Noname manuscript No.
(will be inserted by the editor)

Horizontal Gene Transfer for Recombining Graphs

Timothy Atkinson · Detlef Plump · Susan Stepney

Accepted: 6 January 2020

Abstract We introduce a form of neutral Horizontal Gene Transfer (HGT) to Evolving Graphs by Graph Programming (EGGP). We introduce the $\mu \times \lambda$ Evolutionary Algorithm (EA), where μ parents each produce λ children who compete only with their parents. HGT events then copy the entire active component of one surviving parent into the inactive component of another parent, exchanging genetic information without reproduction. Experimental results from symbolic regression problems show that the introduction of the $\mu \times \lambda$ EA and HGT events improve the performance of EGGP. Comparisons with Genetic Programming (GP) and Cartesian Genetic Programming (CGP) strongly favour our proposed approach. We also investigate the effect of using HGT events in neuroevolution tasks. We again find that the introduction of HGT improves the performance of EGGP, demonstrating that HGT is an effective cross-domain mechanism for recombining graphs.

Keywords Graph-based Genetic Programming · Neuroevolution · Horizontal Gene Transfer

1 Introduction

Recombination of genetic material is commonly viewed as a key component of a successful GP (Genetic Programming) system. Koza [21] recommends that most offspring be produced by crossover, rather than by asexual reproduction and mutation. In contrast, CGP (Cartesian Genetic Programming) [24] traditionally uses the elitist $1 + \lambda$ evolutionary strategy, where all offspring are produced by asexual reproduction and mutation; variation and the ability to

Department of Computer Science
University of York
York
United Kingdom
E-mail: tja511@york.ac.uk, detlef.plump@york.ac.uk, susan.stepney@york.ac.uk

leave local optima are a byproduct of neutral drift in the neutral parts of the genome [25].

EGGP (Evolving Graphs by Graph Programming) [1] is a recently introduced graph-based GP approach that operates directly on graph-structured individuals, rather through some 'cartesian' grid encoding as used in CGP and PDGP (Parallel Distributed Genetic Programming) [29]. Each EGGP individual (graph) has an 'active' component that contributes directly to the fitness, and a 'neutral' component that can drift without affecting the fitness. Like CGP, existing work on EGGP has used only asexual reproduction and mutation. Here we extend EGGP to incorporate Horizontal Gene Transfer (HGT) 'events', introduced in [3], where the genetic information of one parent is shared with another. Our system operates using the elitist '$\mu \times \lambda$' EA, such that in each generation there are μ parents, which each produce λ children, which compete only with their own parent. This is effectively μ parallel $1 + \lambda$ EAs, with genetic information shared horizontally between elite individuals. To avoid disrupting elitism (by modifying the active components of individuals) or sharing junk (by copying neutral components of individuals), we copy only the active components of one parent onto the neutral component of another; it may later be activated through mutation. The work that we present is an extension to the concepts and experiments presented in [3].

EGGP's individuals, represented as (non-encoded) graphs, are directly modified through the probabilistic graph programming language P-GP 2 [2]. This direct approach eases the conception and implementation of graph-based operations. For example, using edge mutations that consider all possibilities that preserve acyclicty, rather than only those possibilities that preserve the ordering of some Cartesian grid, has been shown to offer faster convergence for standard digital circuit benchmark problems [1]. Additionally, it is possible to incorporate domain specific knowledge, such as Semantic Neutral Drift [4], where logical equivalence laws are applied directly to individuals to create neutral drift in their active components.

Here we replace neutral components with new material directly. This is inspired by Horizontal Gene Transfer (or Lateral Gene Transfer) found in nature. HGT is the movement of genetic material between individuals without mating, and is distinct from normal 'vertical' movement from parents to offspring [18]. HGT plays a key role in the spread of anti-microbial resistance in bacteria [13] and evidence has been found of plant-plant HGT [43] and plant-animal HGT [30]. The mechanism of HGT in transferring a segment of DNA into another individual's DNA may have a clear analogy when considering bit-string based Genetic Algorithms such as the Microbial GA [14], the equivalent analogy is not as obvious when dealing with graphs. Hence we use the term metaphorically: when we refer to HGT, we mean the movement of genetic material between individuals without mating. This is the new mechanism we present in this work.

Our approach is not the first work to recombine and share genetic information in graph-like programs. PDGP uses Subgraph Active-Active Node (SAAN) crossover [29] to share material within a population of Cartesian grid-

based programs. A number of crossover operators have been used in CGP, including uniform crossover [23], arithmetic crossover on a vector representation [5], and subgraph crossover [17]. Empirical comparison [15] shows that these crossover operators do not always aid performance, and that CGP with mutation only can sometimes be the best performing approach. Current advice [24] is that the 'standard' CGP approach remains to use mutation only. Our recombination features no modification of active components and does not produce children; nevertheless HGT events followed by edge mutations may perform operations very similar to PDGP SAAN crossover [29] and CGP subgraph crossover [17]. However, our precise mechanism, where active components are pasted into neutral components without any limitations to accessibility, does not obviously translate to PDGP and CGP, which are limited to Cartesian grids. Further, existing graph-based GP crossover techniques focus on the direct recombination of active sub-graphs or the recombination of the entire graph representation whereas our proposed approach offers indirect recombination of genetic material via the inactive components of the graph representation.

The rest of this work is organised as follows. In Section 2 we introduce EGGP with a new feature: depth control. In Section 3 we describe our Horizontal Gene Transfer approach, and the $\mu \times \lambda$ EA. In Section 4 we describe experimental settings for comparing our HGT approach to the existing EGGP approach, and to CGP and GP on symbolic regression problems. In Section 5 we present the results of our symbolic regression experiments. In Section 6 we describe experimental settings for the study of the HGT approach on neuroevolution problems. In Section 7 we present the results of our neuroevolution experiments. Finally, we conclude in Section 8 and describe directions for future work.

2 Evolving Graphs by Graph Programming (EGGP)

EGGP is a graph-based GP approach where individuals are represented directly as graphs, rather than through some encoding, and are manipulated through graph programming [2]. In this Section, we describe the EGGP approach including details of its representation, initialisation, mutation operators and a new extension, depth control. To distinguish between the original EGGP [1] and EGGP with depth control, we call the former EGGP and the latter EGGP_{DC}.

2.1 Representation

In EGGP an individual is a graph. The graph contains indexed input and output nodes, each corresponding to a particular input or output of a given problem. All other nodes are function nodes associated with functions from a chosen function set F. If a node v is associated with function $f \in F$ and

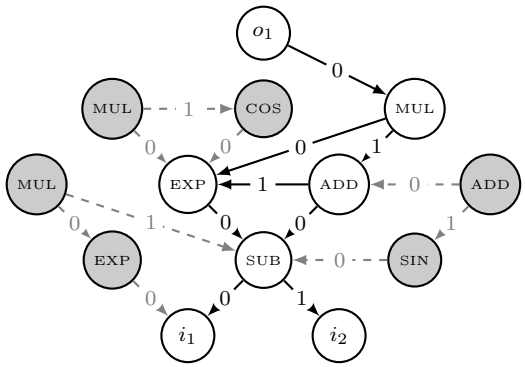


Fig. 1: An example EGGP individual. The single output computes the function $o_1 = e^{(i_1 - i_2)} \times \big(\big(i_1 - i_2\big) + e^{(i_1 - i_2)}\big)$

the arity of f is $a\big(f\big)$, then v has exactly $a\big(f\big)$ *outgoing* edges, which indicate the inputs that that function node takes. These outgoing edges are ordered; each edge is labelled with an integer to indicate its position in the order. Ordering removes ambiguity when dealing with non-commutative functions such as division and subtraction. Output nodes have exactly one outgoing edge, indicating that the function computed for that output is given by the behaviour of the node targeted by this single outgoing edge. Output nodes must have no incoming edges, as this would induce some undefined recurrent behaviour.

Here, the graph is restricted to be acyclic; this ensures that the evolved function is non-recursive. (This constraint could be relaxed to evolve recurrent programs, as in recurrent CGP [36].) An individual is therefore a DAG, with all output nodes as roots, and all input nodes as leaves. Input nodes can be both roots and leaves and other function nodes can also be roots; e.g. if they are targeted by no function nodes or outputs. Wherever there is no directed path from an output node to some node v, v and its outgoing edges are said to be 'neutral', as it does not contribute to the behaviour of the individual. EGGP can undergo 'neutral drift' on its 'neutral' components in a similar manner to CGP [25].

An example EGGP individual is given in Figure 1. There is a single output node, o_1, and two input nodes, i_1 and i_2. neutral nodes and their outgoing edges are coloured grey and dashed respectively; this is a visual aid only. Edge ordering starts at 0; the two outgoing edges of the active SUB node indicate that this node computes the function $i_1 - i_2$, rather than $i_2 - i_1$.

2.2 Initialisation

To generate an individual in EGGP, we begin by creating a graph with i input nodes corresponding to the i inputs associated with a given problem. The

parameter n describes the fixed number of function nodes in each individual solution. To generate these n nodes, we repeatedly pick some function f from the function set F and create a new node v_x associated with that function. We then insert edges connecting v_x to any existing node (chosen uniformly at random) until v_x's outdegree matches the function's arity $a(f)$. We repeat this process until there are n function nodes. When using depth control, the inserted edges may only target nodes that would not lead to the individual exceeding the specified maximum depth. Finally, we insert o output nodes corresponding to the o outputs associated with a given problem; each is then connected at random to any other (non output) node in the individual. This approach to initialisation guarantees the generation of an acyclic individual, and in the case of depth control, that generated individuals do not exceed the maximum depth.

2.3 Mutation

2.3.1 Node Mutation

Node mutation is performed by picking uniformly at random a function node to mutate, and changing that node's associated function to some other function chosen at random. Then two fix-up operations are performed.

Firstly, the outdegree (number of outgoing edges) of the mutated node is corrected to match the new function's arity. If the node's outdegree is greater than the new function's arity, edges are chosen uniformly at random and deleted until the outdegree and arity match. If the node's outdegree is less than the new function's arity, edges are inserted targeting valid nodes chosen uniformly at random. A valid node is a node that preserves acyclicity and maximum depth (see Section 2.4). For the original form of EGGP, preserving a maximum depth is not a consideration when choosing a node to target.

Secondly, we reorder the node's outgoing edges. We remove all ordering information from the node's outgoing edges, and assign a new valid random ordering. This process avoids bias in non-commutative functions; for example a node computing $x + y$ can be mutated to compute $x - y$ or $y - x$.

2.3.2 Edge Mutation

Edge mutation is performed by picking uniformly at random an edge to mutate. We then identify all valid targets for that edge (those nodes which preserve acyclicity and maximum depth, excluding the edge's existing target), and redirect the edge to target one of these nodes chosen uniformly at random. In the original form of EGGP, preserving a maximum depth is not a consideration when choosing a node to target.

2.3.3 Mutation Rate

The mutation rate of an individual is m_r. Certain mutations may prevent other mutations. For example, mutating one edge to target some node may then prevent other mutations of that node's outdoing edges with respect to preserving acyclicity. Therefore, iterating through the individual and considering each node or edge in turn for mutation may introduce bias. So our point mutations first choose a random point to mutate, and then mutate it.

We calculate the number of node or edge mutations to apply based on binomial distributions. For an individual with v_f function nodes and e edges, with mutation rate m_r, we sample a number of node mutations $m_v \in \mathcal{B}(v_f, m_r)$ and edge mutations $m_e \in \mathcal{B}(e, m_r)$, where $\mathcal{B}(n, p)$ indicates a binomial distribution with n trials and p probability of success. We then place all $m_v + m_e$ mutations in a list, and shuffle the list, applying mutations in a random order. While this approach is likely to have some biases, it guarantees reproducible probabilistic behaviour. The overall expected number of mutations is $m_r(v_f + e)$.

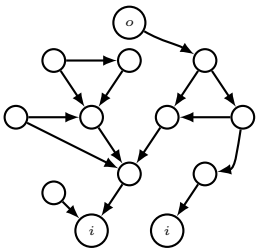
2.4 Depth Control

Here we introduce the notion of *depth control* to EGGP. This prevents mutations that would cause a child to exceed a given maximum depth D. We annotate individuals with information regarding the depth associated with each node. The 'depth up' u (or 'depth down' d) of a node is the length of the longest path from that node to a root (or leaf) node. We label each node v with the values (u, d). An exception is made for output nodes, which have $u = -1$ as their outgoing edges are not considered part of the 'depth' of the individual.

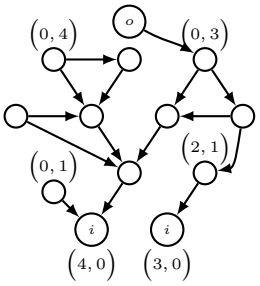
Once an individual has been annotated, we can identify pairs of nodes that, if an edge were inserted between them, would cause the individual to exceed the maximum depth D. If we wish to insert an outgoing edge for node v_1, then we eliminate any other node v_2 as a viable candidate on the following criteria: If the depth up value of v_1 is u_1, and the depth down value of v_2 is d_2, then it is impossible to insert an edge and preserve the maximum depth D if $u_1 + d_2 + 1 > d$: we have a path of length u_1 from v_1 to a root node, and a path of length d_2 from v_2 to a leaf node, hence the overall path from a root to a leaf would be $u_1 + d_2 + 1$, which exceeds D. If $u_1 + d_2 + 1 \leq d$, inserting an edge from v_1 to v_2 would preserve D.

We use this strategy in both edge mutation and node mutation. In edge mutation, we use annotations to identify invalid targets for the mutating edge. In node mutation, we use annotations to identify invalid targets for new edges to be inserted for the mutating node. We give an example of depth preserving edge mutation in Figure 2; an edge of an individual is mutated, but all possible targets that would break acyclicity or a maximum depth $D = 4$ are ignored.

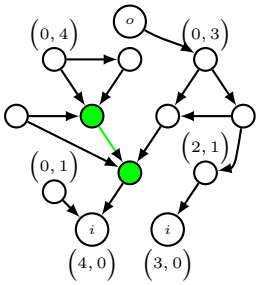
This individual is to undergo an edge mutation preserving acyclicity and a maximum depth $D = 4$.



(1) The individual is annotated with depth information. Each node has an associated 'depth up' value u indicating the length of the longest path to a root node (excl. outputs), and a 'depth down' value d indicating the length of the longest path to a leaf node. These are listed as a pair $\big(u, d\big)$ for each node.

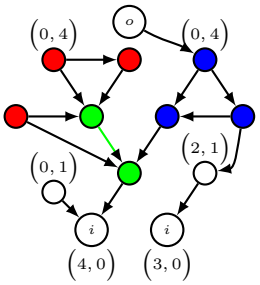


(2) An edge to mutate is chosen at random and marked (green) alongside its source node s and target node t.



x e

(3) Invalid candidate nodes for redirection are identified. If a node v has a directed path to s it is marked invalid (red), as targeting it would introduce a cycle. If the depth down value of a node v is d_v and the depth up value of s is u_s, when $u_s + d_v + 1 > D$, v is marked invalid (blue), as targeting it would exceed the maximum depth.



(4) The edge e (now shown in red) is mutated to target some randomly chosen unmarked (non-output) node, preserving acyclicity and maximum depth D. Finally, all annotations are removed.

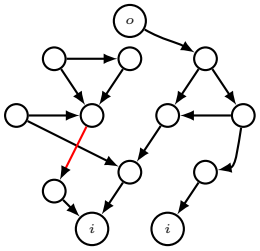


Fig. 2: An example of edge mutation preserving acyclicity and depth. Some annotations from step (1) are omitted for visual clarity.

3 Horizontal Gene Transfer in EGGP

In this Section we describe the introduction of Horizontal Gene Transfer events (HGT) to EGGP. HGT events involve the transfer of active material from a donor to the neutral region of a recipient (Section 3.1). To accommodate the need for multiple surviving individuals, we introduce the $\mu \times \lambda$ EA (Section 3.2) as an alternative to the $1 + \lambda$ algorithm previously used in EGGP.

3.1 Active-Neutral Transfer

HGT involves the movement of genetic material between individuals of a population without reproduction. Given a population P, we choose a donor and recipient individual. We copy the entire active component of the donor (excluding output nodes); we remove sufficient neutral material at random from the recipient to fit this active component within the fixed representation size. The copied active component is inserted into the recipient's neutral component, where it remains neutral until it is activated by some mutation. This type of HGT, which we refer to as 'Active-Neutral Transfer', is guaranteed to preserve the fitness of both the donor and recipient, preventing it from disrupting the elitism of the EA. The intention is to promote the production of higher quality offspring by the recipient, by activating its received genetic material through mutation. This process is mutually beneficial; the donor has a mechanism for propagating its genes, while the recipient stands to improve the survivability of its own offspring.

Once material has been transferred, there are a number of possible consequences: the neutral donor material can drift, or become active, through mutation. In this way it is possible for processes such as SAAN crossover in PDGP [29] or block based crossover in CGP [17] to arise out of Active-Neutral transfer followed by mutation.

Our strategy for choosing a donor and recipient is as follows. A recipient is first chosen based on a uniform distribution over the population P, excluding the best performing member. We refer to this 'best performing member' as the 'leader', which we exclude from receiving genetic material so that it can undergo neutral drift without any disruption. Throughout the evolutionary process, it is likely that the leader will change several times, meaning that the entire population is likely to receive genetic material at some point. Once a recipient is chosen, a donor is selected from the population excluding the recipient based on a roulette wheel. The donor may be the leader, allowing the leader to propagate its own genes to other members of the population. The use of a roulette wheel means that any individual can donate material, but the better performing individuals are more likely to do so.

We give an example of Active-Neutral transfer in Figure 3. The entire active component of a gene donor is copied into the neutral material of the recipient while maintaining the overall representation size.

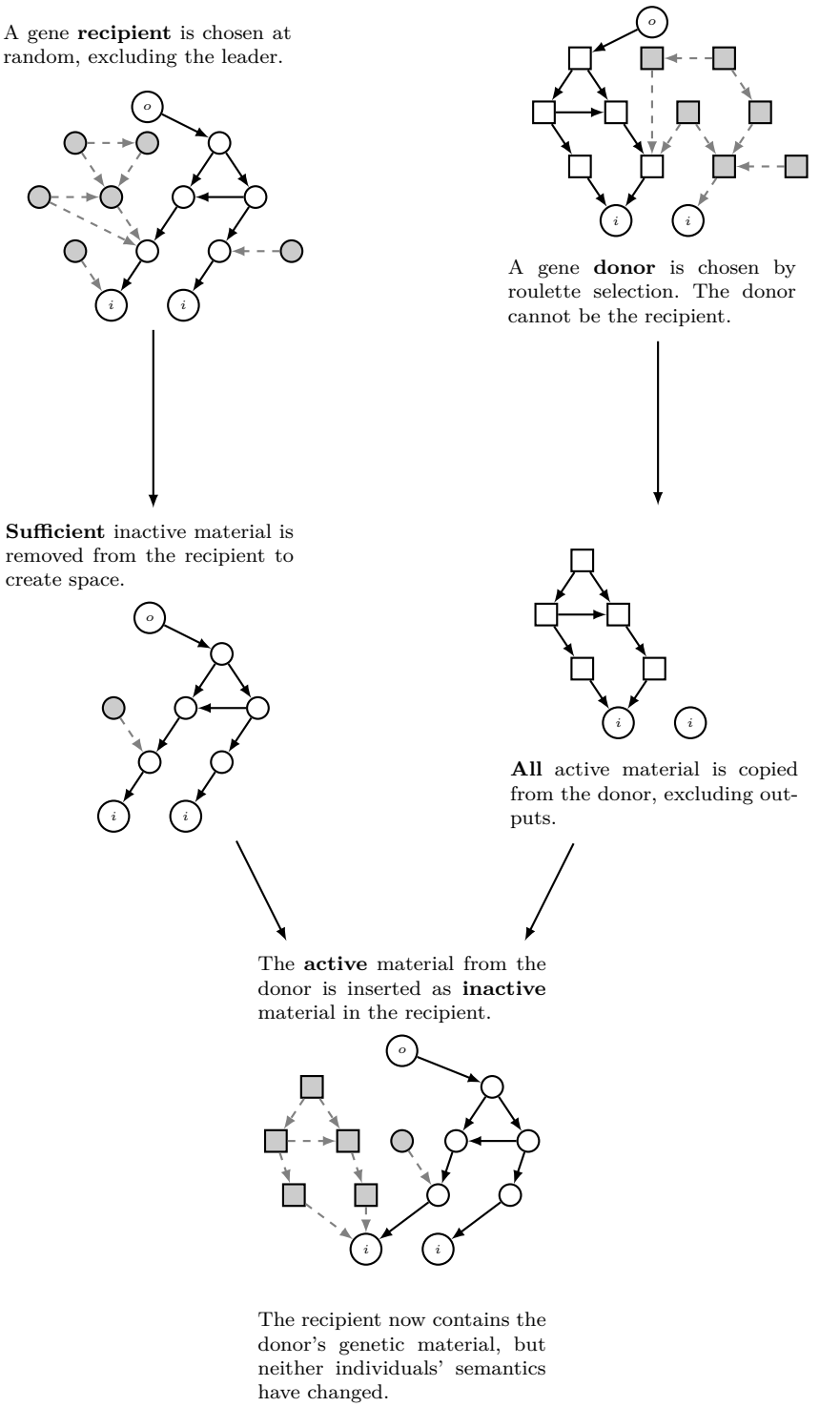


Fig. 3: An example of Active-Neutral transfer. The active material of a donor is copied into the neutral material of a recipient. Neither individuals' semantics is changed by this process. Grey nodes and dashed edges indicate the neutral material of individuals; they do not indicate any actual information stored on the individual. The donor's function nodes are shown as squares for clarity.

3.2 The $\mu \times \lambda$ EA

We cannot use Active-Neutral transfer with the $1 + \lambda$ algorithm except for sharing genetic material between the offspring; this is likely to be ineffective as direct offspring have much material in common. We therefore introduce the $\mu \times \lambda$ EA, a special case of the $\mu + \lambda$ algorithm. In each generation of the $\mu \times \lambda$ EA, there are μ parents. Each of the μ parents generates λ offspring, and compete for survival only with their own offspring. Without HGT, this effectively creates multiple parallel $1 + \lambda$ algorithms.

In each generation we perform a single Active-Neutral transfer operation with probability p_{HGT}. We then follow the procedure set out in section 3.1 by selecting a gene recipient from the μ parents (ignoring the best performing parent, the 'leader') and selecting a donor from the remaining $\mu - 1$ parents by roulette selection. We note that the $\mu \times \lambda$ EA clearly resembles search algorithms introduced in other work, such as CLONALG [7] and aiNet [6], albeit without the additional immune-system inspired concepts.

4 Symbolic Regression Experiments

Here we detail our experimental settings for benchmarking our HGT approach, EGGP_{HGT}, on various symbolic regression problems. We compare EGGP_{HGT} to: standard EGGP; the depth control variant EGGP_{DC}; the depth control variant using the $\mu \times \lambda$ algorithm (and no HGT), $\text{EGGP}_{\mu \times \lambda}$. These experiments allow us to test the following null hypotheses:

- H_1: there are no statistical differences when using the depth control variant EGGP_{DC} in comparison to standard EGGP.
- H_2: there are no statistical differences when using the $\mu \times \lambda$ algorithm for EGGP in comparison to the $1 + \lambda$ algorithm, with both approaches using depth control.
- H_3: there are no statistical differences when using the HGT approach for EGGP in comparison to using the $\mu \times \lambda$ algorithm without HGT, with both approaches using depth control.
- H_4: there are no statistical differences when using the HGT approach for EGGP in comparison to standard EGGP.

We test these null hypotheses for each benchmark problem. From these tests, we build an image of how the various features contribute to the performance of EGGP_{HGT}, and clarify whether the added HGT feature is truly improving performance by isolating it from the other new features.

We also compare our HGT approach to two other approaches from the literature. We compare to tree-based Genetic Programming (GP) [21] for a general measure of how our proposed approach compares to standard GP techniques. We also compare to CGP [24] as this is the most closely related graph-based GP technique to our work. These experiments allow us to test the following null hypotheses:

- H_5: there are no statistical differences when using EGGP$_{HGT}$ in comparison to GP.
- H_6: there are no statistical differences when using EGGP$_{HGT}$ in comparison to CGP.

Again, we test each of these null hypotheses for each benchmark problem. H_5 and H_6 allow us to measure the progress made by introducing HGT to EGGP in comparison to other approaches in literature.

4.1 Benchmark Problems

We benchmark the approaches on 21 synthetic symbolic regression problems [27]. That work justifies the exclusion of Grammatical Evolution (GE) [28], as it finds that GP generally outperforms GE on these problems. For all 21 problems, see [27]; one example is:

$$F_7(x_1, x_2) = \frac{(x_1 - 3)^4 + (x_2 - 3)^2 + (x_2 - 3)}{(x_2 - 2)^4 + 10} \tag{1}$$

These benchmarks were introduced in response to various criticisms of the GP community for 'arbitrarily' chosen benchmark problems, and the reasoning for their design is set out in detail in [27]. We view these problems as good measures of performance of a GP system. Of the 21 problems, 9 take 2 inputs, 1 takes 3 inputs, 8 take 5 inputs, 1 takes 6 inputs and 3 take 10 inputs. Each function's input variables are randomly sampled from the interval $[-5, 5]$.

We use 1000 training samples, 10,000 validation samples and 40,000 test samples. The training data is used to guide the different approaches, while every solution explored is evaluated on the validation data. The globally best performing individual (with respect to the validation data) is returned at the end of a run, and then evaluated on the test data to produce a test performance measure.

The function set for these problems is that of [27]:

$$\{+, -, \times, \div, e^x, \ln(x), \sin(x), \tanh(x), \sqrt{x}\} \tag{2}$$

Each approach has access to the 18 constants -0.9, -0.8, ..., $-0.1, 0.1, 0.2$, ..., 0.9. In GP these are constants, whereas in the EGGP variants and CGP, they are further input nodes.

4.2 Experimental Settings

We evaluate all individuals using the Mean Square Error (MSE) fitness function.We measure statistics taken over 100 independent runs of each approach on each dataset.

For all EGGP variants, we use a fixed 100 nodes and a mutation rate $m_r = 0.03$. For EGGP and EGGP$_{DC}$ we use the $1 + \lambda$ EA with $\lambda = 4$; for

$\text{EGGP}_{\mu\times\lambda}$ and EGGP_{HGT} we use $\mu = 3$ and $\lambda = 1$. This induces a 'minimal' version of the $\mu \times \lambda$ algorithm with $\mu = 3$ being the minimal value we could choose for μ such that HGT occurs not only from the 'leading' thread, but also between threads, and $\lambda = 1$ being the minimal value for λ. For EGGP_{DC}, $\text{EGGP}_{\mu\times\lambda}$ and EGGP_{HGT} we set a maximum depth of $D = 10$, and limit the maximum size to 50 active nodes. The maximum active size is ensured by removing and replacing any generated individual that exceeds the maximum size; it is necessary to prevent errors in the HGT approach where, for example, the size of the donor's active component exceeds that of the recipient's neutral component (causing the overall number of nodes to grow when copying the entire active component over). In practice, this condition is used in very few instances, as depth control constrains the size. The rate p_{HGT} is 0.5.

For CGP, we use the experimental parameters in [39], [24, Ch.3], at which values CGP outperforms GP on symbolic regression problems. We use 100 fixed nodes, and a mutation rate of 0.03. We use the $1 + \lambda$ EA with $\lambda = 4$. We do not use any of the published CGP crossover operators, as their usefulness, particularly on symbolic regression problems, remains disputed [15], and [37, 24] recommend the $1 + \lambda$ approach. We also use no form of depth control with CGP, as the approach is known to have inherent anti-bloat biases [34].

For GP, we use the experimental parameters in [27] with a minor adjustment. The population size is 500, with 1 elite individual surviving in each generation. Subtree crossover is used with a probability of 0.9, and when it is not used, the 'depth steady' subtree replacement mutation operator is used, which, when replacing a subtree of depth d generates a new subtree of depth between 0 and d [27]. Tournament selection is used to select reproducing individuals, with a tournament size of 4, and the maximum depth allowed of any individual is 10. Unusually for GP, we add each new individual to the population one-by-one, discarding one of the children produced by each crossover operator. This allows us to immediately replace invalid individuals with respect to the maximum depth, guaranteeing that every individual in a new population is valid and should be evaluated. To initialize the population, we use the ramped half-and-half technique [21], with a minimum depth of 1 and a maximum depth of 5.

For all experiments, the maximum number of evaluations allowed is 24 950, a value taken from [27] (50 generations with a population size of 500 and 1 elite individual that does not require re-evaluating). In GP this is achieved by allowing the search to run for 50 generations. In EGGP and CGP, we use the optimisation from [24, Ch.2], where individuals are evaluated only when their active components are mutated; there is no fixed number of mutations, and the search continues until the total number of evaluations is performed. There is no analogous optimisation for GP, as GP individuals contain no neutral material. This optimisation makes a large difference to the depth of search; for example, in CGP running on F_1, the median number of generations is 12 385, but if all individuals are evaluated (rather than only those with active region mutations), the number of generations would be capped at 6237 (assuming elite individuals are never re-evaluated).

4.3 Implementation

Our implementation of the EGGP variants described here is based upon the publicly available core EGGP implementation[1]. EGGP mutation operators and depth annotation are prototyped as P-GP 2 programs [2], then re-implemented in more efficient C code for the actual experiments. HGT events are implemented as P-GP 2 programs.

Our CGP experiments are based on the publicly available CGP library [37] with modifications made to accommodate the 'active evaluations only' optimisation and the use of validation and training sets. Our GP experiments are based on the DEAP evolutionary computation framework [9] with modifications made to accommodate our crossover strategy, mutation operator, and use of validation and training sets.

5 Results

	EGGP		**EGGP**$_{DC}$		**EGGP**$_{\mu\times\lambda}$		**EGGP**$_{HGT}$		**GP**		**CGP**	
F	MF	IQR	MF	IQR	MF	IQR	MF	IQR	MF	IQR	MF	IQR
F1	4.45E-3	7.35E-3	6.26E-3	6.45E-3	3.59E-3	1.39E-3	**2.47E-3**	1.79E-3	5.77E-3	3.40E-3	6.74E-3	4.30E-3
F2	8.17E6	6.05E6	1.41E7	9.95E6	8.06E6	5.02E6	**5.94E6**	3.06E6	1.28E7	7.86E6	1.73E7	2.54E6
F3	1.18E-2	7.34E-3	1.48E-2	4.27E-3	9.92E-3	3.82E-3	**7.22E-3**	4.00E-3	1.04E-2	3.56E-3	1.48E-2	4.39E-3
F4	2.58E13	1.05E9	**2.58E13**	3.57E8	2.58E13	7.51E10	2.58E13	1.96E9	3.55E13	8.35E13	2.58E13	2.35E9
F5	3.96E0	3.56E0	4.48E0	4.30E0	2.30E0	2.61E0	**6.90E-1**	2.08E0	5.13E0	3.81E0	7.17E0	1.47E0
F6	1.69E1	2.24E1	2.11E1	3.99E1	7.23E0	1.18E1	4.46E0	6.24E0	**2.61E0**	6.86E0	9.28E0	2.03E1
F7	3.06E2	7.40E2	4.16E2	6.76E2	2.20E2	1.53E2	**1.51E2**	9.62E1	4.20E2	3.50E2	5.76E2	4.39E2
F8	3.91E-2	7.43E-2	1.03E-1	1.13E-1	2.85E-2	2.00E-2	**2.19E-2**	1.21E-2	1.09E-1	4.99E-2	4.49E-2	9.59E-2
F9	7.09E2	5.40E3	2.59E3	1.36E4	1.81E2	3.68E2	1.57E2	3.53E2	**1.46E2**	3.04E1	1.71E2	1.11E3
F10	1.52E-1	2.05E-1	2.36E-1	2.22E-1	1.07E-1	8.30E-2	**7.69E-2**	5.75E-2	3.22E-1	5.62E-2	1.66E-1	1.42E-1
F11	3.93E1	7.26E1	4.53E1	6.33E1	2.43E1	1.37E1	**1.59E1**	1.20E1	3.88E1	3.37E1	4.96E1	4.73E1
F12	1.21E3	5.25E2	1.22E3	5.20E2	6.95E2	1.19E2	**6.83E2**	1.44E2	1.25E3	5.02E1	7.08E2	5.19E2
F18	4.07E4	9.27E3	4.08E4	3.91E4	4.40E3	3.86E4	**3.69E-1**	2.07E4	4.13E4	3.54E2	1.20E2	4.10E4
F21	1.07E0	6.16E-4	**1.07E0**	1.38E-5	1.07E0	7.74E-4	1.07E0	6.88E-4	1.07E0	4.90E-4	1.07E0	1.53E-5

Table 1: Results from Symbolic Regression benchmarks as described in Section 4. MF indicates the **M**edian **F**itness over observed runs; the lowest (best) MF result across all algorithms is highlighted in **bold**. IQR indicates the Interquartile range in fitness.

Our experimental results are given in Table 1. Results for benchmarks F13–17, 19, 20 (omitted here) show very little variety in performance; the results of [27] suggests these are poor benchmark problems in that the functions are almost invariant on their inputs. While F1–3 also exhibit relatively invariant

[1] https://github.com/UoYCS-plasma/EGGP

	H_1		H_2		H_3		H_4		H_5		H_6	
F	p	A	p	A	p	A	p	A	p	A	p	A
F1	0.08	-	$< \alpha$	**0.76**	$< \alpha$	**0.71**	$< \alpha$	**0.76**	$< \alpha$	**0.92**	$< \alpha$	**0.91**
F2	$< \alpha$	0.70	$< \alpha$	**0.76**	$< \alpha$	0.68	$< \alpha$	**0.71**	$< \alpha$	**0.87**	$< \alpha$	**0.95**
F3	$< \alpha$	0.68	$< \alpha$	**0.82**	$< \alpha$	0.70	$< \alpha$	**0.72**	$< \alpha$	**0.75**	$< \alpha$	**0.91**
F4	0.98	-	0.33	-	0.08	-	0.52	-	$< \alpha$	0.68	0.89	-
F5	0.06	-	$< \alpha$	**0.76**	$< \alpha$	0.70	$< \alpha$	**0.84**	$< \alpha$	**0.86**	$< \alpha$	**0.99**
F6	0.26	-	$< \alpha$	**0.78**	$< \alpha$	0.63	$< \alpha$	**0.84**	0.37	-	$< \alpha$	0.63
F7	0.12	-	$< \alpha$	**0.74**	$< \alpha$	**0.71**	$< \alpha$	**0.76**	$< \alpha$	**0.93**	$< \alpha$	**0.94**
F8	$\geq \alpha$	-	$< \alpha$	**0.75**	$< \alpha$	0.62	$< \alpha$	**0.77**	$< \alpha$	**0.95**	$< \alpha$	**0.79**
F9	0.02	-	$< \alpha$	**0.78**	0.77	-	$< \alpha$	0.69	0.23	-	0.17	-
F10	0.01	-	$< \alpha$	**0.74**	$< \alpha$	0.65	$< \alpha$	**0.76**	$< \alpha$	**0.99**	$< \alpha$	**0.81**
F11	0.57	-	$< \alpha$	**0.76**	$< \alpha$	**0.73**	$< \alpha$	**0.85**	$< \alpha$	**0.90**	$< \alpha$	**0.89**
F12	0.85	-	$< \alpha$	**0.76**	0.12	-	$< \alpha$	**0.81**	$< \alpha$	**0.89**	0.15	-
F18	0.84	-	$< \alpha$	**0.71**	$< \alpha$	0.68	$< \alpha$	**0.85**	$< \alpha$	**0.91**	$< \alpha$	0.62
F21	$\geq \alpha$	-	$< \alpha$	0.66	0.11	-	0.57	-	0.32	-	$< \alpha$	0.62

Table 2: Statistical tests for hypotheses H_1–H_6. The p value is from the two-tailed Mann-Whitney U test. The corrected threshold for statistical significance is $\alpha = \frac{0.05}{14}$. Where $p < \alpha$, the effect size from the Vargha-Delaney A test is shown; large effect sizes ($A > 0.71$) are shown in **bold**. Where $\alpha \leq p < 0.005$, p is listed as $\geq \alpha$.

responses, approaches here and in [27] produce a variety of performances that compel their inclusion. Similarly, while F4 and F21 do not show a variety of performances, the functions themselves produce a variety of responses on different inputs, again compelling their inclusion.

Table 1 lists the median fitness (MF) and inter-quartile range in fitness (IQR) of each approach on each dataset over 100 runs. Overall, the lowest MF score is achieved by EGGP$_{HGT}$ in 10 cases, EGGP$_{DC}$ in 2 cases and GP in 2 cases. There are no cases where EGGP, EGGP$_{\mu \times \lambda}$ or CGP achieve the lowest MF score.

To test for statistical significance we use the two-tailed Mann-Whitney U test [22], which (essentially) tests the null hypothesis that two distributions have the same medians (this non-parametric analogue of the t-test does not assume normally distributed data). We use a significance threshold of 0.05 and perform a Bonferroni correction for each hypothesis giving a corrected significance threshold of $\alpha = \frac{0.05}{14}$. Where we get a statistically significant result ($p < \alpha$), we also calculate the effect size, using the non-parametric Vargha-Delaney A Test [38]. $A \geq 0.71$ corresponds to a large effect size. These results of these statistical tests for all hypotheses are given in Table 2.

5.1 Building EGGP$_{HGT}$: H_1, H_2, H_3, H_4

The introduction of depth control (H_1) appears to have relatively little effect and is sometimes detrimental. In 12 of our benchmark problems, we observe no significant difference when introducing the feature. On 2 problems, standard EGGP achieves a statistically significant lower (better) median fitness than EGGP$_{DC}$, but never with large effect. These results indicate that depth control is not necessarily a helpful feature for EGGP, but never causes EGGP to outperform EGGP$_{DC}$ with large effect, and in many cases makes no significant difference to performance. This implies that the performance of EGGP$_{HGT}$ (discussed later) cannot be explained by its new depth control feature alone. We suggest that these results may be due to neutral material contributing to active nodes' 'depth up' values, preventing the active component from undergoing certain mutations even if these mutations would produce an active component of a valid depth. There may be circumstances where this restriction of the landscape hinders the performance of EGGP$_{DC}$.

Comparing EGGP$_{\mu\times\lambda}$ and EGGP$_{DC}$ (H_2) we find that the introduction of the $\mu \times \lambda$ algorithm yields a statistically significant lower median fitness and a large effect size on 12 of the 14 problems. On 1 problem (F4) there is no significant difference, and on 1 problem (F21) EGGP$_{DC}$ achieves a statistically significant lower median fitness, but without large effect. Overall, our study of H_2 provides substantial evidence that the $\mu\times\lambda$ algorithm aids the performance of EGGP, and should potentially be adopted generally.

The differences between EGGP$_{HGT}$ and EGGP$_{\mu\times\lambda}$ (H_3) are more subtle than the comparison of H_2, but there is a prevalent trend. The introduction of HGT yields a statistically significant lower median fitness in 10 problems, 3 of which occur with large effect, and no significant differences on the other 4. These results suggest that HGT is, generally, a beneficial feature capable of yielding major differences in performance. We observe no instances where HGT leads to a significant decrease in performance.

Overall, the results from studying our hypotheses H_1, H_2 and H_3 allow us to explain the success of EGGP$_{HGT}$ in comparison to GP and CGP (discussed in Section 5.2) as a composition of the core EGGP approach, the use of the $\mu \times \lambda$ EA and the introduction of Active-Neutral HGT events. Each of our 3 new features has been added to our overall approach in isolation, allowing us to isolate the beneficial properties of $\mu \times \lambda$ and HGT events. The role of depth control remains unclear from our investigations; it appears to be an unhelpful feature alone but may interact with the HGT process with respect to maintaining smaller individuals. An extended investigation into the role of depth control in our designed approach is desirable in the future.

H_4 compares our final proposed approach EGGP$_{HGT}$ to our original EGGP approach. The proposed approach achieves a statistically significant lower median fitness in 12 of the 14 problems; 11 of which occur with large effect. On the 2 remaining problems, we observe no significant differences. Therefore the combination of our 3 features – depth control, $\mu \times \lambda$ and horizontal gene

transfer – lead to a marked improvement over standard EGGP for the studied problems.

5.2 EGGP$_{HGT}$ vs. GP & CGP: H_4, H_6

EGGP$_{HGT}$ achieves a statistically significant lower median fitness in comparison to GP (H_5) on 11 problems, 10 of which show a large effect. On the other 3 problems, we observe no statistical differences. On a clear majority of the studied problems, EGGP$_{HGT}$ significantly outperforms a standard GP system, and is never outperformed by that GP system.

EGGP$_{HGT}$ achieves a statistically significant lower median fitness in comparison to CGP (H_6) on 11 problems, 9 of which show a large effect. On 3 of the other 4 problems, there is no significant difference, and on only one problem (F21) is there a statistical difference favouring CGP, but without large effect. Hence we have EGGP$_{HGT}$ significantly outperforming CGP under similar conditions on a majority of benchmark problems, and only outperformed on one problem.

Collectively, these results place EGGP$_{HGT}$ favourably in comparison to the literature. Although our experiments are not exhaustive – they are not the product of full parameter sweeps, but rather are testing approaches under standard conditions – they demonstrate that EGGP with Horizontal Gene Transfer is a viable and competitive approach for symbolic regression problems.

6 Neuroevolution Experiments

Here we evaluate the HGT mechanism for a very different class of graphs: Artificial Neural Networks (ANNs). With small modifications, our EGGP system and our HGT mechanism together form a neuroevolution system.

There are a number of significant differences between the types of graphs we are studying in this section and those of the previous symbolic regression experiments. The graphs seen in the previous experiments have a large number of nodes (100) and are relatively sparse (1–2 edges per node). In comparison the graphs in these neuroevolution experiments have fewer nodes (10) but are much more dense (10 edges per node). In Section 6.1 we explain the Pole Balancing Benchmark problems. In Section 6.2 we describe our representation of neural networks and genetic operators. In Section 6.3 we describe our experimental configuration.

6.1 Pole Balancing Benchmarks

Pole balancing problems have a long and extensive history of use as benchmarking problems for neural network training. The form of problem we use

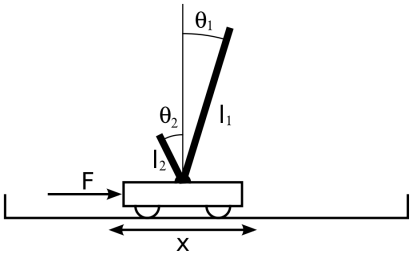


Fig. 4: Pole balancing simulations. Figure taken from [20]

here is described in detail in [40]. The main concept of a pole balancing problem is that there exists a cart upon which N poles are attached. The cart is restricted to moving left or right along a single dimension of a 2-dimensional plane, and its movements, alongside gravity, affect the angles of the poles with respect to the vertical. If any of the poles fall outside a certain angle from the vertical, or if the cart moves beyond a certain distance from its starting point, the simulation is considered a failure. The neural network being evaluated controls the cart by applying horizontal forces to it. This enables the network to accelerate the cart to the left or the right, thereby balancing the poles and keeping the cart within a given distance from its starting points. The equations of motion governing the dynamics of the N-pole pole balancing problem are as follows:

The displacement of the cart from the origin, 0, is x; we denote the cart's velocity and acceleration by $\dot{x}$ and $\ddot{x}$, respectively. The acceleration of the cart is given by

$$\ddot{x} = \frac{F - \mu_c \operatorname{sign}(\dot{x}) + \sum_{i=1}^{N} \tilde{F}_i}{M + \sum_{i=1}^{N} \tilde{m}_i} \tag{3}$$

where $\tilde{F}_i$ is the effective force associated with the ith pole, given by

$$\tilde{F}_i = m_i l_i \dot{\theta}_i^2 \sin\theta_i + \frac{3}{4} m_i \cos\theta_i \left(\frac{\mu_{pi} \dot{\theta}_i}{m_i l_i} + g \sin\theta_i \right) \tag{4}$$

and $\tilde{m}_i$ is the effective mass associated with the ith pole, given by

$$\tilde{m}_i = m_i \left(1 - \frac{3}{4} \cos^2\theta_i \right) \tag{5}$$

where $i = 1, \ldots, N$.

Once the cart's acceleration, $\ddot{x}$, has been calculated, it is possible to calculate the angular acceleration of the ith pole. We denote the angle of each pole by θ_i, measured in radians, with 0 being vertical. Then $\dot{\theta}_i$ is the angular velocity of the ith pole; $\ddot{\theta}_i$ is the angular acceleration of the ith pole, given by

$$\ddot{\theta}_i = \frac{-3}{4 l_i} \left(\ddot{x} \cos\theta_i + g \sin\theta_i + \frac{\mu_{pi} \dot{\theta}_i}{m_i l_i} \right) \tag{6}$$

Symbol	Units	Description
x	m	Horizontal displacement of the cart from 0.
$\dot{x}$	$\mathrm{m\,s^{-1}}$	Velocity of the cart.
$\ddot{x}$	$\mathrm{m\,s^{-2}}$	Acceleration of the cart.
θ_i	rad	Angle of the ith pole from vertical.
$\dot{\theta}_i$	$\mathrm{rad\,s^{-1}}$	Angular velocity of the ith pole.
$\ddot{\theta}_i$	$\mathrm{rad\,s^{-2}}$	Angular acceleration of the ith pole.
F	N	The force applied to the cart by the controller.

Table 3: Variables used in pole balancing experiments.

Symbol	Value	Description
μ_c	5×10^{-4}	Friction between the cart and the track.
μ_{pi}	$\mu_{p1} = \mu_{p2} = 2 \times 10^{-6}$	Friction between the ith pole and the cart.
M	1.0 kg	Mass of the cart.
m_i	$m_1 = 0.1$ kg, $m_2 = 0.01$ kg	Mass of the ith pole.
l_i	$l_1 = 0.5$ m, $l_2 = 0.05$ m	Length of the ith pole.
g	$-9.81\ \mathrm{m\,s^{-2}}$	Acceleration due to gravity.

Table 4: Constants used in pole balancing experiments. These values are taken from [12] (who do not provide units for the friction constants, see [8]).

In our experiments we consider 2-pole problems, so $N = 2$. Variables used in these equations are listed in Table 3. Constants used in these equations are listed in Table 4 In general, we take constant values from [12].

The initial configuration and simulation of the system is taken from [12]. This is done to maximise the strength of our comparisons with other approaches from the literature; a number of techniques are evaluated on these tasks in [12]. The initial state of the system is

$$x = 0, \quad \dot{x} = 0, \quad \theta_1 = \frac{4\pi}{180}, \quad \dot{\theta}_1 = 0, \quad \theta_2 = 0, \quad \dot{\theta}_2 = 0 \tag{7}$$

The cart starts in the centre of the track, with the longer pole p_1 four degrees from vertical, and the shorter pole p_2 inline with the vertical. The limits, beyond which a simulation ends, are that displacement x is bounded to the range $[-2.4, 2.4]$ m, and that the pole angles, θ_1 and θ_2, are both bounded to the ranges $[\frac{-36\pi}{180}, \frac{36\pi}{180}]$ radians, that is, they must stay within 36 degrees from the vertical. The system is simulated using the 4th order Runge–Kutta approximation and a time-step of 0.1 s. The neural network is updated every 2 time steps, and its output is scaled to the range $[-10, 10]$ N, which is then used as the force F applied to the cart. A solution is considered successful if it is able to keep both poles upright, and the cart within the bounds of the track, for $100,000$ simulated time-steps. Otherwise, the fitness assigned to a network is equal to $100,000$ minus the number of time steps the network was able to keep the poles upright and the cart within the track. We are therefore

minimising the fitness value, and the evolutionary run successfully terminates once we find a network with a fitness of 0.

In our experiments, we study 2 problems; Markovian and non-Markovian. In the Markovian case, the network is presented with the full state of the system, with 6 input variables made up of the position and velocity of the cart and the angles and angular velocities of both poles. In the non-Markovian case, the network is presented with only the position of the cart and the angles of both poles. The latter problem is generally believed to be more difficult as it requires the network to internally account for the velocities of the cart and the poles based on observations. We rescale these values to present to the neural network, by dividing x by 1.2, $\dot{x}$ by 1.5, each θ_i by $\frac{36\pi}{180}$ and each $\dot{\theta}_i$ by $\frac{115\pi}{180}$.

6.2 Representation and Genetic Operators

The graphs we study in our neurevolution are similar in behaviour to the graphs conventionally used with EGGP, but there are 2 significant differences. Firstly, their edges are also labelled with weights, which here are represented as integers and converted to rationals by dividing by 1000. Secondly, their edges may be *recurrent*, as indicated by an additional binary component of their labels, and may therefore target any (non-output) node in the graph. A recurrent edge accesses a node's previously computed value, thereby allowing our graphs to be stateful and have memory.

Our topological operators are the same as those used in [3] with minor modifications. We use edge mutation, which may produce recurrent edges with probability p_{rec}. When a recurrent edge is produced, it may target any (non-output) node in the graph. When a non-recurrent edge is produced, the problem of respecting acyclicity is constrained to the subgraph induced by non-recurrent edges. We fix our nodes' functions to be the bi-sigmoidal activation function

$$\text{bisig}(x) = \frac{1 - e^{-x}}{1 + e^{-x}} \tag{8}$$

and therefore do not require function mutations. We do, however, require a new mutation operator to modify weights. This is implemented with a single-rule P-GP 2 program that matches an edge uniformly at random and rewrites its weight to a uniformly chosen value from the specified weight range. We can therefore distinguish between mutation rates; edge redirections may be applied according to a binomial distribution with edge mutation rate m_{re}, and weight mutations may be applied according to a binomial distribution with weight mutation rate m_{rw}.

For HGT to be viable we require that the number of active function nodes in solutions be at most half the total function nodes. If we initialise our relatively dense neural networks with recurrent connections using the previous approach (section 2), it may take exceptionally long to find a starting point that satisfies this constraint. We therefore modify that initialisation procedure for these experiments. With probability p_{rec}, recurrent edges are added

immediately after nodes are added, rather than after all nodes are added. This design decision ensures relatively small generated individuals, making our implementation more viable, but also prevents cycles from existing in the initial graphs. Cycles can be introduced throughout the evolutionary process via mutation.

6.3 Experimental Settings

We deliberately choose representation parameters that cause the graphs we study here to be topologically distinct from the graphs we have studied for symbolic regression in Section 4. By doing this, we further verify HGT as a cross-domain technique that is applicable in a variety of scenarios.

We use a fixed representation size of 10 nodes, with a maximum permitted number of active nodes of 5. Hence, in terms of the number of function nodes, the graphs we study here are much smaller than the 100-function node graphs we studied earlier. Each function node has an arity of 10, that is, there are 10 connections per neuron. Therefore the graphs we study here are significantly more dense, with respect to the number of edges, than the graphs we studied earlier where each function node had 1 or 2 outgoing edges. We are learning potentially cyclic graphs with recurrent edges, and set the probability of recurrent edges, $p_{rec} = 0.1$. In contrast, the graphs studied earlier were acyclic. Finally, our edges are associated with weights, with a weight range of $[-2.0, 2.0]$. In contrast, the edges we studied earlier did not feature edge weights. Overall, the graphs we study in these experiments are distinct from those studied in Section 4 in that they are much smaller, much more dense, may contain recurrent edges and cycles, and have edge weights.

In all experiments we again use the $\mu \times \lambda$ EA with $\mu = 3$ and $\lambda = 1$. Whenever we generate an individual that exceeds the permitted size of 5, we discard it and immediately generate a new one. We set the edge mutation rate $m_{re} = 0.05$, and the weight mutation rate $m_{rw} = 0.1$. We find that very occasional runs take a long time to terminate due to local optima. This is likely because of the small representation size that we have deliberately opted for, which allows for very little inactive material. To make our experiments computationally tractable while still having every evolutionary run terminate, we therefore introduce a restarting procedure; if an evolutionary run has not seen improvement in 1000 generations, its population is randomised.

We study 2 variants of EGGP:

1. EGGP$_{HGT}$ is the $\mu \times \lambda$ EA with HGT as described and $p_{HGT} = 1$.
2. EGGP$_{\mu\times\lambda}$ is simply the $\mu \times \lambda$ EA without HGT. This variant is used as a control for HGT.

We run each algorithm on each problem 200 times. These experiments allow us to test the null hypotheses that there are no statistical differences when using the HGT mechanism in comparison to the $\mu \times \lambda$ EA alone. We carry out statistical tests to test for significant differences introduced by the

	$\mathbf{EGGP}_{HGT}$		$\mathbf{EGGP}_{\mu\times\lambda}$			
Problem	ME	IQR	ME	IQR	p	A
Markovian	812	848	1,194	1,478	10^{-4}	0.61
Non-Markovian	6,230	8,928	10,577	17,074	10^{-6}	0.63

Table 5: Results from pole balancing benchmarks for EGGP_{HGT} and $\text{EGGP}_{\mu\times\lambda}$. The p value is from the two-tailed Mann–Whitney U test. The effect size A from the Vargha–Delaney A test is shown.

HGT mechanism on the studied problems. If our statistical tests reject the null hypothesis, and we see lower Median Evaluations (MEs) required for each problem when using HGT, then we can infer that the HGT mechanism is indeed improving performance for these neuroevolution tasks. We note that it is common in literature to also report the processing time required to solve each task [12]; we do not report these values as we are using these problems as a proxy for measuring the efficiency of search.

7 Neuroevolution Results

The results from our neuroevolution experiments are given in Table 5. For each problem and algorithm, we list the MEs and IQRs in evaluations. To test for statistical significance we use the two-tailed Mann–Whitney U test [22], which (essentially) tests the null hypothesis that two distributions have the same medians. We use a significance threshold of 0.05 and perform a Bonferroni procedure for each hypothesis giving a corrected significance threshold of $\alpha = \frac{0.05}{2}$. Where we get a statistically significant result ($p < \alpha$), we also calculate the effect size, using the non-parametric Vargha–Delaney A Test [38]. $A \geq 0.71$ corresponds to a large effect size.

As we can see in Table 5, on both problems we record lower MEs for EGGP_{HGT} in comparison to $\text{EGGP}_{\mu\times\lambda}$. Our Mann–Whitney U test reveals both results to be statistically significant ($p < \frac{0.05}{2}$), although without large effect. We give box-plots of the results of both problems in Figure 5, highlighting the degree to which HGT improves the efficiency of search. Taking into account the MEs and statistical significance, we can infer that HGT is indeed improving performance for these neuroevolution tasks. However, that we observe no large effect suggests that the change in MEs as a result of HGT is not large. This lack of large effect is in line with our statistical tests comparing EGGP_{HGT} and $\text{EGGP}_{\mu\times\lambda}$ in Section 5.

Empirical comparison with other neuroevolutions on these problems is a difficult task. When these problems have been studied in the literature, they have not been standardised in many respects. For example, some implementations use Euler integration [35,19], whereas others use Runge-Kutta integration [31,12]. In some cases the longer pole starts at 1 degree from vertical [35, 31] and in others it starts at 4 degrees from vertical [12]. Some publications

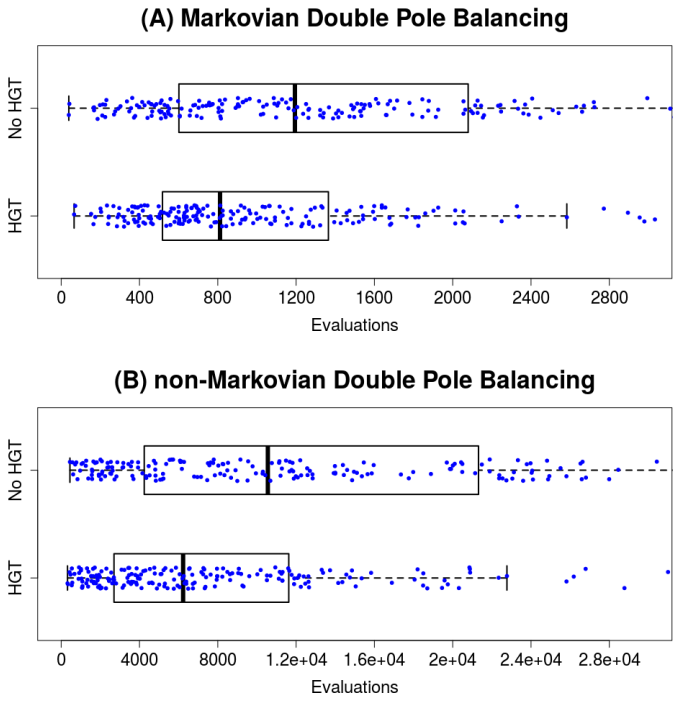


Fig. 5: Box-plots with data overlayed for both neuroevolution problems. We give results for EGGP$_{HGT}$ (HGT) and EGGP$_{\mu\times\lambda}$ (No HGT); (A) Markovian, (B) non-Markovian. Overlayed data is jittered for visual clarity.

use 'bang-bang' force (where the network outputs ± 10 N) [19], whereas others have networks output continuous force [35,12] as we have done. These distinctions, in combination with a general lack of publicly available implementations and that even standardising these conditions may unfairly bias against certain approaches and chosen parameters, make a conventional statistical comparison difficult. For a more detailed discussion of problems in drawing comparisons between methods on these tasks, see [33, Chapter 11].

However, the intention of our experiments is not to propose a state-of-the-art neuroevolution technique. Instead, we are investigating whether HGT works for graphs very different from those studied for symbolic regression. With this in mind, we list in Table 6 the MEs used by EGGP$_{HGT}$ in comparison to results reported in literature. While not a direct empirical comparison, this does give some notion of how the proposed algorithm compares. To this effect, results in [12] are helpful in that they have standardised comparisons over a number of approaches. In Table 6 we can see that EGGP$_{HGT}$ does quite well on Markovian pole balancing, outperforming a number of techniques and performing similarly to CGPANN [35], which used much larger representation and had the longer pole starting at 1 degree from the vertical. However,

Technique	Mean Evaluations	
	Markovian	Non-Markovian
CNE [12] [41]	22,100	76,906
SANE [12] [26]	12,600	262,700
RPG [12] [42]	4,981	5,649
ESP [12] [11]	3,800	7,374
NEAT [31]	3,600	20,918
NEVa[32]	2,177	-
EGGP$_{HGT}$	**1,175**	**8,891**
CGPANN[35]	1,111	-
CoSyNE[12]	954	1,249
CMA-ES[12] [16]	895	3,521

Table 6: MEs reported from various literature. Where a result is given, the publication it is taken from is referenced. A number of results are taken from comparative experiments in [12], in which case we also provide a reference for the approach after the reference to [12]. Results are ordered by MEs on Markovian pole balancing.

the non-Markovian results are less impressive, with EGGP$_{HGT}$ being outperformed by all but three techniques from the literature. We do take some reassurance from the fact that, on both problems, EGGP$_{HGT}$ outperforms the popular neuroevolution technique, 'Neuroevolution of Augmenting Topologies' (NEAT) [31].

The cause of the disparity between the two studied problems in comparison with the literature may be a result of our chosen parameters. We chose a recurrent edge rate of $p_{rec} = 0.1$, and it is generally believed that solutions to the non-Markovian problem are more dependent of memory than solutions to the Markovian problem. Therefore increasing p_{rec} and thereby increasing the amount of memory usage in the network may improve performance. Additionally, the non-Markovian problem is generally viewed as harder, and we may have hampered our search process by choosing such small, dense graphs. This may have reduced the evolvability of the system and the effect of this may be more prevalent on the harder problem, particularly if it has more local optima. Clearly, additional experiments with respect to parameterisation are required to establish the cause of this and improve EGGP$_{HGT}$'s performance on the non-Markovian task.

8 Conclusions and Future Work

In this work we have introduced a new and effective form of neutral HGT in the EGGP approach. Our approach utilises Active-Neutral transfer to copy the active components of one elite parent into the neutral material of another. Experimental results show that both HGT and the introduction of the $\mu \times \lambda$ EA lead to improvements in performance on benchmark symbolic regression

problems. Comparing the final approach, EGGP$_{HGT}$, to GP and CGP yields highly favourable results on a majority of problems.

We have also carried out neuroevolution experiments with HGT. Empirical comparisons on double pole balancing problems reveal that, for both Markovian and non-Markovian tasks, HGT improves the efficiency of search. This result is particularly interesting for two reasons. Firstly, we have evidence of positive effect of HGT for both symbolic regression and neuroevolution problems suggesting that this technique may function as a cross-domain recombination operator. Secondly, we deliberately chose to evolve very small, dense, graphs in our neuroevolution experiments to make the differences with our symbolic regression benchmarks more stark. That HGT remained beneficial reinforces the idea that it is useful.

These results have implications for broader research in EAs and GP. The reuse and recombination of genetic material is generally assumed to be a useful feature of an evolutionary system (e.g. GP crossover [21]), but our Active-Neutral HGT events achieve reuse without altering the active components of individuals. Hence our approach contributes evidence to the notion that neutral drift aids evolutionary search [10]. Active-Neutral HGT events move beyond neutrality through mutation; we are effectively biasing the neutral components of individuals towards areas of the landscape we know to be 'good' with respect to the fitness function. While this is empirically beneficial here, it remains unknown whether this neutral biasing is helpful outside of the EGGP approach. Our favourable comparisons with GP and CGP support this direction of thought; GP offers recombination without neutral drift, whereas (vanilla) CGP offers neutral drift without recombination.

Our work here opens up a number of avenues for further research. It is desirable to investigate the influence of population parameters μ, λ and the HGT rate p_{HGT} the performance of the described approach. Here, we have chosen small values of μ and λ and relatively high values of p_{HGT}; it is therefore interesting to consider whether larger values of μ and λ help or hinder the HGT process, and whether it is necessary to introduce multiple HGT events in a single generation when using larger populations. A possible way to investigate this could be through a graph equivalent of the Microbial Genetic Algorithm [14] as this could work as a minimal extension that supports the use of a larger population. Additionally, an investigation isolating depth control from HGT would help clarify whether HGT is more useful when individuals are smaller or larger.

There are two variants of HGT that should be investigated further. The first is a 'partial' HGT mechanism, where only a small subgraph of the active component of the donor is copied into the recipient. With such a mechanism, it would even be possible to take fragments of genetic material from several donors during a HGT event, thereby increasing the variance in the recipients received genetic material. However, empirical comparisons would certainly be necessary to clarify whether this is a preferable approach, and it is not yet clear how such a mechanism should be parameterised. Open questions are how should a subgraph be selected? how large should a subgraph be? how

many subgraphs should be copied into the recipient, and how many donors should they come from?

Another interesting variant of HGT is 'headless chicken' HGT where the donor is substituted with a randomly generated individual. In this case, we would be replacing neutral material with randomly generated material. An empirical comparison between this variant and standard HGT could reveal any side effects caused by the HGT mechanism; if the headless chicken mechanism is effective, then we may have to reconsider our explanations for the effectiveness of HGT. However, we doubt that the headless chicken mechanism would compete with or outperform HGT, particularly in the symbolic regression problems, as we already have a large degree of neutral material in the genotype which undergoes neutral drift thereby achieving a similar randomising effect.

Acknowledgments

T. Atkinson is supported by a Doctoral Training Grant from the Engineering and Physical Sciences Research Council (EPSRC) in the UK.

References

1. Atkinson, T., Plump, D., Stepney, S.: Evolving graphs by graph programming. In: Proc. European Conference on Genetic Programming, EuroGP 2018, *LNCS*, vol. 10781, pp. 35–51. Springer (2018)
2. Atkinson, T., Plump, D., Stepney, S.: Probabilistic graph programs for randomised and evolutionary algorithms. In: Proc. International Conference on Graph Transformation, ICGT 2018, *LNCS*, vol. 10887, pp. 63–78. Springer (2018)
3. Atkinson, T., Plump, D., Stepney, S.: Evolving graphs with horizontal gene transfer. In: Proc. Genetic and Evolutionary Computation Conference, GECCO 2019, pp. 968–976. ACM (2019). doi:10.1145/3321707.3321788
4. Atkinson, T., Plump, D., Stepney, S.: Evolving graphs with semantic neutral drift. Natural Computing (2019). URL `https://arxiv.org/abs/1810.10453`. (to appear)
5. Clegg, J., Walker, J.A., Miller, J.F.: A new crossover technique for cartesian genetic programming. In: Proc. 9th annual conference on Genetic and evolutionary computation, GECCO 2007, pp. 1580–1587. ACM (2007)
6. De Castro, L.N., Timmis, J.: An artificial immune network for multimodal function optimization. In: Proc. 2002 Congress on Evolutionary Computation. CEC'02, vol. 1, pp. 699–704. IEEE (2002)
7. De Castro, L.N., Von Zuben, F.J.: Learning and optimization using the clonal selection principle. IEEE transactions on evolutionary computation **6**(3), 239–251 (2002)
8. Florian, R.V.: Correct equations for the dynamics of the cart-pole system (2005). URL `https://coneural.org/florian/papers/05_cart_pole.pdf`
9. Fortin, F.A., Rainville, F.M.D., Gardner, M.A., Parizeau, M., Gagné, C.: Deap: Evolutionary algorithms made easy. Journal of Machine Learning Research **13**(Jul), 2171–2175 (2012)
10. Galván-López, E., Poli, R., Kattan, A., O'Neill, M., Brabazon, A.: Neutrality in evolutionary algorithms... what do we know? Evolving Systems **2**(3), 145—-163 (2011)
11. Gomez, F., Miikkulainen, R.: Incremental evolution of complex general behavior. Adaptive Behavior **5**(3-4), 317–342 (1997). doi:10.1177/105971239700500305
12. Gomez, F., Schmidhuber, J., Miikkulainen, R.: Accelerated neural evolution through cooperatively coevolved synapses. J. Machine Learning Research **9**(May), 937–965 (2008)

13. Gyles, C., Boerlin, P.: Horizontally transferred genetic elements and their role in pathogenesis of bacterial disease. Veterinary pathology **51**(2), 328–340 (2014)
14. Harvey, I.: The microbial genetic algorithm. In: European Conference on Artificial Life, ECAL 2009, *LNCS*, vol. 5778, pp. 126–133. Springer (2009)
15. Husa, J., Kalkreuth, R.: A comparative study on crossover in cartesian genetic programming. In: Proc. European Conference on Genetic Programming, EuroGP 2018, *LNCS*, vol. 10781, pp. 203–219. Springer (2018)
16. Igel, C.: Neuroevolution for reinforcement learning using evolution strategies. In: IEEE Congress on Evolutionary Computation, CEC 2003, vol. 4, pp. 2588–2595. IEEE, IEEE (2003). doi:10.1109/CEC.2003.1299414
17. Kalkreuth, R., Rudolph, G., Droschinsky, A.: A new subgraph crossover for cartesian genetic programming. In: Proc. European Conference on Genetic Programming, EuroGP 2017, *LNCS*, vol. 10196, pp. 294–310. Springer (2017)
18. Keeling, P.J., Palmer, J.D.: Horizontal gene transfer in eukaryotic evolution. Nature Reviews Genetics **9**(8), 605 (2008)
19. Khan, M.M., Khan, G.M., Miller, J.F.: Efficient representation of recurrent neural networks for markovian/non-markovian non-linear control problems. In: Proc. 10th International Conference on Intelligent Systems Design and Applications, (ISDA 2010), pp. 615–620 (2010). doi:10.1109/ISDA.2010.5687197. URL `https://doi.org/10.1109/ISDA.2010.5687197`
20. Koutnik, J., Gomez, F., Schmidhuber, J.: Evolving neural networks in compressed weight space. In: Proc. Genetic and Evolutionary Computation Conference, GECCO 2010, pp. 619–626. ACM (2010). doi:10.1145/1830483.1830596
21. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection, vol. 1. MIT Press (1992)
22. Mann, H.B., Whitney, D.R.: On a test of whether one of two random variables is stochastically larger than the other. Ann. Math. Statist. **18**(1), 50–60 (1947)
23. Miller, J.F.: An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In: Proc. 1st Annual Conference on Genetic and Evolutionary Computation, GECCO '99, vol. 2, pp. 1135–1142. Morgan Kaufmann Publishers Inc. (1999)
24. Miller, J.F.: Cartesian genetic programming. In: Cartesian Genetic Programming, pp. 17–34. Springer (2011)
25. Miller, J.F., Smith, S.L.: Redundancy and computational efficiency in cartesian genetic programming. IEEE Transactions on Evolutionary Computation **10**(2), 167–174 (2006)
26. Moriarty, D.E.: Symbiotic evolution of neural networks in sequential decision tasks. Ph.D. thesis, University of Texas at Austin USA (1997)
27. Nicolau, M., Agapitos, A., O'Neill, M., Brabazon, A.: Guidelines for defining benchmark problems in genetic programming. In: 2015 IEEE Congress on Evolutionary Computation, CEC, pp. 1152–1159 (2015)
28. O'Neill, M., Ryan, C.: Grammatical evolution. IEEE Transactions on Evolutionary Computation **5**(4), 349–358 (2001)
29. Poli, R.: Evolution of graph-like programs with parallel distributed genetic programming. In: Proc. International Conference on Genetic Algorithms, ICGA, pp. 346–353. Morgan Kaufmann (1997)
30. Schwartz, J.A., Curtis, N.E., Pierce, S.K.: Fish labeling reveals a horizontally transferred algal (vaucheria litorea) nuclear gene on a sea slug (elysia chlorotica) chromosome. The Biological Bulletin **227**(3), 300–312 (2014)
31. Stanley, K.O.: Efficient evolution of neural networks through complexification. Ph.D. thesis, The University of Texas at Austin (2004). URL `http://nn.cs.utexas.edu/?stanley:phd2004`
32. Tsoy, Y., Spitsyn, V.: Using genetic algorithm with adaptive mutation mechanism for neural networks design and training. In: Proce. 9th Russian-Korean International Symposium on Science and Technology, KORUS 2005, pp. 709–714. IEEE (2005). doi:10.1109/KORUS.2005.1507882
33. Turner, A.: Evolving artificial neural networks using Cartesian Genetic Programming. Ph.D. thesis, University of York (2015). URL `http://etheses.whiterose.ac.uk/12035/1/thesis.pdf`

34. Turner, A., Miller, J.: Cartesian genetic programming: Why no bloat? In: Proc. European Conference on Genetic Programming, EuroGP 2014, *LNCS*, vol. 8599, pp. 222–233. Springer (2014)
35. Turner, A.J., Miller, J.F.: Cartesian Genetic Programming encoded artificial neural networks: a comparison using three benchmarks. In: Proc. Genetic and Evolutionary Computation Conference, GECCO 2013, pp. 1005–1012. ACM (2013). doi:10.1145/2463372.2463484
36. Turner, A.J., Miller, J.F.: Recurrent cartesian genetic programming. In: Proc. International Conference on Parallel Problem Solving from Nature, PPSN 2014, *LNCS*, vol. 8672, pp. 476–486. Springer (2014)
37. Turner, A.J., Miller, J.F.: Introducing a cross platform open source cartesian genetic programming library. Genetic Programming and Evolvable Machines **16**(1), 83–91 (2015)
38. Vargha, A., Delaney, H.D.: A critique and improvement of the CL common language effect size statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics **25**(2), 101–132 (2000)
39. Walker, J.A., Miller, J.F.: The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. IEEE Transactions on Evolutionary Computation **12**(4), 397–417 (2008)
40. Wieland, A.P.: Evolving controls for unstable systems. In: Connectionist Models, pp. 91–102. Elsevier (1991)
41. Wieland, A.P.: Evolving neural network controllers for unstable systems. In: Seattle International Joint Conference on Neural Networks, IJCNN 91, vol. 2, pp. 667–673. IEEE (1991). doi:10.1109/IJCNN.1991.155416
42. Wierstra, D., Foerster, A., Peters, J., Schmidhuber, J.: Solving deep memory POMDPs with recurrent policy gradients. In: Proc. Artificial Neural Networks, ICANN 2007, *LNCS*, vol. 4668, pp. 697–706. Springer (2007). doi:10.1007/978-3-540-74690-4_71
43. Yoshida, S., Maruyama, S., Nozaki, H., Shirasu, K.: Horizontal gene transfer by the parasitic plant striga hermonthica. Science **328**(5982), 1128–1128 (2010)