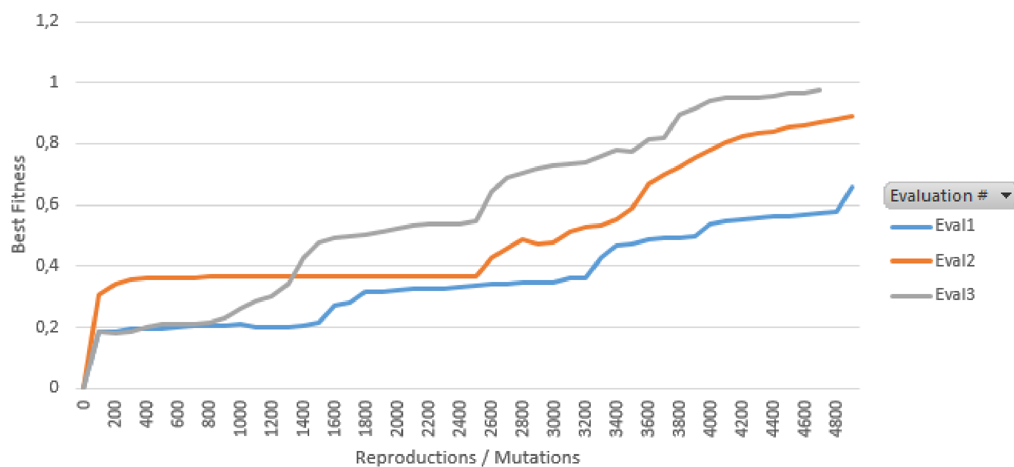

Fig26. XOR – rtNEAT with exploitative search.

6.1.4 RtNEAT with exploitative & explorative search

The results from this combination and exploration was quite surprising, because in evaluation 3, a big part of the population were extensively mutated, yet the population managed to converge, in fact quite early, which may as well be by chance caused by a random good innovation. Despite this, the algorithm was able to maintain good mutations and boost the overall population fitness.

One theory that can explain the result shown here (fig27) is that when additional exploitative and explorative mutations concentrated around the center of the population in the case of evaluation 3, top performing agents could keep the best innovations while the worst performing agents were replaced with offsprings by the reproduction mechanism. This may have allowed for a balance between exploitation and exploration as a large part of the entire population were contributing to search.

It's also important to keep in mind that during evolution, the time in between reproductions also increased due to increase in computational power was needed to mutate a larger number of agents.

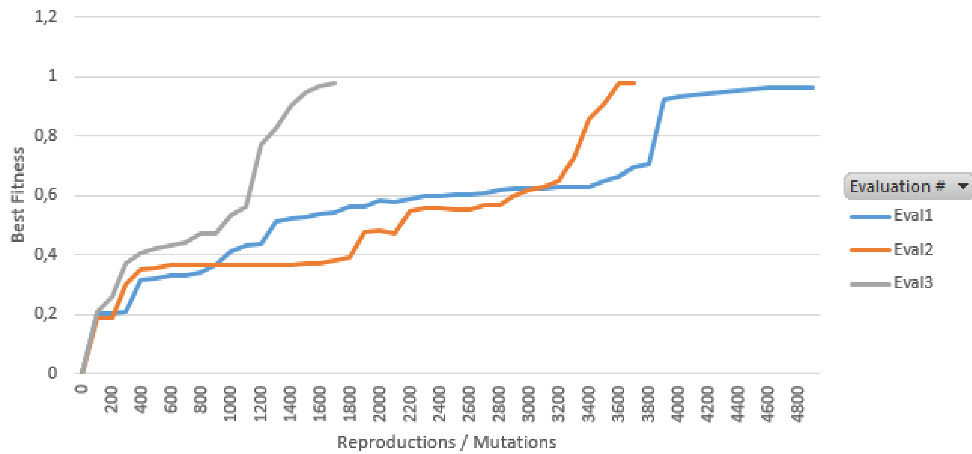

Fig27. XOR – rtNEAT with exploitative & explorative search.

6.1.5 RtNEAT with EDA exploitative search

This experiment shows that by incorporating EDA algorithm may as well be a good technique to allow for better exploitation. Even though the results in 6.1.4 beat the results (fig28) in this experiment by a large margin, but this may as well be by chance, and perhaps with further tuning EDA may perform as well.

Anyhow, the results here still beat pure rtNEAT, rtNEAT with explorative and rtNEAT with exploitative mutations. In combination with results from 6.1.4, these experiments show that by combining explorative and exploitative search, performance will be improved noticeably.

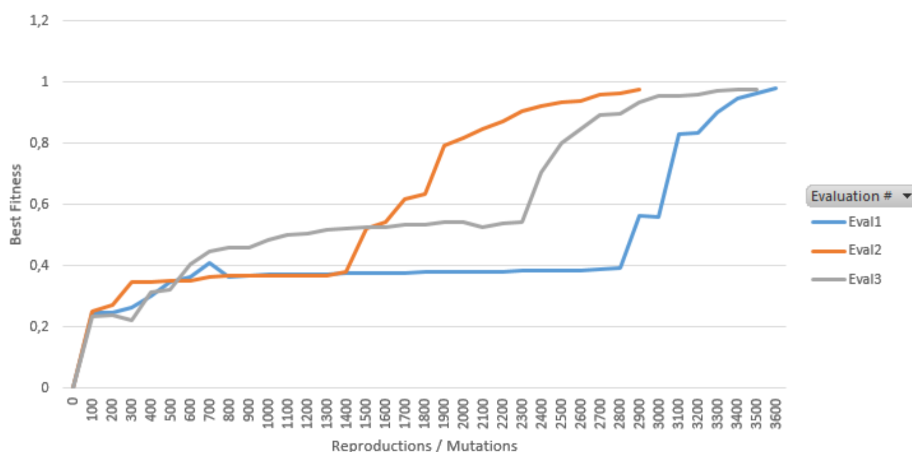

Fig28. XOR – rtNEAT with EDA exploitative search.

6.2 Game

This section will present the results for the experiments in the game environment.

6.2.1 RtNEAT only

The results from this experiment shows that rtNEAT can very well be used to optimize for real-time game environments. Even when the objective is very small, rtNEAT could eventually locate and influence the population to move towards it, and finally cluster inside the objective circle.

Clearly, in search to find smaller objectives, more reproduction cycles were needed before the objective could be located. During evolution the game environment ran smoothly without any lag, meaning the implemented rtNEAT does satisfy the intended design, which is to run in real time.

(Fig29) shows the results of all evaluation while (fig30) shows the final convergence of agents in the environment when the objective was found.

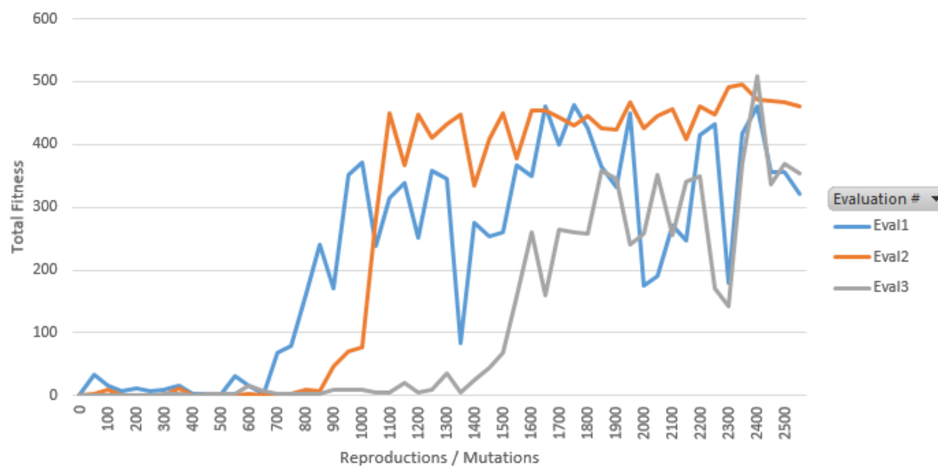

Fig29. Game – rtNEAT only results.

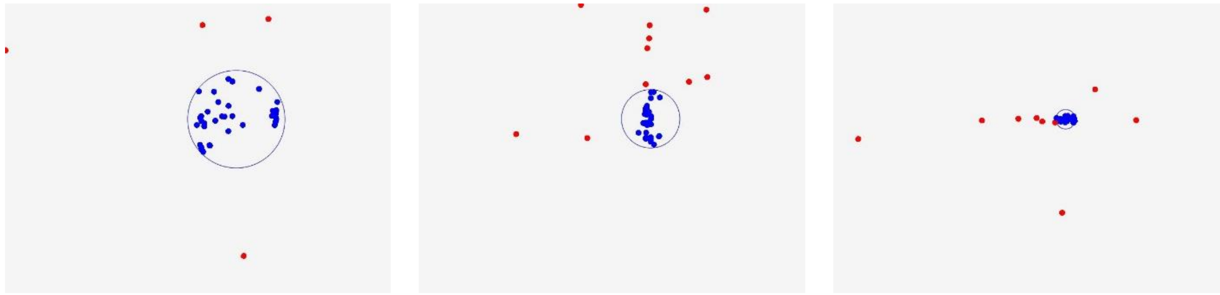

Fig30. Game – rtNEAT only, final result screenshots. From left to right: Eval1, Eval2 and Eval3.

6.2.2 RtNEAT with exploitative & explorative search

This experiment was a surprise, because the results (fig31) displays some very strange statistics. How could it be that evaluation 1 could perfectly find the objective while the other evaluations couldn't? The initial explanation for this phenomenon was that since evaluation 1 did not mutate so many agents, which was the main reason for its convergence. But after a closer look at how the agents moved around in the environment, a strange behavior was detected. Several of the agents were "twitching" or did micro-jumps. It was then later realized that because the mutations were happening at the same frequency as the reproduction cycle, more and more agents would "twitch" whenever reproduction occurred.

Now the "twitching" behavior itself was not the cause for the detrimental results from evaluation 2 and 3, the main cause was (thanks to the observation of twitching) that agents were mutated regardless of how long they have been alive. What this means is that as more and more agents were mutated frequently, led to the fact that more and more agents did not have time to explore and optimize.

This experiment proves yet another factor that could affect the overall fitness of the population, namely the minimum lifespan is crucial in protecting newborn agents in the quest to explore the surrounding landscape and build their fitness. The final screenshots of how agents ended up being spread across the environment in evaluation 2 and 3 are shown in (fig32).

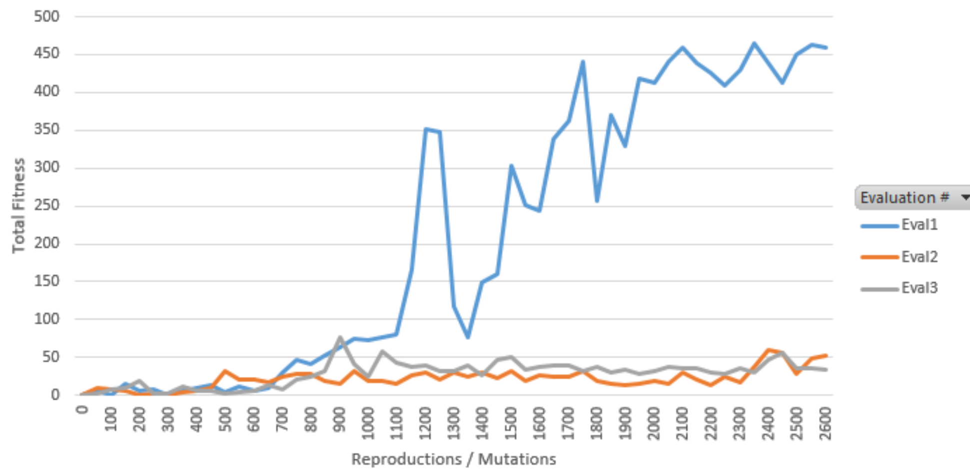

Fig31. Game – rtNEAT with exploitative & explorative search results.

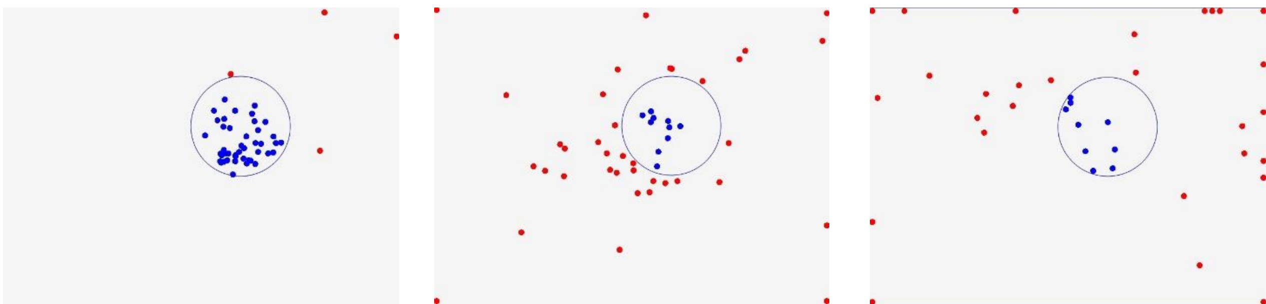

Fig32. Game – rtNEAT with exploitative & explorative search, final result screenshots. From left to right: Eval1, Eval2 and Eval3.

6.2.3 RtNEAT with exploitative EDA

The results from this experiment are quite interesting (fig33). First of all because of the diverse variation between evaluations, showing no clear pattern. This could be due to chance as with any algorithms where random variables are used. In order to understand this statistics better, more exhaustive experiments need to be carried out in order to rule out the hypothesis that this may be caused by chance.

Nonetheless, it is quite interesting to see how evaluation 1 performed well, while evaluation 2 did significantly better, but evaluation 3 could not really find the objective. One of the reason for evaluation 3 failing could be the same reason as for the results in 6.2.2, because both the explorative and EDA mutations here were applied without taking into account the minimum lifespan of agents, as this was much later discovered to be a possible issue. Yet, it is surprising how evaluation 2 managed to skyrocket in fitness while in experiment results of 6.2.2 both evaluations 2 and 3 failed. Surely more experiments need to be carried out in order to understand this phenomenon.

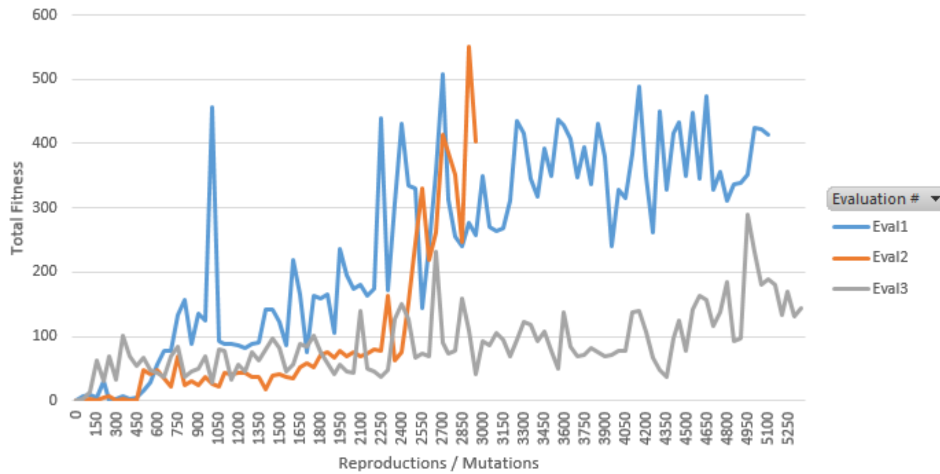

Fig33. Game – rtNEAT with exploitative EDA results.

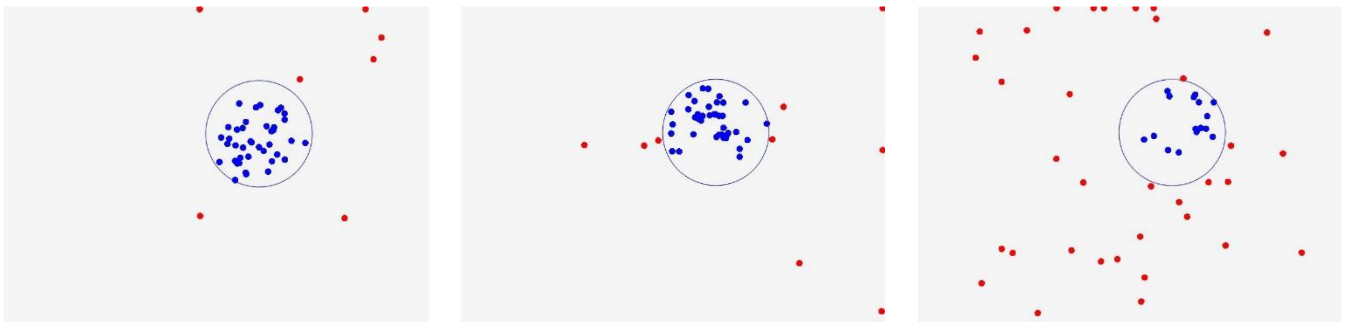

Fig34. Game – rtNEAT with exploitative EDA, final result screenshots. From left to right: Eval1, Eval2 and Eval3.

7 Discussion, Conclusion and Future Work

This chapter will summarize an overall discussion over the work done in this thesis, as well as concluding and propose possible future work.

7.1 Discussion

Throughout this thesis we have been introduced to the elements of a neural network, then to how different learning algorithms incorporate ANN to enable general optimization, and how all of them are constrained by a common problem of parameter adjustments and controlling algorithmic factors to allow for better search.

On the level of neurons and perceptrons to neural networks, different factors can play a key role in what kind of problems they can be optimized for, e.g. how using non-monotonic transfer functions could solve the XOR problem without introducing hidden layers, or how adding bias connections in neural networks would allow for better function approximator.

Moving further into the realm of reinforcement learning, we have seen how evaluative functions can be important in solving for problems where the fitness landscape is initially unknown. Reinforcement learning behaves like a gradient explorer where the gradient of the fitness landscapes are revealed over time through interactions between agents and environment.

On the other hand, moving away from direct feedback of interactions between agents and environment, we get population based metaheuristic search algorithms such as evolutionary algorithms. These algorithms takes a bigger step away from reinforcement learning in regard to estimating the gradient of the fitness landscape. Instead of estimating the gradient, evolutionary algorithms utilizes a single fitness function to evolve a population of agents, in hope that over time agents would accumulate enough knowledge about the environment and navigate it well. This leads to the question of how to define a proper fitness function, as well as how to set the search parameters so that the agents can navigate the environment via the fitness landscape.

Certainly exploration and exploitation plays a big role in the questions above, because exploration leverages the ability for agents to search and find new solutions, which is essential for exploring and discovering optima in the fitness landscape. While exploitation would allow agents to narrow down the search space on a particular area in order to find the exact point of a global optimum.

The work in this thesis have demonstration some of the key issues regarding finding the perfect search parameters as well as how particularly mutation affects search performance.

Because most of the mutation parameters used in the experiments of this thesis were found experimentally, this does not provide much knowledge on how different parameters directly affect search. But some cases were shown that when mutation parameters values were too great, genomes could grow too big and cripple evolution. Also many mini experiments have been carried out during development of the system, which also led to the discovery of the parameters used in the presented experiments. This could have biased the experiment design as to how they were designed in order to show certain effects of different mutation techniques applied.

Nonetheless the experiments in this thesis have provided important understanding of how mutation affects search, how explorative, exploitative as well as how EDA affects the overall search performance of a population of agents. Some interesting

cases were also presented, such as the results from 6.2.3, which indicates that further work is needed in order to have a better understand of the results.

7.2 Conclusion

As a conclusion, I would like to answer the research questions of this thesis:

1. **How do different elements of learning algorithms, particularly different mutation techniques combined with rtNEAT influence the search behavior?**

2. **What are the common issues in tuning algorithmic parameters for balance between explorative and exploitative search?**

3. **How well does rtNEAT perform particularly in a game environment when applying different mutation techniques?**

As for the first question, it seems like when using exploitative and explorative mutations in combination would provide better search performance than without, or when only one of them is used. Through several experiments both with the XOR and Game, exploitative combined explorative search had shown an improvement in search performance.

For the second question, it have been discussed in previous studies [67][65] that some of the most challenging factors in balancing exploration and exploitation is firstly to identify the difference between them, whether there are at all. Secondly for evolutionary algorithms, it is necessary to identify how different factors in a search algorithm contribute to explorative and exploitative behaviors. In the work of this thesis, an attempt to identify the difference between exploration and exploitation was to test whether bad performing agents could be used to explore while top performing agents could be used to exploit. Additionally different parameters were used for exploration and exploitation. Exploitative parameters were less disruptive to the neural network structures, while the explorative parameters were more aggressive. An indication from the work of this thesis is that certainly it is possible to find traits to what factors contributes to exploitive and explorative behaviors, but more work is needed to order to rule out possible misconceptions.

Lastly, to answer the third research question, results from 6.2.2 and 6.2.3 need to be considered. The initial impression is that applying additional mutation techniques seems to improve search in the real time game environment, but unfortunately due to the fact that mutation also destroyed the ability for newborn agents to explore, which may have severely ruined the results of these experiments. Nonetheless, a conclusion can be drawn here by observing that applying additional mutations may

as well improve search performance, but it must be done with care as careless mutations can destroy the underlying search mechanism of rtNEAT.

Lastly the summarized conclusion is that in order to create better learning agents in game environments, one must consider the detailed description of the underlying algorithms as well as taking into account all of its moving parts, as each of them can contribute to different search behaviors which in turn affects the overall behavior of agents.

7.3 Future Work

This thesis had lay ground for several possible future work in the quest of understanding how to balance exploration and exploitation for learning algorithms as well as how different parts can affect the overall behavior. The framework is done and several experiments can be designed in the future to test other aspects of combining rtNEAT with other algorithms as well as how to fine tune search parameters.

Furthermore, the implementation of rtNEAT in this thesis had also shown that Python may be a good language for development but it lacks the performance needed to run complex experiments. Future work may involve optimizing existing code or porting it to better performing languages.

Nonetheless this thesis have provided a guideline for future research in the same direction.

References

[1] L. Se-dol, “Artificial intelligence: Google’s AlphaGo beats Go master Lee Se-dol,” *BBC News*, pp. 1–17, 2016.

[2] V. Mnih *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[3] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone, “A neuroevolution approach to general atari game playing,” *IEEE Trans. Comput. Intell. AI Games*, vol. 6, no. 4, pp. 355–366, 2014.

[4] K. O. Stanley, B. D. Bryant, and R. Miikkulainen, “Real-time neuroevolution in the NERO video game,” *IEEE Trans. Evol. Comput.*, vol. 9, no. 6, pp. 653–668, 2005.

[5] A. K. Jain and J. Mao, “Artificial Neural Network: A Tutorial,” *Communications*, vol. 29, pp. 31–44, 1996.

[6] D. Svozil, V. Kvasnička, and J. Pospíchal, “Introduction to multi-layer feed-forward neural networks,” in *Chemometrics and Intelligent Laboratory Systems*, 1997, vol. 39, no. 1, pp. 43–62.

[7] A. Krogh, “What are artificial neural networks?,” *Nat Biotechnol*, vol. 26, no. 2, pp. 195–197, 2008.

[8] W. Gerstner and W. M. Kistler, *Spiking Neuron Models*. 2002.

[9] N. Kasabov, “To spike or not to spike: A probabilistic spiking neuron model,” *Neural Networks*, vol. 23, no. 1, pp. 16–19, 2010.

[10] F. Rosenblatt, “The perceptron: a probabilistic model for information storage and organization in the brain.,” *Psychol. Rev.*, vol. 65, no. 6, pp. 386–408, 1958.

[11] W. S. McCulloch and W. Pitts, “A Logical Calculus of the Idea Immanent in Nervous Activity,” *Bull. Math. Biophys.*, vol. 5, pp. 115–133, 1943.

[12] S. Hayman, “The McCulloch-Pitts model,” in *IJCNN’99. International Joint Conference on Neural Networks. Proceedings (Cat. No.99CH36339)*, 1999, vol. 6, pp. 4438–4439.

[13] J. Jantzen, “Introduction To Perceptron Networks,” *Neural Networks*, vol. 873, no. 98, pp. 1–32, 1998.

[14] Y. Zhao, B. Deng, and Z. Wang, “Analysis and study of perceptron to solve XOR problem,” in *Proceedings - 2nd International Workshop on Autonomous Decentralized System, IWADS 2002*, 2002, pp. 168–173.

[15] T. M. Kwon, “Gaussian perceptron: experimental results,” *Conf. Proc. 1991 IEEE Int. Conf. Syst. Man, Cybern.*, no. 3, pp. 1593–1598, 1991.

[16] J. Schmidhuber, “Deep learning in neural networks: An overview,” *Neural*

Networks, vol. 61, pp. 85–117, Jan. 2015.

[17] A. K. Jain, R. P. W. Duin, and J. Mao, “Statistical pattern recognition: a review,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 22, no. 1, pp. 4–37, 2000.

[18] A. Sperduti and A. Starita, “Supervised neural networks for the classification of structures,” *IEEE Trans. Neural Networks*, vol. 8, no. 3, pp. 714–735, 1997.

[19] S. B. Kotsiantis, “Supervised machine learning: A review of classification techniques,” *Informatica*, vol. 31, pp. 249–268, 2007.

[20] M. Gori and A. Tesi, “On the problem of local minima in backpropagation,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 14, no. 1, pp. 76–86, 1992.

[21] X. G. Wang, Z. Tang, H. Tamura, M. Ishii, and W. D. Sun, “An improved backpropagation algorithm to avoid the local minima problem,” *Neurocomputing*, vol. 56, no. 1–4, pp. 455–460, 2004.

[22] Y. H. Zweiri, J. F. Whidborne, and L. D. Seneviratne, “A three-term backpropagation algorithm,” *Neurocomputing*, vol. 50, pp. 305–318, Jan. 2003.

[23] R. S. Sutton and A. G. Barto, “Reinforcement learning,” *Learning*, vol. 3, no. 9, p. 322, 2012.

[24] M. L. Littman, “Reinforcement learning improves behaviour from evaluative feedback.,” *Nature*, vol. 521, no. 7553, pp. 445–51, 2015.

[25] C. john cornish hellaby Watkins, “Learning From Delayed Rewards,” Cambridge University, 1989.

[26] V. Mnih *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[27] C. J. C. H. Watkins and P. Dayan, “Q-learning,” *Mach. Learn.*, vol. 8, no. 3–4, pp. 279–292, 1992.

[28] M. Riedmiller, “Neural fitted Q iteration - First experiences with a data efficient neural Reinforcement Learning method,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2005, vol. 3720 LNAI, pp. 317–328.

[29] H. van Hasselt, A. Guez, and D. Silver, “Deep Reinforcement Learning with Double Q-learning,” *arXiv1509.06461 [cs]*, no. 2, pp. 1–5, 2015.

[30] G. Lample and D. S. Chaplot, “Playing FPS Games with Deep Reinforcement Learning,” *arXiv cs.AI*, vol. 9, p. 5521, 2016.

[31] I. Fister, X. S. Yang, J. Brest, and D. Fister, “A brief review of nature-inspired algorithms for optimization,” *Elektrotehniski Vestnik/Electrotechnical Review*, vol. 80, no. 3. pp. 116–122, 2013.

[32] P. Krömer, J. Platoš, and V. Snášel, “Nature-Inspired Meta-Heuristics on Modern GPUs: State of the Art and Brief Survey of Selected Algorithms,” *Int. J. Parallel Program.*, vol. 42, no. 5, pp. 681–709, Oct. 2014.

[33] X.-S. Yang, "Nature-Inspired Algorithms: Success and Challenges," in *Computational Methods in Applied Sciences*, vol. 38, 2015, pp. 129–143.

[34] K. S??rensen, "Metaheuristics-the metaphor exposed," *Int. Trans. Oper. Res.*, vol. 22, no. 1, pp. 3–18, 2015.

[35] Peter E. Hart ; Nils J. Nilsson ; Bertram Raphael, "Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. 4, no. 2, pp. 100–107, 1968.

[36] D. Whitley, "A genetic algorithm tutorial," *Stat. Comput.*, vol. 4, no. 2, pp. 65–85, 1994.

[37] X. Yang, *Firefly Algorithm*. 2010.

[38] C. Blum, "Ant colony optimization: Introduction and recent trends," *Phys. Life Rev.*, vol. 2, no. 4, pp. 353–373, Dec. 2005.

[39] J. H. Holland, "Genetic Algorithms," *Sci. Am.*, vol. 267, no. 1, pp. 66–72, 1992.

[40] K. O. Stanley and R. Miikkulainen, "Evolving Neural Network through Augmenting Topologies," *Evol. Comput.*, vol. 10, no. 2, pp. 99–127, 2002.

[41] M. Kumar, M. Husian, N. Upreti, and D. Gupta, "Genetic Algorithm: Review and Application," *Int. J. Inf. Technol. Knowl. Manag.*, vol. 2, no. 2, pp. 451–454, 2010.

[42] K. Sastry, D. Goldberg, and G. Kendall, "Genetic Algorithms," in *Search Methodologies*, Boston, MA: Springer US, 2005, pp. 97–125.

[43] X. S. Yang and S. Deb, "Cuckoo search via Lévy flights," in *2009 World Congress on Nature and Biologically Inspired Computing, NABIC 2009 - Proceedings*, 2009, pp. 210–214.

[44] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. 1989.

[45] U.-M. O'Reilly and F. Oppacher, "The Troubling Aspects of a Building Block Hypothesis for Genetic Programming," *Found. Genet. Algorithms 3*, pp. 73–88, 1994.

[46] J. Lehman and R. Miikkulainen, "Neuroevolution," *Scholarpedia*, vol. 8, no. 6, p. 30977, 2013.

[47] K. Socha and C. Blum, "An ant colony optimization algorithm for continuous optimization: Application to feed-forward neural network training," *Neural Comput. Appl.*, vol. 16, no. 3, pp. 235–247, 2007.

[48] D. B. Fogel, L. J. Fogel, and V. W. Porto, "Evolving neural networks," *Biol. Cybern.*, vol. 63, no. 6, pp. 487–493, 1990.

[49] X. Yao, "Evolving artificial neural networks," *Proc. IEEE*, vol. 87, no. 9, pp. 1423–1447, 1999.

[50] A. J. Turner and J. F. Miller, "NeuroEvolution: Evolving Heterogeneous Artificial Neural Networks," *Evol. Intell.*, vol. 7, no. 3, pp. 135–154, 2014.

[51] S. Risi, J. Lehman, and K. O. Stanley, "Evolving the placement and density of neurons in the hyperneat substrate," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation - GECCO '10*, 2010, no. Gecco, p. 563.

[52] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci, "A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks," *Artif. Life*, vol. 15, no. 2, pp. 185–212, Apr. 2009.

[53] S. Risi and K. O. Stanley, "A unified approach to evolving plasticity and neural geometry," in *Proceedings of the International Joint Conference on Neural Networks*, 2012.

[54] S. Risi and K. O. Stanley, "Indirectly encoding neural plasticity as a pattern of local rules," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2010, vol. 6226 LNAI, pp. 533–543.

[55] D. J. Montana and L. Davis, "Training Feedforward Neural Networks Using Genetic Algorithms," *Proc. 11th Int. Jt. Conf. Artif. Intell. - Vol. 1*, vol. 89, pp. 762–767, 1989.

[56] N. J. Radcliffe, "Genetic set recombination and its application to neural network topology optimisation," *Neural Comput. Appl.*, vol. 1, no. 1, pp. 67–90, 1993.

[57] K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies," *Evol. Comput.*, vol. 10, no. 2, pp. 99–127, Jun. 2002.

[58] S. Mahfoud, "Niching methods for genetic algorithms," *Urbana*, no. 95001, p. 251, 1995.

[59] B. Sareni and L. Krahenbuhl, "Fitness sharing and niching methods revisited," *IEEE Trans. Evol. Comput.*, vol. 2, no. 3, pp. 97–106, 1998.

[60] E. Dolson and D. Park, "Applying neural pruning to NEAT," pp. 1–11, 2012.

[61] E. Cant, "Pruning Neural Networks with Distribution Estimation Algorithms," *Neural Networks*, pp. 790–800, 2003.

[62] S. Whiteson and P. Stone, *Evolutionary Function Approximation for Reinforcement Learning*, vol. 7, no. AI05-320. 2006.

[63] L. Cardamone, D. Loiacono, and P. L. Lanzi, "Learning to drive in the open racing car simulator using online neuroevolution," *IEEE Trans. Comput. Intell. AI Games*, vol. 2, no. 3, pp. 176–190, 2010.

[64] S. Whiteson, P. Stone, K. O. Stanley, R. Miikkulainen, and N. Kohl, "Automatic feature selection in neuroevolution," *Proc. 2005 Conf. Genet. Evol. Comput. - GECCO '05*, pp. 1225–1232, 2005.

[65] M. Črepinšek, S.-H. Liu, and M. Mernik, "Exploration and exploitation in evolutionary algorithms," *ACM Comput. Surv.*, vol. 45, no. 3, pp. 1–33, Jun. 2013.

[66] a E. Eiben and C. a Schippers, "On Evolutionary Exploration and Exploitation," *Fundam. Informaticae*, vol. 35, pp. 35–50, 1998.

[67] K. Mehlhorn *et al.*, "Unpacking the exploration–exploitation tradeoff: A synthesis of human and animal literatures.," *Decision*, vol. 2, no. 3, pp. 191–215, 2015.

[68] H. Mühlenbein and G. Paass, "From recombination of genes to the estimation of distributions I Binary parameters," in *PPSN IV Proceedings of the 4th International Conference on Parallel Problem Solving from Nature*, 1996, pp. 178–187.

[69] M. El-Abd, "Preventing premature convergence in a PSO and EDA hybrid," in *2009 IEEE Congress on Evolutionary Computation*, 2009, pp. 3060–3066.

[70] J. SUN, Q. ZHANG, and E. TSANG, "DE/EDA: A new evolutionary algorithm for global optimization," *Inf. Sci. (Ny).*, vol. 169, no. 3–4, pp. 249–262, Feb. 2005.

[71] M. Pelikan, D. E. Goldberg, and E. Cantú-Paz, "BOA: The Bayesian Optimization Algorithm," *Genet. Evol. Comput.*, vol. 1, pp. 525–532, 1999.

[72] D. Silver and D. Hassabis, "AlphaGo: Mastering the ancient game of Go with Machine Learning," *Google Research Blog*, 2016. [Online]. Available: https://research.googleblog.com/2016/01/alphago-mastering-ancient-game-of-go.html.

[73] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[74] L. Galway, D. Charles, and M. Black, "Machine learning in digital games: a survey," *Artif. Intell. Rev.*, vol. 29, no. 2, pp. 123–161, 2009.

[75] R. Lara-Cabrera, C. Cotta, and A. J. Fernandez-Leiva, "A review of computational intelligence in RTS games," in *Proceedings of the 2013 IEEE Symposium on Foundations of Computational Intelligence, FOCI 2013 - 2013 IEEE Symposium Series on Computational Intelligence, SSCI 2013*, 2013, pp. 114–121.

[76] J. K. Olesen, G. N. Yannakakis, and J. Hallam, "Real-time challenge balance in an RTS game using rtNEAT," in *2008 IEEE Symposium On Computational Intelligence and Games*, 2008, pp. 87–94.

[77] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaskowski, "ViZDoom: A Doom-based AI research platform for visual reinforcement learning," in *IEEE Conference on Computatonal Intelligence and Games, CIG*, 2017.

[78] D. M. Yoon and K. J. Kim, "Challenges and Opportunities in Game Artificial Intelligence Education Using Angry Birds," *IEEE Access*, vol. 3, pp. 793–804, 2015.

[79] J. D. Bayliss, “Teaching game AI through Minecraft mods,” in *4th International IEEE Consumer Electronic Society - Games Innovation Conference, IGiC 2012*, 2012.

[80] S. Sosnowski, T. Ernsberger, F. Cao, and S. Ray, “SEPIA : A Scalable Game Environment for Artificial Intelligence Teaching and Research,” in *Fourth AAAI Symposium on Educational Advances in Artificial Intelligence*, 2013, pp. 1592–1597.

[81] D. Perez-liebana, S. Samothrakis, J. Togelius, T. Schaul, and S. M. Lucas, “General Video Game AI: Competition, Challenges and Opportunities,” *Proc. 30th Conf. Artif. Intell. (AAAI 2016)*, no. Schaul, pp. 4335–4337, 2016.

[82] T. S. Nielsen, G. A. B. Barros, J. Togelius, and M. J. Nelson, “Towards generating arcade game rules with VGDL,” in *2015 IEEE Conference on Computational Intelligence and Games, CIG 2015 - Proceedings*, 2015, pp. 185–192.

[83] S. Risi and J. Togelius, “Neuroevolution in Games: State of the Art and Open Challenges,” *IEEE Trans. Comput. Intell. AI Games*, vol. 9, no. 1, pp. 25–41, Mar. 2017.

[84] G. N. Yannakakis and J. Togelius, “A Panorama of Artificial and Computational Intelligence in Games,” *IEEE Trans. Comput. Intell. AI Games*, vol. 7, no. 4, pp. 317–335, 2015.