Ryerson University
Digital Commons @ Ryerson

Theses and dissertations

1-1-2011

Study of estimation of distribution algorithms applied to neuroevolution

Graham Holker
Ryerson University

Follow this and additional works at: http://digitalcommons.ryerson.ca/dissertations

Part of the Computer Sciences Commons

Recommended Citation
Holker, Graham, "Study of estimation of distribution algorithms applied to neuroevolution" (2011). *Theses and dissertations.* Paper 1058.

This Thesis is brought to you for free and open access by Digital Commons @ Ryerson. It has been accepted for inclusion in Theses and dissertations by an authorized administrator of Digital Commons @ Ryerson. For more information, please contact bcameron@ryerson.ca.

STUDY OF ESTIMATION OF DISTRIBUTION ALGORITHMS APPLIED TO NEUROEVOLUTION

by

Graham Holker

B.Sc., Queen's University, 2006

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Program of

Computer Science

Toronto, Canada, 2011

© Graham Holker 2011

Author's Declaration

I hereby declare that I am the sole author of this thesis.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Signed:________________________________

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Signed:________________________________

STUDY OF ESTIMATION OF DISTRIBUTION ALGORITHMS APPLIED TO NEUROEVOLUTION

Graham Holker

M. Sc. in Computer Science,

Ryerson University, Toronto, Canada

Abstract

This thesis proposes a methodology for the automatic design of neural networks via Estimation of Distribution Algorithms (EDA). The method evolves both topology and weights. To do so, topology is represented with a fixed-length, indirect encoding; weights are represented as a bitwise encoding. The topology and weights are searched via an incremental learning algorithm and a Guided Mutation operator. To explore suitable EDA ensembles, the study presented here interchangeably combined two representations for topology, two for weights, and two learning algorithms. Tests used in the analysis include: XOR, 6-bit Multiplexer, Pole-Balancing, and the Retina Problem. The results demonstrate that: (1) the Guided Mutation operator accelerates optimization on problems with a fixed fitness function; (2) the EDA approach introduced here is competitive with similar GP methods and is a viable method for Neuroevolution; (3) our methodology scales well to harder problems and automatically discovers modularity.

Acknowledgements

I am grateful to Marcus dos Santos, my supervisor, whose guidance and encouragement made this thesis possible.

I am also indebted to the committee, Dr. Harley, Dr. Misic, and Dr. Ferworn, for reading and providing feedback on my work. It is both intimidating and an honour to have the attention of such an intelligent group of people.

I am grateful for my family, especially my parents who made it possible for me to explore computers from a young age and to study them in University. I'm appreciative of the accommodations provided by my brother Brendan.

I'd like to thank the following friends who contributed to my time at Ryerson through support and inspiration: Elmira Ghoulbeigi, Mahsa Mostowfi, Ervis Sofroni, Leyla Vakilian, Shahin Talaei, Rishabh Saxena, Shermineh Ghasemi, Bart Gajderowicz, Joseph Paes, Joseph Ho, Matthew Dorrance, Edward Ho, Ian D'Mello, Trevor Park, Brendan Holness, Christopher Shilton, Robert Galloway, Leanne Idzerda, Heather Wilson, Andrew MacDonald, Tobias Mankis, Kevin Lu, Paul Grouchy, and Morgan Lincoln (who encouraged me to apply).

Graham Holker

Ryerson University

September 7, 2011

Dedication

For my parents, I strive to make the most of everything you have given me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The central hypothesis of this dissertation is that an Estimation of Distribution Algorithm can be used to efficiently generate Artificial Neural Networks.

Estimation of Distribution Algorithms (EDA) is a new and interesting method of *Evolutionary Computation* (EC). EDA are a *Machine Learning* (ML) approach for optimization. EDA eliminate the need for powerful search operators for exploring the search space, as in population-based EC methods, and instead create a probabilistic model of the population. EDA continually refine their model based on the performance of a population of sampled individuals. Generally, the modifications are such that the probability of sampling the best performing individual is increased.

An *Artificial Neural Network* (ANN) is a computer program made of a number of independent computing devices known as neurons. While inspired by the biological neuron, the function of an artificial neuron is simplified. An ANN can be seen as a black box with any number of inputs and outputs. The structure of the neurons, *i.e.* how they are connected, and the numerical weights of those connections determine the function performed. ANN are used in applications such as: function approximation, classification, and robot control tasks.

1.1 Motivation

There are many applications where the employment of human programmers to develop programs would be inefficient. Such problems as computer vision, character recognition, robot locomotion, natural language processing, and game playing contain examples of problems with a large space of possible inputs and outputs. For example, a written version of the character ‘A’ when scanned and digitized can be encoded in many ways. The task of writing a program to recognize the letter ‘A’ (or any letter) would require a standard input method (say 32 by 32 pixels) and a large set of digitized characters to analyze. Looking at each input and finding patterns would be a laborious task for a human programmer, but a machine can efficiently process large sets of data. The machine could do the analysis and write the program in place of a human programmer, as long as we can provide it an algorithm capable of exploiting that large data set.

Machine Learning employs computers to analyze a great quantity of data and attempt to create generalized algorithms for pattern recognition and task execution. In the field of ML, the large set of data provides *experiences* for the computer to *learn* from.

A topic such as game playing or robotic control has both the challenge of recognizing the state of the environment as well as learning to respond correctly to that state. While a portion of the learning is spent in the recognition of situations, another portion is spent attempting responses to the situations. While it would be preferable to separate the two learning challenges (recognition and response), it is only possible when the many permutations of the environment are numerable (*e.g.* in the game of Checkers). It would be difficult to separate the two learning activities in situations where the player (or *agent*) interacts in an ever-changing environment. *Reinforcement Learning* (RL) is the study of such problems.

ANNs present an interesting opportunity for solving Reinforcement Learning problems because they are computer programs with a number of inputs and outputs that are capable of recognition and response. For example, an agent attempting to keep a pendulum, that is connected to a cart on a track, balanced and upright could be given the angle and velocity of the pole and the position and velocity of the cart as inputs. The ANN then attempts to recognize the situation and decide on an appropriate response, in particular, the ANN agent would provide as output the direction the cart should move in.

Although we cannot always train the ANN on measured input-output pairs, we can define an appropriate measure of the ANN controller's performance and use *Evolutionary Algorithms* (EA) to search for a well performing ANN. We do this by creating ANN randomly and learning from the performance of those individuals and generating a new set of individuals based on the well performing ANN from the previous set.

Evolutionary Algorithms, such as *Genetic Algorithms* (GA) and *Genetic Programming* (GP), are a class of population-based search algorithms that use stochastic processes to find solutions. An initially random population is created (the first *generation*) and each individual's performance is evaluated to create a numeric value (known as *fitness*). Individuals are selected based on their fitness and are then modified and placed into the next generation. The selection scheme creates *selection pressure* which is intended to increase the average fitness of the subsequent generations. The process is eventually stopped, often because an individual reaches the target fitness or the maximum number of generations has been reached. It is advantageous to use EA given the challenges in developing ANNs manually.

Another class of EA, called *Estimation of Distribution Algorithms* (EDA), replaces the population with a probabilistic model. Individuals are generated by sampling the model and a selection of the best individuals is used to update the model. EDA is an interesting technique because it captures *building blocks*. Building blocks are particular values for a portion of the representation that will result in an above-average fitness for the individual. The model captures a building block by increasing its likelihood of occurring within sampled individuals.

1.2 Objectives

Our purpose is to study the optimization of Artificial Neural Networks (ANN) using EA, also known as *Neuroevolution* (NE). From the perspective of artificial evolution, ANN are simply another type of program and can be induced using EC techniques. NE has the added complexity of evolving both the topology of the neurons and the weights of the connections.

EDA have been shown to improve performance in other problem domains [6, 17, 35]. It is our goal to evaluate EDA performance in NE for tasks such as classification and reinforcement learning [31].

The implementation includes two different representations of the network structure, one based on the Gene Expression Programming (GEP) [5] encoding and the other based on Cartesian Genetic Programming (CGP) [29]. The implementation also includes two different methods of searching network weights, one based on a stochastic hill climbing algorithm [38] and the other based on a GA applied to a 17-bit real-valued representation. Finally, the implementation includes two different EDA, namely Population Based Incremental Learning (PBIL) [1] and Probabilistic Incremental Program Evolution (PIPE) [40]. The methodologies are tested in combination, one representation for structure, one for weights, and one EDA to navigate the search-space.

1.3 Methodology

To begin, we required a representation of the ANN's structure. Early NE systems used a fixed-topology or had a limited evolvability, such as a number of hidden nodes in a single layer[1], but connection weights were evolved. Current systems, known as *Topology and Weight Evolving Artificial Neural Networks* (TWEANN), are capable of defining the structure without constraints. They can define a number of nodes and connections in any configuration, and the only constraints come from the choice of representation (*e.g.* tree or graph). In this study, we used the solution representations of GEP and CGP because they are easily mapped to traditional EDA, and they have a many-to-one genotype to phenotype mapping. As shown in [4, 17, 19], linear representations that support a many-to-one mapping between the structures subject to genetic modification and the structures subject to selection have become instrumental for the evolution of highly complex structures, in particular ANN.

[1]ANN are often defined as inputs and outputs with a number of hidden layers in between.

Next, we defined a probabilistic model of the representations. Since both GEP and CGP encode a solution to a problem as a fixed-length array of characters or numbers, the probabilistic model is also a fixed-length array. The fixed-length array introduces a constraint to the size of the ANN represented, but can be alleviated by using a longer array. In the place where the representation would store a single character or number, the model stores an unordered set of probabilities representing the likelihood of a particular value. For example, if the possible values at a particular point are $1, 2, \text{and} 3$, then there would be 3 probabilities, where each represents the probability of one of the possible values.

To be able to generate the real-valued weights of an ANN (when sampling), the probabilistic model stores a mean and variance for each weight and, based on the best values (sampled using a normal distribution), shifts the mean and reduces the variance such that the likelihood of sampling a good value is increased. This method is known as Stochastic Hill Climbing (SHC). The second approach uses a binary string encoding of a floating-point value. The model is then an array of probabilities representing the likelihood of a 1 being sampled at a given point in the array. The bit-string model is updated using one of the EDA introduced below.

Finally, we decided how to update the probabilistic model such that the search was likely to find an individual with the target fitness (or maximal fitness) using only previously sampled individuals as a guide. We chose to explore two different population modelling methods, both of which use the best individual from the current generation. The probabilities at every point in the representation are updated such that the likelihood of the best individual being sampled is increased. This is done by increasing the probability that for each element in the gene the probability of that element being the one from the best of the current generation is increased. One, PBIL, updates the probability table based on its learning rate. The other, PIPE, defines a target probability and increments the probabilities until the target is met.

In our study, we looked at several possible combinations of the methodologies described above. Each was tested against the following problems: XOR, a simple benchmark that is interesting because it requires a hidden neuron in order to be satisfied; 6-bit Multiplexer, a more challenging problem used to compare performance with GEP; Pole Balancing, a standard benchmark in NE; and the Retina Problem, a challenging classification problem. The Retina Problem is meant to test the ability of EDA to evolve *modularity*; where modularity is defined as the encapsulation of functionality [27, 47].

1.4 Results and Contributions

Our results show that EDA for NE is a viable methodology and is actually an advantageous approach for problems that depend on finding and exploiting modularity. On simple problems the EDA shows no benefit or reduced performance, but as we move to harder problems, the EDA approach is able to scale far better than population-based approaches. On a simple problem, specifically XOR, the EDA approach did not perform as well as the population-based alternative. On pole-balancing, the performance is comparable to population-based approaches. On 6-bit multiplexer and the Retina problem, the performance far exceeds other population-based approaches. A qualitative assessment of Retina solutions shows some modularity. Other approaches, that have been developed with modularity in mind, have solutions which appear more modular, however, it is important to note that our approach involved no specific re-tooling in order to induce modularity; the modularity found is an emergent property of the EDA-based system.

Our contribution is a novel EDA approach to NE and a study showing that EDA can perform well at NE and can even perform better than population-based approaches. We've also shown how the Guided Mutation operator can accelerate the search on problems with a fixed fitness function.

1.5 Structure of this Thesis

Chapter 2 provides an overview of current research in the fields related to the work presented here. Initially it reviews Evolutionary Computation approaches, *viz.* Genetic Algorithms, Genetic Programming, and Grammar-Based Genetic Programming (GBGP). Next it reviews conventional Estimation of Distribution Algorithms and their GP and GBGP extensions. Lastly, it reviews works on Neuroevolution.

Chapter 3 begins by presenting the materials that were developed to verify the hypothesis put forth in this work, viz. ANN, EC (GEP, CGP), and EDA (SHC, PIPE, and PBIL). Chapter 3 ends with detailed description of the benchmarks and problems we attempted.

Chapter 4 presents the results of testing the digital circuit problems (XOR, 6-Bit Multiplexer), the Reinforcement Learning problem (Single-Pole Balancing), and the classification task (Retina Problem).

Finally, chapter 5 discusses the results and their impact, and suggests directions for future work.

Chapter 2

Literature Review

This work is a study of Estimation of Distribution Algorithms (EDA) for the discovery of optimal topology and weights for a Neural Network. This chapter reviews the state of the art of works in areas of study pertinent to this thesis, namely: Genetic and Evolutionary Algorithms, Estimation of Distribution Algorithms, and the evolution of Artificial Neural Networks.

2.1 Genetic and Evolutionary Algorithms

2.1.1 Machine Learning

> "A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." (T. Mitchell) [31]

Machine Learning (ML) is the study of computer programs capable of improving their performance at a given task through experience. ML has found success in applications such as data mining and difficult-to-program applications [30]. There are many approaches to ML and evolving Neural Networks was chosen because the approach is capable of performing tasks such as robotic control [2, 22, 33][1], vision [14, 34], game playing [28, 44], and classification [26]. This particular topic is of interest because there has been limited research in using an EDA for evolving Neural Networks. EDA have proven to be effective in problems that require structurally modular solutions. In this work, we explore this feature of EDA when used to evolve ANNs.

[1]It is possible to approach robotic control from other means, including Genetic Programming [11].

2.1.2 Evolutionary Algorithms

This work presents a methodology for developing a computer program that finds a functional solution to a given task by means of an Evolutionary Algorithm (EA). EA are inspired by processes found in biological evolution. There are four major components of the algorithm: a population of candidate solutions (also called *individuals*), the representation, the fitness test, and the search operators. The representation is the data structure or encoding of the individuals. The fitness test is a performance measure that maps individuals to numerical values (the individual's fitness). The search operators propose new individuals and guide the search. The basic steps of an EA are shown in Algorithm 1 [9, 25].

Algorithm 1: Basic EA

```
1: Create a random set of individuals, the population.
2: repeat
3:     Determine the fitness of each individual in the population.
4:     if a satisfactory solution is discovered then
5:         stop.
6:     end if
7:     Generate new individuals by applying the search operators to the current population.
8: until Termination Criterion is met
```

Therefore, in the context of this work, Mitchell's definition of ML can be applied to EA as follows: The ML notion of a performance measure maps quite well to the fitness test in EA; the task in ML can also be the task given to an EA. The ML notion of experience can be mapped to the fitness test performed on the candidates. The EA does not learn directly from each fitness test. Instead, the resulting numerical fitness enables the algorithm to improve the performance of individuals in the subsequent generation and the algorithm eventually finds an adequate individual. Therefore, an EA can be said to learn by evolutionary means.

2.1.3 Genetic Algorithms and Genetic Programming

There are two popular types of EAs known as *Genetic Algorithms* (GA) [9, 16] and *Genetic Programming* (GP) [23]. The population in GA consists of (traditionally) binary strings, whereas in GP, the individuals are program trees. Individuals are encoded in a machine readable format which we shall call the *representation* or *genome* (borrowed from the analogous term in genetics). A population is *evolved* by applying the search operators (*i.e.*, genetic operators) of selection, mutation, and crossover (also called recombination). The candidates are selected stochastically where the randomization is weighted based on the candidates fitness. While some individuals will simply be copied to the next generation, many will have random modifications performed on their genome (known as mutations). Others will be chosen in pairs to 'mate' and their genomes will be combined, resulting in a child (or multiple children).

The representation used in a GA (or GP) characterizes the search space in which a solution will be found. The genetic operators determine how the search will proceed by determining the new individuals to be evaluated. Quite often the population will become stuck in a local maximum and will never find the global maximum because the genetic operators, in general, make small, incremental changes, which combined with selection pressure creates an environment that is detrimental to exploration. That is, individuals are rewarded for their performance and not for their location in the search-space. Hence, the individuals may gather around a particularly good solution where small mutations would only decrease their fitness. The algorithm may never reach a higher maximum in the search-space if doing so requires traversing an area of low fitness. Thus, choice of genetic operators is very important to search performance. Given the initial population is completely random, it is important to be able to avoid local maxima since during any given run the algorithm may encounter a local maximum before the global maximum. Approaches to the problem of getting caught in local maxima include niching [9, 41] and speciation [46].

The focus of this work is on a GP system that searches the solution space by sampling a probabilistic model, as opposed to generating new individuals by means of genetic operators. By using a probabilistic model, the algorithm can more easily sample from throughout the search space since it is not bound to permutations of the current generation of individuals.

2.1.4 Genetic Algorithms

Genetic Algorithms use a fixed-length, one-to-one encoding of the problem solution and search the solution-space via genetic operators. Genetic operators include mutation (flipping a bit) and crossover (one half of individual A is swapped for one half of individual B to create two individuals). Of course, there are other possible operators and implementations.

In order to evaluate the performance of a binary string, it must first be decoded into the solution representation. For example, if we were performing parameter-tuning, the bits may represent a series of numbers that are the values of the parameters. Each time an individual is evaluated, the bits are decoded into a series of numbers and performance is measured using those numbers as parameters. In the biology manner of speaking, such a binary string is known as the *genotype* and the array of numbers is known as the *phenotype*.

Applications for GA include: automated design, parameter tuning, rule-set discovery, and other search problems.

2.1.5 Genetic Programming

Genetic Programming (GP) is similar to GA except it replaces fixed-length encodings with program trees as the solution representation. The tree may have any number of nodes which are either *functions* or *terminals*. Functions are nodes which take a number of inputs and provide some output. Terminals are the leaves of the tree and accept no input; they are usually variables, inputs, or values[2]. For example, for symbolic regression, a search for a formula that best fits the curve

[2]Terminals can also be a function with no input, *e.g.* a random function that takes no input and returns a random value.

of input data, the functions would likely be any number of mathematical functions (multiply, divide, add, subtract, sine, cosine) and the terminals would be the input variables $(X, Y, ...)$ or values $(1, 2, 3, ...)$.

One can use the operators of mutation and crossover, but they must be modified. For mutation, the system will randomly select a node in the tree and change its value. In the symbolic regression example, we can change terminals (X becomes Y) or functions (multiply becomes divide) and even change a function to a terminal (and vice versa). In the case of a terminal becoming a function, provisions must be made to add terminals to the new function.

2.1.6 Gene Expression Programming

Gene Expression Programming (GEP) is a GP technique that uses a fixed-length character array (known in GEP parlance as a *gene*) to represent a program tree [5]. A gene consists of characters representing functions and terminals (*e.g.* $+, -, \times, \div, X_1, X_2, 1, 2$). To ensure that a gene represents a working program, its structure is split into head and tail regions. The head region, the first half of the gene, can have functions and terminals, whereas the tail can have only terminals. The length of the tail region is defined based on the length of the head such that there will always be enough terminals for the functions.

Such a gene simplifies the crossover and mutation operators. In the case of crossover, in place of traversing a tree, the two genes are lined up and the elements after a randomly selected position in the gene are swapped. For mutation, a character is simply switched and there is no chance that a terminal has become a function and requires new nodes to be its inputs. In essence, instead of manipulating trees as in canonical GP, GEP manipulates arrays.

Another advantage of GEP genes is that they are fixed-length and cannot become excessively large (*i.e.* there is no possibility for *bloat*). In canonical GP, it is possible that trees will become quite large by swapping large branches during crossover. Large trees become slow to evaluate and this is detrimental to the algorithm's performance. Much research pertains to mitigating this problem [36, 52].

One representation studied in this work borrows the GEP representation as it has been modified for evolving ANN.

2.1.7 Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) [29] is a fixed-length array representation that is functionally equivalent to GP. The array is the genotype and it encodes a graph structure that consists of nodes spatially arranged as a two-dimensional grid with a number of rows and columns (thus the name *Cartesian*). Each node has access to the inputs and the outputs of the nodes in previous columns[3].

The inputs to the system are numbered starting from 0 and the nodes are numbered starting from the number of inputs. For each node, there is a series of numbers in the array representing a node from a previous column, or an input to the network, that represents the incoming connections to the node. The final numbers are the nodes whose outputs will be used as the program's output.

CGP arrays are evaluated recursively from the outputs. The output nodes are evaluated first. When a node is evaluated, it evaluates the value of its inputs; this will continue until all necessary nodes are evaluated. Alternatively, you could simply evaluate each node in order.

The baseline CGP has been extended in many ways, including: the addition of *Automatically Defined Functions* [49], self-modifying programs [13], and the evolution of neural networks [21]. This work focuses on the canonical CGP with inspiration from the neural network extensions.

CGP has some of the same benefits of GEP (fixed-length structure, many-to-one genotype to phenotype mapping), but with the added benefits of: multiple outputs[4], and that a graph representation is a closer match to an ANN than a tree.

[3]There is a provision for limiting access to columns based on distance from the current column.
[4]GEP can provide multiple outputs via multiple genes.

2.1.8 Grammar-Based Genetic Programming

It is worth noting another sub-area of Evolutionary Algorithms, although it was not used in this work. *Grammar-Based Genetic Programming* (GBGP) is a GP technique that makes use of a context-free grammar [50]. In the canonical methodology[5] the grammar is defined and the representation encodes the production rules followed.

The technique has been applied to Neuroevolution [48], but the method was limited to developing the structure (the weights had to be trained using a data set). Later, an approach for evolving structure and weights using a grammar approach was developed [12]. We chose to pursue a more direct representation such that we could evaluate the performance of just EDA on NE.

2.2 Estimation of Distribution Algorithms

Over the last decade an increasing number of works in EC has focused on alternative search strategies based on algorithms that build a probabilistic model of the search space, and sample from that model to generate individuals. The fittest individual(s) is (are) then used to update the probabilistic model. This class of algorithms, most popularly known as *Estimation of Distribution Algorithms* (EDA)[6], has been shown to be successful in a variety of complex optimization applications [35,43].

EDA can be separated into two major categories: those that consider each element an independent variable, and those that consider the relationships between the variables. The first class maintains a probability of an element being a particular value independent of the other elements sampled, whereas the second class captures dependencies between random variables, such as: what is the probability of the element being X given the previous element was Y. The methods used in this study are part of the first class. The study presented in [7] introduced an EDA (from the independent variable class) for evolving GP-like trees using an indirect representation. That work

[5]The original description of the CGP algorithm is also known as the Canonical CGP.

[6]EDA are also known as *Probabilistic Model Building Genetic Algorithms* (PMBGA).

outperformed GEP and PIPE on a selection of nontrivial problems. Also, the approach presented in [15] was inspired by the EDAs introduced in [7, 40] and on the indirect encoding scheme inspired by GEP-NN [4] to evolve ANNs. Unlike PIPE [40], these methods [7,15] use a fixed-length linear structure for the model, as opposed to a tree.

Next we review other prominent EDA related to this work.

2.2.1 Population-Based Incremental Learning

Population-Based Incremental Learning (PBIL) is an EDA for modeling a population of binary strings that, in comparison tests, performed better than a similar GA [1]. The model is a probability vector where each probability represents the chance that a 1 will occur at the given position in the individual. Initially, each probability is 0.5, representing an equal chance that the bit will be 0 or 1.

After sampling a population and finding the best individual, PBIL updates the probability vector so as to increase the probability of that individual being sampled. In PBIL this is done by increasing the probability by a percentage known as the *Learning Rate*. The probability will decrease if the bit was a zero.

In this study, PBIL was tested as a method to learn values for the ANN weights (encoded as binary strings). It was also modified to be used for more than binary strings. Through modification (and inspiration from the method described in section 2.2.2) we were able to use PBIL with probability tables, such that an integer encoding could be used (as opposed to binary).

2.2.2 Probabilistic Incremental Program Evolution

Since PBIL was created for binary strings, it is appropriate for us to explore the use of an EDA designed for Genetic Programming. *Probabilistic Incremental Program Evolution* (PIPE) is an EDA for GP that uses a probabilistic tree structure to model the population [40]; the tree is known as the

Probabilistic Prototype Tree (PPT). The tree is similar to a GP program tree, except that each node contains an array of probabilities. The probabilities are normalized[7] and each represent a different possible element for the node (*e.g.* for symbolic regression they might be: $+, -, \times, \div, x_1, x_2, R$)[8].

In order to sample the PPT the tree is traversed sampling from each node until all branches end in terminals. During this process, new nodes are created on the fly as required, *i.e.* when a function requires inputs. The newly created node has equal probabilities for each possible element. While the tree *grows* during sampling it is *pruned* during the update phase, if the probability of a node being a terminal surpasses a given threshold, the nodes beneath it are removed.

To update the PPT, first, the best individual is selected, then a target probability is calculated based on the learning rate and other factors (details provided in Chapter 3). Next, the probability for each node is incremented in the program tree. The increments continue until the probability of generating the chosen individual is equal to the target probability. Finally, the probabilities are normalized before the next generation is sampled.

In this work, we use PIPE on a GEP-based and a CGP-based representation in order to generate ANNs. Given that GEP and CGP are fixed-length representations, the model (unlike the PPT) does not have to grow and shrink. The limited size also has the benefit of requiring less processing time. Both representations have been successfully used in a variety of applications [5, 20, 29].

2.3 Evolution of Artificial Neural Networks

In this section, we shall describe what Artificial Neural Networks (ANN) are and review works that have shown that they can be effectively trained to perform many tasks. Then we shall give the motivation for using Evolutionary Computation approaches to train (or create functional) ANN for a particular task.

[7]Normalized probabilities sum to 1.0.

[8]The *R* element, the *ephemeral constant*, mimics the value terminal in GP by providing a random value.

2.3.1 Artificial Neural Networks

An *Artificial Neural Network* (ANN) is a computer program based on a mathematical model of the neurons found in biological nervous systems. They consist of an interconnected group of nodes. The nodes represent neurons and weighted edges represent the *activation level* of the neural synapses (the connections between the neurons).

Each neuron performs a relatively simple function that combined in a network of neurons can result in complex behaviours. First, it computes a weighted-sum of its inputs. Then it uses an *activation function* to determine the neuron's level of activation (its output). The inputs could be from outside the network, or the output of another neuron in the network. There are several possible activation functions, binary or real-valued.

2.3.2 Neural Network Training

Depending on the application domain, three basic learning techniques can be used to tune the connection weights of the network: supervised learning, unsupervised learning, and reinforcement learning.

Supervised learning uses a predefined network topology and tunes the weights using the error between the output and the expected output (from measured input-output data). It is generally used in pattern recognition and function approximation.

Traditionally ANN are constructed by arranging neurons in a capable topology[9] and then employing a suitable learning algorithm to *train* the network, such as the *back-propagation* algorithm or *Hebbian* learning. Training a neural network with one of these methods is a type of supervised learning.

Back-Propagation (BP) [39] uses an input-output set[10] and trains the network based on the

[9]There exists some functions such that a particular topology cannot be trained to approximate it. For example, XOR (binary exclusive or) requires a hidden neuron.

[10]Values for the expected output of a network when given the corresponding input.

difference between the actual output and the expected output. Moving from the output to the input, the connection weights are modified, slightly, such that the actual output will be closer to the expected output. The process is run a predetermined number of times over the set of data. BP is a capable technique, but is ill-suited to reinforcement learning where input-output sets cannot be created.

Hebbian learning also uses an input-output set. The connection weights are updated such that the weight is increased (by a given learning rate) when the input and the output have the same value.

Unsupervised learning, on the other hand, does not use input-output data. Instead, there is a cost function (defined by a combination of the input and the networks output) that is minimized when training the connection weights. It is commonly used in estimation problems.

Reinforcement Learning (RL) uses a specified environment in place of data. The neural network controlled agent interacts with the environment and *learns* based on feedback (instantaneous cost) from the environment. RL can also be done without the use of ANN. In this work, we look at a system of evolving ANN. Such systems have been shown to be more efficient and powerful than RL without ANN [10].

2.3.3 Weight-Only Evolving Artificial Neural Networks

Since the advent of evolutionary computation (EC) as a means of solving combinatorial optimization problems, researchers have explored evolutionary algorithms (EAs) and RL techniques to induce ANN [4, 10, 32, 45]. The application of new EC techniques to the evolution of ANN has attracted increasing interest in the last decade, giving rise to the creation of a sub-area of EC currently known as *Neuroevolution*. Early works in this area focused on evolving the weights of neural networks with a fixed topology as opposed to evolving the complete network (*i.e.* weights and topology).

2.3.4 Symbiotic Adaptive Neuro-Evolution

The method used in the *Symbiotic Adaptive Neuro-Evolution* (SANE) approach [32] uses a single hidden layer topology with an input layer, an output layer, and a hidden layer of neurons. The algorithm evolves a set of neurons and, simultaneously, a set of network blueprints that encode which neurons will be used together. When evolving the neurons, the input and output connections as well as the weights of those connections are evolved.

2.3.5 Topology and Weight Evolving Artificial Neural Networks

Approaches that have focused on the evolution of the full neural network design are known as *Topology and Weight Evolving Artificial Neural Networks* (TWEANN). Examples include: Koza's approach to evolving neural networks using Genetic Programming (GP-NN) [24], Gene Expression Programming applied to Neural Networks (GEP-NN) [4], Cartesian Genetic Programming for Artificial Neural Networks (CGP-ANN) [21], and Neuro-Evolution of Augmenting Technologies (NEAT) [45].

Competing Conventions is a problem in TWEANN where different individuals use different methods to perform the same function [42]. Crossover between two ANN using different *conventions* often creates children of worse fitness than the parents. This problem can be imagined as two ANN where one is the mirror of the other. When crossover is applied to these mirrored networks, the offspring may have none of the functionality of their parents.

Neuro-Evolution of Augmenting Technologies (NEAT) [45] uses a dynamically sized genotype that keeps track of nodes and connections. The initial population is the simplest ANN possible (no hidden neurons) and over time *add connection* and *add neuron* mutations increase the size of the genome. When these mutations occur the new element (node or connection) is given a number, known as a *historical marking*, that tracks when it was created in the hopes that it will represent the functionality of the node or connection. The historical marking allows NEAT to mitigate problems during crossover due to the competing conventions problem. Historical markings also allow NEAT to perform *speciation*, which attempts to protect innovation.

Historical markings are used in the NEAT crossover function to allow the genomes to be arranged according to functionality. The genes that are transferred between parents are chosen stochastically. Generally, crossover is a stochastic process based only on the encoding of the individuals, but with historical markings we are able to perform crossover with some awareness of the functionality of a given portion of the genotype.

Speciation is the process of placing individuals into sets of similar individuals. NEAT uses the notion of species to protect innovations. When a new species is started (a mutation or crossover creates a new gene significantly different from the others) the fitness may still be low. In the hopes that this new species will eventually supersede the current, its first member(s) is kept and mutated in the next generation. This is done by including species size in the fitness function, reducing the fitness of those in a large species, and increasing the fitness of individuals in the new species. By increasing the fitness of members of a new species we give both species a chance at continued existence in the population. Again, historical markings allow the genomes to be compared by functionality and hence the species are defined by function, not by an arbitrary encoding[11].

Although NEAT is a powerful TWEANN technique, due to the difficulty of implementing historical markings in an EDA, we have not implemented such features. While it would be possible to create a probabilistic model that uses a representation similar to NEAT, in an EDA the nature of sampling would mean that the historical markings in such a system would not necessarily indicate functionality so much as order of appearance. It is certainly a potential area of research.

2.4 Modularity

Modularity can be defined as the encapsulation of function. While EC approaches may develop

[11]NEAT also includes provisions for limiting the number of species.

very efficient solutions, it is often the case that the efficient solution lacks modularity [18, 27].

Modularity should not be confused with *regularity*. Regularity is the repetition or re-use of form or function. Regularity and modularity can be seen in the wheels of a car where each wheel provides an encapsulated function and is similar in form. Modularity without regularity would be something like the steering wheel on a car, the function is encapsulated, but there is only one per car.

In GP, modularity has been attempted through the use of *Automatically Defined Functions* (ADF) [25]. This method forces encapsulation, but does not apply any pressure to find modularity inherent to the problem being solved.

It is possible to induce modularity by using a fitness function that biases toward modular solutions. One such method switched the fitness function at fixed intervals between two functions that require similar functionality, *e.g.,* switching between (X XOR Y) OR (Z XOR W) and (X XOR Y) AND (Z XOR W) [18].

It has been shown that HyperNEAT (an extension of NEAT) was unable to find modular solutions [3]. In response, an extension was proposed that would induce modularity by constraining connections to geometrically-local neurons, known as HyperNEAT-LEO. HyperNEAT-LEO was successful in finding modular solutions [47].

Here, on the other hand, we show that modularity is an emergent property of the evolutionary process used in our approach.

Chapter 3

Methodology

The methodology presented in this section aims to provide satisfactory answers for the following questions: (1) Can an EDA-GP (an EDA for a GP task) methodology that maintains a distribution of a linear representation of ANN and *indirectly* searches the solution space of network topology and weights via a many-to-one genotype-to-phenotype mapping, evolve the necessary structures of an ANN? (2) Can an EDA-GP find solutions more effectively than state of the art Neuroevolution systems?

Given that there is no prevalent EDA for ANN in the literature, we have taken the approach of studying several possible methodologies and comparing them to each other and current NE techniques. The systems we have defined combine structural representation (GEP or CGP), weight representation (binary-string representation or mean-variance pairs), and algorithms for estimating the population (PBIL or PIPE).

We have separated the task into three parts: structure representations, weight representations, and population models. We present two methods for each part, each of which can be used interchangeably with the others. Structure representations include: GEP and CGP; weight representations include: mean-variance pairs and bitwise representation; population models include: PIPE and PBIL.

We begin by describing the structure and weight representations of ANN and follow with population modeling techniques for sampling individuals and updating the model. Finally, we present the experiments used to compare the techniques to each other and current NE systems.

3.1 Structure Representation

The structure representation describes the nodes and connections between nodes of the induced ANN. Here we describe the representations, but how they are modeled is described in section 3.3.

3.1.1 The GEP-NN Representation

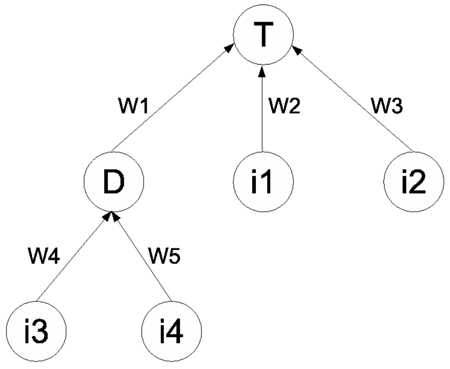
In conventional GEP, a *gene* consists of a fixed-size symbolic string encoding a program tree. Symbols in the genotype represent terminals and functions of the encoded program tree. A GEP representation can either be uni-genic, *i.e.*, the GEP chromosome consists of a single gene, or it can be multi-genic, in which case the chromosome consists of a number of genes. The gene is separated into two sections known as the head and tail regions. The tail includes only terminals whereas the head contains functions and terminals. The tail region's length t is defined as $t = h \cdot (a_{max} - 1) + 1$, where h is the length of the head region and a_{max} is the maximum number of parameters to the functions (*a.k.a.* maximum arity). Every gene represents a functionally correct program tree. The tail is long enough such that in the case that the head consists entirely of functions of maximum arity there will be enough terminals to place as parameters. The opposing extreme is when the first portion of the head is a terminal. In that situation the output is the value of that terminal and the remaining elements of the gene are unused.

In the neural network extension, called GEP-NN [4], the functions and terminals are substituted for neurons and inputs, respectively. Neurons are functions which perform a weighted sum on the given inputs. Neurons with 2, 3, and 4 inputs are called D, T, and Q respectively. Also, a third region of the gene is defined. This region, called the weight region, comes after the head and tail regions and encodes the weights. The weight region's length l_w is defined by the expression $h \cdot a_{max}$. The values of each weight are kept in an array W and are retrieved when necessary. For simplicity, the elements of the weight region are pointers to, or indices of, weights in W. Figure (3.1.a) shows an example of an uni-genic GEP-NN chromosome.

Before an ANN can be evaluated it must be generated from its gene encoding. The generative process begins at the left-most element in the gene and constructs a tree in a breadth-first manner. The weights are then added in a breadth-first manner as well. Figure (3.1.b) shows the ANN encoded by the uni-genic chromosome shown in Figure (3.1.a).

$TDi_1i_2i_3i_4i_1i_2i_3W_1W_2W_3W_4W_5W_8W_6W_9$

(a)



(b)

Figure 3.1: (a) Uni-genic GEP-NN chromosome. D and T are neurons with 2 and 3 inputs, respectively, i_1 through i_4 are inputs to the network, and W_1 through W_{10} represent the weights. (b) The represented Neural Network.

To combine multiple genes, in a multi-genic system, the networks are generated and their outputs are combined with an OR function. The OR function, when in a binary ANN, is a neuron with as many inputs as genes with a weight of 1.0 on each connection.

The representation introduced above constrains the topology of a network in the following ways: it can only represent feed-forward ANN, meaning they contain no loops, the networks are limited to having only one output, and the neurons are limited to only having one output.

A network may consist of any number of neurons. The output of a neuron of arity n is defined by the expression $f(i_1,\cdots,i_n) = k(\sum_{j=1}^{n} i_j \cdot w_j)$, where w_i is a real number denoting the weight associated with the i_{th} input signal to the neuron, and $k(x)$ is the *activation function* (defined in Section 3.6).

3.1.2 Cartesian Genetic Programming Representation

Cartesian Genetic Programming for Artificial Neural Networks (CGP-ANN) (proposed in: [19]) was used as an alternative to the GEP representation to compare performance. CGP-ANN is similar to the GEP representation in that it is a fixed-size array, but the genotype encodes a graph, as opposed to a tree. The graph can have any number of inputs and outputs, which better represents ANN than a tree since the network and neurons can have multiple outputs, which is often the case.

The graph consists of a number of nodes, organized in rows and columns. Each node receives an identification number $N_{id} = N_r \cdot c + r + N_i$, where N_r is the number of rows, N_i is the number of inputs, and c and r are the column number and row number respectively.

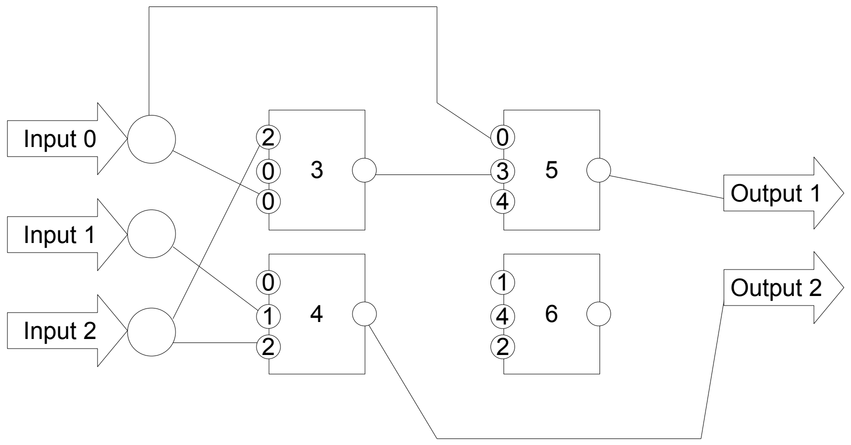
Each node has an equal number of inputs (similar to the arity of a GEP function). The possible input nodes are determined by: 1) whether the graph / neural network is feed-forward or recurrent[1] and 2) the *levels back* parameter, defining the maximum number of columns between the current column and the input column. However, the inputs are accessible to any node.

The graph is encoded as an array. The array has a set of numbers for each node and a final set for the outputs. In canonical CGP, each node has a set of indices, one for each input, followed by a number representing the node's function (*e.g.* $+, -, \times, \div$).

Given that the arity is the same for all nodes, and hence all neurons would have the same number of inputs, CGP-ANN modifies the number of inputs by way of a connection bit. Each connection is either on or off as dictated by the value of its connection bit. The connection bit is encoded as the value immediately following the input's index (as shown in Figure 3.2.c).

The final numbers in the array are the outputs. The value is the index of the node (or input) whose output value should be the network's output. There is no connection weight for the outputs.

[1]This work uses only feed-forward networks.



(a)

Node Identifier	3	4	5	6	Outputs
CGP Array	2100010	0011210	0131400	1141200	6 4

(b)

Input Node	Conn. Bit	Input Node	Conn. Bit	Input Node	Conn. Bit	Function
0	1	3	1	4	0	0

(c)

Figure 3.2: Example CGP Graph (a), the corresponding genotype (b), and (c) the genotype of a neuron (in this case node 5).

3.1.2.1 CGP Weights

In CGP-ANN [19], the weights are enumerated based on the input and output of the connection. For each possible connection there is a weight associated with the type of connection. For example, take a graph with $N_r = 2$, $N_c = 4$, and $N_i = 3$ as the number of rows, columns, and inputs respectively. The number of possible connections can be calculated as $N_{pc} = N_i \cdot N_r \cdot N_c + \sum_{col=0}^{N_c-1} col \cdot N_r^2 = 24 + 24 = 48$. Therefore, this particular graph would have 48 weights.

Since the number of weights can become quite large in the CGP-ANN method, we experimented with a system similar to GEP where the inputs to each node have a weight index. That way we can choose the number of weights for the system. The weight index is encoded after the connection bit in the array.

3.2 Weight Representation

We have implemented two methods for representing the weights: one a bitwise representation (inspired by CPG-ANN) and another as mean-variance pairs.

3.2.1 Bitwise Weight Representation

In CGP-ANN, the authors of [19] used a 17-bit representation for the weights. However, they did not specify how this representation worked, so we have provided our own method. The first bit encodes the sign (+/−) and the rest of the bits encode the value. Such that the weight would be: $W = (-1)^{b_{16}} \cdot \frac{\sum_{i=0}^{15} 2^i \cdot b_i}{2^{16}}$, where b_i is the bit value at index i.

3.2.2 Mean-Variance Pair Representation

In both GA and GP, real or integer values can be directly encoded in the representation, *i.e.* simply place the value (its binary encoding, in the case of GA) into the genome.

When creating a probabilistic model, the purpose is to represent the likelihood of a value being sampled. In the case of a discrete genotype domain, such as that of binary strings, one can use probabilities to represent the likelihood of each bit being a 1. On the other hand, for a continuous domain, such as that of real values, one alternative is to use mean-variance pairs as a model, as shown in the EDA method introduced in [38] for optimizing a vector of real values. The idea is that the mean-variance pair represents the probability of selecting a floating-point value using a normal distribution. During learning, one updates the probability by shifting the mean toward the values with the highest fitness and reducing the variance.

3.3 Population Model

The population model is similar in structure to the representation it models. Here we describe models for GEP, CGP, Bitwise Weights, and Stochastic Hill Climbing.

3.3.1 GEP Model: The Probabilistic Prototype Chromosome

Inspired by PIPE's Probabilistic Prototype Tree [40], we represent the probability distribution of GEP genes via the Probabilistic Prototype Chromosome (PPC) (see Figure 3.3).

The PPC is a linear structure consisting of two sections: the gene section D_g and the weights section D_w. D_g models the distribution of symbols occurring in the head, tail, and weight regions of the gene. Its structure is the same length as the GEP-NN genotype presented in Section 3.1.1. The difference is that it stores a probability table for each gene position. The values in each probability table are normalized (*i.e.* sum to 1.0). Note that the weight region of D_g models the distribution of the indices that point to the weight values stored in the W array of the GEP gene.

When sampled, the PPC produces a GEP-NN gene. The weights for this gene are produced from another model (configuration dependent).

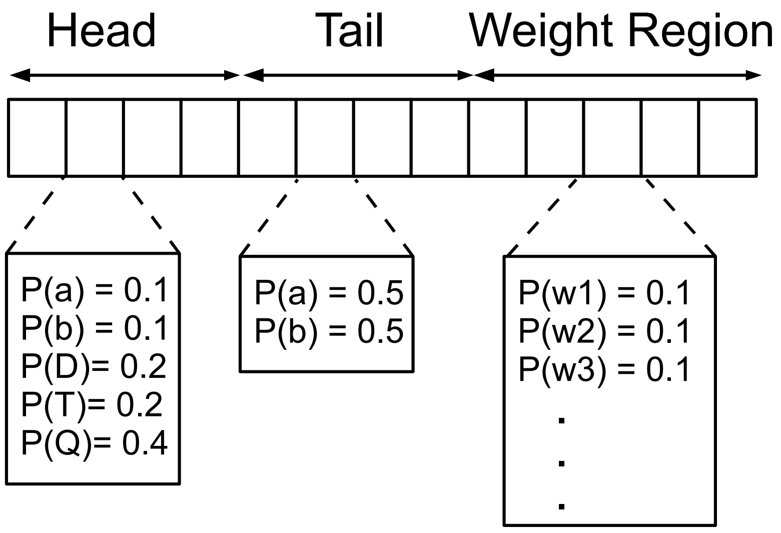


Figure 3.3: Probabilistic Prototype Chromosome: Head, Tail, and Weight Regions

3.3.2 CGP Model: The Probabilistic Prototype Array

The CGP Model (the Probabilistic Prototype Array) is similar in implementation to the PPC. For each position in the CGP array we have a probability table. The number of probabilities at any position is determined by where it is in the graph and what function it serves.

An example PPA segment is shown in Figure 3.4; the segment shows an example of a node model with arity 1. Generally, nodes will have higher arity and the node, connection bit, and weight portion will be replicated for each incoming connection; there will only be one probability table for functions per node[2]. The node portion will have varying probability tables depending on the column the modeled node is in. This segment will be replicated for each node in the array and will be followed by a probability table for each output, which would simply be a table with the possible output nodes.

When sampled, the PPA produces a CGP-ANN array, and the weights will be sampled from another model.

[2]In the figure, b and s, denote binary and sigmoid, but in the experiments of this study we used one or the other.

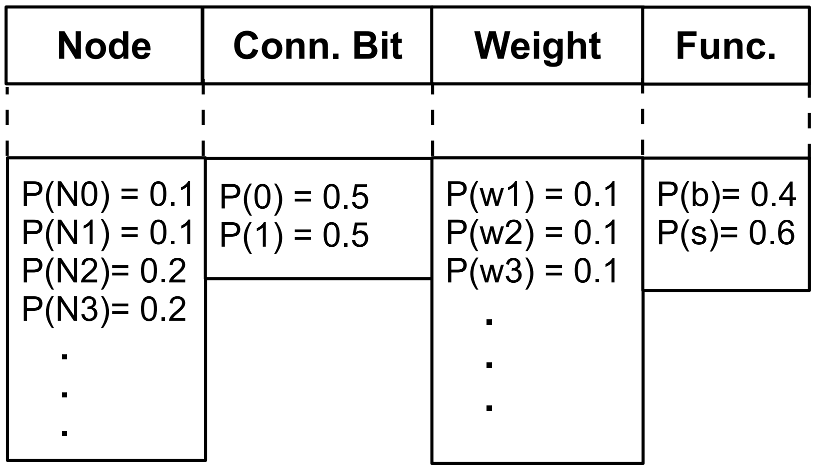


Figure 3.4: Probabilistic Prototype Array Segment: a node will have a segment like this except the node, conn. bit, and weight would be replicated once for each incoming connection.

3.3.3 Bitwise model

A binary string can be modeled as simply as a single array of probabilities representing the probability of the bit at the given position being 1. Our implementation actually uses two probabilities for each bit since it made the implementation simpler, and the system is functionally equivalent.

In any system we will have a number of weights. For each weight, there will be a model as described. When sampled, the Bitwise model produces a set of binary strings, one for each weight. The binary strings are then decoded into floating-point values.

3.3.4 Mean-Variance model

The model for SHC consists of a list of tuples $\langle \mu, \sigma \rangle$, one for each weight value; each tuple represents the mean μ and standard deviation σ of a normal distribution.

When sampled, the tuples become a set of weights.

3.4 The Estimation of Distribution Algorithm

Each of the models described (with the exception of SHC) can be updated in a similar fashion. A given individual (usually the best of the generation) is used as a target. Each probability table in the model is updated such that the probability of the target individual is increased.

Our implementation includes two methodologies for updating the models: PIPE and PBIL. We shall conclude with a description of the SHC updating algorithm.

3.4.1 Probabilistic Incremental Program Evolution (PIPE)

The high-level description of PIPE [40] is shown in Algorithm 2.

Algorithm 2: PIPE

```
 1: Initialize the model with uniform probability.
 2: repeat
 3:    Flip a coin with head probability $P_{\text{Elitist Learning}}$
 4:    if the head turns up and this is not the first iteration then
 5:       Update probabilities based on best individual found so far
 6:    else
 7:       Create population from the model
 8:       Evaluate fitness of individuals
 9:       Update probabilities based on best individual of generation
10:       Mutate the model
11:    end if
12: until Termination Criterion is met
```

Initialization of the PPC: Each list of probabilities in the model is initialized with uniform probability.

Creation of individuals: The population is created by sampling the model. To create an individual we sample from the PPC or PPA to generate a gene or array and we sample the weight model to generate an array of weights. To sample the PPC or PPA or a bitwise model, we perform roulette wheel selection based on the probability tables for each position.

Learning from best individual of generation: Let b denote the best gene found in the current generation. The idea is to modify the model such that the probability of sampling b increases. We use the PIPE learning algorithm [40] to adapt the probability distribution of the model of the structure or the bitwise weights.

Adapting the model (M): Let P_{b_i} denote the probability of the i-th element of b. The expression below defines P_b, the probability of creating gene b:

$$P_b = \prod_{i=1}^{|M|} P_{b_i}, \text{ where } |M| \text{ is the length of the model.} \tag{3.1}$$

P_{target}, the target probability we would like an individual to have, is defined as in [40]:

$$P_{\text{target}} = P_b + (1 - P_b) \cdot \lambda \cdot \frac{(\varepsilon + f_E)}{(\varepsilon + f_B)} \tag{3.2}$$

where λ is the learning rate, ε is the fitness constant, f_E is the fitness of the best individual of all generations (*i.e.* the elite individual), and f_B is the fitness of the best individual of the current generation.

The structure of b is then learned in the PPC by incrementing the probabilities of the elements of b towards P_{target} as follows: while $P_b < P_{\text{target}}$, $P_{b_i} = P_{b_i} + 0.1 \cdot \lambda \cdot (1 - P_{b_i})$, for $i = 1$ to $|M|$.

Learning from the elite individual: Let el be the best gene found so far. Similar to the method shown above to adapt the model, we calculate P_{el} (instead of P_b) and P_{target} using expressions 3.1 and 3.2, respectively, and use the same process shown above to adapt M towards el. We proceed likewise to adapt the weight model.

Model Mutation: After every generation the model is mutated as follows:

Mutation of the model: As in [40], first we find the probability of mutating a single element in the model, defined as: $P_{me} = P_M / (N_e \cdot \sqrt{|b|})$, where P_M is a user-defined parameter defining the overall mutation probability, $|b|$ denotes the size (number of nodes) of b, and N_e is the number of elements in the table (varies depending on position in the model).

We modify each probability table in the model with probability P_{me}, but only the tables that were used in creating the generations best individual b. We add to the probability of the i-th element selected a small amount: $P_{b_i} += M_R \cdot (1 - P_{b_i})$, where M_R is the user-defined mutation rate.

3.4.2 Population Based Incremental Learning (PBIL)

Population Based Incremental Learning (PBIL) is an EDA-GA technique. PBIL is a conventional EDA that is based on a binary string solution representation. Its methodology is similar to that of PIPE. A population of individuals is sampled and evaluated and the best of the generation is found. This best is used to adjust the model. Also, mutations are applied to the model after each generation.

PBIL differs from PIPE in that the adjustment is made in one calculation, as opposed to defining a target probability and incrementing toward it. Also, PBIL does not have a notion of elitist learning.

Adapting the model: We update the probability at index i using the following equation: $p_i = p_i \cdot (1.0 - LR) + best_i \cdot LR$. Where p_i is the probability at index i, $best_i$ is the value (1 or 0) in the best individual, and LR is the Learning Rate.

Mutating the model: After each generation the model is mutated. Each table in the model is mutated with probability P_m, as per the following equation: $p_i = p_i \cdot (1.0 - M_S) + R_{[0,1]} \cdot M_S$. Where M_S is the mutation shift parameter and $R_{[0,1]}$ is randomly selected as either 0 or 1.

While PBIL was originally used for binary arrays, we have expanded the usefulness by combining PBIL adaption and PIPE normalization. This works in a manner similar to PIPE except that that probability is adjusted as per the above equations.

3.4.3 Stochastic Hill Climbing

Initializing the model: The mean and standard deviation of each tuple $\langle\mu,\sigma\rangle$ is initialized as suggested in [38], more specifically: $\mu = r_{min} + \frac{(r_{max}-r_{min})}{2.0}$ and $\sigma = (r_{max} - r_{min}) \cdot \mathcal{C}$, where the constant $\mathcal{C}$ is chosen so that the sampled weight values cover well beyond the user-defined range $r_{max} - r_{min}$.

Sampling the model: To sample weights, we sample the normal distributions represented by the given mean and standard deviation tuples.

Adapting D_w: In each generation we adjust each mean to be closer to the mean of the values in V, where V is a vector storing the three best values found for a given weight. *I.e.* $\mu_i := \mu_i + \mu_{\text{move}} \cdot (\frac{1}{n}\sum_{j=1}^{n} V_j)$, for $i = 0$ to $|M|$[3]. And we reduce each standard deviation, as follows: $\sigma_i := \sigma_{\text{reduce}} \cdot \sigma_i$.

3.5 Proximate Optimality Principle and Guided Mutation

According to the proximate optimality principle [8], for most combinatorial optimization problems the solutions for a particular problem will have a similar structure. An ideal genetic operator should therefore be able to produce an offspring which is close to the best solution found so far. In evolutionary algorithms (EA), recombination is not an ideal operator, as the resulting offspring are often less viable than their parents. Conventional mutation is also not ideal. Although it may produce an offspring similar to the parent, such offspring may be more distant from other better solutions since the operator does not make use of statistics extracted from the population.

Although EDA are able to cope with this problem, their convergence on a solution may be hindered because the sampling distribution is represented by models with limited degrees of freedom. For example, in a problem with multiple local optima, the EDA may decide to represent only one peak or all of the peaks (including the valleys, *i.e.* the unfit areas). If the algorithm chooses the wrong peak, it may get stuck and never find the global optimum. If, on the other hand, it tries to

[3]Where := is the assignment operator.

encompass all peaks, it may get bogged down sampling irrelevant, unfit solutions. A special operator, known as *Guided Mutation* (GM) [53], is a possible workaround to this problem. GM extends an EDA with a search mechanism that directly controls the similarity between new solutions and a given solution.

A percentage of the population is sampled from the model and the remaining individuals are created through GM [53] which combines global statistical information and a given individual.

To create an individual via GM, we either sample a gene element from the model (based on a user-defined probability β), or clone (with probability $1-\beta$) a gene location from the best solution found so far, better known as the elite individual el. Algorithm 3 describes in more detail the creation of an individual via GM. In the algorithm, T denotes the topology model, W denotes the weight model, and i subscript denotes the i-th element of a linear structure. For example, $T[0]$ denotes the probability table at the first position in the topology model and $T_{new}[0]$ denotes the value at the first position in the topology gene or array of the new individual.

The method described in Algorithm 3 describes the weight process with respect to SHC weights. The entire weight is cloned from the individual. If the weights used a bitwise representation, GM would be used for each bit in the model same as is done for the structural representation.

3.6 Experimental Design

In our study of EDA-GP techniques for Neuroevolution we used the following benchmarks: XOR, Six-Bit Multiplexer, Single-Pole Balancing, and the Retina Problem.

3.6.1 XOR and Six-Bit Multiplexer

In order to show a system's effectiveness in finding appropriate topologies and weights we first performed the XOR test. In this test the network represents a binary logic function where the output is true when only one of the inputs is true. This is a good test for topology discovery

Algorithm 3: Creation of individual via GM

```
Let $el$ be the best individual found so far
Let $new$ be the newly created individual
Let $|T|$ be the length of the modeled topology
Let $|W|$ be the number of weights in the model
Let $\beta$ be the probability of using GM
for $i = 1$ to $|T|$ do
    Flip a coin with head probability $\beta$
    if head turns up then
        $T_{new}[i]$ = roulette wheel according to $T[i]$
    else
        $T_{new}[i] = T_{el}[i]$
    end if
end for
for $i = 1$ to $|W|$ do
    Flip a coin with head probability $\beta$
    if head turns up then
        $W_{new}[i] = Norm(W[i])$
    else
        $W_{new}[i] = W_{el}[i]$
    end if
end for
```

because it requires a hidden neuron.

The Six-Bit Multiplexer (6-MUX) test was used as a more challenging binary circuit problem. The 6-MUX takes six binary inputs, two of which represent an address pointing to one of the other four inputs. The output will be the same as the input addressed by the two address inputs.

On the XOR and 6-MUX problems the neurons used a discrete activation function defined as: $k(x) = 1, \text{if } x \geq 1$; $k(x) = 0$ otherwise.

In neural networks, the activation function has a threshold parameter. In binary neurons, the threshold is the value that determines if the neuron's output is active (value is 1). For some problems, it is necessary or beneficial to have different threshold values for the neurons in the network. In NE, either we must evolve a set of theshold values (similarly to the way GEP-NN evolves weights [4]), or we must introduce an input known as a *bias*. Generally, the bias is a constant with value 1.0. Using a bias is as simple as having a connection between the bias and the neuron. The threshold in neurons with a binary activation function is the value that the weighted sum must exceed in order for the neuron's output to be active (value of 1). For example, if the weight is 1.0, the threshold has been effectively reduced to 0.0 because 1.0 has been added to the sum via the bias.

3.6.2 Single Pole Balancing

Single Pole Balancing (SPB) is a control theory problem that has become a benchmark of NE techniques. The test includes a cart with a hinged pole at its centre (Figure 3.5). The pole will swing down unless the cart, by moving backward and forward on the rail, is able to keep the pole up.

We set up the SPB experiments as described in [32]. The Euler method was used to calculate the dynamics of the system, with a step size of 0.02s (seconds). All state variables were scaled to $[0, 1]$ before being fed to the network. The inputs are: cart position x, cart velocity x', pole

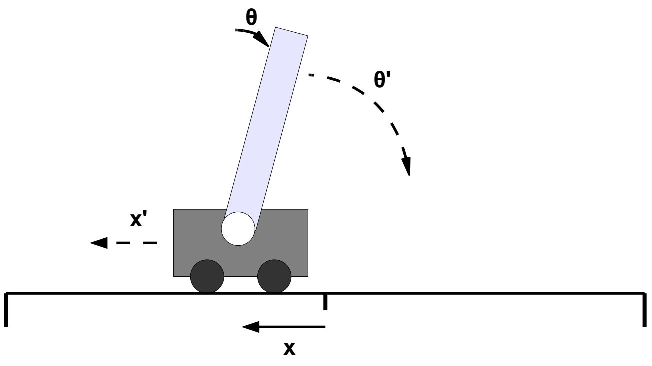


Figure 3.5: Diagram of the Single Pole Balancing Problem

position θ, pole velocity θ′, and a bias of 1.0. There are two outputs from the network, one for ‘left’ movement and the other for ‘right’ movement. If the ‘left’ is greater than the ‘right’ then the force on the cart is -10, otherwise it is +10. In the SPB problem the neurons used a continuous activation function defined as: $k(x) = (1+e^{-4.924273x})^{-1}$.

For 30 simulated minutes (1500 time steps), the cart must move left or right in order to balance the pole. The pole must remain within 12 degrees of directly upward and the cart must remain within 2.4 meters of the origin. The pole position, pole velocity, cart position, and cart velocity all begin with random values within given ranges. The fitness function consists of the sum of fitness values from ten random start states and the maximum fitness attainable is 15,000.

In the case of a GEP based solution, multiple outputs are simulated by evolving two genes, one for each output. CGP, on the other hand, is capable of having multiple outputs without two genes.

The pole and cart are simulated using the following equations:

$$\theta'' = \frac{-9.8 \cdot \sin\theta(m_p + m_c) - \cos\theta(F + m_p l_p \theta'^2 \sin\theta)}{l_p(\frac{4}{3}(m_p + m_c) - m_p \cos^2\theta)}$$

$$\theta'[t+\tau] = \theta'[t] + \tau\theta''[t]$$

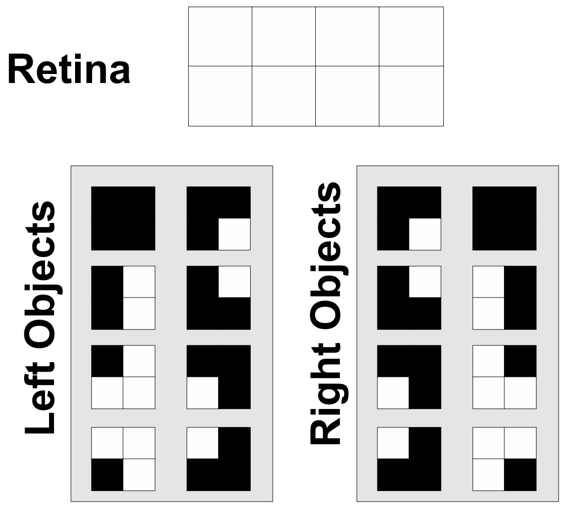


Figure 3.6: Diagram of the Retina Problem from [18]. Black indicates that the bit is on.

$$\theta[t+\tau] = \theta[t] + \tau\theta'[t]$$

$$x'' = \frac{F + m_p l_p \theta'^2 \sin\theta - m_p \cdot l_p \theta'' \cos\theta}{m_p + m_c}$$

$$x'[t+\tau] = x'[t] + \tau x''[t]$$

$$x[t+\tau] = x[t] + \tau x'[t]$$

where x is the cart position (meters), θ is the pole position (radians), τ is the amount of time between simulated steps ($0.02s$), m_p, m_c are the mass of the pole and the mass of the cart, respectively (Kg), l_p is the length of the pole (m), F is the force on the cart (newtons).

3.6.3 Retina Problem

The Retina Problem introduced in [18] is a classification task where the controller must recognize patterns on two retina (left and right). In this test we were not only evaluating performance, we also wished to verify if our approach is able to evolve modular solutions.

There are eight inputs, four per retina, that are arranged two high and four wide, see Figure 3.6. The test is to identify whether the inputs in the left retina represent a left object and the inputs in the right retina are a right object. A left object is defined as one where three or four of the inputs are on, the two inputs are on in the left-most column, or one input is on in the left-most column. Conversely the right object is similarly defined, except that if only one or two inputs are on then they must be in the right-most column.

For this problem, the activation function was binary as in the XOR and 6-MUX problems. Kashtan and Alon [18] varied the fitness function between identifying Left and Right objects and identifying a Left or Right object. For this study, the training was done with the Left-and-Right objects fitness function, as was used to study HyperNEAT-LEO [47]. HyperNEAT is an extension of NEAT created for the purpose of inducing regularity in the evolved structures. HyperNEAT-LEO is an extension of HyperNEAT that uses locality information in evolving connections for the purpose of inducing modularity.

Chapter 4

Results

4.1 Preamble

The fitness function in each of the binary experiments (XOR, 6-MUX, Retina) is a count of the number of cases that the solution ANN is able to satisfy. For Pole Balancing, the fitness is defined as the number of time periods the pole remains balanced (between -12° and 12°) and the cart stays on the track (between -2.4 and 2.4 m). The maximum fitness values for the fitness tests are listed in Table 4.1.

Performance measures attempt to show the usefulness of the algorithm. A Neuroevolution system is useful if it is able to automatically create ANN that satisfy the given task. In our tests, we performed a number of runs and defined our success rate to be the ratio between runs which found a solution and the total number of runs. If the system is capable of a 100% success rate, then the measure used is the number of individuals we need to create and test before we find a satisfactory individual – known as the mean evaluations.

All charts presented in the results section are an average of 100 runs, unless otherwise stated. Tables presenting the number of evaluations do not include failed runs when calculating the mean and standard deviation. Level of significance was calculated with a Z-test with $\alpha = 1\%$.

XOR	4
6-MUX	64
Retina	256
Pole Balancing	15000

Table 4.1: Maximum Fitness Values

Parameter	**Setting**
Weights array length	10
CGP:	
Levels Back	Columns
PIPE:	
Fitness Constant, ε	10^{-6}
Probability of elitist learning, P_{EL}	0.01
Stochastic Hill Climbing:	
μ_{move}	0.05
σ_{reduce} (Where G is number of generations)	$\sqrt[G]{\frac{1}{1000}}$
$\mathcal{C}$	0.25

Table 4.2: Fixed Parameters

Unless otherwise specified, the parameters listed in Table 4.2 remained consistent throughout experimentation.

As mentioned in Chapter 3, we have taken the approach of studying several possible methodologies and comparing them to each other and current NE techniques. We use the following notation to indicate the system setup: <structure>.<weight >.<EDA>. For example, a test setup might be GEP.SHC.PBIL, meaning the configuration uses a GEP representation for the structure, Stochastic Hill Climbing to learn the weight distribution (mean, variance), and PBIL to model the population structure. There are 8 possible systems; examples include: GEP.BW.PIPE, CGP.SHC.PBIL, CGP.BW.PIPE.

4.2 Exclusive OR

Results were obtained by doing a number of rounds of experiments. The purpose of the first round was to find the best performing system so that the subsequent rounds could optimize the parameters of that system. The parameter settings for the first round were selected from the related works [1, 4, 29, 38, 40]. It was found that CGP.BW.PIPE was the highest performing system. In the next round we tested a number of parameters, including: population size, graph size, and PIPE parameters. The third round began with a test of the different systems with the parameters from the previous run. CGP.BW.PIPE was replaced by CGP.SHC.PIPE as the system being optimized. For the fourth round, we decided to test CGP.BW.PIPE and CGP.SHC.PIPE after discovering that SHC had the advantage of unconstrained weights[1]. A full description follows. The settings of the rounds can be found in Table 4.3.

The first round, denoted XOR.R1, included a test of each possible system and a calculation of its mean evaluations and success rate. Possible systems included those using Guided Mutation (GM), the technique we used for exploring the vicinity of the elite individual. The configuration using a CGP representation, Bitwise encoded weights, and the PIPE EDA for learning the structure (CGP.BW.PIPE) and its GM counterpart (CGP.BW.PIPE.GM) had the highest success rate at approximately 17% (mean), whereas the next best competitor was GEP.BW.PIPE.GM at about 9%. We performed 100 experiments with 100 runs, and the result was significant, with $\alpha = 1\%$. In the tests of the mean, since we did not have at least 30 samples (the success rate was less than 30), statistical significance could not be used. Results can be found in Table 4.4.

The next round of experiments, denoted XOR.R2, took the highest performer (CGP.BW.PIPE) and tested it with a number of parameter changes. Tests included: population size, graph size, Guided Mutation, GM on elite or best, PIPE mutation parameters, PIPE learning rate, and using PIPE or PBIL on the BW weights. Results are shown in Table 4.4. In the table, SR denotes success rate and Evals denotes mean evaluations.

The most significant result for performance was population size. When changing the population size we kept the maximum number of evaluations consistent by changing the number of generations. GEP-NN used population size 30 and 50 generations, so the maximum number of evaluations was 1500. Setting the population size to 5 resulted in a 29% success rate. It should be noted that PIPE used a population size of 10 in some experiments [40].

[1]The bitwise representation has a strict minimum and maximum; SHC has no restriction for sampled values except how much it can move the mean each generation.

Setting	**XOR.R1**	**XOR.R2**	**XOR.R3**	**XOR.R4**
Struct. Rep.	-	CGP	CGP	CGP
Weight Rep.	-	BW	SHC	BW, SHC
Struct. Learn.	-	PIPE	PIPE	PIPE
Weight Learn.	-	PIPE	SHC	PIPE, SHC
GM	-	No	Yes	-
BW Range	$[-2,2]$	$[-2,2]$	$[-2,2]$	$[-8,8]$
Experiments	100	100	100	30
Runs	100	100	200	100
Generations	50	50	300	300
Pop. Size	30	30	5	5
GEP:				
Min. Arity	2	2	2	2
Max. Arity	3	3	3	3
Head Length	4	4	4	4
Genes	1	1	1	1
CGP:				
Arity	5	5	5	5
Rows	3	3	1	1
Columns	2	2	6	4
PIPE:				
LR, λ	0.01	0.01	0.005	0.0015
P_M	0.4	0.4	0.5	0.5
M_R	0.4	0.4	0.4	0.3
PBIL:				
LR	0.10	0.10	0.10	0.10
P_M	0.02	0.02	0.02	0.02
M_{shift}	0.05	0.05	0.05	0.05
GM:				
Probability, β	0.5	0.5	0.5	0.5
Pop. Percent	50%	50%	50%	50%

Table 4.3: XOR Configuration

Round	Test	Measure	Mean	Best	Worst	SD	Failures
XOR.R1	CGP.BW.PIPE	SR	17.0	27	9	4	
	CGP.BW.PIPE.GM	SR	16.4	28	10	3	
	GEP.BW.PIPE.GM	SR	9.4	16	3	2	
	GEP.BW.PIPE	SR	8.7	17	2	2	
XOR.R2	Pop. Size						
	Pop. Size 5	Evals	726	170	1270	257	71
	Pop. Size 30	Evals	784	207	1421	336	78
	Pop. Size 10	Evals	901	134	1408	336	73
	Graph Size						
	Row,Col: 1,6	Evals	669	272	1243	311	90
	Row,Col: 2,6	Evals	726	176	1154	358	91
	Row,Col: 1,8	Evals	795	296	1454	370	85
	PIPE Mutation						
	$P_M = 0.5, M_R = 0.4$	Evals	716	162	1300	335	76
	$P_M = 0.3, M_R = 0.4$	Evals	841	233	1499	439	86
	$P_M = 0.4, M_R = 0.3$	Evals	869	131	1489	481	83
XOR.R3	CGP.SHC.PIPE.GM	SR	28.8	41	18	4	
	GEP.SHC.PIPE.GM	SR	24.1	33	12	4	
	CGP.BW.PIPE.GM	SR	22.1	33	13	3	
	CGP.SHC.PIPE	SR	19.9	28	11	3	
	CGP.BW.PIPE	SR	15.3	23	7	3	
	PIPE LR						
	$\lambda = 0.02$	Evals	806	104	1442	344	151
	$\lambda = 0.005$	Evals	811	157	1444	370	151
	$\lambda = 0.0175$	Evals	824	2	1470	411	138
XOR.R4	$(\lambda = 0.015)$						
	GGP.BW.PIPE.GM	SR	59.4	66	50	3	
	GGP.BW.PIPE	SR	57.0	71	44	6	
	GGP.SHC.PIPE.GM	SR	23.0	30	12	4	
	GGP.SHC.PIPE	SR	18.6	30	9	4	
	$(\lambda = 0.0015)$						
	GGP.BW.PIPE.GM	SR	64.7	75	56	4	
	GGP.BW.PIPE	SR	63.2	73	52	4	
	GGP.SHC.PIPE.GM	SR	26.5	32	19	3	
	GGP.SHC.PIPE	SR	18.9	27	12	3	

Table 4.4: XOR Results (SR denotes success rate and Evals denotes mean evaluations).

The PIPE versus PBIL on BW weights result was that PIPE outperforms PBIL. For GM on elite or best, GM on the elite outperformed using the best by about 200 mean evaluations[2]. The graph size showed us that CGP prefers to have a single row with a number of columns (see Table 4.4). Intuitively, we can see that CGP can then use the neurons in any configuration, including one with a number of rows.

After fixing the parameters to their best performing values a third round was run, denoted XOR.R3, in which we tested: population size, graph size, PIPE learning rate, and PIPE mutation. These tests were done with 100 experiments and 100 runs so that we could measure the mean success rate. Population size 5 achieved a 28% success rate and was better than 10 and 20 with $\alpha = 1\%$. For graph size, increasing the number of neurons increases the success rate, but because we want to compare with GEP-NN we kept the size to 1 row, 4 columns. Therefore there are 4 neurons in the graph, similarly GEP-NN was setup to have a maximum of 4 neurons (head size was 4).

For the learning rate, there was inconsistency in the results, 0.02 and 0.005 had similar results (see Table 4.4). Therefore the final round tested with a low (0.0015) and a high (0.015) learning rate. The result showed that the low learning rate outperformed the high.

It is important to note that in GEP-NN tests presented in [4], there is no mention of a bias or evolved thresholds. For the XOR and 6-Mux experiments, it is possible to solve the problem without a bias. In the case of XOR, given the pair $\langle 0,0 \rangle$ as the input, the ANN *must* produce 0 at the output, because no neuron could ever be activated with those inputs. For our tests, we added a bias and this may affect the result and how compatible they are with GEP-NN. Since we added the bias, we also went ahead and made our neuron arity one higher to allow use of the bias.

Another point in which we have tainted the comparability of the results is by increasing the weight range from $-2..2$ to $-8..8$. The SHC method of weight learning has no such restraint on the weight values. It would be challenging (and contrary to performance) to restrain SHC

[2]Detailed results are not in the table in order to reduce the amount of numbers inundating the reader.

from moving the mean and sampling values outside of the range and so we have not done so. Unfortunately, by restraining BW, we were unable to compare BW and SHC directly. SHC was performing better (on 6-MUX and Retina) until we discovered that the reason might be its use of values outside of the range. For the final round, we tested with a range of $-8, 8$ on the BW weights and its performance vastly improved making it the highest performing system.

For a comparison between GEP-NN and the system presented here see Section 4.6. The CGP.BW.PIPE.GM result from XOR.R4 was compared with the GEP-NN result.

4.3 Single Pole Balancing

Tests for Single-Pole Balancing (SPB) proceeded just as they did with XOR. For SPB, the maximum number of evaluations was chosen to be comparable with SANE [32], and its value was 50,000. In general, a negligible number of runs failed therefore success rate is not considered here, and in its place we use mean evaluations to measure performance. The failure rate is reported for completeness.

The first round of experimentation, denoted SPB.R1, was a naive run that tested all possible systems with parameters chosen based on the previous study of the methods [1, 4, 29, 38, 40]. SPB.R1 revealed CGP.BW.PIPE to have the highest performance and it became the system of study for the remaining rounds. Round 2 tested CGP.BW.PIPE on several parameters followed by a third round that used the parameters discovered in the second round. The fourth round illustrates the complexity of optimizing the parameters: we used the parameters as indicated by SPB.R3 and this resulted in worse performance. The complexity comes from the interplay of parameters and from the fact that the results are often not statistically significant and so are not definitive. The fifth and final round used settings from the SPB.R3 result with the lowest mean evaluations. It was decided that since the differences were statistically insignificant and not any better than SPB.R3, it was unlikely we'd find a better result, therefore we halted our optimization. Settings for the five rounds of experimentation can be found in Table 4.5 and a detailed analysis of the results follows.

Setting	**SPB.R1**	**SPB.R2**	**SPB.R3**	**SPB.R4**	**SPB.R5**
Struct. Rep.	-	CGP	CGP	CGP	CGP
Weight Rep.	-	BW	BW	BW	BW
Struct. Learn.	-	PIPE	PIPE	PIPE	PIPE
Weight Learn.	PBIL	PIPE	PIPE	PIPE	PIPE
GM	-	No	No	No	No
BW Range	$[-2,2]$	$[-2,2]$	$[-2,2]$	$[-2,2]$	$[-2,2]$
Runs	100	100	100	100	100
Generations	250	250	5000	3334	5000
Pop. Size	200	200	10	15	10
GEP:					
Min. Arity	1	-	-	-	-
Max. Arity	5	-	-	-	-
Head Length	10	-	-	-	-
Genes	1	-	-	-	-
CGP:					
Arity	5	5	5	5	5
Rows	3	3	1	1	1
Columns	3	3	12	10	12
PIPE:					
LR, λ	0.01	0.01	0.015	0.015	0.015
P_M	0.4	0.4	0.5	0.4	0.5
M_R	0.4	0.4	0.4	0.3	0.4
PBIL:					
LR	0.10	-	-	-	-
P_M	0.02	-	-	-	-
M_{shift}	0.05	-	-	-	-
GM:					
Probability, β	0.5	-	-	-	-
Pop. Percent	50%	-	-	-	-

Table 4.5: SPB Configuration

The first round (SPB.R1) found that CGP.BW.PIPE had the highest performance with 5940 mean evaluations. The difference in mean evaluations between CGP.BW.PIPE.GM and GEP.BW.PBIL.GM was not statistically significant ($\alpha = 1\%$). See Table 4.6 for the results of the first and second rounds.

During round 2, the same set of tests was run on SPB as was XOR: population size, graph size, Guided Mutation, using PIPE or PBIL for bitwise weights, GM on best or elite, PIPE mutation, PIPE learning rate. Results of which can be seen in Table 4.6.

The second round of SPB experiments, denoted SPB.R2 , showed that: small populations performed better than larger populations, PIPE performed better on the bitwise weights, graphs perform best with a single row, and that using the best of generation or the elite for GM made a negligible difference.

For the third round of experiments, denoted SPB.R3, the settings were tuned in accordance with the results from SPB.R2. We also added a test to see if additional weights would benefit the system. Performance differences between systems with more weights was not statistically significant with $\alpha = 1\%$ (values tried: $10-25$ with an interval of 5). Results for SPB.R3 and the remaining rounds can be found in Table 4.7.

In SPB.R3, we found that a GM system performed worse than the same system without GM (with statistical significance, $\alpha = 1\%$). A possible reason is that the SPB fitness function is randomized. It is possible, even likely, that the best of generation or the elite individual, had a higher fitness by random chance. Using GM with an individual whose fitness is not in proportion to its performance would be contrary to the purpose of GM.

We further tweaked the PIPE Mutation and Learning Rate. Interestingly, using a combination of the values from SPB.R3 resulted in a higher mean evaluations in SPB.R4. Therefore, we back tracked and based SPB.R5 on the settings with the lowest mean. As can be seen in Table 4.7, the results of SPB.R5 are worse or vary minutely from those of SPB.R3. We used the best result of SPB.R5 in our comparison to SANE (see Section 4.6).

4.4 Six-Bit Multiplexer

For the Six-Bit Multiplexer benchmark (herein referred to as MUX), there were only two rounds of experimentation: one naive run without any optimization and one based on optimizations learned

Round	Test	Mean	Best	Worst	SD	Failures
SPB.R1	CGP.BW.PIPE	5940	499	24081	3620	0
	CGP.BW.PIPE.GM	6847	1158	36508	5136	0
	GEP.BW.PBIL.GM	7000	1708	20992	3330	0
SPB.R2	Pop. Size					
	Population Size 20	2061	211	7414	1514	0
	Population Size 10	2076	302	5344	1390	0
	Population Size 30	2099	111	10383	1816	0
	Graph Size					
	CGP Rows,Columns: 1,12	4045	594	21563	3944	1
	CGP Rows,Columns: 2,6	4285	277	22522	4239	0
	CGP Rows,Columns: 2,4	4927	668	34043	5483	0
	CGP Rows,Columns: 1,8	5021	802	38389	6308	0
	PIPE Mutation					
	$P_M = 0.5, M_R = 0.4$	3592	603	10967	2368	0
	$P_M = 0.5, M_R = 0.3$	3703	344	13293	2759	0
	$P_M = 0.4, M_R = 0.4$	3765	524	30636	4433	0
	PIPE LR					
	$\lambda = 0.015$	2961	340	12102	2780	0
	$\lambda = 0.0175$	3807	413	16326	3723	0
	$\lambda = 0.0025$	3821	781	12620	2565	0
	PIPE vs. PBIL on BW					
	PIPE	5165	624	43871	6041	0
	PBIL	5856	454	21440	3371	0
	GM on Elite or Best					
	GM on Elite	6093	524	29638	6119	0
	GM on Best	6138	524	23292	5107	0

Table 4.6: SPB Results (Part 1)

Round	Test	Mean	Best	Worst	SD	Failures
SPB.R3	CGP.BW.PIPE	1851	332	5905	1323	0
	CGP.BW.PIPE.GM	3095	97	18102	3379	1
	Pop. Size					
	Population Size 15	2030	113	6819	1422	0
	Population Size 10	2087	189	9426	1485	0
	Population Size 20	2261	63	11246	1787	0
	Population Size 5	2672	92	15880	2537	0
	Graph Size					
	CGP Rows,Columns: 1,10	2075	164	8351	1514	0
	CGP Rows,Columns: 1,12	2133	251	10131	1697	0
	CGP Rows,Columns: 1,6	2367	169	8748	1699	0
	CGP Rows,Columns: 1,8	2596	275	9356	1706	0
	PIPE Mutation					
	$P_M 0.4, M_R 0.3$	1686	81	9132	1187	0
	$P_M 0.6, M_R 0.3$	1882	266	5872	1294	0
	$P_M 0.4, M_R 0.4$	1933	133	7864	1338	0
	PIPE LR					
	$\lambda = 0.0175$	1949	22	6707	1347	0
	$\lambda = 0.015$	2027	211	9250	1498	0
	$\lambda = 0.02$	2066	242	14280	1784	0
SPB.R4	$\lambda = 0.015$	2213	117	29697	3284	0
	$\lambda = 0.0175$	2996	153	40784	4812	0
SPB.R5	$P_M = 0.6, M_R = 0.3$	1657	105	8766	1373	0
	$P_M = 0.4, M_R = 0.3$	2041	108	9587	1615	0
	$P_M = 0.7, M_R = 0.3$	2138	141	9377	1685	0
	Pop. Size 20	1993	89	7784	1573	0
	$\lambda = 0.0175$	1827	154	7440	1317	0

Table 4.7: SPB Results (Part 2)

from the XOR and SPB experiments. Settings for the naive run were gathered from [1,4,29,38,40]. Table 4.8 summarizes the settings for the rounds.

Setting	**MUX.R1**	**MUX.R2**
Struct. Rep.	-	CGP
Weight Rep.	-	BW; SHC
Struct. Learn.	-	PIPE
Weight Learn.	PBIL	PIPE
GM	-	-
BW Range	$[-2,2]$	$[-8,8]$
Experiments	50	30
Runs	50	50
Generations	2000	10000
Pop. Size	50	10
GEP:		
Min. Arity	2	-
Max. Arity	3	-
Head Length	5	-
Genes	4	-
CGP:		
Arity	5	5
Rows	4	1
Columns	6	20
PIPE:		
LR, λ	0.01	0.01
P_M	0.4	0.5
M_R	0.4	0.3
PBIL:		
LR	0.10	0.015
P_M	0.02	0.02
M_{shift}	0.05	0.05
GM:		
Probability, β	0.5	0.5
Pop. Percent	50%	50%

Table 4.8: 6-MUX Configuration

The MUX results are summarized in Table 4.9. The second round of experiments, denoted MUX.R2, showed a considerable increase in performance over MUX.R1: for CGP.SHC.PIPE.GM there was a 40% increase in success rate, and a 30% reduction in mean evaluations. CGP.BW.PIPE and its GM counterpart showed considerable gains in MUX.R2 over their performance in MUX.R1 that do not appear to be the result of the parameter optimization. Instead, it is likely the results are attributed to the increased range of the bitwise weight encoding from [-2,2] to [-8,8].

Round	**Test**	**Measure**	**Mean**	**Best**	**Worst**	**SD**	**Failures**
MUX.R1	CGP.SHC.PIPE.GM	SR	64.3	76	42	7	
	CGP.SHC.PIPE	SR	63.4	74	52	6	
	GEP.SHC.PIPE.GM	SR	42.4	56	26	6	
	GEP.BW.PIPE.GM	SR	38.5	50	24	5	
	GEP.BW.PBIL.GM	SR	34.6	52	22	6	
	GEP.SHC.PBIL.GM	SR	33.7	50	20	7	
	GEP.BW.PBIL.GM	Evals	53K	16K	95K	27K	33
	GEP.BW.PIPE	Evals	53K	14K	87K	21K	37
	CGP.SHC.PIPE	Evals	57K	30K	99K	16K	22
	CGP.SHC.PIPE.GM	Evals	63K	28K	94K	20K	17
	GEP.SHC.PIPE.GM	Evals	71K	27K	98K	20K	32
	GEP.SHC.PBIL.GM	Evals	80K	60K	99K	13K	39
MUX.R2	CGP.SHC.PIPE	SR	91.7	98	80	4	
	CGP.SHC.PIPE.GM	SR	91.4	98	82	3	
	CGP.BW.PIPE.GM	SR	88.9	96	80	5	
	CGP.BW.PIPE	SR	87.4	92	80	3	
	CGP.SHC.PIPE	Evals	54K	26K	95K	16K	1
	CGP.SHC.PIPE.GM	Evals	43K	14K	87K	16K	5
	CGP.BW.PIPE.GM	Evals	30K	5K	87K	15K	9
	CGP.BW.PIPE	Evals	39K	8K	92K	12K	9

Table 4.9: 6-MUX Results (SR denotes success rate; Evals denotes mean evaluations.)

MUX.R2 results are interesting as a comparison between the systems. SHC based systems achieved a higher success rate and at the same time, the BW based systems had a lower mean evaluations. The success rate is possibly attributable to the unrestricted values of SHC. The difference may be small because it is so rare to evolve to a solution that requires a weight with absolute value greater than 8. Other possible reasons for the difference exist, including parameter optimization that favours SHC. The lower mean evaluations is likely because PIPE compensates for the low probability of the individual it is adapting toward. SHC will move the mean a set percentage no matter how distant the target mean is.

It is clear from the results that GM was able to reduce the number of evaluations the systems required to find a solution. A greater than 20% reduction in mean evaluations occurred in both cases.

4.5 Retina Problem

The Retina Problem had two rounds similar to those of the MUX experiment. The first was naive (based on [1, 4, 29, 38, 40]), except that the population size was set to 10^3. We have denoted the first round: RET.R1, and the second: RET.R2. The settings can be found in Table 4.10.

Figure 4.1.a shows a solution created with CGP.SHC.PIPE.GM and taken from the first round. The result was arranged to emphasize the modularity of the two networks (one for left and one for right) which only have a couple of connections crossing between modules. Figures 4.1.b and 4.1.c show results from the second round. There are less distinct layers in these examples because the structure of the graph was a single row with 13 columns, as opposed to 4 rows and 4 columns in the first round. There is a similar level of modularity in the networks even without the rows constraining topology. There is not much difference in modularity when using GM and when not using GM as shown in Figures 4.1.b and 4.1.c.

Quantitative results for the Retina Problem can be found in Table 4.11 and a qualitative analysis of the solutions compared to other methods can be found in Section 4.6.

CGP.SHC.PIPE and its GM counterpart performed similarly in both rounds, but there was a noticeable improvement, likely due to the parameter settings taken from the previous experiments. CGP.BW.PIPE and its GM counterpart had a vast improvement in the second round. The likely reason for CGP.BW.PIPE's improvement is the increase in range of the bitwise weight encoding.

Since there is only one data point for the success rate, we cannot comment on the difference

[3]We had previously determined that a small population size performed best.

Setting	**RET.R1**	**RET.R2**
Struct. Rep.	-	CGP
Weight Rep.	-	BW; SHC
Struct. Learn.	-	PIPE
Weight Learn.	PBIL	PIPE
GM	-	-
BW Range	$[-2,2]$	$[-8,8]$
Runs	100	100
Generations	250000	250000
Pop. Size	10	10
GEP:		
Min. Arity	1	-
Max. Arity	5	-
Head Length	13	-
Genes	1	-
CGP:		
Arity	5	5
Rows	4	1
Columns	4	13
PIPE:		
LR, λ	0.01	0.015
P_M	0.4	0.5
M_R	0.4	0.3
PBIL:		
LR	0.10	-
P_M	0.02	-
M_{shift}	0.05	-
GM:		
Probability, β	0.5	0.5
Pop. Percent	50%	50%

Table 4.10: Retina Problem Configuration

Round	Test	Mean	Best	Worst	SD	Failures
RET.R1	CGP.SHC.PIPE.GM	950K	169K	2.4M	N/A	6
	CGP.SHC.PIPE	1.4M	455K	2.5M	N/A	19
	CGP.BW.PIPE.GM	733K	55K	2.3M	11K	21
	GEP.SHC.PIPE.GM	1.2M	420K	2.5M	18K	65
	CGP.BW.PIPE	1.3M	411K	2.4M	N/A	70
	GEP.BW.PIPE.GM	995K	75K	2.5M	20K	74
	GEP.SHC.PBIL.GM	1.2M	76K	2.0M	N/A	94
RET.R2	CGP.SHC.PIPE.GM	928K	149K	2.2M	N/A	4
	CGP.BW.PIPE.GM	567K	43K	2.3M	4.6K	7
	CGP.SHC.PIPE	1.3M	337K	2.5M	N/A	20
	CGP.BW.PIPE	916K	162K	2.2M	N/A	21

Table 4.11: Retina Problem Results

between SHC and BW systems as far as success rate is concerned. However, the results expand on the MUX results, by stressing the systems on a larger problem.

Performing the Retina Problem, GM reduced the SHC mean evaluations result by approximately 28% and the BW result by 38%, much more than GM reduced the mean evaluations performing MUX. Even though we only have one data point to consider for success rate, the difference between the success rate for the systems with GM versus without GM is quite large and is likely part of a trend. It would be interesting to gather more data points and show that without GM the success rate is much lower since this is the first problem where GM has made such a a great impact on success rate.

We used the GM results from RET.R2 in our comparisons with third-party systems in the following section.

4.6 Comparisons

Table 4.12 summarizes and compares our results for XOR, SPB, MUX, and Retina Problem with those reported by Ferreira's GEP-NN [4]; Moriarty and Miikkulainen's SANE [32]; Kashtan and

Alon's function-switchings [18]; and Verbancsics and Stanley's HyperNEAT-LEO [47].

Assuming a similar standard deviation to CGP.BW.PIPE.GM for GEP-NN on XOR, the difference between the two (CGP.BW.PIPE.GM and GEP-NN) is statistically significant with $\alpha = 1\%$. For SPB, the difference between SANE and CGP.BW.PIPE is not statistically significant, with $\alpha = 1\%$. For the MUX problem, the difference between CGP.SHC.PIPE and GEP-NN is statistically significant, $\alpha = 1\%$ (again assuming a similar standard deviation for both systems). Our system is worse on XOR, similar on SPB, and better on MUX.

Problem	**Method**	**Mean**	**Best**	**Worst**	**SD**	**Failures**
XOR	GEP-NN	77%	-	-	N/A	
	GGP.BW.PIPE.GM	65%	75	56	4	
SPB	SANE	1691	46	4461	984	0
	CGP.BW.PIPE	1657	105	8766	1373	0
MUX	GEP-NN	6%	-	-	N/A	
	CGP.SHC.PIPE	92%	98	80	4	
RET	HyperNEAT-LEO	2.5M[a]	-	-	-	9
	Function-Switching	1.7M	-	-	-	-
	CGP.SHC.PIPE.GM	928K	149K	2.2M	N/A	4
	CGP.BW.PIPE.GM	567K	43K	2.3M	4.6K	7

Table 4.12: Results Comparison

[a]This is the maximum evaluations used, and a mean evaluations was not reported.

For the Retina Problem, and Kashtan and Alon's function-switching method [18], our result is statistically significant, $\alpha = 1\%$, assuming they had a number of runs greater than 30 and a similar standard deviation. Though it should be noted that the purpose of their result was to illustrate a method of spontaneously evolving modularity; their resulting solutions also appear more modular (see rationale in the next paragraph and in Figure 4.1). The function-switching method's network appeared similar to that of HyperNEAT-LEO in Figure 4.2.b.

We have insufficient information to make a claim of statistical significance with HyperNEAT-LEO. We can, however, point out that our system has not been re-tooled in order to induce modularity; it is simply an inherent property of the EDA proposed in this work to capture modularity. While our sample solution does not appear as modular as those from HyperNEAT-LEO (Figure 4.2.b), it only has two connections crossing between the two sides of the network which indicates modules for the left and right retinas (see Figure 4.1). This was quite a good result considering HyperNEAT (without the locality constraints) failed to discover modularity and resulted in a fully connected network (Figure 4.2.a).

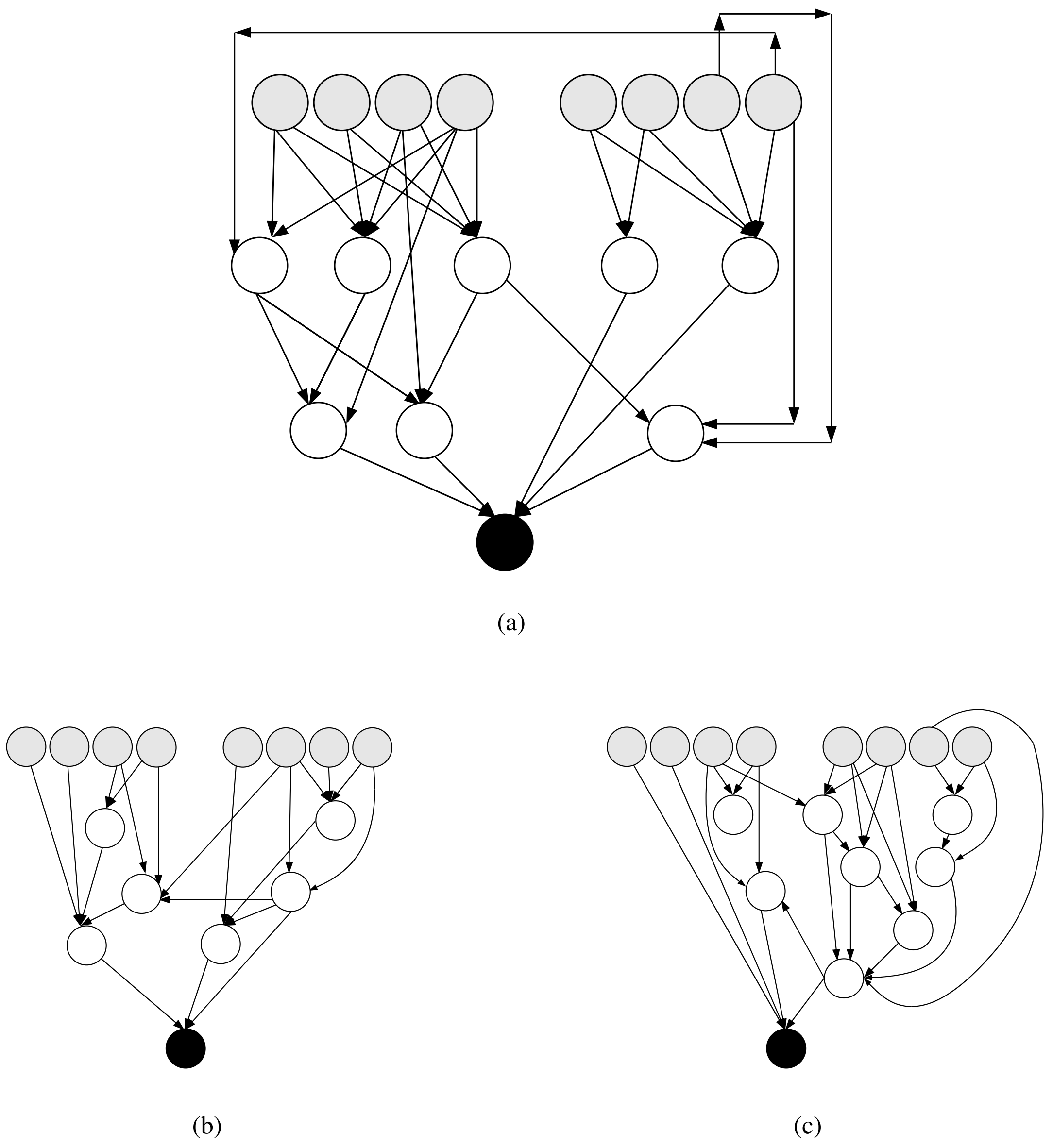
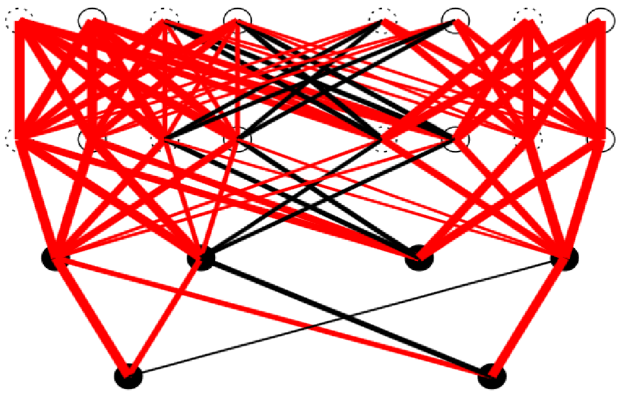
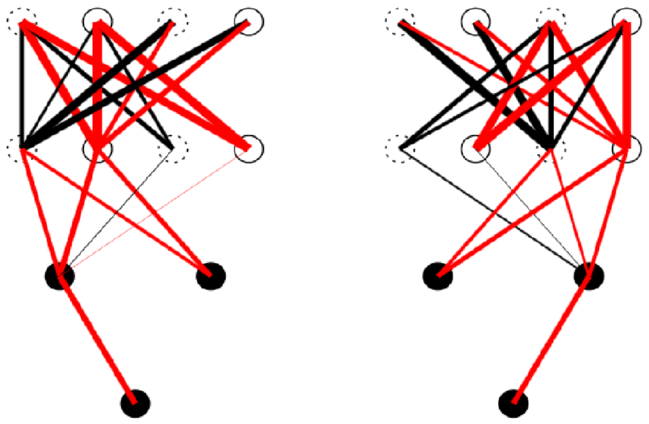


Figure 4.1: Example Retina Solutions: (a) from RET.R1, CGP.SHC.PIPE.GM; (b) from RET.R2, CGP.BW.PIPE.GM; (c) from RET.R2, CGP.BW.PIPE.



(a) Without locality constraints.



(b) With locality constraints.

Figure 4.2: Example Solutions from HyperNEAT-LEO. The outline of the node indicates its 3-D position. [47]

Chapter 5

Conclusions and Future Work

This work presented a study of Neuroevolution by means of Estimation of Distribution Algorithms. The analysis was divided between the representation of the structure (GEP and CGP) and the weights (Bitwise and Stochastic Hill Climbing) and two learning algorithms for estimating the population (PBIL and PIPE). Also included was an analysis of how the Guided Mutation operator could be used to improve the quality of the results.

Experiments showed that the dominant systems were CGP.BW.PIPE and CGP.SHC.PIPE. GM could be used on either, but results regarding the effectiveness of using GM varied.

On XOR GM improved the success rate of systems using SHC more than for the systems using BW. On SPB, GM was shown to cause a reduction in performance. The MUX results showed a negligible difference in success rate for both CGP.BW.PIPE and CGP.SHC.PIPE between the GM and without GM system, but a reduction in mean evaluations (of about 10,000) for systems with GM with a negligible difference in standard deviation. Results from the Retina Problem showed that CGP.BW.PIPE and CGP.SHC.PIPE both demonstrated a considerable improvement in success rate and a large reduction in mean evaluations with GM.

The XOR result could be related to the mean move parameter of SHC. Given the mean is slowly moving toward the optimal value, sampling is unlikely to find a good value until enough generations have passed. GM avoids re-sampling and uses a previously successful weight-value. Given more evaluations, XOR might not have shown as big a difference in success rate since there would be enough generations for the mean to move to the appropriate value. The SPB result is likely attributed to the random nature of the start states of the fitness function. If GM is used on an individual which by chance had an easier[1] set of start states, it might be exploring the area surrounding an individual that does not merit its fitness value. The Retina and MUX problems both have fixed fitness functions and the results showed how well GM can accelerate optimization for such problems.

The Bitwise Weight representation outperformed SHC on the XOR problem with a large improvement in success rate (140% increase). On the MUX problem there was a negligible difference in success rate between BW and SHC (though SHC is higher, by more than 3%), but the mean evaluations was about 10K lower for the BW representation. The Retina Problem showed similar success rates for SHC and BW, but a significantly lower mean evaluations for the BW representation.

The XOR result can be attributed to the small number of evaluations available and the slow speed of the SHC mean move. The MUX and Retina results can be attributed to a slow speed of mean movement, but might also be affected by the speed at which the standard deviation is reduced. Tests could be done with a higher mean move value, but it's likely BW with PIPE will still outperform it. One reason is that the mean move is based on the mean of the top 3 values, which can have conflicting values (*e.g.,* -2 and 2), although this is somewhat mitigated by the fact that there are 3 values and so the third value will likely side with one or the other of the conflicting values. The second possibility is the method of updating of SHC versus PIPE. When PIPE adapts to a new best individual that has a low probability it compensates for the low probability and raises the probabilities considerably. SHC, on the other hand, would require that the good value (or one in its range) be discovered twice before the mean of the best is closer to the new value, and even after this, SHC does not compensate for a large difference between the means and will more slowly move the mean toward the new value.

CGP outperformed GEP in all scenarios. Notwithstanding, considering the no-free-lunch the-

[1]Easier here means that achieving a higher fitness is made simpler by the nature of the start state. The state could begin at rest or with the pole at 11° with a positive pole velocity (the limit being 12°).

orem [51], it is possible had we optimized a given parameter setting for a GEP system, we could have found a problem where GEP outperformed CGP.

CGP performed better than GEP, and this can be attributed to a few points. First, the GEP string: when values in the first couple of elements of the head change, the structure of the tree can change considerably and this may disrupt the learning process. CGP has no such problem. Second, CGP can re-use nodes (or nodes can have multiple outputs), therefore GEP might have to evolve functionality twice where CGP only needs to do so once (plus evolving the second connection). Third, on multiple output problems, CGP can re-use functional nodes as described previously. Finally, even if GEP could be made to perform at a similar level, CGP's ability to have connections cross multiple levels of the graph makes it a more powerful representation allowing the learning algorithm to discover structures that do not resemble trees.

PIPE performed better than PBIL. PBIL does not compensate for the current probability of an individual, therefore it is slower to respond to a new, unlikely solution. If it's too slow it will lose that individual if it does not get re-sampled in the next generation.

Therefore, we've argued that CGP.BW.PIPE is the most recommendable system of those tested and that GM should be used for problems with a fixed fitness test, not those with a stochastic fitness test.

Finally, it is interesting to note the modularity that appeared as an emergent property of using an EDA to optimize the solution. In population-based methods it is possible that genetic code can be preserved in a fit individual as long as its presence does not reduce the individual's fitness considerably. Since EDAs re-sample the population every generation, this junk code is likely to be discarded unless it is sampled in good solutions often enough that it becomes represented in the model. This is one possible explanation why EDA are better at finding modular solutions than population-based approaches.

5.1 Contributions

We have presented CGP.BW.PIPE as a novel, and well performing EDA-NE technique. We've also shown how the Guided Mutation operator can be used to accelerate optimization in EDAs, particularly in problems with a fixed fitness function.

We have shown that EDAs are particularly good at solving problems that can benefit from the use of modularity. Also, that EDAs can perform as well as population-based approaches at NE and scale well to harder problems, likely due to the ability to discover modularity.

This work has illustrated and evaluated several methods of performing Neuroevolution with an EDA. The work can be used to guide future work in the area.

5.2 Future Work

Our results have shown the performance of an EDA using independent variables; it would be interesting to see the results of an EDA that is able to capture dependencies between the variables. Such techniques may be able to address problems that require highly structured solutions.

The approach presented evolved topology and weights simultaneously, but the approach was unable to determine the contribution by the set of weights or the structure in the chosen individual. A better approach might be one that promotes cooperation and specialization of topologies and weights. Previous works have shown that cooperation is key for the induction of ANN topology and weights [32, 37].

The CGP representation has many benefits that come from its fixed-size indirect encoding, but a fixed-size can be detrimental to continuous evolution. NEAT has been shown to be a powerful representation with no limitation to the size of the network and no need to specify the network size *a priori*. NEAT is capable of evolving continuously using competitive individuals, creating a evolutionary technology race. In order to add this feature, we could extend the CGP representation to evolve size simultaneously. It is also possible that the representation used by NEAT could be optimized with an EDA approach.

Through experimentation, it was shown that starting evolution with minimal structures was advantageous in Neuroevolution with NEAT. Although the PIPE methodology did not use a fixed-length, it did not start evolution with minimal size. It would be interesting to see what affect starting minimally would have on an EDA.

Speciation is also an important part of the NEAT algorithm.When using Guided Mutation, our system has implemented a simple method of speciation where a species is developed around the elite individual. This is contrary to the purpose of speciation in NEAT, which is to preserve innovation in low fitness individuals that might be lost to selection. EDA are unlike population-based methods in that EDA start by representing the entire search-space and incrementally constrain the space to the area of highest fitness, whereas population-based approaches start with a number of randomly selected individuals and through mutation and crossover traverse the search-space. Thus our use of GM on the elite individual.

While population-based methods might use speciation to encourage exploration, an EDA could use speciation to improve the model's convergence on a peak. In our case, we have used GM to represent a species around the current elite. It is possible that we could use more than just the current elite as a species representative. That way we could explore more peaks and reduce the risk involved in exploring a single peak. This would be especially beneficial in search-spaces with many peaks.

EDA-NE has shown strong performance on feed-forward ANN, but we have not examined the use of recurrent connections. Recurrent connections provide the neuron with its output from the previous time-step as an input. This allows the network to perform a task such as pole balancing without the velocity information (for the cart and the pole). The CGP-ANN [20] has an encoding for recurrent connections, and so it should be trivial to implement with our EDA.

The result of switching the fitness function to induce modularity is quite interesting. It would be of interest to use the same setup with our system. While, the switching may produce better modularity, the switching fitness function may reduce the performance of GM since it might continue to explore the area of a solution for the previous fitness function. The simplest solution to that would be to use GM on the best individual from the previous generation, not the elite individual.

Our EDA-NE study has been limited to only 4 problems thus far. Given this method's favourable performance on the Retina Problem, it would be interesting to test it further on other large scale problems. Of particular interest would be problems requiring the discovery of modular solutions.

Glossary

activation function is a mathematical function applied to the weighted sum of the inputs to a neuron and defines the output of the neuron, *i.e.* the neuron's level of activation.

algorithm is a well-defined, ordered set of instructions for performing a task or calculation.

arity refers to the number of inputs to a function, neuron, or node.

Artificial Neural Network (ANN) is a type of computer program inspired by the analogue found in biological nervous systems. They have a number of inputs and outputs and can be used for function approximation and as robotic controllers, among other things. In this work, Artificial Neural Network is a synonym of Neural Network.

building blocks Building blocks are particular values for a portion of the representation that will result in an above-average fitness for the individual..

Cartesian Genetic Programming (CGP) is a Genetic Programming technique the uses a fixed-length, indirect encoding of programs structured as graphs. See Section 3.1.2.

CGP-ANN is an extension to Cartesian Genetic Programming (CGP) for the purpose of evolving Artificial Neural Networks.

controller is software and/or hardware used to coordinate the actions of a robot or simulated agent.

crossover is a genetic operator that combines two genetic encodings resulting in one or more different encodings; also known as recombination.

encoding is the computerized representation of an individual, *e.g.* a LISP program tree. An indirect encoding is an encoding that must be transformed to become the individual whose performance is evaluated.

Estimation of Distribution Algorithms (EDA) is an Evolutionary Algorithms approach that replaces genetic operators and populations with a probabilistic model from which to sample individuals. See Section 2.2.

Evolutionary Algorithms (EA) is the study of algorithms inspired by natural evolution for use in optimization problems; *i.e.* population-based search techniques.

Evolutionary Computation (EC) is a field of Artificial Intelligence that studies population-based search techniques inspired by biological evolution.

game playing is the participation in a competition with a number of rules and a winning scenario, including Chess, mazes, and complex simulations.

Gene Expression Programming (GEP) is a Genetic Programming technique that uses a fixed-length, indirect encoding of program trees. See Section 3.1.1.

Genetic Algorithms (GA) is the subfield of Evolutionary Algorithms that studies the optimization of a value (or set of values) by manipulating a population of encoded solutions; the population is manipulated by methods inspired by evolution, *e.g.*, mutation, crossover, and selection of the fittest. See Section 2.1.4.

genetic operators are methods in population-based search techniques used to manipulate the population through mutating and selecting individuals; see also search operators.

Genetic Programming (GP) is the subfield of Evolutionary Algorithms that optimizes computer programs to perform a given task. See Section 2.1.5.

genotype is an encoding that represents an individual, but does not get evaluated for fitness. The genotype is transform in the phenotype which is then evaluated for fitness. Then it is the genotype which is manipulated to generate new individuals (via crossover and mutation).

GEP-NN is an extension to Gene Expression Programming (GEP) for the purpose of evolving Neural Networks.

Guided Mutation (GM) is a search operator for Estimation of Distribution Algorithms that is meant to explore the search space around a given individual or encoding.

hidden neuron is a neuron that is not an input or an output from a neural network.

HyperNEAT is an extension to Neuro-Evolution of Augmenting Technologies meant to induce regularity in solutions.

HyperNEAT-LEO is an extension to Neuro-Evolution of Augmenting Technologies meant to induce modularity by constraining connections to geometrically-local connections (if possible).

input-output pairs are a set of values indicating the input to the program and the expected output from the program.

Machine Learning (ML) is a field of study that explores technology capable of improving itself at a given task.

modularity is the encapsulation of function in programs, including Neural Networks. See Section 2.4.

mutation is a genetic operator whose method takes a genetic encoding and changes at least a single element resulting in a different encoding.

Neuro-Evolution of Augmenting Technologies (NEAT) is a population-based, Neuroevolution technique capable of evolving Topology and Weights of a Neural Network. See Section 2.3.5.

Neuroevolution (NE) is Genetic Programming applied to the optimization of Artificial Neural Networks.

neuron is the base computing device in Artificial Neural Networks. They perform a weighted sum of their inputs and output the result of an activation function applied to that sum.

optimization is any method of finding or selecting a value considered the best by a given measure.

phenotype is the representation of the individual that is evaluated for fitness and is generated from the genotype. For example, the genotype could be a set of numbers representing the weights of connections in a neural network. The evolutionary process searches for the correct weights. The weights are encoded in the genotype, but the fitness value comes from evaluating the neural network with the genotype's weights.

Population Based Incremental Learning (PBIL) is an Estimation of Distribution Algorithms approach to Genetic Algorithms canonically used to model bit strings. See Section 3.4.2.

population-based search techniques are methods of finding solutions (optimization) that uses a set of candidate solutions to aid its search.

Probabilistic Incremental Program Evolution (PIPE) is an Estimation of Distribution Algorithms approach to Genetic Programming canonically used to model program trees. See Section 3.4.1.

program is an implementation of an algorithm.

regularity is the repetition or re-use of form or function. See Section 2.4.

Reinforcement Learning (RL) is a Machine Learning methodology that is concerned with how an agent can improve its performance at acting on a given environment based on the state of the environment and occasional rewards.

search operators are methods in population-based search techniques used to generate the individuals and guide the search; see also genetic operators.

selection pressure is the effect created by choosing the best individuals of a population for the subsequent generation; this effect produces a population of higher fitness than the previous.

stochastic hill climbing is a method of optimizing real values via incrementally updated mean and standard deviation pairs.

Symbiotic Adaptive Neuro-Evolution (SANE) is a Neuroevolution technique capable of some topology evolution and full weight evolution. See Section 2.3.4.

Bibliography

[1] S. Baluja and R. Caruana. Removing the genetics from the standard genetic algorithm. In *Proceedings of the Twelfth International Conference on Machine Learning*, 1995.

[2] R. Barate and A. Manzanera. Evolution of visual controllers for obstacle avoidance in mobile robotics. *Evol. Intel.*, 2(3):85–102, Dec 2009.

[3] J. Clune, B. Beckmann, P. McKinley, and C. Ofria. Investigating whether hyperneat produces modular neural networks. In *GECCO '10: Proceedings of the 12th annual conference on Genetic and evolutionary computation*, Jul 2010.

[4] C. Ferreira. Designing neural networks using gene expression programming. In *Applied Soft Computing Technologies: The Challenge of Complexity*, volume 34 of *Advances in Soft Computing*, pages 517–535. Springer Berlin / Heidelberg, 2006.

[5] C. Ferreira. *Gene Expression Programming: Mathematical Modeling by an Artificial Intelligence*. Springer, 2006.

[6] E. Ghoulbeigi and M. dos Santos. Probabilistic developmental program evolution. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010.

[7] E. Ghoulbeigi and M. Santos. Probabilistic developmental program evolution. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, Mar 2010.

[8] F. Glover and M. Laguna. Tabu search. *Modern heuristic techniques for combinatorial problems*, 1993.

[9] D. Goldberg. Genetic algorithms in search, optimization, and machine learning. *Addison-Wesley Professional*, 1989.

[10] F. Gomez and J. Schmidhuber. Accelerated neural evolution through cooperatively coevolved synapses. *The Journal of Machine Learning Research*, 9:937–965, June 2008.

[11] P. Grouchy and G. D'Eleuterio. Supplanting neural networks with odes in evolutionary robotics. *Simulated Evolution and Learning 2010, LNCS 6457*, 2010.

[12] F. Gruau, D. Whitley, and L. Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In *Proceedings of the First Annual Conference on Genetic Programming*, 1996.

[13] S. Harding, J. Miller, and W. Bangzhaf. Self-modifying cartesian genetic programming. In *GECCO'07: Proceedings of the 2007 conference on Genetic and Evolutionary Computation*, Jul 2007.

[14] M. A. Hasanat, S. Harun, D. Ramachandram, and M. Rajeswari. Object class recognition using neat-evolved artificial neural network. In *Fifth International Conference on Computer Graphics, Imaging and Visualisation, 2008. CGIV '08.*

[15] G. Holker and M. Santos. Toward an estimation of distribution algorithm for the evolution of artificial neural networks. In *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, May 2010.

[16] J. Holland. Genetic algorithms. *Scientific American*, Jul 1992.

[17] S. Johns and M. Santos. On the evolution of neural networks for pairwise classification using gene expression programming. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, Jul 2009.

[18] N. Kashtan and U. Alon. Spontaneous evolution of modularity and network motifs. In *Proceedings of the National Academy of Science*, volume 102, Sep 2005.

[19] G. Khan, J. Miller, and D. Halliday. A developmental model of neural computation using cartesian genetic programming. In *GECCO '07: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, Jul 2007.

[20] M. Khan and G. Khan. A novel neuroevolutionary algorithm: Cartesian genetic programming evolved artificial neural network. In *FIT '10: Proceedings of the 8th International Conference on Frontiers of Information Technology*, Dec 2010.

[21] M. Khan, G. Khan, and J. Miller. Evolution of neural networks using cartesian genetic programming. In *2010 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, july 2010.

[22] R. Koppejan and S. Whiteson. Neuroevolutionary reinforcement learning for generalized helicopter control. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 145–152, 2009.

[23] J. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112, 1994.

[24] J. Koza. Genetic programming: On the programming of computers by means of natural selection. *The MIT Press, Cambridge, MA*, Dec 1996.

[25] J. Koza and R. Poli. Genetic programming. In *Search Methodologies*, chapter 5, pages 127–164. 2005.

[26] R. Lippmann. Pattern classification using neural networks. *IEEE Communications Magazine*, 27:47–50,59–64, Nov 1989.

[27] H. Lipson. Principles of modularity, regularity, and hierarchy for scalable systems. *Journal of Biological Physics and Chemistry*, 7(4):125–128, 2007.

[28] R. Miikkulainen. Creating intelligent agents in games. *The Bridge*, pages 5–13, 2006.

[29] J. F. Miller and P. Thomson. Cartesian genetic programming. *Genetic Programming*, 1802, 2000.

[30] T. Mitchell. Does machine learning really work? *AI magazine*, 18(3), 1997.

[31] T. Mitchell. *Machine learning*. McGraw-Hill, 1997.

[32] D. Moriarty and R. Mikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine learning*, 22(1), 1996.

[33] K. Ohkura, T. Yasuda, and Y. Matsumura. Coordinating the adaptive behavior for swarm robotic systems by using topology and weight evolving artificial neural networks. In *2010 IEEE Congress on Evolutionary Computation (CEC)*, 2010.

[34] M. Parker and B. Bryant. Lamarckian neuroevolution for visual control in the quake ii environment. In *2009 IEEE Congress on Evolutionary Computation*, pages 2630–2637, 2009.

[35] M. Pelikan and D. Goldberg. A survey of optimization by building and using probabilistic models. *Computational optimization and applications*, 21, 2002.

[36] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[37] M. Potter and K. Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8(1), Mar 2000.

[38] S. Rudlof and M. Köppen. Stochastic hill climbing with learning by vectors of normal distributions. *In 1st Online Workshop on Soft Computing*, 1996.

[39] D. Rumelhart, G. Hinton, and R. Williams. Learning representations by back-propagating errors. *Nature*, 323, Oct 1986.

[40] R. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2), 1997.

[41] B. Sareni. Fitness sharing and niching methods revisited. *IEEE Transactions on Evolutionary Computation*, 2(3):97–106, 1998.

[42] J. Schaffer, D. Whitley, and L. Eshelman. Combinations of genetic algorithms and neural networks: A survey of the state of the art. In *Proceedings of COGANN '92: Workshop on Combinations of Genetic Algorithms and Neural Networks*, 1992.

[43] Y. Shan, R. McKay, D. Essam, and H. Abbass. A survey of probabilistic model building genetic programming. In *Scalable Optimization via Probabilistic Modeling*, volume 33 of *Studies in Computational Intelligence*, pages 121–160. Springer Berlin / Heidelberg, 2006.

[44] K. Stanley. Real-time neuroevolution in the nero video game. *IEEE Transactions on Evolutionary Computation*, 9(6):653–668, Dec 2006.

[45] K. Stanley and R. Miikkulainen. Efficient evolution of neural network topologies. In *Proceedings of the 2002 Congress on Evolutionary Computation, 2002. CEC '02.*, volume 2, pages 1757–1762, 2002.

[46] K. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2), 2002.

[47] P. Verbancsics and K. Stanley. Constraining connectivity to encourage modularity in hyperneat. In *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, Jul 2011.

[48] E. Vonk. Using genetic algorithms with grammar encoding to generate neural networks. *Proceedings., IEEE International Conference on Neural Networks 1995*, 4, 1995.

[49] J. Walker. Evolution and acquisition of modules in cartesian genetic programming. In *Proceedings of the Genetic Programming 7th European Conference, EuroGP 2004*, 2004.

[50] P. Whigham. Grammatically-based genetic programming. In *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, Jul 1995.

[51] Wolpert and MacReady. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1), 1997.

[52] B. Zhang and H. Mühlenbein. Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation*, 3(1):17–38, Mar 1995.

[53] Q. Zhang, J. Sun, and E. Tsang. An evolutionary algorithm with guided mutation for the maximum clique problem. *IEEE Transactions on Evolutionary Computation*, 9(2):192–199, 2005.