Alan Turing was the first person to suggest that natural evolution may provide inspiration for approaches to artificial intelligence, in the famous “Intelligent Machinery” report he produced in 1948 (published 1968). The first implementations of such evolutionary computing emerged during the following decades, including pioneering work in the UK by Richard Forsyth with his BEAGLE system (1981). The first event dedicated to evolutionary computing in the UK was held as part of the 1994 AISB Convention in Leeds, organised by Terry Fogarty. The workshop continued to be held at the annual convention for a number of years thereafter, with the proceedings, entitled Evolutionary Computing, published by Springer each year.

The 50th anniversary of the Society for the Study of Artificial Intelligence and Simulation of Behaviour’s (AISB) annual convention is also the 20th anniversary of that first evolutionary computing workshop. And so a one-day event was arranged to mark the anniversary, the contributions to which are included here.

Larry Bull, UWE
Chair
Contents

Exploiting generalisation symmetries in accuracy-based learning classifier systems: An initial study.
L. Bull 1-6

An Evolvable Representation for High-Dimensional Evolutionary Robotics.
A. Churchill, V. Vasas, G. Goren & C. Fernando 7-14

Towards the Evolution of Vertical-Axis Wind Turbines Using Supershapes.
R. Preen & L. Bull 15-22

A review of Hyper-Heuristic frameworks.
P. Ryser-Welch & J.F. Miller 23-29

Immune Clonal Multi-objective Optimization based Simultaneous Clustering and Classification for Classification.
R. Shang, L. Jiao, Y. Su & Y. Li 30-37

On the Interaction between Self-adaptive Mutation and Memetic Learning.
J. Smith 38-45

A Model for Characterising the Collective Dynamic Behaviour of Evolutionary Algorithms.
M. Turkey & R. Poli 46-53

NeuroEvolution: The Importance of Transfer Function Evolution and Heterogeneous Networks.
A.F. Turner & J.F. Miller 54-61
Exploiting generalisation symmetries in accuracy-based learning classifier systems: An initial study

Larry Bull

Abstract. Modern learning classifier systems typically exploit a niched genetic algorithm to facilitate rule discovery. When used for reinforcement learning, such rules represent generalisations over the state-action-reward space. Whilst encouraging maximal generality, the niching can potentially hinder the formation of generalisations in the state space which are symmetrical, or very similar, over different actions. This paper introduces the use of rules which contain multiple actions, maintaining accuracy and reward metrics for each action. It is shown that problem symmetries can be exploited, improving performance, whilst not degrading performance when symmetries are reduced.

1 INTRODUCTION

Learning Classifier Systems (LCS) [Holland, 1976] are rule-based systems, where the rules are usually in the traditional production system form of “IF condition THEN assertion”. An evolutionary algorithm and/or other heuristics are used to search the space of possible rules, whilst another learning process is used to assign utility to existing rules, thereby guiding the search for better rules. LCS are typically used as a form of reinforcement learner, although variants also exist for supervised [Bernadó Mansilla & Garrell, 2003], unsupervised [Tammee et al., 2007] and function [Wilson, 2002] learning. Almost twenty years ago, Stewart Wilson introduced a form of LCS in which rule utility is calculated solely by the accuracy of the predicted consequences of rule assertions/actions – the “eXtended Classifier System” (XCS) [Wilson, 1995]. Importantly, XCS makes a clear connection between LCS and modern reinforcement learning (see [Sutton & Barto, 1998]): XCS uses a genetic algorithm (GA) [Holland, 1975] to discover regularities in the problem thereby enabling generalisations over the complete state-action-reward space. It has been found able to solve a number of well-known problems optimally (e.g., see [Butz, 2006]). Modern LCS, primarily XCS and its derivatives, have been applied to a number of real-world problems (e.g., see [Bull, 2004]), particularly data mining (e.g., see [Bull et al., 2008]), to great effect. Formal understanding of modern LCS has also increased in recent years (e.g., see [Bull & Kovacs, 2005]).

XCS uses a niched GA, that is, it runs the GA over rules which are concurrently active. Initially, following [Booker, 1985] (see also [Fogarty, 1994]), the GA was run in the match set [M], i.e., the subset of rules whose condition matches the current state. The primary motivation for restricting the GA in this way is to avoid the recombinating of rule conditions which generalise over very different areas of the problem space. Wilson [1998] later increased the niching to action sets [A], i.e., the subset of [M] whose action matches the chosen output of the system. Wilson correctly highlighted that for tasks with asymmetrical generalisations per action, the GA would still have the potential to unhelpfully recombine rules working over different sub-regions of the input space unless it is moved to [A]. Using two simple benchmark tasks, he didn’t show significant changes in performance but did show a decrease in the number of unique rules maintained when some asymmetry existed from the use in [A]. Modern XCS uses the [A] form of GA, which has been studied formally in various ways (e.g., see [Bull, 2002; 2005][Butz et al., 2004][Butz et al., 2007]). It can be noted that the first LCS maintained separate GA populations per action [Holland & Reitman, 1978] (see [Wilson, 1985] for a similar scheme).

The degree of symmetry in the state-action-reward space across all problems is a continuum. As noted, running the GA in niches of concurrently active rules identifies those whose conditions overlap in the problem space. However, using the GA in [A] means that any common structure in the problem space discovered by a rule with one action must wait to be shared through the appropriate mutation of its action. Otherwise it must be rediscovered by the GA for rules with another action(s). As the degree of symmetry in the problem increases, so the potentially negative effect of using the GA in [A] on the search process increases.

This paper proposes a change in the standard rule structure to address the issue and demonstrates it using a slightly simplified version of XCS, termed YCS [Bull, 2005].

2 YCS: A SIMPLE ACCURACY-BASED LCS

YCS is without internal memory, the rule-base consists of a number (P) of condition-action rules in which the condition is a string of characters from the traditional ternary alphabet {0,1,#} and the action is represented by a binary string. Associated with each rule is a predicted reward value (r), a scalar which indicates the error (e) in the rule’s predicted reward and an estimate of the average size of the niches in which that rule participates (n). The initial random population has these parameters initialized, somewhat arbitrarily, to 10.

On receipt of an input message, the rule-base is scanned, and any rule whose condition matches the message at each position is tagged as a member of the current match set [M]. An action is then chosen from those proposed by the members of the match set and all rules proposing the selected action form an action set [A]. XCS’s explore/exploit action selection scheme will be used here. That is, on one cycle an action is chosen at random and on the following the action with the highest average fitness-weighted reward is chosen deterministically.
The simplest case of immediate reward $R$ is considered here. Reinforcement in YCS consists of updating the error, the niche size estimate and then the reward estimate of each member of the current [A] using the Widrow-Hoff delta rule with learning rate $\beta$:

$$
\varepsilon_j \leftarrow \varepsilon_j + \beta (|R - r_j| - \varepsilon_j) \quad (1)
$$

$$
r_j \leftarrow r_j + \beta (R - r_j) \quad (2)
$$

$$
\sigma_j \leftarrow \sigma_j + \beta (|A| - \sigma_j) \quad (3)
$$

The original YCS employs two discovery mechanisms, a panmictic (standard global) GA and a covering operator. On each time-step there is a probability $g$ of GA invocation. The GA uses roulette wheel selection to determine two parent rules based on the inverse of their error:

$$
f_j = \left( \frac{1}{(\varepsilon_j^g + 1)} \right) \quad (4)
$$

Here the exponent $v$ enables control of the fitness pressure within the system by facilitating tuneable fitness separation under fitness proportionate selection (see [Bull, 2005] for discussions). Offspring are produced via mutation (probability $\mu$) and crossover (single point with probability $\chi$), inheriting the parents’ parameter values or their average if crossover is invoked. Replacement of existing members of the rulebase is global and uses roulette wheel selection based on estimated niche size. If no rules match on a given time step, then a covering operator is used which creates a rule with the message as its condition (augmented with wildcards at the rate $p_3$) and a random action, which then replaces an existing member of the rulebase selected as under the GA. Parameter updating and the GA are not used on exploit trials.

$$
\text{Figure 1: Schematic of YCS as used here.}
$$

The niche GA mechanism used here is XCS’s time-based approach under which each rule maintains a time-stamp of the last system cycle upon which it was part of a GA (a development of [Booker, 1989]). The GA is applied within the current action set [A] when the average number of system cycles since the last GA in the set is over a threshold $\Delta_t$. If this condition is met, the GA time-stamp of each rule is set to the current system time, two parents are chosen according to their fitness using standard roulette-wheel selection, and their offspring are potentially crossed and mutated, before being inserted into the rule-base as described above.

YCS is therefore a simple accuracy-based LCS which captures the fundamental characteristics of XCS: “[E]ach classifier maintains a prediction of expected payoff, but the classifier’s fitness is not given by the prediction. Instead the fitness is a separate number based on an inverse function of the classifier’s average prediction error” [Wilson, 1995] and a “classifier’s deletion probability is set proportional to the [niche] size estimate, which tends to make all [niches] have about the same size, so that classifier resources are allocated more or less equally to all niches” [ibid]. However, YCS does not include a number of other mechanisms within XCS, such as niche-based fitness sharing, which are known to have beneficial effects in some domains (see [Buzt et al., 2004]).

The pressure within XCS and its derivatives to evolve maximally general rules over the problem space comes from the triggered niche GA. Selection for reproduction is based upon the accuracy of prediction, as described. Thus within a niche, accurate rules are more likely to be selected. However, more general rules participate in more niches as they match more inputs. Rules which are both general and accurate therefore typically reproduce the most: the more general and accurate, the more a rule is likely to be selected. Any rule which is less general but equally accurate will have fewer chances to reproduce. Any rule which is over general will have more chances to reproduce but a lower accuracy (see [Buzt et al., 2004] for detailed analysis).

Under the new rule representation scheme introduced here each rule consists of a single condition and each possible action. Associated with each action are the two parameters updated according to equations 1 and 2:

Traditional rule – condition: action: reward: error: niche


All other processing remains the same as described but with each action of each rule using its associated error and reward parameters, e.g., in the GA. In this way, any symmetry is directly exploitable by a single rule whilst still limiting the possibility for recombining rules covering different parts of the problem space since the GA is run in [A], as Wilson [1998] described. Any action which is not correctly associated with the generalisation over the problem space represented by the condition will have a low accuracy and can be ignored in any post processing of rules for knowledge discovery. The generalisation process of modern LCS is implicitly extended to evolve rules which are accurate over as many actions as possible since they will participate in more niches. Note that the niche size estimate can become
noisier than in standard YCS/XCS as it is an estimate of the size of [M]. Similarly, any effects from the potential maintenance of inaccurate generalisations in some niches due to their being accurate in other niches are not explored here. Initial results do not indicate any significant disruption however.

3 EXPERIMENTATION

3.1 Symmetry
Following [Wilson, 1995], the multiplexer task is used in this paper. These Boolean functions are defined for binary strings of length \( l = k + 2^k \) under which the first \( k \) bits index into the remaining \( 2^k \) bits, returning the value of the indexed bit. A correct classification results in a payoff of 1000, otherwise 0. For example, in the \( k=4 \) multiplexer the following traditional rules form one optimal [M] (error and niche size not shown):

11111111111111111111111111: 1: 1000
11111111111111111111111110: 0: 0

Figure 2 shows the performance of YCS using the new multi-action rule representation on the 20-bit multiplexer \((k=4)\) problem with \( P=1000, \ p_\text{#}=0.6, \ \mu=0.04, \ \nu=10, \ \gamma=0.5, \ \theta_{\text{GA}}=25\) and \( \beta=0.2 \). After [Wilson, 1995], performance, taken here to mean the fraction of correct responses, is shown from exploit trials only, using a 50-point running average, averaged over twenty runs. It can be seen that optimal performance is reached around 60,000 trials. Figure 2 also shows the average specificity of all rules, taken here to mean the fraction of non-# bits in a condition, for the LCS - the amount of generalization produced. The maximally general solution to the 20-bit multiplexer has specificity \( 5/20 = 0.25 \) and YCS can be seen to produce rule-bases with an average specificity very close to the optimum. The average error of rules can also be seen to decrease over time.

Figure 3 shows the performance of YCS using the traditional rule representation with the same parameters. As can be seen, optimal performance is not reliably reached in the allowed time. Figure 4 shows the performance of the same system with \( P=2000 \), with optimality reached around 60,000 trials (matching that of XCS with the same equivalent parameters, e.g., [Butz et al., 2004]). That is, with double the rule-base resource, the GA is able to reliably (re)discover the problem structure in all \([A]\) over the same time period using the traditional rule representation. Hence, in a problem with complete symmetry between \([A]\), the new rule representation presented here significantly improves the efficiency of the GA.

3.2 Less Symmetry
To reduce the symmetry in the multiplexer in a simple way, an extra bit can be added. Here an incorrect response becomes sensitive to the value of the extra input bit: if it is set, the reward is 500, otherwise it is 0. That is, using the new rule representation, it is no longer possible for just one rule to use the same generalisation over the input space to accurately predict the reward for each action in a given [M]. The following traditional rules represent one optimal [M]:

11111111111111111111111111#: 1: 1000
11111111111111111111111111: 0: 500
11111111111111111111111110: 0: 0

Figure 2: Performance of new rule representation.
Figure 3: Performance of traditional rule representation.
Figure 4: As Figure 3 but with larger population size.
Figure 5 shows how YCS is unable to solve the less symmetrical 20-bit multiplexer using the new rule representation with $P=1000$. Figures 6 and 7 show how the performance of YCS with and without the new representation (respectively) is optimal and roughly equal with $P=2000$. Note that the new representation still only requires two rules per [M], as opposed to three in the traditional scheme. However, although there is a slight increase in learning speed with the new scheme, it is not statistically significant ($T$-test, time taken to reach and maintain optimality over 50 subsequent exploit cycles, $p>0.05$). Figures 8 and 9 show there is significant benefit ($p\leq0.05$) from the new representation when $k=5$, i.e., the harder 37-bit multiplexer ($P=5000$).

### 3.3 Multiple Actions

Multiplexers are binary classification problems. To create a multi-class/multi-action variant in a simple way the case where the data bit is a ‘1’ is altered to require an action equal to the value of the address bits for a correct response. In this way there are $2^k$ possible actions/classes. Under the new format with $k=3$, one optimal [M] could be represented as the single rule:

$$111#####1: 7: 1000$$

$$6: 0$$

$$5: 0$$

$$4: 0$$

$$3: 0$$

$$2: 0$$

$$1: 0$$

$$0: 0$$

Figures 10 and 11 show the performance of YCS with and without the new representation (respectively) with $k=3$ and $P=2000$. As can be seen, both representations are capable of optimal performance with the parameters used but the new representation learns significantly faster ($p\leq0.05$).
3.4 Imbalance

The frequency of state visitation is rarely close to uniform in most reinforcement learning tasks. For example, in a spatial maze navigation task, those states at or near a goal will typically be visited more often than those states far from a goal. In data mining, real-world data does not typically contain equal examples of all cases of the underlying concept space - known as the class imbalance problem, and often tackled through under/over sampling. This bias of sampling the problem space can cause difficulties in the production of accurate generalisations since over general rules can come to dominate niches due to their frequency of use (updating and reproduction) in more frequently visited states. Orriols-Puig and Bernado Mansilla [2008] introduced a heuristic specifically for (limited to) binary classification tasks which dynamically alters the learning rate ($\beta$) and frequency of GA activity ($\theta_{GA}$) to address the issue in accuracy-based LCS. They show improved learning in both imbalanced multiplexers and well-known data sets.

The new rule representation would appear to have some potential to address the issue of imbalance generally when there is symmetry in the underlying problem space, i.e., both for reinforcement learning and data mining. Since all actions are maintained by all rules, information about all actions is maintained in the population. Whilst over general conditions will quickly emerge for the same reasons as for the traditional representation, later in the search, the use and updating of the correct actions for less frequently visited states will indicate their true value and the GA will (potentially) adjust generalisations appropriately. An imbalanced multiplexer (akin to [Orriols-Puig & Bernado Mansilla, 2008]) can be created by simply introducing a probabilistic bias in sampling action ‘1’ compared to ‘0’. Figures 12 and 13 show the performance of YCS with and without the new representation (respectively) with $k=4$, $P=2000$ and a bias of 80% (4:1). Exploit cycle testing remains unbiased, as before. As can be seen, the new representation is able to cope with the bias, whereas the equivalent traditional rule representation is not. The same was generally found to be true for various levels of bias, $k$, etc. (not shown).
4 CONCLUSIONS & FUTURE WORK

This paper has proposed the use of rules which contain multiple actions, maintaining accuracy and reward metrics for each action. This somewhat minor alteration appears to provide benefits over the traditional approach in a variety of scenarios. Future work should also consider the new, general rule structure proposed here with more complex representations such as real-valued intervals (e.g., see [Stone & Bull, 2013]) or genetic programming (e.g., see [Preen & Bull, 2013]), together with delayed reward tasks.

Kovacs and Tindale [2013] have recently highlighted issues regarding the niche GA, particularly with respect to overlapping problems. They compare the performance of an accuracy-based LCS with a global GA (see also [Bull, 2005]), a niche GA, and a global GA which uses the calculated selective probabilities of rules under a niche GA. The aim being to avoid the reduced actual selection of accurate, general rules due to overlap within a given niche. Using the 11-bit multiplexer (k=3) problem they show a possible slight increase in performance from their new scheme over the niche GA, with the global GA performing worst. Their new scheme shows an increase in the number of unique rules maintained compared to the niche GA and they postulate this increase in rule diversity may explain the suggested difference in performance. This seems likely given the multiplexer does not contain any overlap. Note that Wilson [1994] proposed using both a global and niche GA together “to offset any inbreeding tendency” within niches. Since they used a supervised form of XCS which only maintains the highest reward entries of the state-action-reward map (UCS) [Bernado Mansilla & Garrell, 2003], the exploitation of symmetry does not help to explain their findings. The effect of the new representation in overlapping problems remains to be explored. The related use of multiple conditions per action, i.e., including an OR relationship, may be a more appropriate approach for the traditional (human readable) conjunctive representations of ternary alphabet or interval rules where overlap may be most significant.

REFERENCES


A Cognitive Architecture made of a Bag of Networks

Alexander W. Churchill\textsuperscript{1} and Vera Vasas\textsuperscript{1} and Goren Gordon\textsuperscript{2} and Chrisantha Fernando\textsuperscript{1}

Abstract. Our aim was to produce a cognitive architecture for modelling some properties of sensorimotor learning in infants, namely the ability to accumulate adaptations and skills over multiple tasks in a manner which allows recombination and re-use of task specific competences. The control architecture we invented consisted of a population of compartments (units of neuroevolution) each containing networks capable of controlling a robot with many degrees of freedom. The nodes of the network undergo internal mutations, and the networks undergo stochastic structural modifications, constrained by a mutational and recombinational grammar. The nodes used consist of dynamical systems such as dynamical movement primitives, continuous time recurrent neural networks and high-level supervised and unsupervised learning algorithms. Edges in the network represent the passing of information from a sending node to a receiving node. The networks in a compartment working together encode a space of possible subsumption-like architectures that are used to successfully evolve a variety of behaviours for a Nao H25 humanoid robot.

1 Introduction

Learning a controller for a complex multiple degree of freedom device such as a child or a robot is a non-trivial problem and the principles by which development achieves goals with such high-dimensional systems has been the subject of considerable investigation from the early work of Bernstein \cite{Bernstein}, the dynamical systems approach of Thelen and Smith \cite{Thelen}, to the robotics experiments of Schaal’s group on Dynamic Movement Primitives \cite{Schaal}. The critical property of developmental systems that fascinated us was their ability to accumulate adaptations without catastrophic forgetting, a topic that has recently been highlighted by Miller’s group \cite{Miller} as being highly suggestive of a role for structural plasticity in the brain along constructivist principles \cite{Miller}. A holy grail in developmental robotics would be to achieve open-ended adaptation over long periods of time, with the same cognitive architecture being able to bootstrap prior knowledge and competences to achieve more and more complex tasks. Whilst incremental evolution has been demonstrated for one or two steps \cite{Churchill}, an open-ended accumulation of adaptation has not been achieved. An attempt has been made in genetic programming to evolve subsumption architectures \cite{Schemmel} but scaling up to high-dimensional robots has not been attempted.

We use a directed graph based control architecture capable of parallel processing, in which cyclic networks are allowed. The network is analogous to a neural network controller, however, the nodes can be much more complex than simple neurons. There is a long history of evolving parallel potentially cyclic directed graph representations, for example Teller and Veloso use parallel evolution for data mining (the PADO algorithm) to solve supervised learning problems \cite{Teller}. Ricardo Poli uses a parallel genetic programming system called PDGP (parallel distributed genetic programming) to evolve feed-forward parallel programs to solve the XOR problem amongst others \cite{Poli}. Cartesian genetic programming \cite{Koza} also evolves feed-forward programs. The dominant paradigm for structural evolution of networks is now NEAT \cite{Stanley}, a powerful system because it allows diversity maintenance, protects innovations, and controls the dimensionality of the genotype carefully. Our system differs from the above approaches in several respects. First, it is intended to be a cognitive architecture for the real-time control of robots, secondly the nodes are intended to be complex machine learning algorithms, not arithmetic primitives or simple mathematical functions, thirdly, our system is intended to be capable of controlling a humanoid robot. As in the existing systems our nodes have multiple functions and transmit information between each other. Our motivation for using the graph representations was to eventually achieve compositionality, systematicity, and productivity because it is possible for specific node types to be physical symbol systems \cite{Churchill} if the rules for operating on them are grammatical. Previous attempts have been made to achieve grammar based genetic programming \cite{Koza} in tree structures, but not in cyclic distributed parallel graph structures of the types described above, to our knowledge. The challenge is to discover an evolvable grammar that generates fit variants with high probability, a task not unlike that faced in language learning by the fluid construction grammar \cite{Gopnik}. The structure of the paper is as follows. The network representation is described followed by the stochastic grammatical variation operators, and the overall evolutionary algorithm that controls the robot in real-time by generating and selecting active units. Some preliminary adaptation results are presented, and the network structures responsible for the behaviour are described. We have not yet achieved the accumulation of adaptation, but we have shown that a variety of distinct tasks can be evolved with disjoint network representations that are suitable for recombination. Several critical problems are encountered and discussed, and an approach is sketched for making progress.

2 Methods

The basic building block of the model is a node, it gets vector inputs, performs a processing operation, and sends vector messages to other nodes with some delay. Nodes when activated, transmit this activity state to downstream nodes, and remain active for a fixed period. Nodes can have internal states. Nodes are connected to evolve subsumption architectures \cite{Schemmel} but scaling up to high-dimensional robots has not been attempted. We use a directed graph based control architecture capable of parallel processing, in which cyclic networks are allowed. The network is analogous to a neural network controller, however, the nodes can be much more complex than simple neurons. There is a long history of evolving parallel potentially cyclic directed graph representations, for example Teller and Veloso use parallel evolution for data mining (the PADO algorithm) to solve supervised learning problems \cite{Teller}. Ricardo Poli uses a parallel genetic programming system called PDGP (parallel distributed genetic programming) to evolve feed-forward parallel programs to solve the XOR problem amongst others \cite{Poli}. Cartesian genetic programming \cite{Koza} also evolves feed-forward programs. The dominant paradigm for structural evolution of networks is now NEAT \cite{Stanley}, a powerful system because it allows diversity maintenance, protects innovations, and controls the dimensionality of the genotype carefully. Our system differs from the above approaches in several respects. First, it is intended to be a cognitive architecture for the real-time control of robots, secondly the nodes are intended to be complex machine learning algorithms, not arithmetic primitives or simple mathematical functions, thirdly, our system is intended to be capable of controlling a humanoid robot. As in the existing systems our nodes have multiple functions and transmit information between each other. Our motivation for using the graph representations was to eventually achieve compositionality, systematicity, and productivity because it is possible for specific node types to be physical symbol systems \cite{Churchill} if the rules for operating on them are grammatical. Previous attempts have been made to achieve grammar based genetic programming \cite{Koza} in tree structures, but not in cyclic distributed parallel graph structures of the types described above, to our knowledge. The challenge is to discover an evolvable grammar that generates fit variants with high probability, a task not unlike that faced in language learning by the fluid construction grammar \cite{Gopnik}.

The structure of the paper is as follows. The network representation is described followed by the stochastic grammatical variation operators, and the overall evolutionary algorithm that controls the robot in real-time by generating and selecting active units. Some preliminary adaptation results are presented, and the network structures responsible for the behaviour are described. We have not yet achieved the accumulation of adaptation, but we have shown that a variety of distinct tasks can be evolved with disjoint network representations that are suitable for recombination. Several critical problems are encountered and discussed, and an approach is sketched for making progress.
are formally sensor nodes because they can be initiator nodes for a graph, i.e. are active unconditionally. **Motor nodes** contain a vector of motors that they control. Their input vector is used to set the motor angles of these motors at each time-step the node is active, and a re-afferent copy or corollary discharge of the command angles is always output [2]. If there is no input, a default motor angle for each motor is stored. **Processing nodes** receive inputs only from other nodes. They include Euclidean distance nodes that gets a vector from its input nodes and calculates the distance to an internally stored vector parameter; a linear transform node that does the dot product of the input vector and an internally stored weight matrix to produce a vector of outputs; an Izhikevich neuron based liquid state machine [2] node. A **reinforcement learning node** has a two part input vector, a data part and a reward part. The data part consists of inputs from a set of other nodes, typically transform nodes or sensory nodes e.g. signalling joint angles. The reward part must be a single scalar value obtained from a transform node, for example this could signal the prediction error or the proximity to a desired state. The output of a learning node is a vector that will typically encode some parameters for downstream motor nodes to use, or it might be a temporal difference error signal. Types of reinforcement learning node include a stochastic hill climbing node, an actor-critic node, a simulated annealing node and a genetic algorithm node. **Supervised learning nodes** undertake online supervised learning based on an input training vector, and an input target vector. They do function approximation (regression) or classification. Such an node is essential for efficiently learning models of the world, e.g. forward models, or inverse models [17][9][5]. The type of model depends purely on the identity of the transform node that sends it the training and target vectors during its training. **Unsupervised learning nodes** take an input vector and compress it, e.g. a k-means clustering node does online clustering and outputs the class of each novel data point input to it; a **Principle Component Analysis node** takes an input vector, has a parameter n which is the number of principle components to output, and the output vector is the intensity of those n first principle components. In the Nao setup, there are high-level face recognition nodes, i.e. a sensor node that receives input from a camera and outputs a label and x,y retinal coordinate of a face; and a speech recognition node, which is a sensor atom that receives input from a microphone and outputs either the text or label of the speech. Other high level nodes in the Nao include walk nodes, motor nodes that control the legs to walk to a particular location and take high level Cartesian instructions as input. To summarise, the repertoire of nodes is essentially unbounded and is only limited by their encoding into the framework. In this manner, any combination of such nodes can form, thus enabling an open-ended evolution of complex controllers. This advantage is also a disadvantage, the number of nodes is huge, and the search space of possible networks is absolutely enormous. This makes evolution in such a space extremely difficult.

Activation of the networks in a compartment always start at the sensory nodes, these activate in parallel the downstream processing nodes with the appropriate delays, which in turn are typically connected to motor nodes. There may be recurrent connections. The unit of neuroevolution is a “bag” of disjoint graphs to which fitness is assigned as a unit. The motivation for this is that it enables the robot to be subject to potentially multiple parallel independent behavioral policies that together control the whole robot in parallel. By allowing graphs to be disjoint, a bag of potentially useful network motifs can be preserved silently and become utilised by the system in later variation events during evolution. This unusual nature of of compartment representations will become more clear when the initialisation of networks and the variation operators acting on them is described below.

### 2.1 Evolvable Action Grammar

We now describe the variation operations or graph rewrite rules [14]. **De novo node constructor** operators make new networks at initialisation giving the robot a primary sensorimotor repertoire of reflexes. For example such a network may consist of a single sensory node with a random number of sensory inputs connected to a linear transform node with a delay. The linear transform unit may have a random initial weight matrix sending output with random delays to three motor nodes. Each motor node may control 1 to 4 motors, randomly chosen. A complex constructor is also used throughout evolution as a macro mutation device for inserting functional motifs that consist of random sensory-(linear transform)-motor triplets in a chain of activation. **Graph replicator** operators produce perfect copies of graphs. These are required for replication and evolution of compartments (bag of graphs). **Graph connectivity operators** only influence the connectivity between already existing nodes within a single graph i.e. the messages that nodes get from other nodes within the molecule, the delays with which the messages influence activation, and the time period for which a node is active after activation. For example, the connection deletor removes a message i.d from the input vector of a node. A connection adder adds a new message i.d of a node in the graph to the input vector of another node in the graph. For now, all nodes are allowed to connect to/from any other node type, but evolvability in future work will demand this to be constrained. A critical feature of the above operators is that deletion of an edge may create two disjoint graphs that remain in the same compartment. **Node duplication** operations make copies of nodes within a single graph. The **parallel node replicator** takes a node and copies it entirely along with its input and output links such that this node now exists in parallel with its parent node, with the same inputs and outputs of the parent node. The **serial node replicator** copies a single node but instead of copying the inputs and outputs of the node, it makes the input to the offspring node the output of the parent node, and makes the offspring node connect to those nodes that the parent node connected to, i.e. it inserts a copy of the parent node downstream of the parent node. A **node deletor** operator removes a node along with its input and output edges, thus potentially disconnecting a graph into two (or more) components. All these operators act on the message i.d list in the input vector of a node. Thus connectivity is encoded at the input end of nodes, not at the output ends. **Intra-node operators** modify the sensory inputs, the motors controlled, the transfer functions and the internal states of the nodes. Intra-node mutations depend on the specific node type. Intra-node operators do not mutate the type of the node. This greatly restricts the mutability of the genome, but increases its evolvability. **Multi-graph recombinators** take N networks and construct a network that is derived from properties of the N parent networks. In analogy with genetic programming [8], a standard operator is the branch crossover operator that is defined only on trees and not recurrent graphs. This chooses two nodes and crosses them over along with any downstream nodes to which they are connected. **Motif crossover operators** allow crossover only between functionally isomorphic digraph motifs, in order to prevent crossover from being destructive due to the competing conventions problem.

The evolutionary algorithm consists of a population of 20-100 units of evolution (compartments). A binary tournament genetic algorithm is used to select pairs of units which are evaluated sequen-

Figure 2: Showing the resting position of the NAO, which it was reset to at the start of each trial. Images are from different angles of the same scene.

3 Results

This section should be read while referring to the accompanying videos in the Supplementary Material. Simulations are conducted in Webots for Nao, and real world experiments are conducted with the Nao robot itself.

3.1 Maximising distance travelled by a humanoid robot

In the first experiment the fitness of a compartment is defined as the distance travelled by the Nao during a single episode. The compartments successfully evolved to increase the distance travelled by the robot in a fixed time period. In order to test the efficacy of the intratomic mutations, an initial graph de novo constructor was used. An example initial graph can be seen in Figure 1 and is formed of three nodes - a sensor that is used for activation only, i.e. with no output, and two motor nodes, which operate the same motors. This graph is called a SMM molecule. The activation and delay times were initialised to create an oscillating effect by alternating between motor angles set to opposite directions in each motor node.

In the first evolutionary run of this system, only the motor angles and the delay and activation times of the actor graph were mutable, not the graph topology. This meant that fitness could only be improved through increasing the speed and strength of oscillations during the fixed time period. Each trial lasted for 50 time steps of 0.1 seconds, and at the start the simulated Nao robot was set to a resting supine position, illustrated in Figure 2. Evolution was performed for 250 generations, each consisting of 10 binary tournaments and 20 evaluations. The fitness of the individuals at every 10 generations is shown in Figure 3(a). There is a clear, although not dramatic increase in fitness (distance travelled) over the course of evolution. Fitness improvements occur after 100 and 120 generations, where the best solutions move from a distance of around 0.3 metres to close to 0.4 and then close to 0.5 metres. In terms of behaviour, the initial setup conditions create a largely symmetrical rocking motion that moves the robot slowly towards its left. The best behaviour after 50 generations rocks the robot with much greater force, achieved by raising the legs higher, which causes it to move further to the left over the course of a time trial. At 100 generations, the legs move higher and are raised for more time, which causes the robot to rotate further, and thus increases the amount of distance travelled. At 150 generations, it employs the same behaviour, moving to its right, but the legs oscillate more quickly. This final behaviour appears to be a local optimum for the robot.

In a second evolutionary run, the complexity of node mutations was increased to make motor identities mutable, but again with graph topology held fixed. The same starting conditions, i.e. the node structure and parameters, were used as in the first run. The fitness of individuals over the course of evolution is shown in Figure 3(b). Here, a similar performance to the first run is achieved after only 70 generations, and by 100 evaluations the system exceeds the best results found on the first run. After 50 generations the behaviour looks similar to the first run. The robot rotates from side to side, biased towards movements on its left side. The molecular graphs in Figure 4 show that after 50 generations, 5 left sided motors and 3 right sided motors are employed. After 100 generations, the behaviour of the best com-

Supplementary material is available at http://www.robozoo.co.uk/research/papers/aisb2014
Activation and Delay time and motor angle mutations only

Activation and Delay time, motor angle and motor mutations

Figure 3: Showing the fitness of actor molecules over 250 generations of evolution on the maximise distance task, for (a) when only activation and delay time and motor angles mutations permitted and (b) where additionally the motors used could be mutated. Each generation consists of 10 binary tournaments.

In the second experiment, compartments were evolved for 3,000 binary tournaments (6,000 evaluations) with all the variation operators included. Fitness was obtained by maximising the z-accelerometer sensor in the torso of Nao over the course of 150 time steps. For example this would be maximised by the robot standing on its head. The behaviour of the best controller after 100 and 5000 evaluations is shown in video V1 in the Supplementary Material, both in simulation and on the real robot.

Figure 4: Showing the atomic structure of the best actor molecules after (a) 50, (b) 100, (c) 150, (d) 200 and (e) 250 generations on the second evolutionary run of the maximise distance task. Sensor atoms (blue) provide activation signals only. Edge numbers indicate delay times and “active” shows the number of time steps a motor atom (red) remains active for. Refer to Figure 1 for the motor key.

Figure 5: Showing the fitness of each evaluated member of the population at each pseudo-generation of 10 binary tournaments. The compartments are clearly increasing in fitness over the course of evolution. There is an especially large jump over the first 50 generations, and then a more gradual but positive ascent throughout the rest of evolution. The graph on the right of Figure 5 shows the earlier fitness history of the population. After 32 generations there is a sudden drop in fitness. This occurs because the Nao moves into a position on its side where it is balanced and unable to return to the rest position on its back. This means that compartments are now selected in effectively a different environment. After 6 generations, Nao moves out of this position and can reset normally again.

The compartment representing the fittest individual found at the end of evolution can be seen in Figure 6. The compartment is made up of 3 disconnected graphs that get activated and produce movement over the course of a trial, labelled as Groups A - C, with a small number of atoms that will lie dormant (never activated). Among the dormant nodes there is a sensor node connected to a linear transform node that will be activated but will have no effect on the behaviour.
of the robot. The fitnesses scored by the three separate active groups can be seen in Table 1. The behaviour of each entry in the table can be seen in Video V2. The compartment containing all three parts performs the best, and the fitness is considerably greater than any one group on its own, showing that the disjoint graphs do interact behaviourally to create a more adaptive behaviour than any single disjoint component of the compartment. The final robot stance obtained by this compartment can be seen in Figure 9. Group A, the largest disjoint graph, has the highest fitness when considered on its own. It causes both of Nao’s legs to move, as well as its right arm, which causes it to move backwards into an arch-like position, raising its torso and so increase its z-accelerometer reading. Group B also scores well by itself, mainly moving Nao’s legs, with a larger emphasis on the right one. This leads it to move its torso backwards. Group C does not produce much movement when starting from a resting position. The combination of Groups A and B moves both legs far back underneath the torso, arching Nao backwards at a sharp angle, as can be seen in Figure 7(b). With all three groups together, the robot moves its legs even further backwards, drawing the torso even closer to a vertical position.

The two highest scoring groups (A and B) both make use of the Z-accelerometer sensor (143 in Figure 6). To test the impact of this sensor, the molecule was modified to replace all occurrences of the Z-accelerometer sensor with the X-accelerometer, which had the effect of oscillating the robot’s torso from side to side, leading to a much reduced fitness score. The final stance produced by the modified compartment can be seen in Figure 7(c). This demonstrates that a true closed-loop solution has been found.

An interesting variant of the above task involves initialising the robot from the prone position rather than the supine position. Figure 5 shows that evolution to max z from a prone position is much faster and more effective evolution to max z from a supine position. Video V3 shows the best behaviour evolved after 1400 evaluations. The solution is quite different, exploits the initial position of the Nao and is elegantly simple.

### 3.3 Evolving bouncing behaviour in a humanoid robot on a Jolly Jumper

In the third set of experiments, a physical NAO robot is placed in a Jolly Jumper infant bouncer (see Figure 9), which consists of a harness attached to a stand by a spring. This is influenced by Thelen’s work [23], which describes how an infant placed in a bouncer discovers configurations of movement dynamics that provide acceptable solutions and then tunes these to produce efficient bouncing. The robot is placed in the harness and suspended with the tip of its feet firmly touching a cushion on the ground, in the first experiment, and just touching the floor in the second. The fitness function is to maximise

<table>
<thead>
<tr>
<th>Group</th>
<th>Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>375</td>
</tr>
<tr>
<td>B</td>
<td>459</td>
</tr>
<tr>
<td>C</td>
<td>272</td>
</tr>
<tr>
<td>A + B</td>
<td>772</td>
</tr>
<tr>
<td>A + C</td>
<td>634</td>
</tr>
<tr>
<td>B + C</td>
<td>590</td>
</tr>
<tr>
<td>A + B + C</td>
<td>807</td>
</tr>
<tr>
<td>A + B + C (Changed Z-accelerometer sensors to X-accelerometer sensors)</td>
<td>467</td>
</tr>
</tbody>
</table>

Table 1: Fitnesses of the individual and combinations of disjoint graphs from best compartment found on the maximise z-accelerometer task, labelled as shown in Figure 6.

The first derivative of the z-accelerometer (z) at each time step (t), \( f(x) = \sum_{t=1}^{T} |z_{t+1} - z_t| \). Each trial lasts for \( T = 100 \) time steps of 100ms each. At the end of the trial, the robot’s joints are relaxed and it rests for 3 seconds. It may not return to the same position at the start of each trial, and the spring will not be completely dampened from the previous appraisal, which introduces noisy fitness evaluations.

In the first experiment, a Microbial GA with a population of 10 is applied for 800 evaluations to a fixed graphical structure, evolving only the parameters. The compartment can be seen in Figure 10 and consists of four Dynamic Motor Primitive (DMP) [15] nodes, connected to the knee and ankle motors, and receiving afferent signals from the motors as input. Additionally, two separate graphs take the force sensitive resisters (FSR) from each foot as input, and output to the ankle and knee motors for their respective sides of the body. These graphs will take control if the FSR signal is above an evolving threshold. Figure 11 shows the fitness of the population over the course of evolution. The run proceeds in several stages, which are shown in Video V4. Initially a kicking motion develops, that enables fitness to increase quickly. After 120 evaluations, the feet get stuck on the cushion, and the robot is unable to use its previous motion. A new strategy emerges, moving the knee as far back as possible before kicking. After 200 evaluations, with the feet now unstuck, a new solution is found, moving the left knee far back and relying on a fast kicking motion in the right leg to bounce. After 350 evaluations, the
cushion is removed, and the robot is now suspended above the ground at the start of each trial. Fitness falls sharply, as previous strategies relying on contact with the ground are much less successful. Fitness quickly improves, and solutions adapt to this new condition, with the knees travelling much greater distances during each oscillation. After 500 evaluations fitness returns to the previous level and then is quickly surpassed, with the best solutions achieving a fitness of over 9. This shows that the presented architecture is able to quickly adapt in a dynamic environment.

A second experiment explores a similar task but with an evolved topology. A more complex evolutionary process is employed in this experiment, inspired by NEAT [19]. A population is seeded with 20 identical copies of a small graph consisting of a DMP node connected to the left knee and receiving an afferent signal as input. This is initialised to perform a basic oscillatory motion. As in NEAT, each atom is assigned a unique id. An individual, \( i \), is sequentially compared to each member, \( j \), of each species using the similarity measure, \( s(i, j) = \frac{1}{2} [s_s(i, j) + s_p(i, j) + s_m(i, j) + s_{shc}(i, j)] \). Here, \( s_s(i, j), s_m(i, j), s_p(i, j) \) are functions which return the number of shared sensors, motors and processing node functions respectively, and \( s_{shc}(i, j) \) returns the number of shared connections between nodes with the same unique id. If \( s(i, j) \) is below a threshold, \( s_{thresh} \) (0.5 in this experiment), \( i \) is assigned to the species of \( j \) and the search is halted. Otherwise, \( i \) is assigned to a new species. The fitness score of each individual is divided by the number individuals in its species. A mutation count, \( mut_c \), initialised at 0, is also assigned to each new individual, and incremented if a mutation event occurs. Structural mutations are only permitted if \( mut_c > 3 \). In this way, innovations are protected and behavioural diversity is encouraged in the population. Additional variation operators pertaining to the DMP atoms are also included in this experiment. Disjoint graphs containing a DMP have an explicit probability of duplicating, and a DMP in one graph can be replicated and replace an existing DMP in another.

Evolution was again run directly on a real NAO robot, over the course of 11 hours. Figure 12 shows the fitness of each individual in the population after each evaluation. There is a clear improvement in competence on this task from the individuals in the initial population to those at the end of the run. The initial behaviour that the population was seeded with provided an acceptable fitness of between 3 and 5. After around 500 evaluations, structural mutations begin to have a noticeable effect on fitness. There is a climb in fitness from evaluations 500 to 1000, where the best individuals improve from fitness of 5 to 6.5, and again from evaluations 1100 to 1500, where the fitness of the best increases to around 8.5. Figure 13 shows the compartment of the best individual from the final population. As with the compartment shown in Figure 6, it contains redundancy, with several graphs lacking sensors or motors. The main driving force is a single DMP, which is used to control the left hip, right hip, left shoulder and right knee. A second molecule also controls the right knee, which can reset actions sent from the DMP, and create additional oscillatory movements. This solution produces rapid kicking motions, enabling it to move quickly along the longitudinal axis, and thus attain fast changes in its z-accelerometer sensor.

![Figure 6](image6.png)

Figure 6: Showing the compartment that represents the best individual found on the maximise Z-accelerometer task. There are three disjoint connected graphs that get activated during a run, and these have been highlighted and labelled A, B and C. Blue show sensors with the numbers representing sensors shown in the key below, green represents linear transform nodes with a maximum of 5 outputs, yellow Euclidian distance nodes with their target vector denoted inside, orange shows PCA nodes with the number of outputs denoted inside, black shows a K-Means node with the number of clusters denoted inside and red shows motor nodes with motors denoted inside (refer to Figure 1 for the motor key). Sensor key: 59:HeadYaw, 68:LKneePitch, 74:RElbowRoll, 77:RHipPitch, 139:GyroscopeY, 143:AccelerometerZ, 204:DistanceZ.

![Figure 7](image7.png)

Figure 7: Showing the final position of the Nao after different molecular groups are used for the actor, labelled as shown in Figure 6.
Figure 8: Evolution to maximise the z-accelerometer from the prone position. (a) There is a punctuated increase in fitness. (b) The final pose obtained is of high fitness.

4 Conclusion

An outline and some preliminary investigations of a modular and evolvable control architecture for high-dimensional robotic systems has been described. It remains to see whether the architecture is capable of the accumulation of adaptation and transfer learning across multiple tasks by the use of an archive of control motifs. The problem with the architecture is simultaneously its strength, i.e. it is too rich. A detailed investigation of the evolvability of the architecture will be needed in future if it is to be made efficient. One aspect missing from the architecture currently is gating and channeling at the inputs and outputs. Evolvable control flow operations are also needed. This adds an even greater complexity to the evolvability problem. An alternative approach is to begin with existing architectures for modular robot control, e.g. [6] [21], encode them within the above framework, and allow them to mutate. What is clear is that methods are acutely needed to deal with the issue of poor evolvability arising from the large space of possible networks in the current system.

ACKNOWLEDGEMENTS

The work is funded by the FQEB Templeton grant "Bayes and Darwin", and the FP-7 FET OPEN Grant INSIGHT.

REFERENCES

[5] G. Gordon and E. Ahissar, ‘Hierarchical curiosity loops and active sens-
Figure 11: Showing the fitness solutions of every 10 evaluations, over 800 evaluations on the first Jolly Jumper experiment.

Figure 12: Showing the fitness of all solutions over 11 hours of evaluations on the second Jolly Jumper experiment.

Figure 13: Showing the best compartment found at the end of evolution on the second Jolly Jumper experiment.
Towards the Evolution of Vertical-Axis Wind Turbines using Supershapes

Richard J. Preen and Larry Bull

Abstract. In this paper, we explore the evolution of three-dimensional objects with a simple generative encoding, known as Gielis Superformula. Evolving three-dimensional objects has long been of interest in a wide array of disciplines, from engineering (e.g., robotics) to biology (e.g., studying morphological evolution). While many representations have been presented, ranging from direct encodings to complex graphs and grammars, the vast majority have possessed complex underlying encodings, which were necessary to produce varied morphologies. Here, it is shown possible to produce very closely matching designs of a complex three-dimensional objects through the evolution of supershapes produced by Gielis Superformula. Subsequently, we explore the evolution of vertical-axis wind turbine prototypes represented as supershapes wherein each individual is physically instantiated and evaluated under approximated wind tunnel conditions.

1 Introduction

In recent years, wind has made an increasing contribution to the world’s energy supply mix. However, there is still much to be done in all areas of the technology for it to reach its full potential. Currently, horizontal-axis wind turbines (HAWTs) are the most commonly used form. However, “modern wind farms comprised of HAWTs require significant land resources to separate each wind turbine from the adjacent turbine wakes. This aerodynamic constraint limits the amount of power that can be extracted from a given wind farm footprint. The resulting inefficiency of HAWT farms is currently compensated by using taller wind turbines to access greater wind resources at high altitudes, but this solution comes at the expense of higher engineering costs and greater visual, acoustic, radar and environmental impact” [11]. This has forced wind energy systems away from high energy demand population centres and towards remote locations with higher distribution costs. In contrast, vertical-axis wind turbines (VAWTs) do not need to be oriented to wind direction and can be positioned closely together, potentially resulting in much higher efficiency. VAWT can also be easier to manufacture, may scale more quickly and cheaply. Three-dimensional printers are now capable of it is now possible to fabricate a wide range of prototype designs quickly and cheaply. Three-dimensional printers are now capable of printing an ever growing array of different materials, including food (e.g., chocolate [23] and meat [38] for culinary design), sugar (e.g., to help create synthetic livers [42]), chemicals (e.g., for custom drug
design [57]), cells (e.g., for functional blood vessels [32] and artificial cartilage [62]), plastic (e.g., Southampton University laser sintered aircraft), thermoplastic (e.g., for electronic sensors [36]), titanium (e.g., for prosthetics such as the synthetic mandible developed by the University of Hasselt and transplanted into an 83-year old woman), and liquid metal (e.g., for stretchable electronics [34]). One potential benefit of the technology is the ability to perform fabrication directly in the target environment; for example, Cohen et al. [10] recently used a three-dimensional printer to perform a minimally invasive repair of the cartilage and bone of a calf femur in situ. Lipson and Pollack [37] were the first to exploit the emerging technology in conjunction with an EA using a simulation of the mechanics and control, ultimately printing mobile robots with embodied neural network controllers.

We have recently undertaken initial experimentation of surrogate-assisted embodied evolutionary algorithms to design VAWT with a vector of integers representing the width of a turbine blade segment [48, 47]. In this paper, we explore the evolution of a simple generative encoding to produce more flexible three-dimensional designs for manufacture by a three-dimensional printer. Initially, the target-based evolution of three-dimensional supershapes is investigated. Subsequently we explore the evolution of VAWT represented as supershapes wherein each individual is physically instantiated and evaluated under approximated wind tunnel conditions.

2 Related Work

The evolution of geometric models to design arbitrary three-dimensional morphologies has been widely explored. Early examples include Watabe and Okino’s lattice deformation approach [61] and McGuire’s sequences of polygonal operators [40]. Sims [55] evolved the morphology and behaviour of virtual creatures that competed in simulated three-dimensional worlds with a directed graph encoding. Bentley [5] investigated the creation of three-dimensional solid objects via the evolution of both fixed and variable length direct encodings. The objects evolved included tables, heatsinks, penta-prisms, boat hulls, aerodynamic cars, as well as hospital department layouts. Eggenberger [12] evolved three-dimensional multicellular organisms with differential gene expression. Jacob and Nazir [31] evolved polyhedral objects with a set of functions to manipulate the designs by adding stellating effects, shrinking, truncating, and indented polygonal shapes. More recently, Jacob and Hushlak [30] used an interactive evolutionary approach with L-systems [50] to create virtual sculptures and furniture designs.

EAs have also been applied to aircraft wing design (e.g., [46]) including aerodynamic transonic aerofoils (e.g., [20, 51]), and multidisciplinary blade design (e.g., [21]). Few evolved designs, however, have been manufactured into physical objects. Conventionally evolved designs tend to be purely descriptive, specifying what to build but not how it should be built. Thus, there is always an inherent risk of evolving interesting yet unbuildable objects. Moreover, high-fidelity simulations are required to ensure that little difference is observed once the virtual design is physically manifested. In highly complex design domains, such as dynamic objects, the difference between simulation and reality is too large to manufacture designs evolved under a simulator, and in others the simulations are extremely computationally expensive.

Funes and Pollack [15] performed one of the earliest attempts to physically instantiate evolved three-dimensional designs by placing physical LEGO bricks according to the schematics of the evolved individuals. A direct encoding of the physical locations of the bricks was used and the fitness was scored using a simulator which predicted the stability of the composed structures. Additionally, Hornby and Pollack [28] used L-systems to evolve furniture designs, which were then manufactured by a three-dimensional printer. They found the generative encoding of L-systems produces designs faster and with higher fitness than a non-generative system. Generative systems are known to produce more compact encodings of solutions and thereby greater scalability than direct approaches (e.g., see [54]).

Compositional pattern producing networks [56] have recently been used to evolve three-dimensional objects which were ultimately fabricated on a three-dimensional printer [2, 3, 9]. Both interactive and target-based approaches were explored.

Recently, Rieffel and Sayles [53] evolved circular two-dimensional shapes where each design was fabricated on a three-dimensional printer before assigning fitness values. Interactive evolution was undertaken wherein the fitness for each printed shape was scored subjectively. Each individual’s genotype consisted of twenty linear instructions which directed the printer to perform discrete movements and extrude the material. As a consequence of performing the fitness evaluation in the environment, that is, after manufacture, the system as a whole can exhibit epigenetic traits, where phenotypic characteristics arise from the mechanics of assembly. One such example was found when selecting shapes that most closely resembled the letter ‘A’. In certain individuals, the cross of the pattern was produced from the print head dragging a thread of material as it moved between different print regions and was not explicitly instructed to do so by the genotype.

Husbands et al. [29] used an interactive evolutionary approach to design three-dimensional objects with a superquadrics formula similar to the Gielis Superformula used here. The Genetic Algorithm (GA) [27] used a directed graph encoded as bitstrings that were translated into a valid geometry. They were the first to combine superquadric primitives and global deformations with a GA, incorporating translation, rotation, scaling, reflection, tapering and twisting. A significant advantage of superquadrics is the compactness of the representation since few parameters are needed for a given deformation that widely extends the range of shapes representable.

3 Gielis Superformula

Gielis [16, 17] found that the forms of a large variety of plants and other living organisms can be modelled by a single, simple, geometric equation, forming a generalisation of a hyper-ellipses, termed the Superformula. Modifying the set of real-valued parameters to the Superformula generates myriad and diverse natural polygons with corresponding degrees of freedom. The Superformula can be used to create three-dimensional objects, supershapes, using the spherical product of two superformulae; in fact, by multiplying additional superformulae it can be extended to \( N \)-dimensions. “In general, one could think of the basic Superformula as a transformation to fold or unfold a system of orthogonal coordinate axes like a fan. This creates a basic symmetry and metrics in which distances can further be deformed by local or global transformations. Such additional transformations increase the plasticity of basic Supershapes” [17]. Gielis’ Superformula can be further generalised to increase the degrees of freedom, adding twist and further rotations, permitting the creation of more complex three-dimensional forms, including shells, möbius strips, and umbilic tori. Gielis’ Superformula, which defines a supershape in 2 dimensions is given in the following equation, where \( r \) is the radius; \( \phi \) is the angle; \( a > 0, b > 0 \) control the size of the supershape and typically \( n = 1 \); and \( m \) (symmetry number), \( n_1, n_2 \) and \( n_3 \) 
(shape coefficients) are the real-valued parameters:

\[
    r = f(\phi) \cdot \frac{1}{\sqrt{\left[\frac{1}{2} \cos(\frac{\phi}{2})\right]^2 + \left[\frac{1}{2} \sin(\frac{\phi}{2})\right]^2}} \tag{1}
\]

Using the spherical product, the extension to three dimensions:

\[
    \begin{align*}
    x &= r_1(\theta) \times \cos(\theta) \times r_2(\phi) \times \cos(\phi) \tag{2} \\
    y &= r_1(\theta) \times \sin(\theta) \times r_2(\phi) \times \cos(\phi) \tag{3} \\
    z &= r_2(\phi) \times \sin(\phi) \tag{4}
    \end{align*}
\]

Where \(-\frac{\pi}{2} \leq \phi \leq \frac{\pi}{2}\) for latitude and \(-\pi \leq \theta \leq \pi\) for longitude.

Example shapes generated with the Superformula can be seen in Figure 1 where the cube, star, and heart can be generated from the same set of eight real-valued parameters; the torus requiring two additional parameters; the shell a total of twelve; and the möbius strip a total of fifteen.

A supershape visualisation tool and its source code, licensed under Creative Commons Attribution-Share Alike 3.0 and GNU GPL license, can be found at http://openprocessing.org/visuals/?visualID=2638.

4 Target-Based Evolution

Given a target shape it is often very useful to identify a representative formula. Optimisation methods, such as the Levenberg-Marquardt (LM) theory [49], have typically been used to identify the best fitting supershape parameters (e.g., [19]). However LM cannot retrieve all of the parameters required for supershape fitting. Bokhabrine et al. [7] used a GA to evolve all supershape parameters for surface reconstruction (i.e., a target-based approach) using an inside-outside function [14] for fitness computation. Voisin et al. [60] later extended this to utilise a pseudo-Euclidean distance for fitness determination, yielding improved performance. Additionally, Morales et al. [43] used a GA to evolve \(N\)-dimensional Superformula for clustering.

The cube, star, and heart shapes (as seen in Figure 1) are here converted into \(50 \times 50 \times 50\) binary voxel arrays and used as the desired targets, where the fitness of an individual is the fraction of voxels that correctly match. The genotype of each individual in the population consists of eight real-valued parameters in the range \([0,5]\) which affect the Superformula, giving rise to the supershape. The GA proceeds with a population, \(P\), of 200 individuals, a per allele mutation rate of 25%, and mutation step size of \pm \text{rand}(5)\), where \text{rand} selects a real-valued number in the range \([0.5]\); a crossover rate of 0%; the GA tournament size for both selection and replacement is set to 3.

Figures 2–4 show the fraction of total voxels matched to the target shapes during evolution of the supershapes; results presented are an average of 10 experiments. Similar to [9], a large number of voxels are quickly matched, however here the target object is not identifiable until approximately 99% are set correctly. As such, the small differences in fitness between the treatments represent substantial differences in whether the target object is recognisable. In all cases, greater than 99.5% fitness is achieved. From Figure 2 it can be seen that, on average, the GA takes approximately 1100 evaluations to reach >99% matching voxels of a target cube object and 3700 evaluations to achieve >99.9%. Figure 3 shows that on average approximately 3900 evaluations are required to reach >99% matching voxels of a target star object and 16100 evaluations to achieve >99.5%. Finally, Figure 4 shows that, on average, >99% matching voxels of a target heart object is reached after 6400 evaluations and >99.5% after 24000 evaluations.

At the end of the experiments, the fittest individual was subsequently fabricated by a three-dimensional printer and can be seen in Figure 5, including the supporting rafts required for manufacture. Figure 6 illustrates a sample of the evolved individuals from one cube experiment, Figure 7 similarly for the star experiment, and Figure 8 for the heart experiment.

5 Rotation Speed-Based Evolution

As previously mentioned, we have recently undertaken initial experimentation of surrogate-assisted embodied evolutionary algorithms to design VAWT with a vector of integers representing the width of a turbine blade segment [48, 47].

The fitness of each individual was scored as the maximum rotation speed achieved during the application of constant wind generated by an approximated wind tunnel after fabrication by a three-dimensional printer. The rotation speed is the significant measure of aerodynamic efficiency since the design space is constrained (including rotor radius and turbine height). However, in future work, the AC voltage
generated would be preferred as it would take into account any slight weight variations that may affect performance. The rotation speed was measured in number of revolutions per minute (rpm) using a PCE-DT62 digital photo laser tachometer by placing a $10 \times 2 \text{mm}$ strip of reflecting tape on the outer tip of one of the treatment’s blades. When measuring a single isolated VAWT the treatment was placed at $30 \text{mm}$ distance from the centre of a $3,500 \text{rpm}$ $300 \text{mm}$ propeller fan generating $4.4 \text{m/s}$ wind speed.

Initially, 20 random designs were generated, fabricated, and evaluated. Since many of the seed treatments were extremely aerodynamically inefficient (only 2 out of 20 yielded $>0 \text{rpm}$), a canonical GA was run for 2 further generations before comparing with a GA assisted by a neural network surrogate model (SGA). The fittest evolved treatments after each generation are reproduced here in Figure 9 for the GA and Figure 10 for the SGA.

Additional challenges are encountered when extracting large amounts of wind energy since multiple turbines must be arranged into a wind farm. As the turbines extract the energy from the wind, the energy content decreases and the amount of turbulence increases downstream from each. See [25] for photographs and explanation of the well-known wake effect at the Horns Rev offshore wind farm in the North Sea. Due to this, HAWTs must be spaced 3–5 turbine diameters apart in the cross-wind direction and 6–10 diameters apart in the downwind direction in order to maintain 90% of the performance of isolated HAWTs [11]. The study of these wake effects is therefore a very complex and important area of research (e.g., see [4]), as is turbine placement (e.g., see [44] for an evolutionary approach). This work has almost exclusively considered HAWT. However, Dabiri et al. [33] have recently highlighted how the spacing constraints of HAWT often do not apply for VAWT, and even that performance can be increased by the exploitation of inter-turbine flow effects. Indeed, it has been shown [11] that power densities an order of magnitude greater can be potentially achieved by arranging VAWTs in layouts utilising counter-rotation that enable them to extract energy from adjacent wakes and from above the wind farm.

Therefore, a surrogate-assisted cooperative coevolutionary approach (SCGA) to design wind farms was explored, utilising the aggregated rotation speed of an array of 2 closely positioned VAWT as fitness. Each VAWT was treated separately by evolution and approximation techniques so that heterogeneous designs could potentially emerge. When measuring the fitness of an individual, one turbine from each species population was positioned $33 \text{mm}$ adjacently and $30 \text{mm}$ from the propeller fan. That is, there was a $3 \text{mm}$ spacing between the blades at their closest point. The treatments in each species population were initially evaluated in collaboration with a single randomly selected treatment from the other species population. Thereafter, the GA was run as before, however alternating between species after each offspring was formed and evaluated with the elite member from the other species. The fittest evolved treatments after each generation are shown in Figure 9 for the coevolutionary GA (CGA) and Figure 12 for the SCGA.

The results showed that EAs are capable of identifying novel and increasingly efficient VAWT designs wherein a sample of prototypes
are fabricated by a three-dimensional printer and examined for utility in the real-world. The use of a neural network surrogate model was found to reduce the number of fabrications required by an EA to attain higher aerodynamic efficiency (rotation speed) of VAWT prototypes. The approach represents the first surrogate-assisted embodied evolutionary algorithm using three-dimensional printing, and completely avoids the use of three-dimensional computer simulations, with their associated processing costs and modelling assumptions. In this case, three-dimensional CFD analysis was avoided, but the approach is equally applicable to other real-world optimisation problems, for example, those requiring computational structural dynamics or computational electro-magnetics simulations. In particular, the wind turbine array experiment showed that it is possible to use surrogate-assisted coevolution to iteratively increase the performance of two closely positioned turbines, taking into account the inter-turbine flow effects, which is especially difficult to achieve under a high-fidelity simulation. The surrogate-assisted GA represents a scalable approach to the design of wind turbine arrays since the number of inputs to the surrogate-models remains constant regardless of the number of turbines undergoing evolution.

One of the drawbacks of the representation used is that it assumes an underlying VAWT structure. In contrast, supershapes open the space of possible designs and yet retain a compact encoding. As a first step towards the evolution of supershapes as VAWT, here a single supershape as described previously becomes a prototype VAWT. A workspace (maximum object size) of $50 \times 50 \times 70\text{mm}$ is used so that the instantiated prototype is small enough for timely production ($\sim 80\text{mins}$) and with low material cost, yet large enough to be sufficient for fitness evaluation. The workspace has a resolution of $100 \times 100 \times 100$ voxels. A central platform is constructed for each individual to enable the object to be placed on to the evaluation equipment. The platform consists of a square torus, 2 voxels in width and with a centre of $10 \times 10$ empty voxels consistent through the $z$-axis, thus creating a hollow tube; see example in Figure 13a.

When production is required, the three-dimensional binary voxel array is converted to stereolithography (STL) format. Once encoded in STL, it then undergoes post-processing with the application of 3 Laplacian smoothing steps using Meshlab2; see example in Figure 13b. Finally the object is converted to printer-readable G-code

\footnote{MeshLab is an open source, portable, and extensible system for the processing and editing of unstructured 3D triangular meshes. http://meshlab.sourceforge.net}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{cube_evolution.png}
\caption{Evolution of a three-dimensional cube.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{star_evolution.png}
\caption{Evolution of a three-dimensional star.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{heart_evolution.png}
\caption{Evolution of a three-dimensional heart.}
\end{figure}

and is subsequently fabricated by a Stratasys Dimension Elite printer using a polylactic acid (PLA) bioplastic. See example in Figure 13c.

The fitness computation for each individual is the maximum rotation speed achieved during the application of constant wind generated by an approximated wind tunnel after fabrication by a three-dimensional printer. The rotation speed is here measured in number of revolutions per minute (rpm) using a PCE-DT62 digital photo laser tachometer by placing a $10\times2\text{mm}$ strip of reflecting tape on the centre of the treatment. The experimental configuration can be seen in Figure 14, which shows the 3, 500rpm 300mm propeller fan and the treatment placed at $30\text{mm}$ distance and offset by $100\text{mm}$ from the centre; that is, with an asymmetric air flow of $4.4\text{m/s}$.

The initial population consists of the star individual from Figure 13 and 19 other individuals whose parameters are each those of the star $\pm \text{rand()} \times 5.0$, where rand() is a random number in the

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{treatment_evolution.png}
\caption{The fittest treatments with $z$-variability produced by the GA each generation.}
\end{figure}
Figure 10: The fittest treatments with $z$-variability produced by the SGA each generation.

range $[-1,1]$; that is, $P = 20$. All initial individuals are subsequently fabricated and evaluated. Thereafter, a generational GA forms the next generation using the evolutionary operators as described for the target-based experiment. The fittest evolved treatment after 4 generations is shown in Figure 15. The parameters to the Superformula specify the length of the blades in addition to the frequency and the population has evolved an individual that forms an ‘X’ shape where the blades extend beyond the length of the workspace. As the blades extend beyond the workspace they are no longer drawn/fabricated and so the hollowness of the shape can be observed. It appears that evolution has identified that longer blades are more efficient under the current experimental conditions and this is also observed with an increase in the average length of the blades throughout the population. Furthermore, the reduction in number of blades from the initial 6 to 4 indicates that fewer blades may be more efficient. In Figure 16 the fittest evolved treatment after 5 generations is shown. As can be seen, overall the shape is more rounded and two of the blades from the ‘X’ have merged closer together in a step towards a 3 bladed shape, resulting in a lighter weight design with an increase in rotation speed.

6 Conclusions

This paper has shown that it is possible to evolve a vector of reals that are used as Superformula parameters to generate three-dimensional objects. Target-based evolution was used to explore the ability of Superformula to create complex objects, particularly those that resemble natural designs. The experiments showed that with target-based evolution very closely matching objects can be identified. In addition, a methodology for the embodied evolution of supershapes as VAWT has been introduced. One significant advantage of the approach over alternative representations is the simplicity and compactness of the encoding, which may be amenable for use in a surrogate-assisted approach.

If the recent speed and material advances in rapid-prototyping continues, along with the current advancement of evolutionary design, it will soon be feasible to perform a wide-array of automated complex engineering optimisation in situ, whether on the micro-scale (e.g., drug design), or the macro-scale (e.g., wind turbine design). That is, instead of using mass manufactured designs, EAs will be used to identify bespoke solutions that are manufactured to compensate and exploit the specific characteristics of the environment in which they are deployed, e.g., local wind conditions, nearby obstacles, and local acoustic and visual requirements for wind turbines.

REFERENCES

[7] Y. Bokhabrine, Y. D. Fougereolles, S. Foufou, and Frédéric Truchetet, ‘Genetic algorithms for Gielis surface recovery from 3D data sets’, in
Figure 12: The fittest SCGA heterogeneous array treatments. Wind direction from the south.

(a) 4th Gen — top view

(b) 4th Gen — side view

(a) Seed VAWT phenotype; genome = [6, 5, 30, 10, 4, 10, 10, 10].

(b) Seed VAWT with 3 Laplacian smoothing steps applied.

(c) Seed VAWT smoothed and printed by a three-dimensional printer; 50 × 50 × 50 mm; 80 mins printing time.

Figure 13: Seed VAWT.


Figure 16: Fittest treatment after 5 generations.


Tao Xu, Kyle W. Binder, Mohammad Z. Albanna, Dennis Dice, Weixin Zhoa, James J. Yoo, and Anthony Atala, ‘Hybrid printing of mechanically and biologically improved constructs for cartilage tissue engineer- ing applications’, Biofabrication, 5(1), (March 2013).
A Review of Hyper-Heuristic Frameworks

Patricia Ryser-Welch 1 and Julian F. Miller 2

Abstract. Hyper-heuristic frameworks have emerged out of the shadows of meta-heuristic techniques. In this very active field, new frameworks are developed all the time. Shared common features that help to classify them in different types of hyper-heuristics. Similarly to an iceberg, this large subfield of artificial intelligence hide a substantial amount of bio-inspired solvers and many research communities. In this paper, the tip of the iceberg is reviewed; recent hyper-heuristic frameworks are surveyed and the overall context of the field is presented. We believe its content complements recent reviews and offers another perspective of this important and rapidly developing field to the research community. Some hyper-heuristic frameworks tend to be largely constrained and prevent the state-of-the-art algorithms being obtained. We suggest in addition to relaxing constraints together with analysis of the evolved algorithms may lead to human-competitive results.

1 Introduction

In recent years, hyper-heuristic frameworks have emerged out of the shadows of meta-heuristic techniques. Those share common features that help to classify them in different types of hyper-heuristics. An analysis of shared common features allows them to be classified into different types of hyper-heuristics. Similarly to an iceberg, this large subfield of artificial intelligence hides a substantial amount of bio-inspired solvers and many research communities.

Instead of exploring a search space of problem solutions, hyper-heuristics automatically produce an algorithm that solves a problem more efficiently. A global optima is not guaranteed to be found with heuristics; however it provides at least one solution whenever the algorithm stops. In the worst case, the algorithm iterates over a large number of candidates solutions before finding the best one. In the best case scenario, the best solution is found rapidly. The "No Free lunch theorem" (NFL) makes us aware that if a good performance is demonstrated by an algorithm on a certain class of problems it will have a trade-off; the algorithm performance will be degraded on others classes. Hyper-heuristics offers a general technique for optimising algorithms. Learning mechanisms can customize algorithms to the unique needs of a restricted class of problems; this should reliably find a more suitable solution faster for a well-defined problem class [46].

Our motivation is to review a variety of hyper-heuristic models and frameworks. Identify their main purpose and the problems they have solved successfully. The next section compares two computing models of hyper-heuristics, before discussing the advantages and disadvantages of this search methodology. The following sections review algorithm-portfolio-based solvers, cross-domain hyper-heuristic and evolutionary frameworks. To conclude we discuss opportunities for further development and the wider applicability of such techniques. In this paper, only the tip of the iceberg can be reviewed; space restrictions prevent us to cover the entire field. We believe its content complements other reviews and offers another perspective to this important and rapidly developing research area.

2 Hyper-heuristic models

2.1 Heuristic, metaheuristic and hyper-heuristic

Heuristic techniques are often referred to as "search algorithms". They solve problems by discovering a solution in the set of all possible solutions for a given problem, which is regarded as the "search space". Non-deterministic search methods such as "evolutionary algorithms", "local search methods", "Simulated annealing", and others search algorithms offer an alternative approach to exhaustive search to solve difficult computational problems in a reasonable amount of time. These methods guarantee finding a solution at any time, but it may not be optimum [11, 23, 34].

Within a metaheuristic context, the focus shifts from solving a specific problem to producing a heuristic that solves the problem. The purpose of such approaches is to find, generate, or select a method or algorithm to solve a problem; their search space is now the collection of all possible heuristics and the outcome can be formulae or algorithms together with a solution of the problem it solves. One example is Genetic Programming (GP) that solves "automatically problems without the users to know or specify the the form or structure of the solution in advance. At the most abstract level GP is a systematic, domain-independent method for getting computers to solve problems automatically starting from a high-level statement of what needs to be done" [32].

In the literature, the terms heuristic and metaheuristic are often used interchangeably and often effectively solve NP-hard problems. Nonetheless, these search strategies can be resource-intensive to implement and develop. Many of these algorithms rely on a stochastic population; runs can produce very different outcomes. Additionally, the algorithms and problem solutions can be unusual as well as challenging to understand; new domains of a heuristic search space may be explored that lead to problem solutions that cannot always be easily explained logically [34].

Hyper-heuristics aim to address some of these issues; this search methodology discovers some algorithms that are capable of solving a whole range of problems, with little or no direct human control. It has been described as "heuristics to choose heuristics" [6] or "hyper-heuristic is an automated methodology for selecting or generating heuristics to solve hard computational problems" [5]. These techniques search the space of algorithms for any given problem in two ways. The first method assesses whether some combinations of pre-existing heuristics can improve the performance of the algorithm.

---

1 York University, England, email: patricia.ryser-welch@york.ac.uk
2 York University, England, email: julian.miller@york.ac.uk
The second option generates some new heuristics with a metaheuristic search mechanism [34, 35].

### 2.2 A two-level model

A modular model separates the functionalities of a given problem from the functionalities of the algorithm optimization process. Described by Cowling, this easy-to-implement architecture has been widely adopted by the hyper-heuristic research community. In this paper, we refer to the top level as “the Hyper level” and the lower level as “the Base level” (see fig 1) [6, 28, 34, 37, 41].

The Base level encapsulates a set of predefined heuristics for the given problem, a fitness evaluation function and a specific search space (see table 1). Its input parameters include a chosen heuristic with its the location memory and a chosen problem instance. Its only output is a performance evaluation of the algorithm. [6, 41].

The Hyper level decides which Base-level heuristic to solve a chosen problem. This can be achieved with a learning mechanism that evaluates the quality of the algorithm solutions, so that they can become general enough to solve unseen instances of a given problem. Grefenstette suggested that hyper-heuristics can be viewed as a form of reinforcement learning. Both methods employ online and offline learning. Online learning learns directly from its experience from its operational environment; self-modifying operators and autoconstructive evolution are examples of this type of hyper-heuristic (see sections 6.3 and 6.4). Offline learning gathers information in the form of programs from a set of training problems; automated design, meta-generic programming, and full evolutionary algorithms are some examples [3, 14, 28].

The Hyper level encapsulates a workspace that acts as repository of its metaheuristic states and the states of the search and should offer more freedom to the learning mechanism. The only input is the performance indicator of the chosen heuristic and its output includes a chosen heuristic with its location memory and a chosen problem instance (see table 2 and fig. 1).

Quite naturally, the domain barrier interface between the Hyper and Base level. Once the Hyper level has selected randomly some heuristics, these are passed to the Base level. The lower level can then pass the performance indicators to the higher level.

**Table 1**: Encapsulation of the problem domain at the Base level as suggested by [41]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>Solution-state space</td>
</tr>
<tr>
<td>( e )</td>
<td>Fitness function for instance of a problem</td>
</tr>
</tbody>
</table>

**2.3 The Algorithm Selection Problem**

The Algorithm Selection Problem simply describes the iterative mechanism that is likely to take place during the learning process. “For a particular problem instance \( p \in P \) with feature vector \( f(p) \in F \), find the selection mapping \( S(f(p)) \) into the algorithm space \( A \), such that the selected algorithm \( a \in A \) maximises the performance measure \( \| y \| \) for \( y(a,p) \in Y \)” [38]. In fact, this architecture illustrates how the Problem space is embedded in the Feature space and subsequently affect the state the Algorithm space. The selection

**Table 2**: Operational environment of the Hyper level as suggested by [41]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( H )</td>
<td>The hyper-heuristic function</td>
</tr>
<tr>
<td>( Q )</td>
<td>Metaheuristic states.</td>
</tr>
<tr>
<td>( W )</td>
<td>Repository for the states of the search</td>
</tr>
<tr>
<td>( i, j, k )</td>
<td>A tuple made of variables i,j,k variables where</td>
</tr>
<tr>
<td>( i )</td>
<td>the chosen heuristic</td>
</tr>
<tr>
<td>( j )</td>
<td>the location memory of this heuristic</td>
</tr>
<tr>
<td>( k )</td>
<td>the chosen problem instance</td>
</tr>
</tbody>
</table>

**Figure 1**: The mathematical model suggested by [41] to represent a hyper-heuristic system.

**Figure 2**: An illustration of the Algorithm Selection Problem, as originally proposed by [33]
process of a new heuristic can employ a variety of search methodologies, including online and offline learning. So we believe this general model cannot only model hyper-heuristic frameworks, but also meta-heuristics and others search methodologies that are outside the scope of this paper [7, 33, 38].

Table 3: Description of the main components of the Algorithm Selection Problem. These components are similar to the variables of the two-level model and simply describes all the elements of a heuristic.

![Table 3](image)

3 Discussion of hyper-heuristic methodologies

Using these four important distinctive components of section 2, the concept offers many advantages:

1. Hyper-heuristics should influence positively the selection of heuristics. The optimized heuristics for a given problem should compute high quality solutions. The learning phase should refine the algorithms, so that the algorithm solutions meet the needs of the training set and subsequently problems of a certain class can be solved more efficiently. Both models complement each other and comply with the “No Free lunch” theorem. Their response mechanism should move towards optimum algorithm solutions in the workspace, as it guides the selection of heuristic. The Algorithm Selection Problem represents in a three-dimensional coordinate system the relationship between a problem instance, an algorithm solution and its performance. Comparatively, the two-level model offers a clear separation between the optimization of an algorithm and the optimization process of a specific problem. This provides a visualisation of the NFL [20, 34, 46].

2. The existence of the two models not only raises questions about the level of generality, but also introduces the concept of plug-and-play of heuristics. Both models at least separates the problem domain from the algorithm search space. Like Lego bricks the models offer elements a degree of freedom to be changed. With very little data being passed between each component, each element can be changed as long as they respect the interfaces in place. For example, the Hyper level search methodologies have no knowledge of the problem-domain concealed in the Base level. In turn, the Base level is not aware of the learning mechanism used to choose its heuristic, in the Hyper level. In comparison, every space of the Algorithm Selection Problem can also change each of its spaces, without affecting of the others [7, 28, 41].

3. Both models explore a greater design space. The stochastic process explores more candidate algorithms in the design space. We can imagine that hyper-heuristics can either produce algorithms that are close to the state-of-the-art methodologies or algorithms that have not yet been thought of by humans. They offer a viable and powerful tool that is able to respond to some performance indicators and probabilistically move the search forward to new areas in a reasonable amount of time. As suggested by [47], the development cost of writing heuristic could be potentially lowered. “In addition Moore’s law states that processor speed is increasing exponentially, while the cost of human labour increases in-line with inflation” [4, 12].

Nonetheless the following issues needs to be considered too.

1. Experienced-based methodologies provide algorithms that may not be guaranteed to be optimal. These algorithms may vary after each run and be challenging to understand intuitively. The chosen heuristic can produce solutions of a lower quality than expected. It may also not be trusted by its users; the algorithm search may have generated an unknown order of instructions. The chosen problem area must then be able to cope with the speculative and randomness of hyper-heuristics. It could be disastrous if the maximum strain of a steel cable is solved with an algorithm of poor quality. Lives could be lost, if the cable is used inappropriately, with a lift with a load that is too heavy [12, 34, 35].

2. The simplicity and modularity of the two models offers the opportunity to represent simple or very complex hyper-heuristics. This varying complexity can be implemented in either one element, several elements or all of them. Adding too much technical knowledge and the programmers’ expertise can result in reducing the reusability and the applicability of a framework. These systems require a lot of effort to understand them. Additionally, the embedded conceptual elements in the application programming interface could become challenging to use again; some logic may not be suitable in another context. In others areas of evolutionary computations (EC), researchers have shown that EC can produce designs that surpass the state-of-the-art. Overly complex frameworks may prevent this creative feature occurring [24, 43].

3. Similarly to the full evolution of an evolutionary algorithm, the training phase could be quite power-hungry with a long training time. Although the performance of computers is improving all the time, this important factor cannot be ignored. The search in the algorithm space could be affected; the domain knowledge may be gained with fewer generations than expected and affect the quality of the learning. Also the produced algorithm may find good quality solutions, but their execution time and number of generations may be too large. To overcome this issue, some hyper-heuristics extend the fitness measure at the Hyper level by including higher level variables such as the execution time [8, 28].

4 Algorithm-Portfolio-based frameworks

Hyper-heuristic frameworks known as Algorithm-Portfolio-based frameworks aim at predicting the running time of algorithms, so that the time they take to solve a problem can be reduced. This idea identified that consideration of the running time of algorithms had been neglected from the Algorithm Selection Problem. The “Algorithm Portfolio” technique provides a means to overcome this problem. This method applies the Algorithm Selection Problem to construct models of algorithms runtimes using statistical regression. Then for a given instance of a problem, each algorithms time is predicted and the fastest predicted algorithm is used to solve the problem until the allotted time is used up or a suitable solution is found [25].

4.1 The SATzilla framework

SATzilla applies the Algorithm portfolio to Boolean satisfiability (SAT) problems; SAT solvers have been built for nearly a decade.
An offline learning process develops first a portfolio of algorithms, before applying each of them against an instance of the problem to select the fastest algorithm and predicts its runtime. More recently, new benchmarks instances and a variety of new base solvers were added to the framework [29,49–51].

4.2 The Snappy framework

The “Simple Neighborhood-based Algorithm Portfolio in PYthon” (Snappy) is a more recent framework. Although this framework also adopts the Algorithm portfolio, its aim is to provide a tool that can improve its own performances through online learning. Instead of using the traditional offline training step, a neighbourhood search predicts the performance of the algorithms. Snappy outperformed state-of-the-art benchmarks problems previously solved by SATzilla [36].

5 Cross-domain hyper-heuristic frameworks

In this section we review some cross-domain frameworks that have been recently mentioned in the literature. All these frameworks are implemented with Java, to provide a library that helps the programmers to write hyper-heuristic algorithms more easily in the Hyper level. All these frameworks offer a range of tools abstracted from iterated local search methodologies, that can be used to quickly create some hyper-heuristics.

5.1 Hyflex and parHyFlex

The motivation of Hyflex was inspired by the two-level hyper-heuristic model (see figure 1). “The emphasis of our HyFlex framework lies in providing the algorithm components that are problem specific, thus liberating the algorithm designers needing to know the problem’s domain’s specific details” [2]. An interface between the Hyper and the Base level is provided, with the main purpose of comparing a variety of hyper-heuristics. In fact, the algorithm designers can only devise new Hyper level algorithms; the Base level contains a library of well-known combinatorial problem domains with their benchmarks. In this context, the low-level heuristic supplies a set of operators that either apply small or large changes in the problem solutions. These perturbations should expand the search to a larger neighborhood and then guarantees better solutions are found [1, 2].

The flexibility offered by object oriented programming gives a simple and convenient method to easily create some hyper-heuristics. The framework structure hides strictly within the domain barrier the problem domain, in order to implement a domain-independent form of hyper-heuristic. “Using the framework, one can implement a hyper-heuristic without any knowledge about the algorithm running on parallel systems” [44]. The “Problem-domain, Hyper-heuristic and Heuristic type” classes decompose the system in explicit templates; a diagram can be found in [1] and [30]. New hyper-heuristics are then derived from those components and only the code that specifically differs from the original problem domains or hyper-heuristics is then written. For example, [45] developed a specific subclass of the Problem Domain for the vehicle routing problem and from the Hyper-heuristic another three subclasses that implement three different adaptive iterated local search. This new class encoded a representation of this NP-hard problem, an evaluation function with some benchmark problems and the current state-of-the-art operations. On the other hand, [27] used Hyflex to implement a more complex Hyper level. The research used again the problem domain

library with an Adaptive Dynamic Heuristic Set strategy enhanced with a learning automaton.

This strict use of templates could limit the ability of Hyflex of solving large real-world problems; such problem-domain preferably require less domain information [35]. Also the algorithm designers are required to structure their code with the explicit definitions of the three components. Finally, the framework seems to only support local search meta-heuristic in the Hyper level, making it very challenging to use Genetic Programming.

5.2 Hyperion

Hyperion applies a general reusable hyper-heuristic solution, to offer the tools to rapidly create a prototype. Its main aim helps identifying the components that contribute to an algorithm’s good performance. A transition function uses the problem domain variables (see table 1) to transform a problem solution into another one: \( Transition : S \rightarrow S \). In this case the transition has been defined as Eqn. 1. These transitions result from a variety of search methodologies that are built in a library. Hyperion also provides the four learning mechanisms described by [31]; the most complex framework recursively aggregates the hyper-heuristic to implement a hierarchy of hyper-heuristics.

\[
Transition = \{(from, fromValue, Operator, to, toValue)\}
\]

from \( \in S \),
\( fromValue \in R \),
\( operator \in O \),
\( to \in S \),
\( toValue \in R \) 

(1)

Experiments using the Boolean satisfiability compared the performance of several neighbourhood techniques [42].

5.3 hMod

Inspired by the previous frameworks, hMod abstracts all the elements of flow charts in a new object-oriented architecture. This model encodes the core of the Hyper level in several modules, referred as algorithm containers. hMod directs the programmer to define the Hyper level heuristic using two separate XML files: one for the heuristic selection process and another one for the acceptance move. These XML files are then read and interpreted with the code [43].

1. Each flowchart has a start and an end. An initial step is encapsulated in an “algorithm” class and the “flow control” in a “step” class. This variable points to the next operation, except for the last operation, which points to nothing.
2. A generic processing step holds a set of instructions that describe a specific behaviour.
3. The “decision” is treated as special step with two flow controls; one if the condition is met and another one if the condition is not met. The decision is useful with iterations and conditional execution.
4. “Input/output” has its own set of data classes with the traditional get and set methods.

At the time of writing, this new framework was only at the proposal stage. No result of its performance was available to allow comment.
6 Evolutionary hyper-heuristics

This second branch of hyper-heuristics optimises algorithms with Genetic Programming (GP) at the hyper level. Unlike the previous frameworks, the top level remains mostly unchanged, and most of the effort is required to encode the problem domain at the Base level.

6.1 Automated design and meta-genetic programming

These methods specialise in the automatic design of components of Genetic Algorithms. Similarly to meta-genetic programming (see [13]), automated design considers components of an evolutionary algorithm with the purpose of improving their performance without losing excessively the generality of this well-known algorithm. Both techniques can be applied to the hyper-heuristic models aforementioned. At the Base level the performance of the operators is assessed by executing several times a Genetic Algorithm with the newly generated operators. While, meta-genetic programming encodes their algorithms in a tree, automated design uses register machines at the Hyper level. Only automated design strongly aims at producing selection or mutation operators with improved performance on a class of problems. The domain knowledge is acquired with a set of instances chosen from a given problem class, during a training phase. The operators’ performance is assessed during the validation phase and the real-world phase can apply fully-developed algorithms on problems of the same given class.

Automated design has successfully improved the performance of selection operators for a GA that solved problem classes of the one-max problem. It was also used to explore mutation operators for a GA that solves mathematical functions with arbitrary chosen bit-strings. In these experiments, the parameters of each problem were defined by a Gaussian distribution. Finally, automated design discovered and generated new statistical distribution for the mutation operators [21,47,48].

6.2 Full evolution of evolutionary algorithms

Evolving evolutionary algorithms (EEA) fully adapts an evolutionary algorithm to the given needs of a problem; it is a very specialised hyper-heuristic method. It lets an EA discover the rules and knowledge, so that it can find the best EA to optimise the solutions of a problem. Several subfields of Genetic Programming have achieved this purpose with some success; Linear Genetic Programming (LGP), Multiple Expression Programming (MEP) and Grammatical Evolution (GE) have produced unknown evolutionary algorithms [9,10,22,26].

To the best of our knowledge, only a few researchers have yet focused on optimising the sequence of an EA to the specific need of a problem. Two approaches demonstrated the feasibility automatically evolving evolutionary algorithms. The first method used linear genetic programming and multi-expression genetic programming, to optimise the EA solving unimodal mathematical functions. An evolutionary algorithm manipulates a sequence of genetic and reproductive operators with their parameters. A solution consists of a new evolutionary algorithm that is capable of outperforming GA when solving a specific class of unimodal test functions. The result was an initial time-consuming and power-hungry learning process, such systems seems to be better suited to test hypothesis rather than actively solving a problem [9,10].

A Grammatical Evolutionary framework also evolved EAs, to instead solve the Royal Road problems. Genetic operators are performed on binary string that encodes the EEA. A mapping process, inspired by the protein synthesis, then transform this simple code into an EEA. The binary string are transcribed into an integer string, which is in turn derived into a tree structure using a set of predefined grammatical rules. Then the EEA is executed to solve classes of Royal Road Problems. Results demonstrates GE can optimise EAs, despite the grammatical rules imposed during evolution seeming to hinder the production of innovative EAs [26].

6.3 Self-modifying operators

Instead of repetitively using GP to select or generate the lower heuristic, before assessing its performance, self-modifying operators integrate the variation operators within the algorithm itself. These added online-learning features in the Hyper level empowered with the capability of adapting operators at the higher-level operate at the same time it is optimising low-level heuristics.

Self-modification Cartesian Genetic Programming (SMCGP) encodes in a graph the chosen low-level heuristic, alongside self-modifying operators. During the evaluation phase, a copy of the CGP program is made. This graph is then executed, and if they are any alterations to be made, then they modify the new graph. The encoded graph is then executed the same manner as a CGP program. SMCGP has been used to solve a number or problems. It has found general solutions to arbitrary parity problems, binary addition and Fibonacci series. It has discovered algorithms for computing e and pi to arbitrary positional accuracy. [15–17].

6.4 Autoconstructive evolution

Another approach lets the Hyper level programs be subjected to changes in their structure, at the same time they are responsible themselves produce their own children. “In fact, autoconstructive evolution is a hyper-heuristic in two ways: reproductive mechanisms are evolved which are then used to vary problem solutions, and reproductive mechanisms vary the reproductive mechanisms.” [18] (see figure 3). Push is a well-known autoconstructive evolution system defined by [39]. A detailed tutorial demonstrates clearly how stacks differentiate in term of data type and store the values of variables. [40] A stack can either represent a simple data type like a boolean, or a more complex data structure like locations within a tree. The latter becomes very useful to encode instructions that adapt the tree structure of a tree dynamically. This type of hyper-heuristic method was successfully applied to solve the order and majority problems, with the following evolutionary elements:

1. An autoconstructive mutation operator was defined as \( f' = f(g) \) and an auto constructive crossover operator had the mathematical function \( f' = f(f, g) \)
2. Each problem domain has its own fitness function defined separately.
3. The Push interpreter maintains a parent and a children population. A tournament selects the parent from the parent population before the autoconstructive operators are applied. The children population only accepts candidate solutions that meet the size requirements, have less errors than their parents and finally are different from their parents.
4. After the Push interpreter is initialised the two selected parent programs \( f \) and \( g \) are copied in the children population. Each of the
autoconstructive genetic operators are executed and solve either the order problem or the majority problem. The program becomes a candidate child program and the acceptance criteria is applied.

The entanglement of the low-level heuristic and its problem domain within the higher-heuristic program can lead to the problem solutions and its algorithm being inseparable. By treating the produced algorithm as a variation operator, the program may be used independently to solve problems belonging to the same class [18, 19, 39].

7 Discussion and conclusion

We have discussed types of heuristics, and compared and discussed the differences between the main hyper-heuristic models and frameworks. The hyper-heuristic research community is very active and have produced impressive results. However, they are still many questions that remain to be considered to advance the field further.

Although hyper-heuristic frameworks are well developed, there has been little discussion of the generated algorithms themselves. Can something be learned from study and analysis of generated algorithms that perform well? It seems that early discussions of these aspects in meta-heuristics have not been continued. The chosen encoding scheme is not expressive enough to allow easy the analysis of computational patterns of operations against their performance.

The operations that have been used to generate new candidate algorithms have been dominated by local search operations. Such operations would be very unlikely to create algorithms which carry out a form of cross-over. In addition, the form that generated algorithm can take appears to be very limited. For instance, any forms of GP cannot be currently produced by these cross-domain hyper-heuristics approaches. Too much human domain knowledge from software engineering has been included in the structure of the frameworks. This seems to prevent the framework from being adapted easily from their current educational and prototyping focus.

Algorithm design for solving NP-hard problems is an area of intense research. Many sophisticated state-of-the-art algorithms exist, yet currently it is unknown whether hyper-heuristic frameworks are capable of expressing such algorithms. To the best of our knowledge, structures of algorithms are seldom analysed or compared with the state-of-the-art algorithms. For instance, in the field of logic synthesis and minimisation there exist number of very effective algorithms. Can such algorithms be improved on using hyper-heuristic methods?

So far, hyper-heuristic frameworks have been restricted to a constrained set of possible algorithms. In general, it may not have enough expressiveness to represent a greater variety of algorithms, closer to the state-of-the-art or even programming languages.

The range of computer languages used to encode cross-domain hyper-heuristic frameworks has been quite limited and needs to take advantage of a greater ranges of programming platforms.

REFERENCES

Immune Clonal Multi-objective Optimization based Simultaneous Clustering and Classification for Classification

Ronghua Shang, Licheng Jiao, Yuchen Su, and Yang Li

Abstract: Clustering learning and classification learning is an important branch of traditional pattern recognition. Cai et al. [1] firstly proposed a multi-objective simultaneous learning framework (MSCC) for both clustering and classification learning which gets a good effect of classification. However, we found that using MOPSO optimization MSCC framework gets only a few Pareto optimal solutions, and it is not conducive to maintain the diversity of the solutions. This paper attempts to choose the immune clonal multi-objective optimization algorithm to optimize the MSCC framework in order to achieve better effect of classification. The immune clone operation, immune gene operation and antibody population updating operation in the proposed algorithm make the MSCC framework to obtain more Pareto optimal solutions and get a better effect of classification. Two groups of experiments on 18 data sets show that the proposed algorithm obtains better diversity, uniformity and convergence of solutions and also gets better classification accuracy.

Keywords: Clustering learning, classification learning, MSCC, immune clonal, multi-objective optimization.

1. Introduction

Clustering learning can be broadly divided into partitioning methods, hierarchical methods, density-based methods, grid-base methods and model-based methods [2–5].

Classical algorithms based on partitioning methods such as C-means algorithm proposed by MacQueen et al [2]. The main idea of the C-means algorithm is that at first randomly identify C points as cluster centers, and calculate the distance of each data to the cluster centers, then based on this distance classify the data to the nearest cluster center, last update the cluster center and repeat until the algorithm convergences. This algorithm is efficient on large data sets, but it has some weak points as the number of cluster centers needs to be determined in advance. It uses hard division and fails to show the relationship between data and the cluster centers and it is only applicable to the convex data sets and sensitive to noise. In order to overcome the shortcoming of the noise-sensitive, Kaufman et al. put forward a method of finding groups in data [3]. The modification of algorithm is the definition of the class center of K-Means algorithm and it makes the algorithm not sensitive to noise and abnormal data, but it fails to deal with small-scale data.

To address this issue, CLARANS was proposed for large data sets [4], but this algorithm did a poor job in terms of calculation efficiency.

The main idea of the hierarchical methods is compositting the data set in accordance with the hierarchical. Classical algorithms such as the BIRCH: it was proposed by Zhang et al [5] and it uses the CF-tree structure to divide the data and it is suitable for large data sets and dynamic clustering. CURE is another classical algorithm which was proposed by Guha et al. [6], in which the data is firstly divided into different classes, and then put the classes which are next to each other together. Typical density-based clustering algorithm such as STING [9] and CLIQUE [10], CLIQUE is a combination algorithm based on grid and density methods.

Classification learning uses the information of data set to build the decision function and according to which to achieve the purpose of classification. Comparing to clustering learning, classification learning focuses more in the division of the data and ignores the relationship between the internal structure and the data set. There is some classic classification learning algorithms such as the Bayes algorithm, C4.5 algorithm [11] and Support Vector Machines algorithm [12]. Bayes algorithm is based on the theory of probability and statistic, and the algorithm model is formed by the classical mathematical theory. Having the advantage of simple, fast, not sensitive to the missing data, it is suitable for large data sets. But Bayes algorithms assume that a given class value does no influence on other class attributes. Obviously the reality situation cannot meet this assumption, which leads to losing of correct rate. In order to overcome this weak point, TAN algorithm (tree augmented Bayes network) and other derivative algorithm were proposed. C4.5 Algorithm based on the decision tree, has higher classification accuracy relative to Bayes algorithm [11]. It is more suitable for associated data set, but the classification efficiency is lower compared to Bayes algorithm.

The difference between clustering learning and classification learning is that clustering learning is based on the internal structure of data set and the classification learning is based on discriminant function which is constructed by data type information. Both algorithms have advantages and disadvantages. Therefore some researchers make a combination of the two algorithms. Such as RBFNN algorithm [13], FRC algorithm [14], VQ+LVQ algorithm [15], CCAS algorithm [16] and ECCAS algorithm [17], etc. However, these algorithms are simply the superposition of clustering learning and classification learning. They are not able to fully integrate
the advantages of cluster learning and classification learning.

2. Related Backgrounds

Single-objective optimization problem contains only one function to be optimized, while multi-objective optimization problem has two or more functions to be optimized at the same time [18]. For multi-objective optimization problem, normally, we cannot find the optimal solution to all functions. Because when a solution is the optimal solution of a function, it is not necessarily so that the very solution in other functions. So, solving multi-objective optimization problems can only get a relatively better solution set, which means that one solution in this set is just not worse than the other ones, and these solutions are called Pareto optimal solutions [19, 20].

With the rapid development of multi-objective optimization techniques, some excellent works were proposed on multi-objective learning. For binary classification problems, multi-objective genetic programming (MOGP) was proposed to obtain a group of non-dominated classifiers to achieve the maximum receiver operating characteristic convex hull (ROCCH) [21]. A multi-objective regularized negative correlation learning (MRNCL) algorithm was proposed, which incorporated an additional regularization term for the ensemble and used the evolutionary multi-objective algorithm to design ensembles. Better performance was achieved by MRNCL [22]. Multi-objective evolutionary algorithms were also used for the construction of neural ensembles which found an optimal trade-off between diversity and accuracy [23]. Multi-objective evolutionary optimization was also introduced as a formidable ensemble construction technique to treat diversity and accuracy as evolutionary pressures which was shown to be effective [24]. As for fully integrating the advantages of cluster learning and classification learning, Cai et al. proposed a Multi-objective Simultaneous Clustering Learning and Classification Learning Algorithm (MSCC), in 2010 [1]. MSCC is based on cluster objective function and classification objective function simultaneously, by which a classifier is made both good in revealing the internal structure of data set and making the decision function. The classifier has a better classification accuracy compared to single classifier based on clustering learning or classification learning.

However, we found that using MOPSO [25] to optimize MSCC framework has a shortage of only gets few Pareto optimal solutions, and it is not conducive to maintain the diversity of the solutions. Therefore, in this paper, immune clonal multi-objective optimization algorithm is chosen to optimize the MSCC framework for classification. Immune clone algorithm comes from the Artificial Immune System (AIS) [26-27]. It is established by analog physiological responses of the body’s immune, which has good convergence and good solution diversity maintaining. Representative algorithms of AIS algorithms such as, Jiao et al. through vaccination and immune selection mechanism, proposed an immune gene algorithm [26]; Clonal Selection Algorithm proposed by de Castro et al [24]; Dynamic Clonal Selection Algorithm proposed by Kim et al [28]; and so on. The proposed algorithm in this paper firstly divides the possible solutions into dominated ones and non-dominated ones. Then immune clone operation and immune gene operation are applied to the non-dominated solutions, in which the immune gene operation including clonal restructure operations and clonal mutation operation. Having the advantage of keeping the diversity of the population, the proposed algorithm is good at keeping the diversity of solutions in the algorithm, and it makes some improvement on the classification accuracy on real data sets.

3. Immune Clonal Multi-objective Optimization based MSCC for Classification

3.1 Objective functions

In MSCC, Cai et al. constructs two objective functions based on the cluster centers: the clustering objective function and classification objective function [1]. Clustering function reflects the relationship of cluster centers and the data in the cluster, and cluster objective function is optimized to help improving the compactness of cluster centers and the data in the cluster. While classification objective function based on Bayes theory builds a valid classification mechanism and the classification objective function is optimized to help improving the classification accuracy. Using multi-objective optimization algorithm can optimize cluster objective function and classification objective function in the same time to achieve both good clustering compactness and good classification accuracy [1].

The following introduce the MSCC objective function [1]:

\[
\min F(v) = (f_1(v), f_2(v))^T (1)
\]

\(V\) is the cluster center combination \(V = [v_1, v_2, ..., v_M]\), and \(N\) is the number of cluster centers. \(f_1(v)\) and \(f_2(v)\) are cluster objective function and classification objective function respectively which will be introduced in the following.

3.1.1 Cluster objective function

\(f_1(v)\) is the clustering objective function based on the cluster centers[1].

\[
f_1(v) = 2\sum_{j=1}^{N_f} \left( \sum_{i=1}^{P_f} \frac{(1 - T(x_j, v_j))^{(m-1)}}{\sum_{j=1}^{N_f} (1 - T(x_j, v_j))^{(m-1)}(1 - T(x_i, v_j))} \right) (2)
\]
\[ T(x_i, v_j) = \exp\left( -\frac{\lambda \| x_i - v_j \|^2}{\max_{x \in X} \| x - \bar{x} \|^2} \right) \]  

\( P \) is the size of data set, \( N \) is the number of cluster centers. \( T(x_i, v_j) \) is the function based on cluster center. \( \bar{x} \) is the mean of \( x \), \( \lambda \) values base on the test, and \( m \) usually takes 2.

3.1.2 Classification objective function

\[ f_2(v) \text{ is the classification objective function [1].} \]

\[ f_i(v) = \frac{1}{N} \sum_{i=1}^{P} \delta(f(x_i), u_j) \]

\( \delta \) function means while \( f(x_i) = u_0 \), \( \delta \) takes 0; else, \( \delta \) takes 1.

\[ f(x_i) = \arg \max_{l \in SL} \left( \sum_{j=1}^{N} p(c_j \mid x_i) p(u_j \mid c_j) \right) \]

\( f(x_i) \) decides which class \( x_i \) belongs to, \( w_i \) means the \( l \)-th class, \( p(u_j \mid c_j) \) is the ratio of the number of antibody belongs to \( w_i \) and \( c_j \) the same time and the number of antibody belongs to \( c_j \).

\[ p(c_j \mid x_i) = \frac{1}{2(1-T(x_i, v_j))} \left( \sum_{j=1}^{N} (2(1-T(x_i, v_j)))^{-1} \right) \]

3.2 Main steps of immune clone algorithm optimizing the MSCC

MOPSO was used as a muti-objective algorithm to optimize the cluster objective function and classification objective function in MSCC framework in Cai et al.’s work [1]. However, we found that the use of MOPSO muti-objective algorithm is not doing a good job in maintaining the non-dominated solutions. To overcome this shortage, we choose the immune clonal muti-objective algorithm. Immune clonal algorithm can get more non-nominated solutions, and the using of immune clone operation can effectively maintain the diversity of the population and thus contribute to the optimization of the population, so as to achieve better classification result.

Using the immune clonal muti-objective optimization algorithm [18, 20] to optimize the MSCC framework, firstly, randomly initialize the antibody population and calculate the cluster objective function and classification objective function. Then allocate clone grade to the antibody population and do clone operation according to the clone grade. Clone operation based on the fitness of antibody population keeps the diversity of antibody population, and then the immune gene operation makes the antibody population evolution. Finally, the antibody population update operation is used to speed up the convergence of the algorithm [18]. The detailed description of each step will introduce in the following.

3.2.1 Initialization

Generate an antibody population of the scale \( P \) randomly: \( V(t) = [v_1(t), v_2(t), ..., v_P(t)] \); initialize the number of iteration \( t = 0 \); calculate the objective function values of all antibodies \( v_i(t) (i = 1, 2, ..., N) \); initialize the number of antibody \( v_i(t) \) dominated by other antibody \( n = 0 \) and dominance grade \( r = 0 \).

3.2.2 Immune clone operation

In this paper, we first calculate antibody non-dominance grade, and then according to the non-dominance grade calculate the antibody clone grade, Lastly clone antibody population depending on the antibody clone grade. For any individual antibody \( v^*_n(t) \in V(t)(n = 1, ..., N) \), if and only if \( v^*_n(t) \) satisfies the following conditions [18]:

\[ \exists v^*_n(t) \neq v^*_n(t), v^*_n(t) \in V(t); \]

\[ \forall f_i(v^*_n(t)) \geq f_i(v^*_n(t)) \land f_j(v^*_n(t)) > f_j(v^*_n(t)) \]  

\[ \| f_i(v^*_n(t)) > f_i(v^*_n(t)) \land (f_j(v^*_n(t)) \geq f_j(v^*_n(t))) \]

\( v^*_n(t) \) is called a non-dominated antibody; else, \( v^*_n(t) \) is called a dominated antibody.

Single out non-dominated antibodies, these antibodies have the highest non-dominate grade, then give these antibodies the highest clone grade. Then, select from the remaining antibodies to find out non-dominated antibodies, give these non-dominate antibodies lower clone grade. Calculate all the antibody clone grade using this method, clone the antibodies with the highest clone grade \( M \) times, and clone the antibodies with the lower clone grade \( M-1 \) times, and so on, clone all the antibodies. Immune clone operation steps are shown in the table 1.

<table>
<thead>
<tr>
<th>Table 1 Immune clone operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Imune clone operation</strong></td>
</tr>
<tr>
<td><strong>P</strong>: size of antibody population;</td>
</tr>
<tr>
<td><strong>k</strong>: clone grade, it takes 0 at the beginning;</td>
</tr>
<tr>
<td><strong>Step1</strong>: For ( i = 1:1:P )</td>
</tr>
<tr>
<td>Initialize antibody ( V[i] ) in antibody population ( V ) the times be dominated by other antibodies ( n = 0 ) and antibody clone grade ( c_i = 0 );</td>
</tr>
<tr>
<td><strong>End</strong></td>
</tr>
<tr>
<td><strong>Step2</strong>: For ( i = 1:1:P )</td>
</tr>
<tr>
<td>For ( j = i+1:1:P )</td>
</tr>
<tr>
<td>if ( V[j] ) dominates ( V[i] ), ( n_j = n_i + 1; )</td>
</tr>
<tr>
<td>if ( V[j] ) dominates ( V[i] ), ( n_j = n_i + 1; )</td>
</tr>
<tr>
<td><strong>End</strong></td>
</tr>
<tr>
<td><strong>End</strong></td>
</tr>
<tr>
<td><strong>Step3</strong>: Single out ( n_i = 0 ) antibodies, for these antibodies clone grade ( c_i = k; )</td>
</tr>
<tr>
<td><strong>Step4</strong>: For the remaining antibodies, initialize the times be dominated to 0, and turn to <strong>Step2</strong>;</td>
</tr>
<tr>
<td><strong>Step5</strong>: Single out ( n_i = 0 ) antibodies in the antibody population, antibodies clone ( c_i = k+1; )</td>
</tr>
<tr>
<td><strong>Step6</strong>: Repeat <strong>Step4</strong> and <strong>Step5</strong>, until all the antibodies in the antibody population get clone grade;</td>
</tr>
<tr>
<td><strong>Step7</strong>: Do clone operation on antibodies depends on antibody</td>
</tr>
</tbody>
</table>
3.2.3 Immune gene operation

The immune gene operation in this paper includes clone restructure operation and clone mutation operation. The following highlights the non-uniform mutation operation used in this paper [29]. How the immune gene operation goes is shown in the table 2. \(v[i,k]\) is \(k\)-th component of the \(i\)-th antibody in the antibody population, and \(p\) takes 1.

<table>
<thead>
<tr>
<th>Table 2 Immune gene operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step1</strong>: for (i=1):</td>
</tr>
<tr>
<td><strong>Step2</strong>: if (\beta) is random number in the range of ((0, 1)) ;</td>
</tr>
<tr>
<td>(\gamma) is random number in the range of ((0, 1));</td>
</tr>
<tr>
<td>(\eta=\text{round}((M-1)^*\text{rand}(1)+1); )</td>
</tr>
<tr>
<td>(k=\eta;)</td>
</tr>
<tr>
<td><strong>Step3</strong>: if (\beta&gt;\rho_m)</td>
</tr>
<tr>
<td>(v[i,k]=v[i,k]+(\eta-\text{v[i,k]})<em>((1-\gamma</em>(1-(t/G)) ^2)); )</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>(v[i,k]=v[i,k]-\text{v[i,k]})<em>((1-\gamma</em>(1-(t/G)) ^2)); )</td>
</tr>
<tr>
<td><strong>Step4</strong>: (i=i+1;)</td>
</tr>
<tr>
<td>If the termination condition is met, stop; Otherwise, turn to <strong>Step2</strong>.</td>
</tr>
</tbody>
</table>

3.2.4 Antibody population updating operation

In order to maintain the diversity of solutions and accelerate the convergence of the algorithm, we add the antibody population updating operation [18]: from the antibody population select the non-dominated solutions and pick out the antibodies which are intensive on the Pareto front to build the new antibody population. The steps of the update operation are shown in table 3. \(Num\) is the maximum dominate grade. \(NM\) is the size excepted to update the antibody population.

<table>
<thead>
<tr>
<th>Table 3 Antibody population updating operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Antibody population updating operation</strong></td>
</tr>
<tr>
<td><strong>while size(vpop,1)&lt;Num</strong></td>
</tr>
<tr>
<td>Find the number of antibody which non-dominate grade is (i);</td>
</tr>
<tr>
<td>If size(vpop,1)=F (&lt;NM)</td>
</tr>
<tr>
<td>Put antibodies which non-dominate grade equal to (i) in the antibody population vpop;</td>
</tr>
<tr>
<td>And put these antibody function value in vp;</td>
</tr>
<tr>
<td>(i=i+1;)</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>(NM=\text{Num-size}(vpop,1);)</td>
</tr>
<tr>
<td>Put antibodies which non-dominate grade equal to (i) in the antibody population tpop;</td>
</tr>
<tr>
<td>And put these antibody function value in tp;</td>
</tr>
<tr>
<td>((\text{ns,C})=\text{size}(tp);)</td>
</tr>
<tr>
<td>Put the antibody non-dominate grade of the antibodies in the antibody population (\text{tpop}) in (\text{tp;};C+1);</td>
</tr>
</tbody>
</table>

3.2.5 The time complexity of the proposed algorithm

The time complexity of the proposed algorithm consists of four parts: the time complexity of the initialization, the time complexity of immune clone operation, the time complexity of immune gene operation and the time complexity of antibody population updating operation. Let \(P\) be the size of data set, \(r\) be the maximum generation, \(M\) be the antibody population size, \(N\) be the number of cluster centers, and \(a\) be the number of objective functions. The time complexity of initialization of the proposed algorithm is \(O(M)^{33}\). The time complexity of immune clone operation consists of two parts: the time complexity of non-dominated operation is \(O(aMNP)\) and the time complexity of clone operation based on the non-dominated grade is \(O(aM)\). Let \(M_c\) be the antibody population size after clone operation. The time complexity of immune gene operation is \(O(M_c)\). Let
$M_{\text{non}}$ is the size of non-dominated antibody population, then the time complexity of antibody population updating operation is $O(aM^2_{\text{non}} M)$. Therefore, the total time complexity of the proposed algorithm is:

$$O(M) + O(aMNP) \ast O(aM) + O(M) + O(aM^2_{\text{non}} M)$$

And the algorithm iterates $t$ times. So the total time complexity of the proposed algorithm is:

$$O(t \ast (M + aMNP aM + M_e + aM^2_{\text{non}} M)).$$

4. Experimental and analysis

18 data sets from UCI machine learning database are chosen to test the proposed algorithm, MOPSO optimizing MSCC framework is used as comparison [1] denoted by “MOPSO” and the proposed algorithm is denoted by “ICMO”. Table 5 shows the data sets selected and data set dimension and the number of classes of the data set.

Table 5 Test data sets

<table>
<thead>
<tr>
<th>Data set</th>
<th>Scale dimension # class</th>
</tr>
</thead>
<tbody>
<tr>
<td>lenses</td>
<td>24 # 4 # 3</td>
</tr>
<tr>
<td>Lung_cancer</td>
<td>32 # 56 # 3</td>
</tr>
<tr>
<td>Soybean_small</td>
<td>47 # 35 # 4</td>
</tr>
<tr>
<td>Iris</td>
<td>150 # 4 # 3</td>
</tr>
<tr>
<td>Wine</td>
<td>178 # 13 # 3</td>
</tr>
<tr>
<td>Sonar</td>
<td>208 # 60 # 2</td>
</tr>
<tr>
<td>Glass</td>
<td>214 # 9 # 6</td>
</tr>
<tr>
<td>Thyroid</td>
<td>215 # 5 # 3</td>
</tr>
<tr>
<td>Heart_disease</td>
<td>270 # 13 # 2</td>
</tr>
<tr>
<td>Ecoli</td>
<td>336 # 7 # 8</td>
</tr>
<tr>
<td>Bupa</td>
<td>345 # 6 # 2</td>
</tr>
<tr>
<td>Vote</td>
<td>435 # 16 # 2</td>
</tr>
<tr>
<td>WDBC</td>
<td>528 # 10 # 11</td>
</tr>
<tr>
<td>Balance_scale</td>
<td>569 # 30 # 2</td>
</tr>
<tr>
<td>Pima_Indian_diabetes</td>
<td>625 # 4 # 3</td>
</tr>
<tr>
<td>Spambase</td>
<td>4601 # 57 # 2</td>
</tr>
<tr>
<td>Waveform</td>
<td>5000 # 21 # 3</td>
</tr>
</tbody>
</table>

Table 6 lists the classification accuracy of the test data sets by the proposed algorithm and MOPSO as comparison. Experimental set as following: antibody population size is 100; maximum generation is 100; crossover probability is 0.9; mutation probability is 1. The value of $\lambda$ is selected from the set {0.001, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 15}. Cluster center k minimum take the class of data set, the biggest one takes 20. In order to select the most appropriate $(k, \lambda)$ combination, different combination of $(k, \lambda)$ are tested 20 times independently, and select out the $(k, \lambda)$ combination with the highest average accuracy. $(k, \lambda)$ combination are shown in table 6. Table 6 also gives the $t$ values of this unpaired test with a significance level of 0.05. It can be seen from Table 6, among the 18 data sets, compared with MOPSO, ICMO achieves higher classification accuracy on 14 data sets, including 5 data sets with the classification accuracy of 100%, achieves approximately the same classification accuracy on 3 data sets and achieves much lower classification accuracy on 1 data set. From a statistical point of view, ICMO is significantly better than MOPSO on 12 test data sets. ICMO has no significant performance difference with MOPSO on 4 test data sets and ICMO is significantly worse than MOPSO on 2 test data sets. From the above analysis it can be concluded that the proposed algorithm improves the performance of the classifier effectively.

Table 6 Classification accuracy obtained by MOPSO and ICMO on both of the training data and testing data (best results in bold) and the t-test values

<table>
<thead>
<tr>
<th>Data sets</th>
<th>k, $\lambda$</th>
<th>MOPSO</th>
<th>ICMO</th>
<th>t-test</th>
</tr>
</thead>
<tbody>
<tr>
<td>lenses</td>
<td>4.0, 0.01</td>
<td>86.7</td>
<td>3.5</td>
<td>100</td>
</tr>
<tr>
<td>Lang_cancer</td>
<td>6.0, 0.001</td>
<td>59.3</td>
<td>3.001</td>
<td>93.2</td>
</tr>
<tr>
<td>Soybean_small</td>
<td>4.1, 0.001</td>
<td>100</td>
<td>5.0</td>
<td>100</td>
</tr>
<tr>
<td>Iris</td>
<td>12.0, 0.001</td>
<td>79.1</td>
<td>17.1</td>
<td>100</td>
</tr>
<tr>
<td>Wine</td>
<td>6.0, 0.01</td>
<td>98.3</td>
<td>17.1</td>
<td>100</td>
</tr>
<tr>
<td>Sonar</td>
<td>18.0, 0.01</td>
<td>85.6</td>
<td>5.1</td>
<td>100</td>
</tr>
<tr>
<td>Glass</td>
<td>20.0, 0.001</td>
<td>68.9</td>
<td>6.001</td>
<td>100</td>
</tr>
<tr>
<td>Thyroid</td>
<td>10.0, 0.001</td>
<td>96.4</td>
<td>11.0</td>
<td>100</td>
</tr>
<tr>
<td>Heart_disease</td>
<td>34.0, 0.001</td>
<td>84.2</td>
<td>18.0</td>
<td>100</td>
</tr>
<tr>
<td>Ecoli</td>
<td>24.0, 0.01</td>
<td>85.0</td>
<td>10.0</td>
<td>100</td>
</tr>
<tr>
<td>Bupa</td>
<td>28.0, 0.01</td>
<td>68.2</td>
<td>2.0</td>
<td>100</td>
</tr>
<tr>
<td>Vote</td>
<td>12.0, 0.85</td>
<td>8.4</td>
<td>4.0</td>
<td>100</td>
</tr>
<tr>
<td>Vowel</td>
<td>15.0, 0.87</td>
<td>39.8</td>
<td>16.0</td>
<td>100</td>
</tr>
<tr>
<td>WDBC</td>
<td>2.0, 0.001</td>
<td>97.3</td>
<td>8.0</td>
<td>100</td>
</tr>
<tr>
<td>Balance_scale</td>
<td>10.0, 0.001</td>
<td>90.8</td>
<td>17.1</td>
<td>100</td>
</tr>
<tr>
<td>Pima_Indian_diabetes</td>
<td>25.0, 0.01</td>
<td>76.5</td>
<td>14.0</td>
<td>100</td>
</tr>
<tr>
<td>Spambase</td>
<td>18.0, 0.001</td>
<td>89.9</td>
<td>3.0</td>
<td>100</td>
</tr>
<tr>
<td>Waveform</td>
<td>100.0, 0.001</td>
<td>86.5</td>
<td>3.0</td>
<td>100</td>
</tr>
</tbody>
</table>

*ICMO is significantly better than MOPSO.
*ICMO has no significant performance difference from MOPSO.
*ICMO is significantly worse than MOPSO.

Table 7 lists the classification accuracy of the test data sets by the proposed algorithm and MOPSO and also C4.5, SVM, KNN as the outward comparison. The C4.5, SVM, KNN are tested on the Weka (Waikato Environment for Knowledge Analysis), which is a JAVA environment based on open source machine learning and data mining software.

Table 7 classification accuracy

<table>
<thead>
<tr>
<th>Data sets</th>
<th>C4.5</th>
<th>SVM</th>
<th>KNN</th>
<th>ICMO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lang_cancer</td>
<td>50%</td>
<td>75%</td>
<td>68.75%</td>
<td>93.2</td>
</tr>
<tr>
<td>Soybean_small</td>
<td>85.63%</td>
<td>90.03%</td>
<td>87.98%</td>
<td>100</td>
</tr>
<tr>
<td>Iris</td>
<td>94.67%</td>
<td>96.00%</td>
<td>96.00%</td>
<td>100</td>
</tr>
<tr>
<td>Wine</td>
<td>93.26%</td>
<td>97.75%</td>
<td>93.26%</td>
<td>100</td>
</tr>
<tr>
<td>Sonar</td>
<td>74.03%</td>
<td>77.88%</td>
<td>88.46%</td>
<td>100</td>
</tr>
<tr>
<td>Glass</td>
<td>60.75%</td>
<td>49.53%</td>
<td>62.62%</td>
<td>67.8</td>
</tr>
<tr>
<td>Thyroid</td>
<td>99.47%</td>
<td>93.55%</td>
<td>90.93%</td>
<td>96.7</td>
</tr>
<tr>
<td>Heart_disease</td>
<td>76.82%</td>
<td>81.46%</td>
<td>76.82%</td>
<td>88.2</td>
</tr>
<tr>
<td>Ecoli</td>
<td>81.54%</td>
<td>76.19%</td>
<td>81.55%</td>
<td>96.7</td>
</tr>
<tr>
<td>Bupa</td>
<td>61.05%</td>
<td>58.72%</td>
<td>58.72%</td>
<td>72.8</td>
</tr>
<tr>
<td>Vote</td>
<td>97.23%</td>
<td>96.77%</td>
<td>93.54%</td>
<td>99.1</td>
</tr>
<tr>
<td>Vowel</td>
<td>69.49%</td>
<td>60.40%</td>
<td>92.12%</td>
<td>100</td>
</tr>
<tr>
<td>WDBC</td>
<td>76.92%</td>
<td>87.18%</td>
<td>79.81%</td>
<td>92.9</td>
</tr>
<tr>
<td>Pima_Indian_diabetes</td>
<td>72.32%</td>
<td>73.37%</td>
<td>69.45%</td>
<td>98.1</td>
</tr>
<tr>
<td>Spambase</td>
<td>91.52%</td>
<td>90.21%</td>
<td>87.78%</td>
<td>98.1</td>
</tr>
<tr>
<td>Waveform</td>
<td>72.56%</td>
<td>85.36%</td>
<td>71.64%</td>
<td>78.1</td>
</tr>
</tbody>
</table>

Proceedings of the AISB50 Symposium on Evolutionary Computing 20
As can be seen from Table 7, among the 17 data sets, compared with C4.5, SVM, KNN, the proposed algorithm achieved higher classification accuracy on the test of 16 data sets, including 5 data sets with the classification accuracy of 100%; achieved lower classification accuracy on the test of 1 data set. From the above analysis it can be concluded that the proposed algorithm performance better of the classification.

In the following, a set of experiments are listed in order to prove that ICMO achieves more Pareto optimal solutions and better distribution of solutions. In these experiments, ICMO obtained abundant Pareto optimal solutions and better distribution of solutions which leads to a better classification accuracy. The Pareto optimal solutions of ICMO and MOPSO are shown in figure 1–figure 3. In these figures, Pareto optimal solutions obtained by MOPSO is shown as circles “o” and those obtained by ICMO is shown as asterisks “*”. The percentage listed next to the circle or the asterisk is the classification accuracy of these Pareto optimal solutions. The abscissa $f_1$ is the value of the cluster objective function, and the vertical axis $f_2$ is the value of the classification objective function.

Figure 1 shows the Pareto optimal solutions and classification accuracy obtained by ICMO and MOPSO on data sets Sonar, Vote, Ecoli, Lung_cancer, Soybean and Pima_indian_diabetes. It can be seen from figure 1 that, for the 4 data sets, the proposed algorithm obtains better classification accuracy and more Pareto optimal solutions, and the figures also show that the proposed algorithm performs better in the aspects of distribution and convergence of the Pareto optimal solutions.

Figure 2 shows the Pareto optimal solutions and classification accuracy obtained by the proposed algorithm and MOPSO of data sets Wine and Glass. It can be seen from figure 2 that, for the two data sets, the proposed algorithm and MOPSO achieved approximately the same classification accuracy. The proposed algorithm achieves better classification accuracy in the test of Wine and MOPSO achieves better classification accuracy in the test of Glass. But the proposed algorithm obtained more Pareto optimal solutions in the test of both data sets and figure 2 also shows that the proposed algorithm performs better in the aspects of distribution and convergence of solutions.
Figure 2 Pareto optimal solutions obtained by MOPSO (denoted by “o”) and ICMO (denoted by “*”) of data sets: (a) Glass, (b) Wine

Figure 3 shows the Pareto optimal solutions and classification accuracy obtained by the proposed algorithm and MOPSO of data sets WDBC and Thyroid. Figure 3 shows that MOPSO achieves better classification accuracy on the test of WDBC and performs better in the aspect convergence of the Pareto optimal solutions, but the proposed algorithm obtains more Pareto optimal solutions.

Figure 3 also shows that the proposed algorithm performs better in the aspect of distribution of the solutions. For the test data Thyroid, the proposed algorithm and MOPSO achieve approximately the same classification accuracy, the MOPSO performs better in the aspect of convergence but the proposed algorithm achieves more Pareto optimal solutions and performs better in distribution of the solutions.

These experiments show that, the proposed algorithm compared to MOPSO generally achieved more Pareto optimal solutions in the test of data sets, performs better in the aspect of distribution of solutions. To a certain extent, the proposed algorithm performs better in the aspects of uniformity and convergence of solutions. And the marked classification accuracy also shows the Pareto optimal solutions obtained by the proposed algorithm have better classification accuracy to a certain extent.

5. Conclusion

Based on the clustering and classification framework MSCC proposed by Cai et al. [21], this paper uses the immune clonal multi-objective algorithm to optimize this framework. The immune clone operation, immune gene operation and the antibody population updating operation in the proposed algorithm make the algorithm to obtain more Pareto optimal solutions and this leads to better classification accuracy. Two groups of experiments are included in this paper to test the classification accuracy of the proposed algorithm and the simulation results show that the proposed algorithm has better diversity, uniformity and convergence, and also better classification accuracy than the compared algorithm.

Acknowledgement

This work was partially supported by the National Natural Science Foundation of China, under Grants 61371201 and 61001202, the EU FP7 project (grant no. 247619) on "NICaiA: Nature Inspired Computation and its Applications", the Fund for Foreign Scholars in University Research and Teaching Programs (the 111 Project) under Grant B07048, the Fundamental Research Funds for the Central Universities, under Grant K5051302028.

Reference


On the Interaction between Self-adaptive Mutation and Memetic Learning

J.E. Smith

Abstract. The “end-game” of evolutionary optimisation is often largely governed by the efficiency and effectiveness of searching regions of space known to contain high quality solutions. In a traditional EA this role is done via mutation, which creates a tension with its other different role of maintaining diversity. One approach to improving the efficiency of this phase is self-adaptation of the mutation rates. This leaves the fitness landscape unchanged, but adapts the shape of the probability distribution function governing the generation of new solutions. A different approach is the incorporation of local search – so-called Memetic Algorithms. Depending on the paradigm, this approach either changes the fitness landscape (Baldwinian learning) or causes a mapping to a reduced subset of the previous fitness landscape (Lamarkian learning). This paper explores the interaction between the effects of mechanisms embodying these two approaches. Initial results suggest that that the reduction in landscape gradients brought about by the Baldwin effect can reduce the effectiveness of self-adaptation. In contrast Lamarkian learning appears to enhance the process of self-adaptation, with very different, but appropriate, behaviours seen on different problems.

1 Introduction

Evolutionary Algorithms (EAs) are a class of population-based global search heuristics that have proved highly successful in many optimisation domains [7]. Much of their success comes from the use of randomised “genetic operators” – mutation and crossover – that create non-uniform probability distribution function (pdf) over the search space for generating new candidate solutions to be sampled. Given a parent pool selected from the current population (a multiset of candidate solutions), the shape of this pdf is governed by the contents of the pool, the choice of recombination and mutation operators, and their associated parameters. A broader pdf allows exploration of the search space, and hence the ability to escape local optima. A narrower pdf allows exploitation of hard-won information by focussing sampling in the vicinity of promising solutions. The way in which the trade-off between these two factors is managed has a major impact on both the effectiveness and efficiency of evolutionary search.

One common approach is to couple the randomised nature of EAs with a more systematic local search method to create Memetic Algorithms. This may be done in a number of ways – see e.g. [15] for a description and taxonomy. This paper will examine the most straightforward and most common: after recombination and mutation, each offspring undergoes local search for a specified number of iterations. In a Baldwinian paradigm [3], akin to “life-term” learning, the offspring has its fitness replaced with that of the fittest neighbour found by the local search. The Lamarkian paradigm is more drastic – both the “genome”(representation) and fitness of the offspring are replaced. Studies of these two paradigms with the Evolutionary Computation literature date back to the mid-1990s (see e.g. [41] and other papers within that special issue), and over the last decade hundreds of papers have documented their successful application to improve the effectiveness and efficiency of evolutionary search - see e.g. [21] for a recent survey. Both process alter the search landscape “seen” by the EA, but do this in different ways – this is discussed in more depth in Section 2.3.

Another very common approach, with proven success, is to apply a method for parameter adaptation, typically with the effect that an initially more uniform pdf is “narrowed” to focus more on promising regions of the search space over time. In both the combinatorial and real-valued domains, the majority of research and applications have focussed on adapting the mutation parameters, since the effect of recombination lessens as the population converges. A good review may be found in [8]. Whether adaptation is driven implicitly (e.g. via self-adaptation) or explicitly via the application or an “external” algorithm, a key factor is the presence of some form of evidence of the utility of an operator, or parameter value in generating high quality solutions from the current population. From early research [5] through to more current algorithms [39, 9, 16, 10], many adaptive algorithms maintain explicit archives - to record the mean improvement caused by different settings and reward the more promising. In contrast, in the self-adaptive paradigm the evidence is more implicit - successful strategies are those that produce offspring that survive, and so propagate via association. The practice and theory of self-adaptation of mutation rates has been documented in the continuous domain since [26, 4], the binary domain since [1, 2, 36, 37, 29] and for permutations [27]. More recent surveys may be seen in [33, 19].

These approaches have been successfully combined in, for example, the COMA framework [30, 32, 28, 34, 35], but much of that work focussed on the issues adaptation at the memetic level to create what Meuth et. al. called “second and third generation MAs” [18]. So far no attention that the author is aware of has been paid to the potential issues even with simple “first-generation” MAs, when the action of local search potentially destroys the link between strategies and offspring survival that is considered essential for successful self-adaptation to occur.

This paper represents a start at building an understanding of this issue by examining the patterns of behaviour observed when applying a simple memetic algorithm with self-adaptation of mutation rates to some well-understood combinatorial problems, where the “building blocks” to be found and propagated are of different orders, so that some cannot be discovered by local search alone. Specifically it examines the following hypotheses:

1 Department of Computer Science and Creative Technologies, University of the West of England, Bristol, UK. email: james.smith@uwe.ac.uk
• H1: That one-step Baldwinian learning has a blurring effect on the fitness landscape which reduces the selection pressure on different mutation rates, slowing the process of self-adaptation.

• H2: That one-step Lamarckian learning behaves differently - the mapping to a reduced search space that occurs when offspring are replaced by fitter neighbours effectively increases the selection pressure towards lower mutation rates.

• H3: That on problems with single-bit building blocks, using multiple steps of local search compounds the effects seen above and increases the selective pressure towards lower mutation rates.

• H4: In contrast, on problems with higher order building blocks, the effect of multiple steps of local search is to act as a repair function, allowing the preservation of the higher mutation rates needed to discover the “optimal” building blocks.

The paper is set out as follows. Section 2 provides a (necessarily) brief introduction to the key concepts of self-adaptation and Lamarckian search and the Baldwin effect. Section 3 describes the algorithms, test problems, and methods used to generate and analyse results. The results are presented in Section 4 and discussed in Section 5 before Section 6 draws conclusions and suggests future work.

2 Background

2.1 Self-Adaptation of Mutation Rates

The practice of using adaptive mechanisms to alter operator choices and parameters has attracted much attention. However the space of operators and parameters is large, and the mapping to the resulting quality of solution found is complex, still not well-understood, and problem dependent. Therefore hand-designed mechanisms have had relatively less success, and there has been natural interest in the application of evolutionary algorithms to search this space. In particular the use of “Self-Adaptation”, where the operator’s parameters are encoded within the individuals and subjected to evolution was established in the continuous domain within Evolution Strategies [26], and Stephens et al. have shown in general that adding self-adaptive genes to encodings can create evolutionary advantages [37].

Bäck’s work in self-adapting the mutation rate to use for binary encodings within generational GAs [1, 2] proved the concept, and established the need for appropriate selection pressure. Smith and Fogarty examined the encoding and conditions necessary to translate this to a steady-state GA [36]. To achieve the necessary selection pressure they employed a cloning mechanism: from the single offspring resulting from crossover they derived a set of clones. The mutation rate of each clone was then modified with a certain probability, and parameters has attracted much attention. However the space of operators and parameters is large, and the mapping to the resulting quality of solution found is complex, still not well-understood, and problem dependent. Therefore hand-designed mechanisms have had relatively less success, and there has been natural interest in the application of evolutionary algorithms to search this space. In particular the use of “Self-Adaptation”, where the operator’s parameters are encoded within the individuals and subjected to evolution was established in the continuous domain within Evolution Strategies [26], and Stephens et al. have shown in general that adding self-adaptive genes to encodings can create evolutionary advantages [37].

Bäck’s work in self-adapting the mutation rate to use for binary encodings within generational GAs [1, 2] proved the concept, and established the need for appropriate selection pressure. Smith and Fogarty examined the encoding and conditions necessary to translate this to a steady-state GA [36]. To achieve the necessary selection pressure they employed a cloning mechanism: from the single offspring resulting from crossover they derived a set of clones. The mutation rate of each clone was then modified with a certain probability, before being applied. The fittest resulting solution was then returned to the population. Results showed that the number of clones that led to the shortest tour length was 5 and that this tallied with previous work in Evolution Strategies (where the ratio of μ to λ is typically in the range 5 – 7) and Bäck’s implementation of truncation selection. They found that adding a self-adaptive mutation rate improved the performance of the GA and removed a non-trivial parameter from the GA. They also examined a variety of different ways of encoding the mutation parameter.

In subsequent work, Stone and Smith showed that for combinatorial problems the use of a continuous variable to encode for the mutation rate, subject to log-normal adaptation was outperformed by a simpler scheme [38]. In their method the value of the gene encoding for the mutation rate had a discrete set of alleles i.e. the mutation rate came from a fixed set, and when subject to mutation was randomly reset with a small probability. This has been examined experimentally and theoretically in [29] and [31]. In particular it was shown that the way the encoded mutation rate is perturbed is important – allowing the operator to work “on-itself” (as per [2, 36]) will lead to premature convergence to sub-optimal attractors. This effect has subsequently been rediscovered elsewhere [24]. Similar results have been found in the continuous domain experimentally (e.g. [12]) and theoretically [25].

Extensive experimental studies using Sequential Parameter Optimization have revealed that in binary search spaces different variants of self-adaptation do offer performance advantages [23]. However the very mixed results clearly indicate the need for a deeper understanding of the processes involved.

2.2 Memetic Algorithms

The field of Memetic Computation encompasses a wide range of algorithms based on the concept of memes as methods for generating or improving individual solutions to one or more problem instances. Rather than just local search-evolutionary hybrids, [21] consider Memetic Computation as a more general paradigm which uses “the notion of meme(s) as units of information encoded in computational representations for the purposes of problem solving”. In their more general view memes might be represented as “decision trees, artificial neural networks, fuzzy system, graphs etc” , and are not necessarily coupled to any evolutionary components at all, requiring simply a method for credit assignment. This enticing view offers the promise of memes capturing useful structural and behavioural patterns which can be carried between instances of the same problem, as is being explored in e.g. [40].

This paper is restricted to the broad class of “Memetic Algorithms” (MAs). Introduced by Moscato [20], these combine population-based global search heuristics (such as EAs) with heuristics that attempt to improve a single solution. Meuth et al. [18] distinguish between:

• First Generation MAs - which they define as “Global search paired with local search”.

• Second Generation MAs - “Global search with multiple local optimizers. Memetic information (Choice of optimizer) passed to offspring (Lamarckian evolution)”.

• Third Generation MAs: - “Global search with multiple local optimizers. Memetic information (Choice of local optimizer) passed to offspring (Lamarckian Evolution). A mapping between evolutionary trajectory and choice of local optimizer is learned”.

Although powerful paradigms with increasing number of successful applications, adaptive second and third generation MAs bring with them a number of issues concerning the credit-assignment problem [22, 35]. For these reasons this initial study concentrates on a simple first generational memetic algorithm based on a greedy bit-flipping mechanism. For a binary string with length l the Hamming distance between two strings as $H(i,j) = \sum_{k=1}^{l} |j[k] - i[k]|$, so this algorithm uses a neighbourhood function $n_{H2}(i) = j : H(i,j) = 1$. This local search algorithm can be illustrated by the pseudocode given in Eq. 1.

2.3 Lamarckianism and the Baldwin Effect

The framework of the local search algorithm outlined above works on the assumption that the current incumbent solution is always replaced by the fitter neighbour when found. Within a memetic algo-
fit to the higher neighbour, and whole swathes of low-fitness points are effectively removed from the search space.

3 Experimental Methodology

3.1 Algorithm

The core Evolutionary Algorithm used a very standard Genetic Algorithm (GA) with a (100,500) model for population management and the following operators and parameters.

Table 1: EA operators and parameters

<table>
<thead>
<tr>
<th>Operator/Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mu )</td>
<td>100</td>
</tr>
<tr>
<td>Representation</td>
<td>Binary Strings</td>
</tr>
<tr>
<td>Solution Length</td>
<td>Problem-dependent</td>
</tr>
<tr>
<td>Crossover</td>
<td>One-Point</td>
</tr>
<tr>
<td>Crossover Probability</td>
<td>0.7</td>
</tr>
<tr>
<td>Mutation Operator</td>
<td>Bit-Flipping</td>
</tr>
<tr>
<td>Mutation Rate Encoding</td>
<td>Integer in range [1,10]</td>
</tr>
<tr>
<td>Mutation Strategy Rate ( P_{m} )</td>
<td>0.05 or 0</td>
</tr>
<tr>
<td>Parent Selection</td>
<td>Tournament Size 1</td>
</tr>
<tr>
<td>Survivor Selection</td>
<td>Truncation</td>
</tr>
<tr>
<td>Local Search Neighbourhood</td>
<td>Hamming Distance 1</td>
</tr>
<tr>
<td>Local search pivot rule</td>
<td>Greedy</td>
</tr>
<tr>
<td>Depth of local search ( d )</td>
<td>1, 2 or 5 iterations</td>
</tr>
</tbody>
</table>

The Self-adaptation process used the scheme outlined in [29, 31, 38]. Rather than attempting to adapt a continuous mutation rate parameter, each solution encodes a choice from a discrete set of values, \( 1.0/l \times \{0.001, 0.005, 0.01, 0.05, 0.1, 0.2, 1.0, 2, 25, 50, 100 \} \) where \( l \) is the length of the problem encoding. Prior to mutating the solution encoding, the gene encoding for the mutation rate is randomly reset with probability \( P_{m} \).

Although these operators and parameter values were taken as fairly standard from the literature, preliminary experimentation (not show for reasons of space) suggests that the effects observed below occur over a wide range of parameter values. One point crossover was chosen for its positional bias which matches that of the problem encoding used for the Royal Road and Trap functions.

3.2 Test Functions

The first test problem used is a 200-bit version of the unimodal One-Max function:

\[
f(i) = u(i) \times 100/l.
\]

where the unitation \( u(i) = \sum_{j=1}^{l} I(j) \).

The effect of Baldwinian learning with depth \( d \) on this landscape is to assign to each genome the fitness of a individual with \( d \) more bits set to 1 - in other words the shape of the landscape is left untouched except for those few solutions with a neighbourhood \( H(i,j) = d \) of the global optimum, where the landscape is flat. The effect of Lamarckian search is to move each point \( d \) steps up the slope of the hill - ie. effectively to remove those points with \( u(i) < d \) from the search space. In both cases the underlying structure of the problem is left unchanged, so except for the more rapid convergence to the global optimum, it is hypothesized that the self-adaptation of mutation rates will follow a similar pattern to the GA.

The second two problems used were 64-bit versions of the type R1 Royal Road fitness function [11] where the fitness is given by the
number of blocks “aligned” to the target string in a problem with \( L \) blocks, each of length \( K \). Assuming (without loss of generality) that the target string is all 1s for each block:

\[
    f_{RL}(i) = \sum_{j=1}^{L} \prod_{j \in \text{block}_k} i_j
\]

(2)

A well known property of these functions is that for \( K > 1 \) they possess “plateaus” of equal fitness, and as discussed in [42] these represent entropic barriers to evolutionary search. As has been extensively documented, search on these problems typically proceeds via a series of “epochs”, and during transitions the entropy of the population is reduced as the correct alignment is found for the next block, and fixated through the population.

To examine the effect of learning two different 64-bit versions with different sized plateaus were used - denoted hereafter as “Royal-4bit” \( (L = 16, K = 4) \) and “Royal-8bit” \( (L = K = 8) \).

To understand the effect of learning on these problems, let us consider the partition of the search space corresponding to a single block. Applying one step of local search means that now \( K \) of the possible \( 2^L \) solutions in that partition now contribute to the global fitness instead of just 1. The effect of multiple steps of learning will depend on whether any of the blocks have unitation of \( K - 1 \). The “Baldwin effect” on these landscapes is that the plateaus effectively grow in size to occupy a proportion \( (K+1)/2^K \) of the partition. Regardless of mutation rate, it becomes more probable that mutation will cause a jump onto the plateau, but higher rates are more likely to destroy previously existing blocks, unless these can be repaired by multiple applications.

The effect of Lamarckian learning is subtly different - points with unitation in the partition between 0 and \( K - 2 \) are unchanged, but those with unitation \( K - 1 \) are removed as offspring created in those regions are move to the single sub-solution with a unitation \( K \). Thus the proportion of the partition corresponding to the high-fitness value is now \( 1/(2^K - K) \) which is smaller than the Baldwin version. Thus more of these points are at Hamming distance greater than 1, so we might expect to see the preferential selection for higher mutation rates which are more likely to cause jumps to points at 1-change remove from the optimal sub-solution.

A third class of problems which present a fitness barrier, rather than an entropic one to evolutionary progress to the global optimum is characterised by so-called \( L \) “Trap” or deceptive functions of size \( K \). This paper will consider functions with a deceptive partition whose fitness varies as a function of the unitation \( u_j(i) \) in the \( j - \text{th} \) partition as given in [6]:

\[
    f(i) = 100.0/L \times \sum_{j=1}^{L} \left\{ \begin{array}{ll}
        a(K - 1 - u_j(i)) & u_j(i) \leq K \\
        1 & \text{otherwise}
    \end{array} \right.
\]

(3)

Experiments used a 80 bit four-trap problem and a 64 bit 8 trap problem - so \( L = 20 \) and 8 respectively, and the constant \( a \) at values 0.1 for \( K = 8 \) and 0.2 for \( K = 4 \) to ensure that the optimal configuration for the partition outscore the deceptive optimum.

For each problem, these values of length used were chosen to provide similar levels and speed of convergence to the other problems given the selection regime and population sizes.

### 3.3 Methods for analysis

Each configuration of EA without local search (GA), and with Baldwin (B) or Lamarckian (L) learning with depths 1, 2 and 5 \((B-d1, ..., B-d5, L-d1 \text{ etc.})\) was run 100 times on each problem, with a termination criteria of 50 generations. After each generation of each run data was recorded for the best, worst and mean fitness, mean and standard deviation of mutation rates in the current population, and the total number of evaluations used.

As this paper is primarily concerned with the effect on the learning of mutation rates, algorithms are mostly compared on a generation-by-generation basis, ignoring the fact that the local search variants make more calls to the evaluation function. Results are displayed graphically, and where claims are made about values at the end of runs, these are confirmed by analysis of the results at generation 49 using Analysis of Variance followed by post-hoc testing using Tukeys HSD test at the 95% confidence level. Statistical analysis was carried out using the SPSS v. 20 software. In separate experiments the mean best fitness, average evaluations to solution and success rates were compared for the seven algorithms above, and variants using a fixed mutation rate of \( pm = 1/\ell \). These were repeated with different selection regimes - namely (100,500) with uniform parent selection, and (500,500) with tournaments of size 5 to select parents. The latter should provide similar selection pressure (according to takeover times) and has been suggested to outperform truncation selection [24].

### 4 Results

#### 4.1 Benchmarking Self-Adaptation

Performing an Analysis of Variance (ANOVA) on the \((100,500)\) results for maximum fitness after 49 generations, with the function, and algorithm as independent factors showed that although there were small differences between algorithms, by that stage there were not statistically significant differences between fixed and self-adaptive mutation rates. Comparing the final mean mutation rates, those of the MA-B-d5 algorithm were significantly higher than the other methods, which were otherwise not significantly different. The overall picture was not different with the \((500,500)\) regime.

Comparing the efficiency, as measured by when the best fitness was recorded for each run, ANOVA with this as the dependant variable, and function and algorithm as independent variables showed that the self-adaptive variants were always faster, more significantly so with increased depth of local search. Lamarckian variants were always significantly faster than their Baldwinian counterparts and each step of increasing depth from 0 (GA) through 1.2 and then 5 caused a significant increase in evaluations.

The mean best fitness results for \((100,500)\) showed that there was no difference between the fixed and adaptive mutation rates for Lamarckian search, but these were always significantly better than the GA and Baldwinian MAs. In contrast, adding self-adaptation to the Baldwinian MAs significantly reduced the mean best fitness for each different depth of search.

In order to examine the interplay of effects further, the next set of experiments concentrate on the effect of selection at the level of mutation rates in the presence of different forms of local search. To this end, the “strategy adaptation” parameter \( P_{str} \) was set to 0, so each member of the initial population had its mutation rate randomly set to one of the permissible values, and offspring inherited mutation rates from their parents, but there was no further perturbation of these strategy parameters.

#### 4.2 Evolution of Mutation Rates for OneMax

Figure 2 shows the evolution of the fitness values in the population, the mean and standard deviation of encoded mutation rates
on the OneMax function. For both learning paradigms the rates stabilise more slowly, and to values that decrease with increasing search depth. However for Baldwinian search, the values at generation 49 are not significantly different to the GA.

The effect of selection is much more noticeable with Lamarkian learning. The mutation rates converge faster, and to lower values than the GA - not significantly so for depth 1, but the evolved rates for depths 2 and 5 are significantly different to the GA, and each other.

![Algorithm Comparison](https://via.placeholder.com/150)

**Figure 2:** Evolving Behaviour on OneMax. Increasing lines are best, mean and worst fitness in population. Dashed and dotted lines are mean and standard deviation of mutation rates.

The reduction in the standard deviation shows that this is a learned effect rather than simple drift. To confirm this, experiments were run where the function switched from OneMax to ZeroMax after 25 generations. Figure 3 shows the evolved behaviour on this “switching” problem. The clear spike in the evolved mutation rates after the switch, and subsequent rapid recovery in fitness values, most notably for MA-L-D5 demonstrates that effective self-adaptation is occurring.

### 4.3 Results for Royal Road Functions

Figure 4 shows the evolution of behaviour on the Royal Road function with different sized blocks. In addition to the difference in effectiveness of search, the key point to note is the consistently higher, and more varied mutation rates for Lamarkian search with depth 5, a feature that increases when the size of the sub-blocks to be optimised increased. Mutation rates also increase with depth of Baldwinian search, but the differences are not statistically significant by generation 49.

![Algorithm Comparison](https://via.placeholder.com/150)

**Figure 3:** Evolving Behaviour on Switching function. Increasing lines are best, mean and worst fitness in population. Dashed and dotted lines are mean and standard deviation of mutation rates.

### 4.4 Results for Deceptive Functions

Figure 5 shows the evolution of behaviour on the deceptive function with different sized blocks: Trap-4 and Trap-8. Note the difference in effectiveness of search. On both functions, at generation 49 the statistically homogenous subsets are, ranked according to increasing fitness; (B-d5, GA, B-d1, B-d2) < (L-d1) < (L-d5, L-d2), where the suffix MA is omitted for brevity.

On the functions with 4-bit partitions, the Baldwin behaviour is not statistically significantly different to the GA, but there are consistently lower mutation rates for the Lamarkian learning. This difference is significant even up to generation 49 when the best value had stopped increasing.

With the trap-8 function, the values are no longer statistically significant by generation 49 - but of course there are far fewer sub-functions to be optimised. Considering instead the mean mutation rates across the whole run, there is now a statistically significant difference - the values for Lamarkian learning are significantly lower than for the GA, and then in turn for the Baldwinian learning. These values reflect the speed of the adaptive process - higher mean values meaning slower adaptation.

### 5 Discussion

The first set of benchmarking comparisons confirmed that self-adaptation outperformed a single fixed mutation rate, as expected - working just as effectively at finding good solutions but more efficiently. Lamarkian learning improved the mean best fitness discovered. However, the interplay between the Baldwin effect and self-adaptation was not always beneficial - particular on the Royal Road...
Figure 4: Evolving Behaviour on Royal Road functions with blocks of size 4 (top) and 8 (bottom). Increasing lines are best, mean and worst fitness in population. Dashed and dotted lines are mean and standard deviation of mutation rates.

Figure 5: Evolving Behaviour on Deceptive functions with blocks of size 4 (top) and 8 (bottom). Increasing lines are best, mean and worst fitness in population. Dashed and dotted lines are mean and standard deviation of mutation rates.
landscares where the plateaus form entropic barriers to improvement and the Baldwin effect extends those plateaus.

On the OneMax function, the hypothesis predicted that Lamarkian learning would demonstrate faster adaptation (H2) and to lower (H3) values of mutation rates than the GA. This was supported by the observations. The hypothesis H1 and H3 suggested competing effects would results from Baldwinian learning. Results confirmed that and indeed with depth 1 a slower adaption to higher rates than the GA was seen, an effect which diminished with increased local search depth, but the differences were not statistically significant by the end of even these relatively brief runs.

The results on the switcher function confirmed that self-adaptation is able to occur effectively and efficiently with Lamarkian learning up to a depth 5, possibly even suggesting a synergistic effect when compared to the GA alone.

On the Royal Road functions the hypothesised effects were not really seen except for with depth 5, where as predicted by H4, the Lamarkian search maintains higher mutation rates - which in turn lead to the continued discovery of sub-solutions. For example even after averaging over 100 runs, the bottom right figure of Figure 4 shows an increase in \( f_{\text{max}} \) around 30 generations.

On the trap functions the differences are most evident in the speed of adaptation: as predicted by H1 the “blurring” effect of Baldwin learning significantly reduces the rate of adaptation to lower mutation values than the GA. In contrast, as predicted by H2, the rate of adaptation is faster for Lamarkian learning than for the GA, and hence the overall mean across all generations is lower.

6 Conclusions

This paper set out to examine the interaction between two different forms of memetic learning, and the self-adaptation of mutation rates. A number of hypotheses were proposed to describe the effects. Results suggest that there are indeed significant interactions, but the hypotheses themselves require significant clarification as in cases they work against each other.

The primary empirical results suggest that whereas Lamarkian learning seems to reinforce the self-adaptation process, the Baldwin effect often hinders the process, sometimes with detrimental results on the effectiveness and efficiency of the overall search process.

The primary message of this paper is therefore perhaps unsurprising: that it is wise to rashly mix algorithmic adaptations that work well in isolation. Clearly further empirical and theoretical studies are needed to model these effects so that the twin forces of memes and self-adaptation can be brought to bear with reliable and predictable results.

REFERENCES


[28] J. Smith, ‘Credit assignment in adaptive memetic algorithms’, in Pro-


A Model for Characterising the Collective Dynamic Behaviour of Evolutionary Algorithms

Mikdam Turkey and Riccardo Poli

Abstract. Exploration and exploitation are considered essential notions in evolutionary algorithms. However, a precise interpretation of what constitutes exploration or exploitation is currently lacking and so are specific measures for characterising such notions. In this paper, we start addressing this issue by presenting new measures that can be used as indicators of the exploitation behaviour of an algorithm. This work by characterising the extent to which available information guides the search. More precisely, they quantify the dependency of a population’s activity on the observed fitness values and genetic material, utilising an empirical model that uses a coarse-grained representation of population dynamics and records information about it. The model uses the \( k \)-means clustering algorithm to identify the population’s “basins of activity”. The exploitation behaviour is then captured by an entropy-based measure based on the model that quantifies the strength of the association between a population’s activity distribution and the observed fitness landscape information.

In experiments, we analysed the effects of the search operators and their parameter settings on the collective dynamic behaviour of populations. We also analysed the effect of using different problems on algorithm behaviours. We define a behavioural landscape for each problem to identify the appropriate behaviour to achieve good results and point out possible applications for the proposed model.

1 Introduction

Population-based search algorithms solve problems using a set of interaction mechanisms which control the generation of new individuals in a population. Based on the information obtained via previous individuals, an algorithm directs the search and changes the density of the population in different regions. This bias in directing the movement of the population is hoped to guide the algorithm towards promising areas in the search space and exploring them with more intensity than less promising areas. As the interaction mechanisms operate on moving, creating and/or eliminating individuals, the algorithm produces an emergent “large-scale” behaviour in the dynamics of the population.

Population dynamics has been a central issue in many theoretical investigations. For example, within evolutionary computation theory, approaches include: the schema theory [8, 22, 23], Markov chain formulations [27], Walsh-function-based analyses [1], statistical-mechanical formulations [21] and also some computational complexity approaches [29, 9, 10]. The study of the dynamics/behaviour of populations with respect to specific problems has also been the subject of a number of studies where empirical measures were defined and studied. For example, semi-empirical measures of problem difficulty, such as the fitness-distance correlation [12] and the negative slope coefficient [17], have been proposed to characterise what makes a problem easy or hard for evolutionary algorithms.

All of these approaches have seen some successes at mathematically or empirically modelling evolutionary algorithms. However, very often researchers describe the behaviour of search algorithms in terms of exploration and exploitation, not of run times, success probabilities, or Walsh coefficients. Exploration refers to behaviour resulting in the discovery of new good regions in the search space, while exploitation refers to the behaviour of exploring previously discovered good regions [8, 3]. Researchers have found that to achieve good performance it is often important to control the trade-off between explorative and exploitative behaviour. This is done by either tuning the parameters of search algorithms prior to a run or by changing them dynamically throughout the run based on features of the population such as fitness or diversity. However, there is no precise definition in the literature of the notions of exploration and exploitation, no precise characterisation of the distinction between them, and no numerical quantification of them.

In our previous work [25], we presented a model to analyse the collective dynamic behaviour of population-based algorithms. Based on that model, we defined many measures to extract emergent features attributed to that behaviour. These features can be used to qualitatively describe the exploitation/exploration behaviour of an algorithm. We used self-organising maps (SOMs) [13] to implement the model. The SOMs track the population’s movement as an algorithm operates on solving a problem and this allowed us to gather information about its activities in each of the regions currently being explored. In [26], we have taken the work further by performing more experiments and presenting a new set of measures to characterise the dependency of different collective behaviour features on observed fitness values. We called these measures fitness dependency measures and used them as indicators of the exploitation behaviour of a population. Building on that work, here we have used the model presented in [26, 25] to define a new measure, the genetic dependency that we have used along with fitness dependency to analyse the influence of the observed fitness values and genetic material in guiding the search activities. Our current model is implemented using the \( k \)-means clustering method [4] to identify regions of activity within the population over a run. Based on this model we define entropy-based measures [18] to quantify the dependency of population activities on fitness values and on the diversity of the available genetic material. The proposed measures are used as indicators of the exploitation behaviour of an algorithm as they characterise the extent by which a population
exploits available information to guide search activities.

The proposed model is an effort to contribute to our understanding of the exploration/exploitation phenomena by trying to define what constitutes exploitation and measuring quantities that assess the degree of that behaviour in an algorithm. An algorithm produces different behaviours when it operates on different problems. In accordance with the No Free Lunch (NFL) theorem [30], only certain types of collective dynamic behaviour will lead the algorithm to perform well on a problem. In this work, we show that, for a given problem, an algorithm’s performance can be modelled as a function of the algorithm’s collective behaviour. In the future, this performance model could be used to implement an algorithm selection mechanism to help identify algorithms with a collective behaviour that likely to achieve better results on a problem. It could also be used to tune an algorithm’s parameters so as to exhibit such behaviour or to guide the design of new algorithms.

The rest of the paper is organised as follows. In Section 2, we present the proposed model and define the dependency measures that will be used to characterise the exploitation behaviour. Section 3 outlines the details of our experiments and presents our results of different genetic operators, their settings and interaction on the collective exploitation behaviour of an algorithm. We consider the possible ways in which our model and indicators can be used in Section 4 and we finish with some conclusions in Section 5.

2 The Proposed Model

We will first describe the proposed model (Section 2.1) and then present the dependency measures that we use to characterise a population’s exploitation behaviour (Section 2.2).

2.1 Formulation

Our model is similar, but not identical, to the one proposed in [25]. The proposed model uses a set of nodes to capture the distribution of population individuals at certain points in time. Each node has a centroid representing the centre of a group of individuals in a localised area of the search space. In addition, nodes store information describing the activities of population in the corresponding area. To track the dynamic behaviour of a population over a run, the model is instantiated every so many time steps to represent the motion of the population and record information about its activities as the algorithm progresses in solving a problem. Formally, a set of nodes, \( C^t \), represents a snapshot of the population distribution at time \( t \). Let \( C_0^t \) be the set of nodes that represents the distribution of the initial individuals in the population, \( P_{ini} = \{(x_1, y_1), \ldots, (x_n, y_n)\} \), where \( n \) is the population size, \( x_i \in \mathbb{R}^D \) represents the position of individual \( i \) in the search space (\( D \) being its dimension) and \( y_i \in \mathbb{R} \) is the associated fitness value. Also, let the time-ordered set \( P = \{(x_1, y_1), \ldots, (x_N, y_N)\} \) represent a sequence of all the individuals created by an algorithm over a period of a run from the initial population, \( N \) being the total number of individuals created by the algorithm (i.e., the total number of calls to the fitness function).

The sets of nodes \( C_1^t, C_2^t, \ldots \) are created every \( \tau \) individuals produced by the algorithm (i.e., \( \tau \) represents the sampling period). For example, \( C_1^t \) captures the distribution of the sequence \( P_1 = \{(x_1, y_1), \ldots, (x_\tau, y_\tau)\} \). Generally speaking, \( C^t \) is created by tracking and recording information about individuals in the sequence \( P_t = \{(x_{k+1}, y_{k+1}), \ldots, (x_{k+\tau}, y_{k+\tau})\}, \) where \( k = \tau \times (t - 1) \).

Formally, a set of nodes, \( C^t \), consists of \( n_t \) nodes, where each node, \( C^t_r \), is represented by the following tuple

\[
C^t_r = (m^t_r, h^t_r, d^t_r, f^t_r)
\]

where in \( C^t_r \), \( t \) is time and \( r \in \{1, \ldots, n_t\} \) is the position in the set.

The elements of \( P_t \) are partitioned into non-overlapping subsets, \( P^t_r \), the individuals of which are associated with the node \( C^t_r \) based on their distance from the \( r \)-th centroid (as explained later in this section). Let \( x^t_r \) and \( y^t_r \) represent sets of positions and fitness values of the individuals associated with \( C^t_r \), respectively. Then the elements of the tuple \( C^t_r \) are computed as follows:

\[
m^t_r \in \mathbb{R}^D \text{ is the node centroid. The node centroid represents a region of a population activity. As the algorithm works on redistributing the population around activity regions, node centroids are recalculated so as to represent newly created individuals. As we will see later in this section, we utilised the } k \text{-means clustering method to identify population’s activity regions, divide the individuals among the nodes and calculate the centroid of each one; } h^t_r \text{ (hit counter) is the number of individuals associated with the node } C^t_r \text{. Formally, } h^t_r = |P^t_r|. \text{ It represents the amount of activity that an algorithm allocates for a particular area of the search space. } d^t_r \text{ (hit distance) is the mean distances between a node’s centroid, } m^t_r \text{, and the positions of the individuals, } x^t_r \text{, associated with } C^t_r. \text{ Formally, } d^t_r = \frac{1}{h^t_r} \sum_{a \in C^t_r} \|a - m^t_r\|. \text{ This feature describes the local genotypic diversity of an activity region. } f^t_r \text{ is the mean fitness of the individuals associated with } C^t_r. \text{ Formally, } f^t_r = \frac{1}{h^t_r} \sum_{a \in C^t_r} f(a). \text{ This feature gives insights on the quality of the region of the search space tracked by this node.}

As mentioned earlier, unlike [26, 25], here we use the \( k \)-means clustering method [4] to partition a sequence of individuals \( P_t \) into groups (we call them nodes \( C^t \)) based on distance and to calculate the centroid of each group (i.e., \( m^t_r \)). This makes our model much simpler. In the previous implementation, SOMs were used to preserve the topological distribution of the individuals in a population and keep track of the changes of that distribution (by observing the changes to a SOM’s centroids). These two factors are essential to calculate some of the emergent features presented in the our previous work. However, in this work, we are only interested in capturing a coarse-grained representation of population distribution in order to extract certain features and produce probability distributions for them. For this reason, here we can use the simpler and more efficient \( k \)-means clustering algorithm instead of SOMs.

The most common \( k \)-means algorithm uses an iterative refinement technique to assign observations (individuals) to clusters (nodes) and to update the centroid of each cluster. The process starts by choosing the initial values for cluster centroids. That can be done by selecting random individuals as the initial centroids or by using a seeding technique. Then, each individual is assigned to the nearest cluster and the centroid of each cluster is computed as \( m^t_r = \frac{1}{|{r^t_r}|} \sum_{a \in {r^t_r}} a \). The process of assigning individuals to clusters and updating the centroids continues until we reach a point where no further improvements in the value of the within-cluster sum of squares can be obtained. This is defined as \( \text{WCS} = \sum_{\{r^t_r\}} \sum_{a \in {r^t_r}} \|a - m^t_r\|^2 \). The \( k \)-means algorithm can result in empty clusters, in which case we remove them from \( C^t \). Hence, the number of nodes, \( n_t \), changes over time. In our implementation, we choose the initial value of the number of clusters as 25, therefore, \( n_t \leq 25 \).
two random variables. From Equation 2, we can see that mutual information is a quantity that measures the mutual dependence of the features. In order to characterise the dependency of population behaviour on available information, we need to compute the probability distribution of the features: hit counter (activity), mean fitness and mean hit distance over all areas of population activity. In our model the probability distribution of a feature is approximated via a histogram of that feature across the nodes in the model. The histogram has $m$ equal-size bins or classes (in this work, we choose $m = 10$) whose width is simply the difference between the maximum and minimum values of a feature over all nodes divided by $m$. The probability distribution of each class is then computed as the proportion of activity (as indicated by the hit counter) the nodes of each class received. More specifically, if $L_i$ for $i = 1 \ldots m$ are the bins (classes) obtained by dividing up nodes according to the values of the feature, then the probability distribution of that feature is approximated as:

$$p(L_i) = \left( \sum_{i \in L_i} h^X \right) / \tau, \quad for \ i = 1 \ldots m$$

where $\tau$ is the total number of hits (created individuals) between two sampling points and $h^X$ is the hit counter of node $X$.

To use the uncertainty coefficient to assess the dependency of a population’s behaviour on fitness and hit distance (local genotypic diversity), we first need to compute the probability distribution of the features: hit counter (activity), mean fitness and mean hit distance over all areas of population activity. In our model the probability distribution of a feature is approximated via a histogram of that feature across the nodes in the model. The histogram has $m$ equal-size bins or classes (in this work, we choose $m = 10$) whose width is simply the difference between the maximum and minimum values of a feature over all nodes divided by $m$. The probability distribution of each class is then computed as the proportion of activity (as indicated by the hit counter) the nodes of each class received. More specifically, if $L_i$ for $i = 1 \ldots m$ are the bins (classes) obtained by dividing up nodes according to the values of the feature, then the probability distribution of that feature is approximated as:

$$p(L_i) = \left( \sum_{i \in L_i} h^X \right) / \tau, \quad for \ i = 1 \ldots m$$

With these approximations of the probability and joint probability distributions of features in hand, the entropy and mutual information (Equations 4 and 3, respectively) and then the uncertainty coefficient (Equation 2) can be calculated. In particular, we have applied these definitions to analyse the dependency of hit counter (activity) on mean fitness and mean hit distance. For simplicity, we will denote the corresponding values of the uncertainty coefficient as $f_{m,f}$ and $g_{d,f}$, respectively. They can be defined as follows. Let $act_i = \{h^X, \forall X \in C_i\}$, $fit_i = \{f^X, \forall X \in C_i\}$ and $hd_i = \{d^X, \forall X \in C_i\}$, then $f_{m,f} = U(\text{act}_i | \text{fit}_i)$ and $g_{d,f} = U(\text{act}_i | \text{hd}_i)$.

The quantity $f_{m,f}$ assesses the dependency of activity on fitness values, therefore, we will refer to it as fitness dependency. The quantity $g_{d,f}$, instead, assesses the dependency of activity on genetic material and we will refer to it as genetic dependency. The two measures are used to assess the exploitation behaviour of algorithm $\alpha$ as it operates on problem $f$ at point of time $t$.

The average fitness dependency over all algorithm run is defined as $f_{d,f} = \frac{1}{M} \sum_{i=1}^{n} f_{m,f}$, where, $M = N/\tau$ represents the total number of sampling points, $N$ is the total number of fitness function evaluations in a run and $\tau$ is the sampling period. $g_{d,f}$ is defined in a similar way. We propose to use these two exploitation measures to represent the dynamic collective behaviour of an algorithm with respect to certain problem over a run.

The $k$-means algorithm is sensitive to the initial values of the cluster centroids, meaning that the clustering method may end up clustering a population’s individuals in a different way depending on initial

![Figure 1](image-url)
conditions. This might have a potential impact on our measures as it can affect the probability distribution of the emergent features. To examine the effects of clustering noise, we tested the extent to which different initial conditions can affect the entropy of $fit_i$ and $hdr_i$. In this experiment, at each sampling point $t$, the individuals $P_i$ were clustered 10 times by k-means (using different random seeds). Then, we computed the entropy of $fit_i$ and $hdr_i$, $H(fit_i)$ and $H(hdr_i)$, for each of the 10 clustering results. Finally, we compute the average and the standard deviation for these 10 values for each feature. Results indicated that initial conditions for k-means minimally affect our measures, as in all cases the entropy values had very low standard deviations. Figure 1 shows an example of typical run of BLX-0.5-0.025-2 on F9. Similarly we also found that results are not particularly sensitive to the choice of $k$ and that using $k = 25$ in the k-means algorithm provides a good balance between accuracy and computation time. For this reason, we used this value.

3 Experimental Results

In this section we will look at our experimental setup and experiments.

3.1 Experimental Setup

The collective dynamic behaviour of an evolutionary algorithm is the result of combining the biases of its search operators and their interaction with the problem landscape. The bias of each operator depends on the nature of the operator itself and the values of its control parameters. Thus, although the exploitation behaviour of an algorithm is a collective phenomenon, it depends on the operator biases, parameter settings and the way operators interact with each other, in addition to the bias generated by the fitness landscape. To examine these effects and interactions in relation to the exploitation behaviour of an algorithm, we applied our dependency measures to evolutionary algorithms using three different crossover methods, different levels of selection pressure and different mutation rates. We used generational evolutionary algorithms with fixed population size (100) and constant crossover rate (0.7). Non-uniform mutation has been used with different mutation rates. To refer to our algorithms, we used a naming convention that summarises details about the algorithm in its name. The details of this naming convention are reported in Appendix A.

An algorithm’s dynamic collective behaviour is greatly influenced by the nature of the problem fitness landscape. To study the impact of the problem on the algorithm collective behaviour and performance, we carried out experiments on a number of 10-dimensional real-coded benchmark problems defined in [24]. Each algorithm (e.g., Arth-0.4-0.025-2 or BLX-0.25-0.05-3, see Appendix A) has been tested in 100 independent runs with each problems. Each run lasted for 100,000 fitness function evaluations.

The crossover methods chosen for our tests are: arithmetic crossover, blending crossover (BLX-α), and heuristic crossover (their details and settings are explained in Appendix B). These crossovers were chosen as they handle the parental genetic material in very different manners: deterministic, stochastic, and fitness-biased, respectively. They exhibit different levels of exploitation/exploration because they make use of the information available to them in very different ways. Also, their behaviour changes significantly depending on their parameter settings.

More specifically, arithmetic crossover (Arth-λ) exhibits an exploration behaviour and increases the population diversity for $λ > 1$ or $λ < 0$, and otherwise reduces it [6]. BLX-α widens the distribution of genetic materials when $α > (\sqrt{3} - 1)/2$, and otherwise reduces it [15]. Heuristic crossover, on the other hand, creates offspring close to the better parent, which makes this operator more exploitative (it does not only make use of the provided genetic material, but also uses the fitness values to bias the search). To obtain a full spectrum of behaviours, in our experiments, we used four control parameter settings for each of the three crossover methods. These made the bias of Arth-λ and BLX-α range from exploitative to explorative and the bias of heuristic crossover from more exploitative to less exploitative.

To examine the effect of mutation on the exploitation behaviour of algorithms, non-uniform mutation [14] has also been used with a range of mutation rates. Non-uniform mutation lowers the mutation step over a run to allow more exploration at the initial generations and fine tuning at the late stages of the search. This mutation is considered very appropriate for real-coded evolutionary algorithms [7]. More details are presented in Appendix B.

Obviously, we cannot assess the effect of genetic operators on the exploitation behaviour of an algorithm based only on the way they combine/alter parental genetic material to produce new individuals. This is because the quality of the operators’ output depends on the quality of the material provided in input, which is chosen from the population by the selection mechanism. Therefore, we used tournament selection with different tournament sizes, so as to vary the selection pressure in the algorithm. Larger tournaments induce a higher selection pressure and lead to a higher exploitation behaviour whereas smaller tournaments lead to a lower pressure and a higher exploration.

3.2 Algorithm Settings and Operators Effects

The collective dynamic behaviour of an algorithm is the result of applying the algorithm on a certain problem. Both the algorithm and the problem have an effect on the population dynamics. In this section, we apply our dependency measures on different algorithms operating on problems with different properties. Figure 2(A) compares the fitness dependency and genetic dependency of two sets of algorithms, Heur-0.2-0.025-x and Heur-0.4-0.025-x for $x = 2, \ldots, 8$, run on problem F9. The two sets of algorithms...
use the heuristic crossover method with different parameter settings (0.2 and 0.4) over differ selection pressures (tournament sizes). From the figure we see that Heur-0.2 algorithms exhibit lower fitness-dependencies and genetic-dependencies compared to Heur-0.4 algorithms. This is due to the fact that Heur-0.2 algorithms, especially with high selection pressure levels, tend to narrow the population distribution and loose the ability to exploit. On the contrary, Heur-0.4 algorithms move more slowly toward better solutions, maintaining a higher population diversity for relatively longer periods. This allows an algorithm to exploit good information for longer resulting in higher average fitness and genetic dependencies. We should also note that fitness dependency is greater than genetic dependency for both sets of algorithms. This is because there are more operators in these algorithms that are biased by fitness values (crossover in addition to selection).3

In Figure 2(B), we compare the collective dynamic behaviour of two algorithms with different crossover methods, BLX-0.5 and Arth-0.4, run on problem F18. The figure shows that using different crossover methods leads to different behaviours. The exploitation behaviour of Arth-0.4 algorithms is higher than that of BLX-0.5 algorithms. The stochastic nature of BLX crossover contributes to reduce the exploitation behaviour in comparison to arithmetic crossover which creates new solutions in a deterministic fashion.

To demonstrate the impact of the problem fitness landscape on the algorithm collective dynamic behaviour, we run the same set of algorithms on two different problems: F4, a noisy unimodal fitness function, and F9, a multimodal function with high number of local optima. We can see that although we have used the same set of algorithms, namely BLX-0.5-0.025-x, for x = 2,...,8, the collective behaviour is totally different. The fitness dependency of algorithms operating on F4 is always less than the genetic dependency. This is likely due to the unimodal nature of this fitness landscape. On the contrary, the genetic dependency is less than the fitness dependency in algorithms run on F9. We will explore more the effects of the fitness landscape on the collective behaviour of algorithms later on in the article.

Many researches have analysed the impact of mutation rate on algorithm performance and behaviour (e.g. [16, 28]). In this work, we examine the effect of changing mutation rate on the collective behaviour of algorithms. As we mentioned earlier, non-uniform mutation has been used with different mutation rates. Figure 3 depicts the change in the collective dynamic behaviour of algorithms in response to changing the mutation rate. We chose a set of 24 algorithms, Arth-{0.2, 0.4}-µ-x, BLX-{0.25, 0.5}-µ-x and Heur-{0.2, 0.4}-µ-x, for x = 2,...,5, run on problem F4. We used four mutation rates, µ = 0.0, 0.025, 0.05, 0.1. The figure presents the collective dynamic behaviour of an algorithm in terms of its fitness dependency and genetic dependency. Each algorithm is represented as a point in fitness- and genetic-dependencies space, which we will refer to as the algorithm behavioural space. From the figure we can see that changing mutation rate actually shifts the algorithm behaviour in the behavioural space. Note that how changing the mutation rate has led the algorithms to form a different niche in the behavioural space.

3 Of course, we have to keep in mind that the collective dynamic behaviour of these algorithms is also dependent on the problem they are operating on and different behaviours can result with different fitness landscapes (more about the effect of problem on algorithm’s behaviour in this and next sections).
3.3 Fitness Landscape Effect

As mentioned before, the nature of the fitness landscape has a great impact on how an algorithm conducts the search and distributes the population in the search space. Our measures of population collective dynamic behaviour, and in fact every performance measure, characterise the association between algorithm and problem. An algorithm exhibits different behaviours depending on the problem it operates on, which is consistent with the NFL theorem. To demonstrate this, we ran a set of 24 algorithms on four different problems. The algorithms are Arth-\{0.2, 0.4\}-0.05-\(x\), BLX-\{0.25, 0.5\}-0.05-\(x\) and Heur-\{0.2, 0.4\}-0.05-\(x\) for \(x = 2, \ldots, 5\). The four problems are \(F_4\), \(F_6\), \(F_{15}\) and \(F_{18}\). Figure 4 shows the distributions of the 24 algorithms with respect to the four problems. It is obvious from the figure that the same set of algorithms has different distribution in the behavioural space.

4 Prospects for Applications of the Proposed Model

In section 3.3, we showed that the collective dynamic behaviour of an algorithm changes according to the problem faced by the algorithm. This means that every time we use an algorithm, \(a\), on a problem, \(f\), the collective dynamic behaviour of \(a\), represented in terms of our dependency measures as a point \((f_{d_a, f}, g_{d_a, f})\) in the behavioural space, will be different from the behaviour of another algorithm, \((f_{d_{a'}, f}, g_{d_{a'}, f})\), or the behaviour of the same algorithm applied to different problem, \((f_{d_{a', f'}}, g_{d_{a', f'}})\), unless that the two algorithms, \(a\) and \(a'\), are qualitatively similar or the two problems, \(f\) and \(f'\), have similar features, respectively.

We analysed the distribution of the collective dynamic behaviours of a large set of algorithms in the behavioural space and noticed that algorithms with good performance values (e.g., best fitness achieved) tend to form niches in the behavioural space. In addition, algorithm performance seems to be a continuous function of behaviour in such a space.

Figure 5 depicts the distribution of 336 algorithms on two different problems, \(F_4\) and \(F_{18}\). The tested algorithms are Arth-\{0.2, 0.4\}-1.2, 1.5-\(x\), BLX-\{0.0, 0.25, 0.5, 0.75\}-\(x\) and Heur-\{0.1, 0.2, 0.3, 0.4\}-\(x\), where \(x = 2, \ldots, 8\) and \(\mu = 0.0, 0.025, 0.05, 0.1\). We can see that certain areas in behavioural space have algorithms with better performance. The continuous change in algorithm performances in the behavioural space demonstrates that the relationship between the collective dynamic behaviour, \((f_{d_a, f}, g_{d_a, f})\), and algorithm’s performance, \(p_a, f\), can be modelled as a smooth function, \(p_a, f = pbl(f_{d_a, f}, g_{d_a, f})\), where we term the function \(pbl\) a behavioural landscape function of problem \(f\).

The behavioural landscape of problems \(F_4\) and \(F_{18}\) can be modelled by using a regression method to define the relationship between algorithm’s behaviour and performance. We analysed the data reported in Figure 5 using a local regression method (LOESS) [5]. The resulting behavioural landscapes are shown in Figure 6. Simple inspection of such landscapes reveals that they may help us to identify the desired collective behaviour to achieve good results on a problem.

For this reason, the proposed model has the potential to present a solution to the algorithm-selection [19] or parameter-tuning problems [2]. Both problems can be viewed as the problem of searching the behavioural landscape to find the right algorithm/parameter settings to produce the desired collective dynamic behaviour and consequently achieve the best results on the problem in hand. Obviously in any real-world application, the shape of the behavioural landscape is unknown and, so, an algorithm selection/parameter tuning mechanism would need to sample such a landscape and direct further the exploration toward areas of algorithms with high performance. This can be done by adjusting the parameters of an algorithm, or by selecting another algorithm, to produce a collective behaviour in a certain region in the behavioural landscape containing algorithms more likely to have better performance.

The behavioural landscape represents a unified approach to characterise a problem and to provide useful knowledge about it to guide algorithm design and implementation. It also gives us a different perspective on what makes a problem hard to solve by certain algorithms.

5 Conclusions and Future Work

In this paper, we have presented an empirical model to represent and analyse the collective dynamic behaviour of population-based algo-
The model uses a coarse-grained representation of populations to identify areas of search activity and record information describing genotypic and phenotypic aspects of this activity. Our aim was to characterise the exploitation behaviour of algorithms by quantifying how much the search is guided by the information available in a population. In other words, we measured the extent by which a population makes use of (exploits) observed fitness values and genetic material to direct and intensify population activities. The k-means clustering method was used to identify activity regions, while the dependency of activity on genetic material and fitness values was assessed by an entropy-based measure, namely the uncertainty coefficient.

In order to analyse the effects of different search operators and their parameter settings on the exploitation behaviour of EAs, we applied the proposed model to algorithms with three different crossover methods, different levels of selection pressure and different mutation rates. The chosen crossover methods combine individual solutions and utilise information presented to them in very different ways. Furthermore, each of them has a control parameter that can be used to tune the exploration/exploitation behaviour of the operator.

The mutation rate is considered a key control parameter of EAs. Our results confirmed this. Changing mutation rate can affect dramatically the collective behaviour of populations. We showed that the mutation rate can affect the bias of other operators and change the dependency on fitness and on genetic material.

The effect of using different problems on the collective dynamic behaviour has been analysed. A range of benchmark problems with a variety of properties have been used. We showed that for a problem, a behavioural landscape of the algorithm collective dynamic behaviour can be defined. We explained that this behavioural landscape can be used to implement algorithm-selection or parameter-tuning mechanisms. We also showed that this landscape can be used to characterise a problem and provide guidelines to practitioners to design and implement algorithms for a certain problem.

Although the proposed model has been used to analyse EAs, different kinds of population-based algorithms could be analysed, as the model is totally algorithm-independent and only observes population individuals as they are created. This gives the proposed model an advantage over many analysis tools that have been tailored to analyse or model a certain kind of algorithms or single operators. In addition, the model can be implemented using clustering techniques other than k-means. Possible candidate tools are self-organising maps [13] or principal component analysis (PCA) [11].

A key contribution of the proposed model is that it helps to identify what constitutes exploitation (or exploration) behaviour and to understand the effects that operators with different biases have on the dynamic behaviour of populations. The model also provides a practical tool to analyse a population’s collective behaviour. This tool can be of use by EA practitioners for analysing the effects of operators and for tuning their control parameters. The model also allows the comparison of different algorithms at a higher level than just performance.

In future work, the notion of behavioural landscape that we have defined in this work could be utilised in implementing a performance prediction model. The correlation between algorithm performance and the fitness or genetic dependency can be analysed and possibly used to define a new kind of behavioural landscape.

ACKNOWLEDGEMENTS

This work has been supported by EPSRC via a Doctoral Training Account (DTA) research studentship.

REFERENCES

A Algorithms Naming Convention

We used a naming convention for EAs to capture four facts about an algorithm: crossover method, crossover control parameter, mutation rate and tournament size (selection pressure). The algorithm name has the format \texttt{CCCC-c-m-t}. Here \texttt{CCCC} refers to the crossover method and it can be \texttt{Arth} for arithmetic crossover, \texttt{BLX} for blending crossover and \texttt{Heur} for heuristic crossover (see Appendix B); \texttt{c} is a real number representing the control parameter of the crossover method used; \texttt{m} is the per-locus mutation rate and \texttt{t} is an integer representing the tournament size. For example, \texttt{Arth-0.4-0.10-4} is an EA using arithmetic crossover with control parameter 0.4, mutation rate 0.10 and tournament size 4.

B Crossover and Mutation Operators

Three crossover methods have been used in this paper. They all return two offspring and all have a control parameter that can be used to tune their exploration/exploitation behaviour. These methods are:

- **Arithmetic crossover** (\texttt{Arth-λ}): Two offspring individuals, \(C_1\) and \(C_2\), are produced from two parents, \(P_1\) and \(P_2\), as follows: \(C_1 = P_1 \times λ + P_2 \times (1 – λ)\) and \(C_2 = P_1 \times (1 – λ) + P_2 \times λ\).

- **Blending crossover** (\texttt{BLX-αλ}): Two offspring individuals are produced by randomly (uniformly) generating values for their genes within an interval that depends on the corresponding parental genes. Suppose that \(p_1^i\) and \(p_2^i\) are the \(i\)th parameter of parents \(P_1\) and \(P_2\), respectively, the corresponding parameter, \(c_1^i\), of offspring \(C_1\) is randomly chosen from the interval \([p_{\text{min}}^i – I \times α, p_{\text{max}}^i + I \times α]\), where \(p_{\text{min}} = \min\{p_1^i, p_2^i\}\), \(p_{\text{max}} = \max\{p_1^i, p_2^i\}\) and \(I = p_{\text{max}} – p_{\text{min}}\).

- **Heuristic crossover** (\texttt{Heur-λ}): This method creates one offspring individual around the parent with the highest fitness. Here we modified it slightly so as to produce two offspring. Suppose that we have two parent individuals, \(P_1\) and \(P_2\), and that \(P_1\) is the one with higher fitness, then the two offspring individuals, \(C_1\) and \(C_2\), are created as: \(C_1 = P_1 – λ \times (P_2 – P_1)\) and \(C_2 = P_1 + λ \times (P_2 – P_1)\).

Non-uniform mutation has been used in this work. This mutation has the ability to adapt over run to allow different degrees of exploration. It is defined as following.

Let \(C = (c_1, \ldots, c_D)\) represents an individual undergoing mutation, where \(D\) represents the problem dimension, and it is set to 10, and \(c_i \in [a_i, b_i]\) is a gene to be mutated and \(a_i\) and \(b_i\) are the lower and upper limits of possible values range, then the mutated gene \(c'_i\) is calculated as follows

\[
c'_i = \begin{cases} 
c_i + \Delta(t, b_i - c_i) & \text{if } q = 1 \\
c_i - \Delta(t, c_i - a_i) & \text{if } q = 0
\end{cases}
\]

with \(q\) is a random number which may have the value of either one or zero, and \(\Delta(t, y) = y(1 - \left(\frac{t}{t_{\text{max}}}\right)^6)\), where \(t\) is the number of current generation, \(t_{\text{max}}\) is the maximum number of generations, \(r\) is a random number from the interval \([0, 1]\) and \(b\) is a parameter determines the degree of dependency on the number of generation \(t\). We chose \(b = 5\).
NeuroEvolution: The Importance of Transfer Function Evolution and Heterogeneous Networks

Andrew James Turner ¹ and Julian Francis Miller ²

Abstract. NeuroEvolution is the application of Evolutionary Algorithms to the training of Artificial Neural Networks. Currently the vast majority of NeuroEvolutionary methods create homogeneous networks of user defined transfer functions. This is despite NeuroEvolution being capable of creating heterogeneous networks where each neuron’s transfer function is not chosen by the user, but selected or optimised during evolution. This paper demonstrates how NeuroEvolution can be used to select or optimise each neuron’s transfer function and empirically shows that doing so significantly aids training. This result is important as most NeuroEvolutionary methods are capable of creating heterogeneous networks using the methods described.

1 Introduction

NeuroEvolution (NE) is the application of Evolutionary Algorithms (EA) to the training of Artificial Neural Networks (ANN). NE’s history began by evolving the connection weights of fixed topology ANNs [23, 35]. This method brought many advantages over the still popular gradient based methods; such as back propagation [24]. These advantages include: being able to escape local optima, being less sensitive to the initial connection weights, being suited to deep ANNs and not requiring that each neuron’s Transfer Function (TF) be differentiable [37]. NE is also suited to reinforcement learning as well as supervised learning; whereas back propagation is only suited to supervised learning. Other ANN training methods such as restricted Boltzmann machines are also suited to unsupervised learning [26].

A significant advantage of NE is its ability to evolve the topology of ANNs; as well as the connection weights. Topology evolving NE methods include: GNARL [1], NEAT [27], SAGA [4] and CGPANN [8, 29]. This ability to automatically create suitable topologies is significant as topology has been shown to strongly influence the effectiveness of back propagation [10] and weight only evolving NE [30]. Evolving the topology of ANNs has even been shown to be more important to training than evolving connection weights [30]. Although some non-evolutionary ANN training methods do adapt topology, they typically achieve this by iteratively adding or removing neurons during training. This approach is akin to a local search of topologies, and is consequently likely to become trapped in locally sub-optimal topologies [1].

Interestingly, NE can also be used to optimise the TF of each neuron within heterogeneous ANNs. However, this capability of NE has been widely overlooked in recent research. Indeed, at the turn of the 21st century many ANN publications stated that more research was required surrounding the optimisation of TFs: “Relatively little has been done on the evolution of node transfer functions, let alone the simultaneous evolution of both topological structure and node transfer functions” [37]. “The current emphasis in neural network research is on learning algorithms and architectures, neglecting the importance of transfer functions” [5] and “Selection and/or optimisation of transfer functions performed by artificial neurons have been so far little explored ways to improve performance of neural networks in complex problems” [6]. However, a search of the literature reveals that there has been little active research in this area. This paper intends to help fill this gap by showing how NE can easily optimise neuron TFS during evolution and that doing so produces strongly beneficial results.

The remained of this paper is structured as follows. Section 2 introduces some background of applying NE to evolving the TFS of ANNs. Section 3 describes the investigations which were undertaken using NE to evolve TFS, with the results given in Section 4. Finally Section 5 discusses the overall findings with final conclusions given in Section 6.

2 Background

There are vast number of ANN TFS found in the literature [6]. However, the majority of NE implementations only evolve homogeneous ANNs of logistic functions or Gaussian functions; which have both been shown capable of universal approximation; [7] and [20] respectively. Of those which do evolve heterogeneous ANNs there are two main methods.

The first method selects the TF of each neuron from a predetermined list of TFS. Training methods which use this method include General Neural Networks (GNN) [11]; which randomly adds or removes logistic or Gaussian TFS using an evolutionary programming method. GNN is also a hybrid approach which makes use of back propagation during training. Other NE methods which select specific TFS for each neuron include Parallel Distributed Genetic Programming (PDGP) [21], a modified Hierarchical Co-evolutionary Genetic Algorithm (HCGA₂) [34] and Cartesian Genetic Programming of Artificial Neural Networks (CGPANN) [8, 29]. These methods use genes to encode which TF is used by each neuron. These genes are then subject to mutation and/or crossover during evolution.

The second way in which NE can optimise neuron TFS is to use TFS which are described by a number of parameters [6]. The training methods then optimises these parameters for each individual neuron. A simple version of this technique has been used by CGPANN [13]; where the widths of Gaussian functions were optimised for each neuron. Again the parameter associated with each neuron was encoded in the chromosome by the addition of an extra gene for each neuron.

¹ The University of York, England, email: andrew.turner@york.ac.uk ² The University of York, England, email: julian.miller@york.ac.uk
A more complex version of this method was used in [2] where each neuron’s TF was itself an evolved Genetic Program. This method allowed for an almost limitless variations of TFs. Another example where each neuron is described by a number of genes, is state-enhanced neural networks [19], where the dynamics of each neuron are evolved. These state-enhanced neural network exhibit memory which can be utilised on partly observable Markov decision tasks.

Up until now however there has been little research which empirically and rigorously investigates if the ability for NE to evolve heterogeneous ANNs actually provides any benefit. This is important as if it is shown to be beneficial it could easily be adopted by other NE methods; as the described methods just require an additional gene(s) per neuron. As discussed there are two ways in which NE can evolve TFs; 1) by choosing the TF of each neuron from a predetermined list or 2) by optimising parameters associated with each individual neuron. Both of these methods are investigated here using two NE strategies and compared to evolving regular homogeneous ANNs.

3 Investigation

The investigation presented in this paper takes three parts. The first is to identify if the choice of TF impacts on the effectiveness of NE when using homogeneous ANNs. The second investigates allowing NE to select each neuron’s TF from a predetermined list. The third investigates using NE to optimise parameters associated with each neuron’s TF. It would also be possible to use NE to evolve ANNs with a range of TFs each of which had separate parameters to be optimised; but this was not undertaken here.

The remainder of this section introduces the NE methods employed by the investigation, the TFs made available and the benchmarks used.

3.1 NeuroEvolutionary Strategies

In order to undertake the described experiments, two NE methods were used; this is to ensure that any conclusions are not specific to a particular type of NE. The chosen NE methods are Conventional NeuroEvolution (CNE) and Cartesian Genetic Programming of Artificial Neural Networks (CGPANN). CNE is the simplest (and oldest) form of NE and only evolves connection weights of fixed topology networks. CGPANN is a more complex NE method which evolves both the weights and topology of ANNs. These two NE methods represent the two main types of NE; those which evolve only connection weights and those which evolve connection weights and topology.

3.1.1 Conventional NeuroEvolution

CNE [23] operates by storing the connection weights of a fixed topology network as an array of floating point numbers; each within a range specified by the user. Each of these arrays represents a chromosome. Mutation is implemented by selecting a new random weight value for each gene (weight) with a given probability. CNE is extended here to be capable of evolving each neuron’s TF by the inclusion of an additional gene per neuron. These TF genes can either be used as an index in a look-up-table of TFs, or as a parameter value to be used by each neuron’s TF. As CNE uses fixed topologies, this topology must be selected in advance by the user.

3.1.2 Cartesian Genetic Programming of Artificial Neural Networks

CGPANN [8, 29] is the application of Cartesian Genetic Programming (CGP) to the evolution of ANNs. CGP [18, 17] is a form of Genetic Programming (GP) which represents computational structures as a directed graph of nodes indexed by Cartesian coordinates. CGP does not suffer from bloat [15, 31]; an Achilles heel of many GP methods [25]. CGP chromosomes also contain non-functioning genes enabling neutral genetic drift during evolution [33, 38]. CGP typically evolves acyclic networks but can also be easily adapted to evolve cyclic or recurrent networks. CGP typically uses point or probabilistic mutation and no crossover. CGP is easily applied to ANNs [8, 29] by the inclusion of connection weight genes and by using TFs suited to ANNs. CGPANN has all of the benefits of CGP and is a NE training method which can evolve the weights, topology [30] and TFs of ANNs. Although CGP freely evolves topology, it is required that the user specifies a maximum network size. This could be considered a drawback, but overestimating the required number of nodes has been shown to be highly beneficial for CGP [16]. Similarly, a maximum neuron arity must be specified, however, the arity of each neuron can be lower than this maximum [29]. This occurs when the chromosome describes two neurons being connected by two or more connections. In this case, multiple connections between two neurons are equivalent to one connection; with the weight value being the sum of the individual connection weights.

It is important to note that the types of ANN created using CGPANN are unconventional and often cannot be described in terms of layers and nodes per layer. Figure 1 gives an example of the type of ANN which can be created using CGPANN. It can be seen that the neuron inputs are highly unrestricted; they can connect to any previous neuron in the network. It can also be seen that the arity of each neuron can vary. Additionally any neuron can be used as an output; including input neurons. Figure 1 demonstrates that when using NE to optimise topology, evolution is capable of utilising topologies which would be unlikely to be considered by a human designer.

![Figure 1. Depiction of the types of ANN created using CGPANN.](image)

3.2 Transfer Functions

The TFs used for the first two parts of the investigation are the Heaviside step function, Equation 1, the Gaussian function, Equation 2, and the logistic function, Equation 3. Each of these TFs is shown

\[ f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \]  

\[ f(x) = e^{-\frac{x^2}{2\sigma^2}} \]  

\[ f(x) = \frac{1}{1 + e^{-x}} \]

\[ \text{The logistic function is often referred to as the sigmoid function in the ANN literature. In fact the term sigmoid function refers to any function which is 'S' shaped. The logistic function is therefore a specific type of sigmoid function along with other functions including the Gompertz function.} \]
graphically in Figure 2. These particular TFs were selected as they are the most commonly used by ANNs.

As can be seen in Equations 2 and 3, the Gaussian and logistic function have been given in a form which contains a $\sigma$ variable. Where $\sigma$ is set as one for the typical form of these TFs. When using NE to evolve parameters associated with each neuron, the $\sigma$ value can be evolved or optimised. Figures 3 and 4 show the Gaussian and logistic function respectively for a range of $\sigma$ values.

$$f(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

$$f(x) = \exp \left( -\frac{x^2}{2\sigma^2} \right) \quad (2)$$

$$f(x) = \frac{1}{1 + \exp(-\sigma x)} \quad (3)$$

![Figure 2](image2.png) Form left to right: Heaviside step function, Gaussian function and the logistic function. With $\sigma = 1$ for the Gaussian and logistic functions.

![Figure 3](image3.png) Variable Gaussian function. From left to right $\sigma = 1, 2$ and 3.

![Figure 4](image4.png) Variable logistic function. From left to right $\sigma = 1, 2$ and 3.

3.3 Benchmarks

In order to draw strong conclusions regarding whether it is beneficial to evolve TFs, it is necessary to examine its effectiveness on a wide range of benchmarks. In this paper five benchmarks were employed. The chosen benchmarks mainly include supervised learning classification tasks, a common application of ANNs, but also include a reinforcement learning control task (ball throwing).

Despite many of the described benchmarks being classification tasks, they each use their own type of fitness function. Although this adds complexity, the fitness functions are those typically used with these benchmarks. This is done to ensure the standardised use of these benchmarks; which is important when comparing machine learning methods.

3.3.1 Ball Throwing

The ball throwing benchmark [9] is a reinforcement learning control task. The task is to design a controller for a driven arm so as to throw a ball a distance of $\geq 9.5$ m. A depiction of the task is given in Figure 5, with the equations describing the dynamics of the arm given in Equations 4 and 5; symbol definitions given in Table 1. The model is simulated using Euler integration with a time step of 0.01 s for 3000 time steps. The control system has two inputs $\theta$ and $\omega$ and outputs two values $T$ and whether or not to release the ball. The inputs to the controller are linearly scaled from $\pm \pi/2$ and $\pm 5$ rad/s to a [0,1] range for $\theta$ and $\omega$ respectively. The first output of the controller sets the torque applied to the arm and is linearly mapped to a $[0,1]$ N range. The ball is released if the second output exceeds a threshold of 0.5. Once the ball is released, Newtonian mechanics are used to calculate the distance the ball is thrown ($d$) which is then used as the fitness value.

$$\left( \dot{\theta}, \dot{\omega} \right) = \left( \omega, -c \cdot \omega + \frac{g \cdot \sin(\theta)}{l} + \frac{T}{m \cdot l^2} \right) \quad (4)$$

$$\omega = 0 \text{ if } |\theta| \geq \frac{\pi}{2} \quad (5)$$

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\theta$</td>
<td>The arm angle</td>
<td>$[-\frac{\pi}{2}, \frac{\pi}{2}]$ rad</td>
</tr>
<tr>
<td>$\omega$</td>
<td>The arm’s angular velocity</td>
<td>$2.5 \text{rad}^{-1}$</td>
</tr>
<tr>
<td>$c$</td>
<td>Friction constant</td>
<td>$2.5 \text{s}^{-1}$</td>
</tr>
<tr>
<td>$l$</td>
<td>Arm length</td>
<td>$2$ m</td>
</tr>
<tr>
<td>$g$</td>
<td>Gravity</td>
<td>$9.8 \text{m/s}^2$</td>
</tr>
<tr>
<td>$m$</td>
<td>Ball mass</td>
<td>$0.1$ kg</td>
</tr>
<tr>
<td>$T$</td>
<td>Torque applied to arm</td>
<td>$[-5, 5]$ Nm</td>
</tr>
</tbody>
</table>
3.3.2 Full Adder

The full adder benchmark is the task of implementing a full adder circuit using an ANN. The ANN has three inputs (two input bits and a carry bit) and two outputs (one for the sum bit and the other for the carry out). Each output is decoded as a ‘1’ if $\geq 0.5$, otherwise it is decoded as a ‘0’. The fitness value assigned to each chromosome is the number of correct output bits generated after every possible input pattern has been applied; see Table 2. This results in a maximum fitness of sixteen.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>Sum</th>
<th>Cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2. Full Adder truth table.

3.3.3 Monks Problem 1

The Monks Problems [28] are a set of three classification benchmarks intended for comparing learning algorithms. The classification tasks are based on the appearance of robots which are described by six attributes, each with a range of values; see Table 3. Only the first classification task is used here, where a robot belongs to a class if head_shape = body_shape OR jacket_color = red. The task uses 124 of the possible 432 combinations for the training set and the remainder for the testing set. The implementation commonly used by ANN is to assign each value of each attribute its own input to the network; totaling seventeen inputs. Each of these inputs is set as ‘1’ if the particular attributes value is present and as ‘0’ otherwise. The ANN classifies each sample as belonging to the class if the single ANN output is $\geq 0.5$. The target fitness is zero percent classification error.

<table>
<thead>
<tr>
<th>Description</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>head_shape</td>
<td>round, square, octagon</td>
</tr>
<tr>
<td>body_shape</td>
<td>round, square, octagon</td>
</tr>
<tr>
<td>is_smiling</td>
<td>yes, no</td>
</tr>
<tr>
<td>holding</td>
<td>sword, balloon, flag</td>
</tr>
<tr>
<td>jacket_color</td>
<td>red, yellow, green, blue</td>
</tr>
<tr>
<td>has_tie</td>
<td>yes, no</td>
</tr>
</tbody>
</table>

Table 3. Monks Problem Robot Descriptions.

3.3.4 Two Spirals

The two spirals classification benchmarks was created in the 1980s and was originally posted on a connectionist mailing list by Alexis Wieland [3]. The benchmarks consists of 194 data points describing samples taken from two spirals in Cartesian space; see Figure 6. The task is to classify to which spiral each sample belongs using only the $(x, y)$ Cartesian coordinates. The target fitness is zero miss classifications. The ANN comprises of two inputs for the $(x, y)$ Cartesian coordinates of each sample, and one output. When the output value is $< 0.5$ it is interpreted as one class and $\geq 0.5$ as the other.

3.3.5 Proben1: Cancer1

The Cancer1 dataset5 is a classification task taken from the Proben1 document [22]. The dataset was originally constructed at the University of Wisconsin Hospital [12]. Each sample in the dataset describes nine values, recorded by a surgeon using fine needle aspiration, of a tumour located in the breast of patients. Each sample is labeled with two mutually exclusive flags, benign and malignant, indicating the tumour type. All of the values are scaled into a $[0, 1]$ range. The dataset contains 699 samples, 65.5% of which represent benign tumours. The first 525 samples are used as the training set with the remainder used for the testing set. The fitness assigned to each chromosome is the squared error percentage, Equation 6. Where $o_{\text{min}}$ and $o_{\text{max}}$ are the minimum and maximum output values form the ANN, $N$ is the number of outputs from the ANN, $P$ is the number of training examples, $o_{pi}$ are the actual output values from the ANN and $t_{pi}$ are the target outputs. Therefore the optimum corresponds to a squared percentage error equal to zero.

$$E = 100 \cdot \frac{o_{\text{max}} - o_{\text{min}}}{N \cdot P} \sum_{p=1}^{P} \sum_{t=1}^{N} (o_{pi} - t_{pi})^2 \tag{6}$$

4 Results

Three experiments are presented here which investigate the influence of TPs when using NE to train ANNs. All of the results presented are taken from fifty repeated runs. Each run was terminated after 1000 generations, all used a $(1 + 4)$-ES, $3\%$ probabilistic mutation and connection weights in the range $\pm 5$. It should be noted that all of the results presented could likely be improved given more generations and are not representative of the maximum ability of any of the employed methods. When using CNE, three hidden layers were used each containing ten neurons; plus one input layer and one output layer. The arity of each neuron was such that the ANN was fully connected between layers. When using CGPANN the maximum number of nodes was set as thirty each with a maximum arity of ten.

Where appropriate, the results are compared using the non-parametric two sided Mann-Whitney U-test and the effect size [32] statistics. A U-test value of $< 0.05$ indicates that the difference between two datasets is statistically significant. The effect size value

---

5 The ‘1’ in ‘Cancer1’ refers to the permutation of the dataset; see [22].

6 Where probabilistic mutation changes each gene to a new valid value with a given probability.
shows the important of this difference considering the spread of the data; with values $> 0.56$ showing small importance, $> 0.64$ medium importance and $> 0.71$ large importance. Therefore if a comparison between results is shown to be statically significant with a medium or large effect size, then we can be reasonably sure that any difference is not due to under sampling and that the difference is significantly large.

### 4.1 Experiment 1 - Homogeneous Networks

In homogeneous ANNs the TF used by each neuron is the same, whereas in heterogeneous ANNs the network uses different types of TF for different neurons. The first experiment identifies whether, and to what extent, the choice of TF impacts on the effectiveness of training homogeneous ANN using NE. As previously discussed, the three TFs used for this investigation are the Heaviside step, Gaussian and logistic functions; see Section 3.2.

The average fitness achieved when using each TF is given for the five benchmarks in Tables 4 and 5; when using CNE and CGPANN respectively. The average fitness value is given in bold if it represents the best fitness for that benchmark; indicating the most suitable TF for that benchmark. When appropriate, the fitness is given for the training and testing sets. Where the testing fitness is the average fitness achieved by each of the fifty runs on the testing set after training on the training set is complete. The statistical significance between the fitnesses achieved using each TF are given in Tables 6 and 7; when using CNE and CGPANN respectively. When the difference is statistically significant the value is given in bold. The effect size of the differences between the fitnesses are given in Tables 8 and 9; when using CNE and CGPANN respectively. When the effect size is of medium or grater importance the value is given in bold.

From the results given in Tables 4 and 5 it can be seen, for both CNE and CGPANN, that the choice of TF has a large impact on the effectiveness of NE. Additionally, in the majority of cases these differences are shown to be statistically significant and with a medium or large effect size. This confirms that the choice of TF has a large impact on the effectiveness of NE. Interestingly the most suitable TF was often dependant on the NE training method used. Interestingly again, for the classification tasks with testing sets, the best TF for training error percentage was not always the best for generalisation.

### Table 4. Average fitness of homogeneous ANN using different TFs trained using CNE.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Step</th>
<th>Gaussian</th>
<th>Logistic</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ball Throwing</td>
<td>5.63</td>
<td>6.41</td>
<td>5.57</td>
<td>5.87</td>
</tr>
<tr>
<td>Full Adder</td>
<td>16.00</td>
<td>15.92</td>
<td>15.86</td>
<td>15.93</td>
</tr>
<tr>
<td>Monks Problem 1 Train</td>
<td>9.82</td>
<td>27.65</td>
<td>11.03</td>
<td>16.17</td>
</tr>
<tr>
<td>Monks Problem 1 Test</td>
<td>27.98</td>
<td>43.16</td>
<td>25.87</td>
<td>32.34</td>
</tr>
<tr>
<td>Two Spirals</td>
<td>70.00</td>
<td>56.54</td>
<td>81.52</td>
<td>96.35</td>
</tr>
<tr>
<td>Proben1: Cancer Train</td>
<td>10.50</td>
<td>5.44</td>
<td>3.35</td>
<td>6.43</td>
</tr>
<tr>
<td>Proben1: Cancer Test</td>
<td>14.44</td>
<td>7.49</td>
<td>3.54</td>
<td>8.49</td>
</tr>
</tbody>
</table>

### Table 5. Average fitness of homogeneous ANN using different TFs trained using CGPANN.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Step</th>
<th>Gaussian</th>
<th>Logistic</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ball Throwing</td>
<td>9.34</td>
<td>7.34</td>
<td>3.80</td>
<td>7.49</td>
</tr>
<tr>
<td>Full Adder</td>
<td>15.94</td>
<td>15.40</td>
<td>15.78</td>
<td>15.71</td>
</tr>
<tr>
<td>Monks Problem 1 Train</td>
<td>10.71</td>
<td>15.27</td>
<td>12.72</td>
<td>12.90</td>
</tr>
<tr>
<td>Monks Problem 1 Test</td>
<td>13.44</td>
<td>21.93</td>
<td>18.79</td>
<td>18.05</td>
</tr>
<tr>
<td>Two Spirals</td>
<td>67.42</td>
<td>66.36</td>
<td>80.64</td>
<td>71.47</td>
</tr>
<tr>
<td>Proben1: Cancer Train</td>
<td>2.16</td>
<td>2.55</td>
<td>2.50</td>
<td>2.40</td>
</tr>
<tr>
<td>Proben1: Cancer Test</td>
<td>2.71</td>
<td>2.74</td>
<td>2.09</td>
<td>2.51</td>
</tr>
</tbody>
</table>

### Table 6. Statistical significance between the CNE results given in Table 4.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Step-Gauss</th>
<th>Step-Log</th>
<th>Gauss-Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ball Throwing</td>
<td>3.55E-1</td>
<td>9.55E-15</td>
<td>4.68E-1</td>
</tr>
<tr>
<td>Full Adder</td>
<td>4.33E-2</td>
<td>6.49E-3</td>
<td>3.43E-1</td>
</tr>
<tr>
<td>Monks Problem 1 Train</td>
<td>6.41E-18</td>
<td>3.81E-2</td>
<td>7.10E-18</td>
</tr>
<tr>
<td>Monks Problem 1 Test</td>
<td>6.92E-18</td>
<td>3.34E-2</td>
<td>6.94E-18</td>
</tr>
<tr>
<td>Two Spirals</td>
<td>2.41E-17</td>
<td>6.86E-18</td>
<td>4.94E-18</td>
</tr>
<tr>
<td>Proben1: Cancer Train</td>
<td>1.72E-14</td>
<td>6.34E-18</td>
<td>3.54E-10</td>
</tr>
<tr>
<td>Proben1: Cancer Test</td>
<td>2.26E-13</td>
<td>6.19E-18</td>
<td>1.49E-11</td>
</tr>
</tbody>
</table>

### Table 7. Statistical significance between the CGPANN results given in Table 5.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Step-Gauss</th>
<th>Step-Log</th>
<th>Gauss-Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ball Throwing</td>
<td>4.09E-10</td>
<td>2.06E-13</td>
<td>1.22E-1</td>
</tr>
<tr>
<td>Full Adder</td>
<td>1.39E-4</td>
<td>1.01E-1</td>
<td>1.52E-2</td>
</tr>
<tr>
<td>Monks Problem 1 Train</td>
<td>6.92E-3</td>
<td>2.12E-2</td>
<td>4.66E-2</td>
</tr>
<tr>
<td>Monks Problem 1 Test</td>
<td>1.75E-5</td>
<td>1.02E-2</td>
<td>9.30E-3</td>
</tr>
<tr>
<td>Two Spirals</td>
<td>3.14E-1</td>
<td>3.83E-17</td>
<td>2.53E-17</td>
</tr>
<tr>
<td>Proben1: Cancer Train</td>
<td>7.12E-7</td>
<td>5.07E-7</td>
<td>7.91E-1</td>
</tr>
<tr>
<td>Proben1: Cancer Test</td>
<td>8.99E-1</td>
<td>6.51E-3</td>
<td>2.85E-3</td>
</tr>
</tbody>
</table>

### Table 8. Effect Size between the CNE results given in Table 4.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Step-Gauss</th>
<th>Step-Log</th>
<th>Gauss-Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ball Throwing</td>
<td>0.573</td>
<td>0.940</td>
<td>0.542</td>
</tr>
<tr>
<td>Full Adder</td>
<td>0.54</td>
<td>0.570</td>
<td>0.530</td>
</tr>
<tr>
<td>Monks Problem 1 Train</td>
<td>~1</td>
<td>0.620</td>
<td>0.999</td>
</tr>
<tr>
<td>Monks Problem 1 Test</td>
<td>~1</td>
<td>0.624</td>
<td>~1</td>
</tr>
<tr>
<td>Two Spirals</td>
<td>0.991</td>
<td>0.998</td>
<td>~1</td>
</tr>
<tr>
<td>Proben1: Cancer Train</td>
<td>0.945</td>
<td>~1</td>
<td>0.863</td>
</tr>
<tr>
<td>Proben1: Cancer Test</td>
<td>0.925</td>
<td>~1</td>
<td>0.890</td>
</tr>
</tbody>
</table>

### Table 9. Effect Size between the CGPANN results given in Table 5.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Step-Gauss</th>
<th>Step-Log</th>
<th>Gauss-Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ball Throwing</td>
<td>0.865</td>
<td>0.412</td>
<td>0.585</td>
</tr>
<tr>
<td>Full Adder</td>
<td>0.657</td>
<td>0.552</td>
<td>0.608</td>
</tr>
<tr>
<td>Monks Problem 1 Train</td>
<td>2.85E-3</td>
<td>0.632</td>
<td>0.615</td>
</tr>
<tr>
<td>Monks Problem 1 Test</td>
<td>2.85E-3</td>
<td>0.647</td>
<td>0.649</td>
</tr>
<tr>
<td>Two Spirals</td>
<td>0.558</td>
<td>0.988</td>
<td>0.590</td>
</tr>
<tr>
<td>Proben1: Cancer Train</td>
<td>2.85E-3</td>
<td>0.787</td>
<td>0.571</td>
</tr>
<tr>
<td>Proben1: Cancer Test</td>
<td>0.507</td>
<td>0.655</td>
<td>0.669</td>
</tr>
</tbody>
</table>
4.2 Experiment 2 - Heterogeneous Networks

The second experiment identifies if allowing NE to select the TF for each neuron from a predetermined list is beneficial; and if so to what extent. Evolving the TF used by each neuron is considered beneficial if the result is better than the average of using each TF individually. This is chosen because when approaching a new task it not generally known which TF would be most suited; therefore a TF would have to be selected arbitrarily. Here the need to make this choice is removed, and hence it should be considered beneficial if it beats the average random choice. The average fitnesses of using each TF individually for the five benchmarks are given in Tables 4 and 5 for CNE and CGPANN respectively.

The average fitnesses achieved when evolving heterogeneous ANN are given in Tables 10 and 11 for CNE and CGPANN respectively. The results are given in bold if the fitness is better that the average of using each TF individually. The percentage of neurons which use each TF is also given in Tables 10 and 11; this is only for the active nodes in the CGPANN case. No statistical analysis can be undertaken for this experiment as the comparison is against the average result of using each TF individually.

As can be seen in Tables 10 and 11, in the majority of cases evolving heterogeneous ANNs outperformed the average result of evolving homogeneous ANNs. This indicates that evolving heterogeneous ANNs is typically a better strategy than evolving homogeneous ANN. This holds unless the user knows in advance which TF is most suited to a given task; in which case that TF should be used.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Monks Problem 1</th>
<th>Two Spirals</th>
<th>Proben1: Cancer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Train</td>
<td>8.83</td>
<td>16.00</td>
<td>3.87</td>
</tr>
<tr>
<td>Step Gaussian</td>
<td>3.60%</td>
<td>32.4%</td>
<td>31.8%</td>
</tr>
<tr>
<td>Logistic</td>
<td>33.1%</td>
<td>34.1%</td>
<td>32.2%</td>
</tr>
<tr>
<td>Average</td>
<td>30.9%</td>
<td>33.5%</td>
<td>36.0%</td>
</tr>
</tbody>
</table>

Table 10. Average fitness of heterogeneous ANN trained using CNE. The percentage of neurons which used each TF is also given.

Table 11. Average fitness of heterogeneous ANN trained using CGPANN. The percentage of neurons which used each TF is also CNGPANN.

4.3 Experiment 3 - Evolving Transfer Function Parameters

The third experiment identifies if optimising parameters associated with each neuron is beneficial for NE. As discussed, the parameters to be optimised vary the shape of the Gaussian and logistic functions; see Section 3.2. In each case the ANNs use a fixed TF, Gaussian or logistic, but a parameter describing the shape of each neuron’s TF is optimised or evolved. Here the parameter values for each TF is limited to the values 1, 2, or 3, see Equations 2 and 3; but this is not a requirement of the method.

Evolving parameters associated with each neuron will be considered beneficial if it produces stronger results than the use of the non-parametrised counterpart e.g. if variable Gaussian produces stronger results than the standard Gaussian TF.

The results of using variable Gaussian and variable logistic functions on the five benchmarks are given in Tables 12 and 13 respectively when using CNE. Similarly Tables 14 and 15 give the results when using CGPANN. In all cases the results are compared to those obtained for the non-variable form of the function. In the given Tables, a bold fitness value indicates that the variable TF performed better than the non-variable form. Additionally, bold values for the U-test and effect size indicate statistical significance and a meaningful difference respectively. For instance, if the fitness, U-test and effect size values are all given in bold then the variable TF is shown to outperform the non-variable counterpart. If however the fitness value is not bold, but the U-test and effect size values are, then this shows that the non-variable TF outperformed the variable counterpart. If either of the U-test or effect size values are not bold then the difference between the two forms of TF is considered insignificant.

It can be seen in Tables 12-15, that in the majority of cases, the variable version of the TF outperformed the non-variable form. Additionally, many instances where the variable form is superior are also shown to be statistically significant with a medium to large effect size. Only three of the twenty cases show the non-variable form outperforming the variable form with statistical significance and medium to large effect size. Seven of the twenty cases showed the variable form to outperform the non-variable form with statistical significance and a medium to large effect size. The remaining ten cases showed no significant difference between the variable and the non-variable TFs. Therefore using variable TFs is shown to be often beneficial and rarely worse. Interestingly, the variable form of the TFs were shown to be more beneficial for CNE than CGPANN. In five of the ten cases which used CNE, the variable TFs outperformed the non-variable TFs, compared to only two out of the ten cases for CGPANN. The spread of improvement between Gaussian TFs and the logistic TFs was roughly even; for the Gaussian TF three of the twenty cases found the variable form to be more beneficial whereas for the logistic TF this was four of the twenty cases.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LogVar</th>
<th>U-test</th>
<th>Effect Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ball Throwing</td>
<td>8.15</td>
<td>2.66E-7</td>
<td>0.799</td>
</tr>
<tr>
<td>Full Adder</td>
<td>15.96</td>
<td>4.07E-1</td>
<td>0.520</td>
</tr>
<tr>
<td>Monks Problem 1</td>
<td>26.24</td>
<td>2.18E-2</td>
<td>0.633</td>
</tr>
<tr>
<td>Monks Problem 1</td>
<td>41.99</td>
<td>9.58E-3</td>
<td>0.650</td>
</tr>
<tr>
<td>Two Spirals</td>
<td>66.26</td>
<td>6.76E-12</td>
<td>0.898</td>
</tr>
<tr>
<td>Proben1: Cancer</td>
<td>3.09</td>
<td>5.28E-12</td>
<td>0.900</td>
</tr>
<tr>
<td>Proben1: Cancer</td>
<td>3.53</td>
<td>2.44E-11</td>
<td>0.886</td>
</tr>
</tbody>
</table>

Table 12. Average fitness of ANNs using the variable Gaussian TF trained using CNE.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Logistic Var</th>
<th>U-test</th>
<th>Effect Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ball Throwing</td>
<td>6.21</td>
<td>6.79E-6</td>
<td>0.744</td>
</tr>
<tr>
<td>Full Adder</td>
<td>16.00</td>
<td>6.49E-3</td>
<td>0.570</td>
</tr>
<tr>
<td>Monks Problem 1</td>
<td>10.45</td>
<td>3.72E-1</td>
<td>0.552</td>
</tr>
<tr>
<td>Monks Problem 1</td>
<td>27.00</td>
<td>1.37E-1</td>
<td>0.586</td>
</tr>
<tr>
<td>Two Spirals</td>
<td>74.28</td>
<td>3.74E-16</td>
<td>0.970</td>
</tr>
<tr>
<td>Proben1: Cancer</td>
<td>3.89</td>
<td>2.94E-3</td>
<td>0.672</td>
</tr>
<tr>
<td>Proben1: Cancer</td>
<td>4.79</td>
<td>1.58E-4</td>
<td>0.718</td>
</tr>
</tbody>
</table>

Table 13. Average fitness of ANNs using the variable logistic TF trained using CNE.
Table 14. Average fitness of ANNs using the variable Gaussian TF trained using CGPANN.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Gaussian</th>
<th>Var U-test</th>
<th>Effect Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ball Throwing</td>
<td>7.62</td>
<td>2.21E-1</td>
<td>0.571</td>
</tr>
<tr>
<td>Full Adder</td>
<td>15.72</td>
<td>6.50E-2</td>
<td>0.586</td>
</tr>
<tr>
<td>Monks Problem 1 Train</td>
<td>15.26</td>
<td>8.21E-1</td>
<td>0.513</td>
</tr>
<tr>
<td>Monks Problem 1 Test</td>
<td>21.80</td>
<td>5.14E-1</td>
<td>0.538</td>
</tr>
<tr>
<td>Two Spirals</td>
<td>69.50</td>
<td>2.77E-3</td>
<td>0.673</td>
</tr>
<tr>
<td>Proben1: Cancer Train</td>
<td>2.48</td>
<td>5.01E-1</td>
<td>0.538</td>
</tr>
<tr>
<td>Proben1: Cancer Test</td>
<td>2.31</td>
<td>5.71E-2</td>
<td>0.608</td>
</tr>
</tbody>
</table>

Table 15. Average fitness of ANNs using the variable logistic TF trained using CGPANN.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Logistic</th>
<th>Var U-test</th>
<th>Effect Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ball Throwing</td>
<td>7.82</td>
<td>2.20E-7</td>
<td>0.766</td>
</tr>
<tr>
<td>Full Adder</td>
<td>15.74</td>
<td>7.76E-1</td>
<td>0.511</td>
</tr>
<tr>
<td>Monks Problem 1 Train</td>
<td>10.07</td>
<td>3.65E-2</td>
<td>0.621</td>
</tr>
<tr>
<td>Monks Problem 1 Test</td>
<td>17.26</td>
<td>1.70E-1</td>
<td>0.579</td>
</tr>
<tr>
<td>Two Spirals</td>
<td>75.60</td>
<td>2.37E-8</td>
<td>0.823</td>
</tr>
<tr>
<td>Proben1: Cancer Train</td>
<td>2.42</td>
<td>1.01E-1</td>
<td>0.592</td>
</tr>
<tr>
<td>Proben1: Cancer Test</td>
<td>2.28</td>
<td>2.77E-1</td>
<td>0.561</td>
</tr>
</tbody>
</table>

5 Discussion

The results presented for the first experiment demonstrate that the choice of TF has a large impact on the effectiveness of NE. This is an intuitive result as it is likely that particular TFs are more or less suited to given tasks; this accords with the ‘No Free Lunch’ theorem [36]. However, although intuitive, it is a significant result as a user is unlikely to know, in advance of training, which TFs are most suited for a given task. The user must therefore except possibly poor results, or repeat the learning process using a range of TFs.

Another interesting result from the first experiment is that, in the majority of cases, the Heaviside step function was found to be the most effective TF. The significantly more popular logistic function was found to be the most effective TF in only one case; comparing the training fitness values for the classification tasks. The step function was the original TF used by the McCulloch and Pitts neuron models [14]. The fact that the step function is incompatible with the back propagation algorithm, and is only suited to tasks with binary outputs, is the likely reason why other TFs have been favoured. Here, however, it has been shown that when using NE the Heaviside step function is still a suitable TF for modern day ANNs; provided the task is compatible with binary outputs.

The second experiment demonstrated that allowing NE to select the TF of each neuron provided a better training method than the average of using homogeneous ANNs of each individual TF. This is significant because, as the first experiment shows, selecting the wrong TF for homogeneous networks has a large impact on the effectiveness of the final ANN. This coupled with the fact there is no way of knowing which TF will be most suited for a given task before training begins, puts homogeneous ANN at a strong disadvantage. This is an important result for NE as the addition of a gene describing the TF used by each neuron is probably compatible with all NE methods. The result may even be strengthened by the inclusion of other TFs not considered here. Also, as NE places no restrictions on the types of TFs used, the range of possible TF is limitless.

A further result from the second experiment concerns the percentage of neurons which used each type of TF in the evolved heterogeneous ANNs. Interestingly, it was never the case that one type of TF strongly dominated the network; which would have indicated that it was the TF evolution found most suitable toward the given task. There was, however, reasonable variation in the percentages of each type of TF used; showing that evolution was providing some form of pressure to use a particular type of TF i.e. it was not simply random. It could also be the case that many of the neurons which used a particular type of TF also used connection weights small in magnitude. For instance, if many of the neurons which used the logistic function also used connections weights which approached zero, then it would be clear that evolution had not found the logistic function useful towards the given task. However the magnitude of the connection weight were not considered here.

The third experiment demonstrated that, in the majority of cases, using NE to optimise parameters associated with each neuron provided either a better training method or had no significant effect compared to using non parametrised TFs. This is also an important result as the the inclusion of an additional gene (or genes) which alter the characteristics of each neuron’s TF is again probably compatible with all NE methods.

A further result from the third experiment was that CNE benefited more from variable TFs than CGPANN. This is interesting as the main difference between CNE and CGPANN is that CGPANN can evolve topology as well as connection weights. This could indicate that there is some form of interaction between evolving topology and evolving TFs. This interaction is likely highly complex.

This paper used a very limited method for allowing NE to optimise parameters associated with each neuron; each neuron had a single parameter and the parameter value was limited to 1, 2 or 3. It does however demonstrate the concept of optimising individual parameters for each neuron and, even in this this simple form, showed it to be advantageous. Further research should therefore allow for more complex TFs described by multiple parameters and placing fewer constraints on the values each parameter can take; such as in [2]. Additional further research could involve a combination of heterogeneous ANN where each neuron also has parameters to be optimised.

6 Conclusion

The use of NE to optimise the weights and topology of ANNs is well established and offers a number of advantages over traditional training methods; such as back propagation. However, the use of NE to optimise the TFs employed by each neuron has been so far under utilised. This paper has demonstrated the use of two, non mutually exclusive, methods for allowing NE to optimise each neuron’s TF. That is, selecting each neuron’s TF from a predetermined list of TFs or by optimising parameters associated with each neuron. This paper has also shown that the effectiveness of using NE to train homogeneous ANNs is highly dependent on the selected TF. Using NE to optimise each neuron’s TF has been empirically demonstrated to alleviate this issue.

The significance of the results presented in this paper are heightened by the fact that all NE methods are probably compatible with the two methods described. That is many NE method could benefit from evolving heterogeneous ANNs.

REFERENCES


