Tweaking a Tower of Blocks Leads to a TMBL: Pursuing Long Term Fitness Growth in Program Evolution

Tony E Lewis and George D Magoulas, *Member, IEEE*

***Abstract*— If a population of programs evolved not for a few hundred generations but for a few hundred thousand or more, could it generate more interesting behaviours and tackle more complex problems?**

We begin to investigate this question by introducing Tweaking Mutation Behaviour Learning (TMBL), a form of evolutionary computation designed to meet this challenge. Whereas Genetic Programming (GP) typically involves creating a large pool of initial solutions and then shuffling them (with crossover and mutation) over relatively few generations, TMBL focuses on the cumulative acquisition of small adaptive mutations over many generations. In particular, we aim to reduce limits on long term fitness growth by encouraging *tweaks*: changes which affect behaviour without ruining the existing functionality. We use this notion to construct a standard representation for TMBL. We then experimentally compare TMBL against linear GP and tree-based GP and find that TMBL shows strong signs of being more conducive to the long term growth of fitness.

I. Introduction

One of the lessons from evolutionary biology is the astounding amount of functional complexity that a cumulative process can build using nothing but small, unguided improvements and plenty of time.

Evolutionary Computation (EC) draws inspiration from evolution to create algorithms that can harness some of its benefits. Various forms of EC have been proposed for tackling different sorts of problems. A Genetic Algorithm (GA) entails evolving a string of characters. For this approach, the implementer must design some suitable interpretation of the string. For other application areas it may be unclear what structure of solution is required or it may be obvious that an algorithm or behaviour must be evolved. These problems may be better suited to Genetic Programming (GP) which entails evolving programs. The standard form of GP program is akin to an arithmetic formula but there are many other forms, some of which are Turing complete [13].

GA and GP share many similarities [16] and there is no clear cutoff between the two (for example Cartesian Genetic Programming (CGP) is a commonly used form of GP but it could be viewed as a GA with a particularly sophisticated fitness evaluation function). Still, it is helpful to use two different names because GA and GP have different aims and so suit different problem areas.

Although GP is often highly effective during the initial generations, it typically stagnates quickly. Despite the ever faster computation being delivered by processor technology, GP remains unsuitable for problems that can only be solved through a long series of improvements. This suggests that computational power may not be the main limit to the useful complexity that GP can evolve. Understanding what that limit is might make it possible to exploit evolution's ability to build deep functional complexity and hence attack new sorts of problems.

This work introduces a new form of EC called Tweaking Mutation Behaviour Learning (TMBL, pronounced "tumble"). Like GP, it entails evolving programs; unlike GP, it prioritises the long term growth of fitness above all else. This may be at the expense of efficiency in the initial generations if necessary.

Any technique for evolving programs will involve some arrangement of instructions and registers. Given the many proposed arrangements, any form of TMBL will inevitably bear strong similarities to existing forms of GP. However TMBL is about shifting the focus to long term fitness growth as much as it is about specific techniques. For this reason, TMBL appears here as a new form of EC, distinct from GP, rather than as one of its flavours.

It is worth acknowledging that a method which pursues long term fitness growth above all else will have to sacrifice other aims:

- Any such method will typically require a lot of data. It is unlikely to be good at generalising from sparse data since the aim is to avoid bias towards simplicity.
- For similar reasons, any such method is likely to be too messy to get perfect results on relatively simple questions. This suggests that it would not suit the "automatic programming" view that is sometimes associated with GP.
- The final result of any such method will hopefully be highly complex which will make it difficult to interpret. This cost is significant since a method that produces results that humans can understand can be used in many more situations than a method that can't.
- Any such method will be computationally expensive. In this context, computational efficiency (the amount of work needed to achieve some pre-specified degree of success) may be a less useful discriminator of methods than the final quality of solution that can be obtained given the maximum available computational resources.

Consequently even if TMBL achieves its goal perfectly, it will still be less effective than GP for the majority of GP's current application areas. This is a sacrifice worth making provided TMBL is able to make new progress on other problems that can only be solved through a long series of improvements.

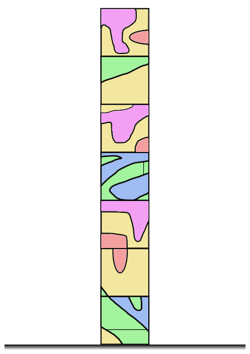
Both authors are with the Department of Computer Science, Birkbeck, University of London, Malet Street, London, WC1E 7HX, United Kingdom (emails: tony@dcs.bbk.ac.uk, gmagoulas@dcs.bbk.ac.uk).

II. What Limits Long Term Fitness Growth?

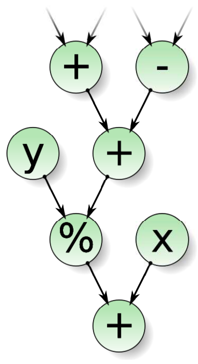
Consider a toy puzzle consisting of many cuboid blocks that must be lined up in some specific order according to the patterns on their surfaces. Assume that it is quite possible to stack the blocks into a tall tower without them toppling. Further assume that the puzzle is sufficiently tricky to require a good deal of trial and error but that planning is forbidden. Now imagine attempting to solve the puzzle by stacking the blocks vertically in a single, free-standing column as depicted for a small example in Figure 1(a). It is intuitively clear that this single-stack, trial-and-error approach is doomed; given a puzzle with enough blocks, the strategy will stagnate.

Why must this be so? This is because once initial progress has been made, it becomes difficult to make changes without ruining previous achievements. Once successful regions are formed in the stack, it becomes extremely difficult to adjust any blocks below without the successful region falling over. Hence as progress is made and as successful regions are formed, the cost of meddling with more and more of the blocks increases and so each next step becomes harder. There might be some easy improvements that can be made, particularly near the top of the stack, but once these get used up, the same problems remain. Eventually the attempt grinds to a halt.

Note that this strategy's attempts may consistently start well and may consistently make moderate progress. Initially, it would be easy to put one block on top of another or to substitute this block for that. This should not mislead; the strategy is limited.



(a) A small tower of blocks



(b) A GP tree

Fig. 1. Illustrations of a small tower of blocks (a) and a GP tree (b). It is claimed that these two challenges face similar limitations.

This analysis also illuminates the stagnation seen in GP. Compare the tower of blocks to an upside-down GP tree as depicted in Figure 1(b). In both cases, it becomes increasingly difficult to improve the structure by a process of trial and error because the more successful material that is built on top of an item, the harder it becomes to change that item without doing more harm than good. In the tower, the lower blocks provide physical support for the blocks above; in the inverted GP tree, the support is functional.

Then how are these processes able to make any progress at all? The early solutions are unremarkable so progress can be made by additions and occasional lucky random alterations. This process continues and the solution collects components built with the best luck seen so far. Consequently it becomes increasingly rare that a new randomly trialed component is better than the component it would replace, which represents the best luck seen so far in that area. The closer to the structure's core that the candidate change would occur, the more damage it is likely to do to the prior achievements and so the more exceptional good luck is needed to succeed.

These arguments emphasise the problem with building a single structure out of such highly interdependent units: as the structure becomes increasingly elaborate, it becomes increasingly difficult to modify the structure without ruining prior achievements. Imagine if biology had somehow been constrained to only allow one gene (translated to one protein chain) per organism. As evolution added ever more functions to this Swiss Army knife protein, new mutations would face ever more formidable constraints to maintain the precise structural configurations required to maintain all the previous functions.

Given plentiful blocks or nodes, why can't the process just keep improving by adding at the fringes until the resources dry up? For one thing, passable components congeal at the heart of the structure and get buried until there is no practical way to improve them. Each new layer of passable component that establishes itself adds new limitations to what remains practically achievable. That said, developing at the top of a tower may continue to improve its value at the same rate until the blocks are used up. For the GP tree, the outlook is worse because most components will tend to restrict the influence and scope of their neighbours further out from the core.

As with the tower, consistent moderate progress in tree-based GP should not mislead; all attention should remain on the situation after the initial progress and on overcoming the obstacles that arise then.

In reality, few players of the blocks game would persist with building a single vertical tower for long before switching to a strategy of assembling the puzzle horizontally. Once the puzzle is laid out flat, changes can easily be made without ruining previous achievements, making better results easier. This concept of a change which affects without ruining is at the heart of this work and is given the name *tweak*. For example, a sub-tree replacement mutation is not a tweak because it completely removes the previous sub-tree and thus requires the new random sub-tree to do a better job in that position than the sub-tree it replaces.

When tweaks are prevented, a candidate alteration must do a better job than the "best luck" component it would replace or damage. When tweaks are encouraged, a candidate alteration may alternatively succeed by making a new contribution to an existing component whilst still allowing it to remain and function as before.

The aims then, are to focus on the situation later on in the run and to find a form of program evolution that is "laid out flat" to encourage tweaks.

III. Background

Previous discussions of stagnation in GP have tended to focus on bloat and diversity. This work takes a different view based on the considerations in Section II and, in particular, tackles the issue by examining representation.

Many representations have been proposed for GP and there are different schemes for classifying them. The most commonly used is based on the sort of structure that is typically used to depict the representation: a line (of instructions), a tree or a graph. Another scheme divides by what the given representation's diagram indicates: the flow of data between locations or the flow of execution between instructions. These schemes suffer the problem that they depend upon a subjective choice in the way a representation was originally presented. For instance, Parallel Algorithm Discovery and Orchestration (PADO) [15] and GRAph structured Program Evolution (GRAPE) [12] are usually thought of as graph-based GP and yet they are functionally similar to Linear Genetic Programming (LGP).

An alternative, more objective scheme is to distinguish representations according to whether they use nodes, the entities out of which standard GP trees are built. It is normally sufficient to treat nodes as primitives that mean "perform this node's instruction on its inputs and then store the result within the node to be made available to other nodes". However for the purposes of comparing node-based systems with node-free systems, it helps to expose the implied instruction and register under the node's bonnet. A node can be thought of as one instruction and one register bound together in a pair such that the instruction only writes to its partner register and is the only instruction that may do so. After a node's instruction has executed, the result held in the node's register may be used as the input to another node's instruction. Hence a node-based system is different from a node-free system in that it imposes these constraints on the relationship between instructions and registers.

Since most computer programs are lists of instructions, the obvious representation for artificial program evolution is a linear, node-free representation. Unfortunately, mutating code written in common programming languages often invalidates the program. This may be solved by carrying out appropriate repair or interpretation to ensure a valid result, an approach taken by the TB [4] and "Pedestrians" [2] representations. It is notable that the resulting structures are node-based trees so the representation is a node-based tree with a linear genotype. The Compiling Genetic Programming System (CGPS) [10] representation directly manipulates linear machine code programs.

A possible extension is to allow the flow of execution to branch based on certain decisions. With branches added, the list of instructions can be teased out into a graph to help illustrate the possible flows of execution. This is how PADO [15], Genetic Network Programming (GNP) [7] and GRAPE [12] are presented.

Any node-free system must choose how to handle the data. For example PADO uses a stack from which inputs are popped and onto which outputs are pushed whereas GRAPE handles the data by carrying a data set around the graph along with the point of execution. Another two node-free representations, Linear-tree [5] and Linear-graph [6], have been proposed which combine linear sections into larger tree and directed acyclic graph structures respectively.

Despite the power of these branching node-free systems, they are potentially brittle because they have a single point of execution which traverses the instructions. A potential alternative is to use a node-based system. Node-based systems require the nodes to be connected to each other to determine which nodes each node uses as its inputs. An illustration of a node-based program typically represents these connections with arrows.

A node-based architecture must also choose how to handle the flow of execution. The most widely used representation in GP researched is tree-based. This is a node-based representation which constrains programs to be trees (meaning there are no loops in the equivalent undirected graph) and in which each node has the output of one node connected to each of its inputs. These constraints define the flow of execution because each node's inputs may be calculated in advance and any such ordering of execution gives the same result (provided there are no nodes with side effects). This also provides a simple crossover operation in which sub-trees from two individuals are swapped over. Furthermore, trees lend themselves to being interpreted by humans. Despite its wide use, some have criticised the tree-based representation for its lack of expressive power [15].

The Parallel Distributed Genetic Programming (PDGP) [11] and CGP [9] representations allow more general structures by allowing the output of each node to be used by more than one other further along in the program. The result is a direct acyclic graph which is capable of reusing the partial results. The drawback of this enhancement is that it is less apparent how to construct a crossover operation. PDGP solves this issue with sophisticated crossover operators that swap sub-graphs and then repair the result as needed. The issue is solved in CGP by an elegant mapping to the graph phenotype from a genotype consisting of a string of integers which means that GA operators can be used with the genotype.

The papers for PDGP and CGP mention the possibility of allowing loops and calculating the results by iteratively updating the graph (so that each node updates based on the inputs from the previous iteration). This is also the approach taken by Neural Programming (NP) [14].

Two representations, Multiple Interacting Programs (MIPs) [1] and Recurrent Network consisting of Trees (RTN) [17] provide distinctive variations by combining evolving trees into a larger graph. Another representation, Tangle Representing Execution And Data (TREAD), involves several additional features including an if-condition associated with each node to determine whether or not it executes [8].

IV. A TMBL Form to Avoid Limitations

Section II recommends focusing on the situation later on in the evolutionary process. The path taken to that position is secondary; the primary concern is what can be done to encourage further development once there. Consider what this situation tends to look like. At this point, the population fitness has typically made substantial progress and the best individuals have a lot to lose from a bad mutation. The initial flurry of improvements has waned and few generations see the population best improve. The individual that most recently improved the population best is likely to be dominating the population through its descendants. Other lineages that do not match the fitness of this top individual disappear quickly and descendants with deleterious mutations rarely last more than a few generations.

For these reasons, there is unlikely to be much diversity in the enduring core of the population, just minor variations on one form of solution. This means there is little for a crossover operator to work with, so although it may or may not confer some additional benefit, it isn't the significant source of functional inventiveness. Instead, the evolutionary process must rely on mutation to provide most adaptive steps.

It was also argued in Section II that to improve long term fitness growth, the focus should be on finding a structure that encourages tweaks (changes which affect behaviour without ruining existing functionality).

What sorts of properties of a program representation might encourage tweaks? Firstly, a change to one component of the program should be able to affect the behaviour of another part of the program without it being necessary to also change that other part. Compare this to the way that a newly evolved gene's product can interact with a pre-existing biological process carried out by other genes' products. This suggests that the program should be built out of actions that affect other entities rather than static components that present their results for use by another part of the program. Instead of the overall behaviour arising from a single structure, it should arise from many parts which evolve to make their own contributions. Secondly, each component should have as few ties with functionally unrelated components as possible. This suggests that the design should not force components of the program to share aspects globally.

This analysis can be used to design a standard form for TMBL. Note that this is just one of many possibilities and other researchers are encouraged to propose their alternative suggestions for meeting the aims of TMBL. The methodology used here is to review the properties that divide the various GP representations and, at each stage, use a fresh focus on tweaks to make a choice.

A. Choosing whether to use nodes

Using nodes makes it hard to modify the behaviour of a program without damaging existing functional behaviour. In other words, using nodes hinders tweaks. This is because the changes affecting the behaviour of a node-based program will involve changes to active nodes (nodes which currently affect the output) but this involves disrupting the contribution that node and its active children were already making to the output.

Changes at the boundary between active and inactive nodes may minimise the number of useful nodes that are disrupted. Unfortunately, such changes at the fringes of the functional structure tend to have restricted influence and tend to create a new fringe with even less influence. As argued in Section II, building at the fringes does not solve the problems under consideration.

Nodes undermine the stated aim of building programs out of actions which can directly affect the behaviour of other parts of the program. Without nodes, it is relatively easy to change an instruction to modify some register without disrupting other instructions that are already using it. For these reasons, nodes will not be used in this representation.

B. Choosing the structure of memory

What structure of memory seems most likely to encourage tweaks? Until now, the word "register" has been used to refer to any part of an EC program's memory. From now on it is worth being more precise because in addition to plain registers, GP systems may alternatively arrange the memory into a stack or indexed memory.

In register-based memory, each instruction is tied to specific registers which it uses for its output and inputs. Unlike when using nodes, each register may be read from or written to by multiple instructions. This means that the instructions must be placed in some order to ensure consistency and to avoid access clashes. Non–node-based GP systems more commonly use stack-based memory. This involves each instruction popping enough data off the stack for its inputs, performing its calculation and then pushing the result back onto the stack. Indexed memory involves providing read/write functions that allow a program to use a run time argument to indicate which slot of memory to access.

The stack approach seems likely to be the most brittle. As a stack-based program develops, it will become increasingly tricky for mutations to affect behaviour (in any way which involves the stack) whilst still preserving the state of the stack well enough to avoid damaging already functioning parts. A stack is too global in the sense that all separate computations in a program are forced to share the same stack.

Indexed memory is a potentially suitable approach which appears less brittle than a stack because it is relatively easy for newly mutated parts of a program to access one area of indexed memory without affecting already functioning parts of the program which access another area. However, since indexed memory is more complicated than register-based memory and requires more of its instructions, it is not included in the scope of this investigation.

Plain registers encourage tweaks because they make it easy for changes to instructions to affect registers being used by other instructions without ruining the actions of those other instructions. Furthermore, different parts of a program can easily avoid sharing resources by using separate registers.

Systems using registers often use relatively few, and an analysis of LGP found that 16 registers was suitable [3]. The consequence of this is to force growth to be vertical in the sense that programs develop their fitness by extending the list of instructions that cooperate in sequence on a small set of registers. That sort of growth is important and may be essential for developing some of the complex parts of an algorithm, but it involves building a complex network of interactions and so makes tweaks increasingly difficult. For this reason it should be complemented by horizontal growth in which programs can develop their fitness by developing new groups of instructions and registers which make (fairly) independent contributions. This suggests that TMBL should use substantially more registers than are normally used in LGP.

C. *Choosing the type of flow control*

Programs that require conditional behaviour should permit some form of flow control in the representation. The most common forms of flow control in non–node-based GP use a single point of execution which flows through the program and jumps to different locations in the program depending on the result of an evaluation each time it reaches certain branch points. This approach is too global for TMBL because it requires that all components of a program must collaborate on a shared flow of execution. As programs develop complexity, it becomes increasingly hard for new parts to exploit their programs' flow control systems without damaging other parts already relying on them.

To encourage tweaks, the TMBL representation should instead use a more local system in which each instruction can determine its own execution status. This is achieved by allowing each instruction to potentially have its own if-condition test. An instruction with an active if-condition is only executed when the value at the if-socket is positive. This system can still be used to generate sophisticated behaviours by repeating the execution through the program for multiple iterations. Similar systems have been discussed for LGP that allow multiple, nested if-conditions [3].

D. *Choosing the type of instructions*

In a final step to encourage tweaks, the instructions are constrained to always have the target register as the first input register. Whereas instructions are normally of the form "overwrite register C with the result of adding register A and B", this constraint restricts them to the form "add register A to register B". This encourages instructions to modify the values in registers without destroying any information that they previously hold and so encourages changes that affect without ruining.

V. A Summary of TMBL's Standard Form

The resulting representation is somewhere between a linear (node-free) representation and a cyclic graph-based (node-based) representation. Like a linear representation, the instructions and registers are not paired together. A stack is not used as is often the case in linear representations and more registers are used than is normally the case (for those linear representations that use them). Like a cyclic graph-based representation, the evaluation is iterated and all nodes are evaluated each iteration (except those that opt out via their if-conditions) rather than there being a single point of execution as is often the case in linear genetic programming. The instructions are constrained to be of the form "add the value in register A to the value in register B". The implementation can be summarised as follows:

- Each individual consists of an ordered list of instructions and two numbers indicating the number of registers and iterations to be used when evaluating the individual.
- Each instruction contains an if-switch, an if-socket, an input-socket, an output-socket and an operation.
- The if-switch is a Boolean value indicating whether the if-condition is to be used.
- Each of the sockets contains the index of a register or of a dimension of the test case. The output-socket may only refer to a register (because instructions should not write to test cases).
- Before evaluation, the registers are all initialised to zero.
- In each iteration, each instruction is evaluated in turn.
- If an instruction has an active if-condition, the instruction is skipped whenever the value pointed to by the if-socket is negative.
- Executing an instruction involves reading the value pointed to by the input-socket and using the operation to apply that value to the register indicated by the output-socket.
- After the last iteration is complete, the output is taken from the last register.

In addition to the standard functions, a TMBL program has the functions SetValue and Copy. The SetValue function sets the target register to some floating point number held within the instruction (which is open to mutation). The Copy function copies the input to the output. It would be simple to modify this representation to allow for operations with arity other than two (although some thought may be required to construct operators that act on a register rather than overwriting it).

Crossover could easily be applied but was not used in these experiments. The mutation operator varies each component of each instruction with some small probability and moves an instruction to some other location in the execution list with some small probability. The probabilities are set such that each individual has a 0.95 probability of having at least one mutation.

VI. The Problem and Experimental Setup

To begin the investigation of this TMBL representation, it was tested on its ability to learn many simple pieces of information — a simple task that presents a significant obstacle to GP. To make this problem suitable, it was expressed as a form of binary classification problem as follows. At the start of a run, a set of test cases are created and each of them is randomly assigned to one of two classes, positive or negative, with equal probability. An individual's fitness is the number of test cases for which the individual outputs a value of the correct sign when evaluated on the test case. Zero is included in the positive class.

Early experiments used consecutive integers for the test cases but this made it difficult to get a feel for the resulting behaviours. To make the behaviours more visually interesting, a second dimension was added and the test cases were chosen randomly from a uniform distribution over $[-100, 100] \times [-100, 100]$.

Early experiments also highlighted that the lack of feedback from the fitness function was slowing down the evolutionary process to no useful end. To provide feedback on the incorrect results, the fitness function was modified to award two points per test case if an output x has the correct sign and $1.001^{-|x|}$ otherwise. This means that all correct answers are awarded the same but incorrect answers are awarded more for being less wrong (ie for having a smaller magnitude).

Common Settings	
Objective	Learn a function which duplicates the sign of test cases randomly distributed in $[-100, 100] \times [-100, 100]$
Fitness	As described in the main text
Functions	+, -, * and % (protected divide)
Number of generations	10000
Population size	1000 (four demes of 250)
Number of instructions	150
Number of instructions executions	$150 * 20$ iterations
Selection	Tournament
Tournament size fraction	0.3
Number of test cases	512
Gap between deme transfers	30 generations
Deme transfer topology	Ring
Deme transfer mechanism	Best replaces random
TMBL specific settings	
Number of registers	150
Additional functions	SetValue and Copy (see text)
Operators	Mutation as described in the text
LGP specific settings	
Number of registers	16
Operators	Mutation (similar as for TMBL)
Tree-based GP specific settings	
Genetic operators	Crossover, hoist mutation, subtree mutation, constant mutation

TABLE I

A SUMMARY OF THE KEY PARAMETERS USED FOR THE EC RUNS GROUPED INTO COMMON SETTINGS AND SETTINGS SPECIFIC TO EACH TECHNIQUE.

More formally, let n be the number of test cases, let $(x_i, y_i)_{i=1\ldots n}$ be the test cases where each x_i and y_i are drawn from the uniform distribution $U(-100, 100)$ and let $(a_i)_{i=1\ldots n}$ be the correct answers where each a_i is randomly chosen to be either -1 or $+1$ with equal probability. Then the fitness of an individual $I(,)$ is defined as:

$$\sum_{i=1}^{n} f(a_i, I(x_i, y_i))$$

where $f(a_i, I(x_i, y_i))$ is defined as:

$$\begin{cases} 2 & \text{if } \frac{I(x_i,y_i)}{a_i} > 0 \\ 2 & \text{if } I(x_i, y_i) = 0 \text{ and } a_i = 1 \\ 1.001^{-|I(x_i,y_i)|} & \text{otherwise} \end{cases}$$

For the experiment, TMBL was compared against LGP, perhaps the closest form of GP, and tree-based GP. A form of register-based LGP was constructed that is similar to TMBL but which had a few differences to make it fit in with standard LGP practice. It has fewer registers than TMBL (16 rather than 150). It has instructions with two inputs that overwrite the output. Finally, rather than if-conditions, it has branching-conditions that allow the point of execution to jump to any other instruction in the program.

Table I gives more details of the parameters of the runs. Each experiment was repeated three times.

VII. Results

Figure 2 shows the best fitness for each technique for 10000 generations averaged over three runs. It is encouraging that TMBL finishes with the highest fitness. It is even more encouraging that the graph indicates that TMBL is doing a better job of avoiding stagnation than the other techniques. Note that TMBL is the worst near the start of the runs and only takes the lead after nearly 3000 generations.

For LGP and TMBL, Figure 3 shows the behaviour of a best individual in the final generation. The behaviours are both fairly complex; there is perhaps some indication that the latter shows slightly more ability to construct independent solid blocks away from the origin but it is not clear that this is significant.

VIII. Conclusion

This work introduces TMBL, a style of EC of programs which prioritises the long term growth of fitness through a slow process of accumulating small improvements. To achieve this, the emphasis is placed on finding a representation that encourages tweaks, mutations which are able to affect an individual without ruining its previous functionality. A representation was identified to suit this aim and it was empirically compared to the most similar form of GP. The results are encouraging: TMBL had the highest mean best fitness at the final generation and TMBL's fitness growth looks much more promising for the long term growth of fitness.

There follows an outline of some of the potential criticisms of this work with the corresponding responses and indications of possible avenues for further research.

- **Insufficiently Thorough Investigation.** This work has presented a prima facie case with some initial supporting evidence but there is much more work to be done to make the case fully convincing (such as introducing more repetitions, longer runs and more varieties of GP and TMBL).
- **Bias of Problem.** The test was chosen to identify whether TMBL has an ability to learn more pieces of

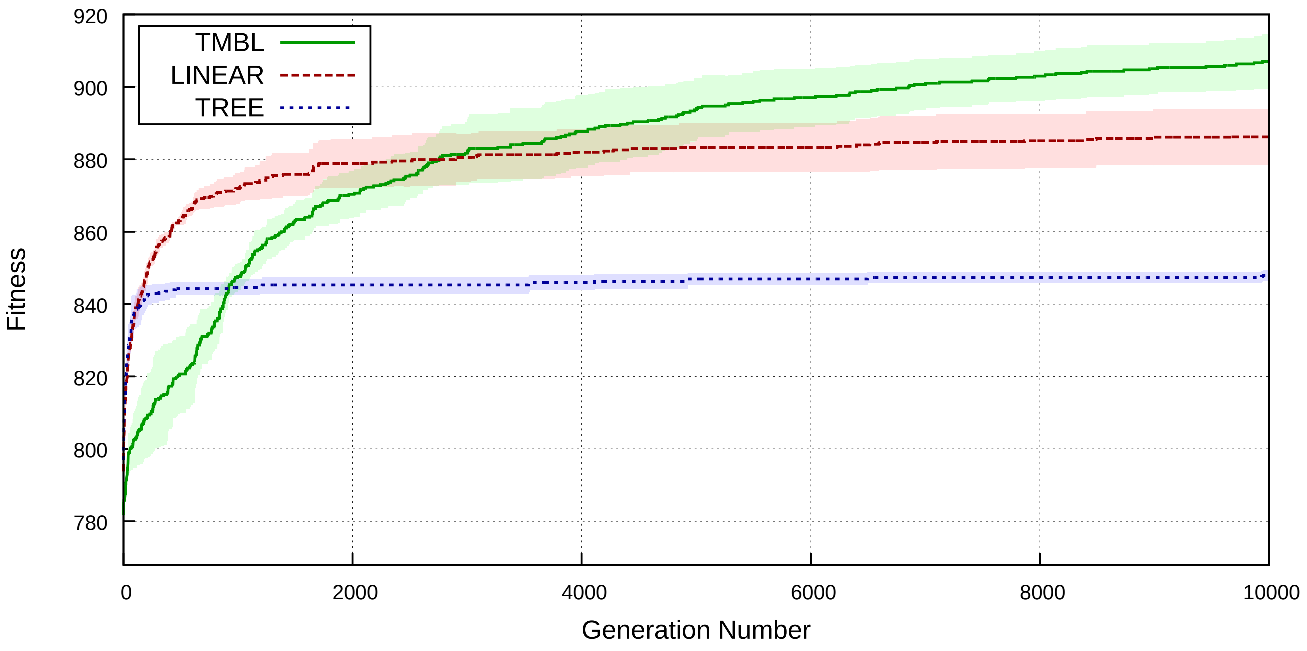
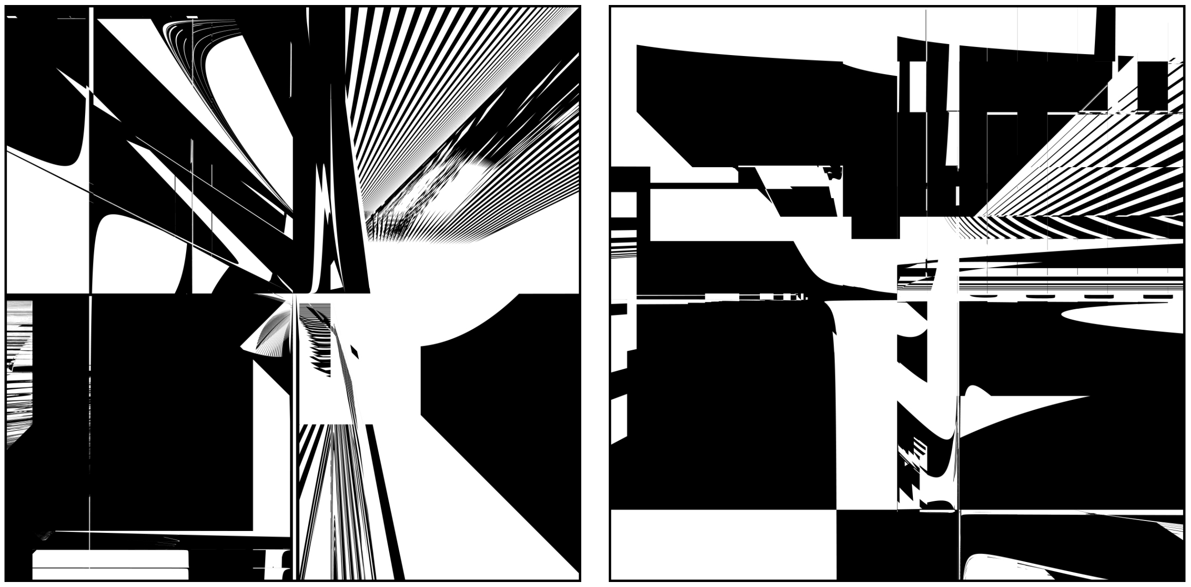


Fig. 2. For each technique, this figure shows the mean best fitness from three runs at each generation. The baseline of the graph is set to 768 which is the expected fitness for an individual which always outputs zero. The maximum possible fitness is 1024. The shaded areas represent values within one estimated standard error of the corresponding mean.



(a) The behaviour of an individual generated using LGP with fitness 897.899 and 386 of the 512 ($\approx 75.39\%$) test cases correct

(b) The behaviour of an individual generated using TMBL with fitness 916 and 404 of the 512 ($\approx 78.91\%$) test cases correct

Fig. 3. Each figure shows how the fittest individual after 10000 generations divides the square between the two classes: negative (black) and non-negative (white). In training, each of the individuals is only assessed on 512 points within this square but here the behaviour of each individual is plotted on (a fine grid of points over) the full range $[-100, 100] \times [-100, 100]$, not just the points that the individual was trained on. The sub-figures are labelled with the number of test cases correctly assigned but they do not show the locations of any of the test cases, correctly assigned or otherwise. Since the test cases are randomly created at the start of each run, each of the individuals shown here was trained on different data.

information than standard forms of GP. However, it might be argued that the resulting problem is too well tuned to the proposed representation, perhaps because the problem is biased in favour of if-conditions. This is an area for further research and identifying the right sort of problem may well be vital to the success of TMBL. Readers are encouraged to offer their suggestions or to try using TMBL themselves.

- **Poor Generalisation of Solutions.** This work has shown that TMBL produces messy solutions which are unlikely to be effective generalisers for problems such as data mining. This was an anticipated cost of this approach as discussed in Section I.
- **Computational Requirements.** The runs involved in this work took a substantial amount of time because they involve over 15 million million instruction evaluations each. Some might argue that this renders the approach impractical.

  That view may be short sighted. Work on accelerating the TMBL runs using a (reasonably old) Graphics Processing Unit (GPU) have been very effective and as processors develop, these computational demands are likely to seem increasingly reasonable.

  Those interested in tackling more complex problems will have to accept the need for more computation. The question is not which methods avoid the need for substantial computational power but which methods can do the most when such power is made available.
- **Inadequate Avoidance of Local Optima.** One further line of potential criticism worth anticipating is an argument from fitness landscape intuitions. Some may argue that because TMBL uses a slow, steady process which does little to preserve diversity and which focuses on relatively small mutations, the method will inevitably get stuck in the fitness landscape's local optima.

  This sort of argument appeals to intuitions drawn from a two dimensional fitness landscape. In two dimensions, a landscape is either relatively trivial or is made up of enough local optima to require diverse populations. However, it should not be assumed that this intuition translates to a very high dimensionality landscape. Perhaps some problems are hard, not because the landscape has no smooth path ascending the fitness landscape but because the path is exceedingly hard to navigate in a space with vast dimensionality. Notwithstanding phenomena such as horizontal gene transfer and sexual reproduction, natural selection has built astonishing functional complexity despite its inability to plan ahead or take large regressive steps to achieve greater aims.

REFERENCES

[1] P. J. Angeline. Multiple interacting programs: A representation for evolving complex behaviors. *Cybernetics and Systems*, 29(8):779–806, 1998.

[2] W. Banzhaf. Genetic programming for pedestrians. In S. Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, page 628, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[3] M. Brameier and W. Banzhaf. *Linear Genetic Programming*. Number XVI in Genetic and Evolutionary Computation. Springer, 2007.

[4] N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In J. J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA, 24-26 July 1985.

[5] W. Kantschik and W. Banzhaf. Linear-tree GP and its comparison with other GP structures. In J. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Genetic Programming: 4th European conference*, pages 302–312, Berlin, 2001. Springer.

[6] W. Kantschik and W. Banzhaf. Linear-graph GP—A new GP structure. In J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi, editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 83–92, Kinsale, Ireland, 3-5 Apr. 2002. Springer-Verlag.

[7] H. Katagiri, K. Hirasawa, J. Hu, and J. Murata. Network structure oriented evolutionary model-genetic network programming-and its comparison with genetic programming. In E. D. Goodman, editor, *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 219–226, San Francisco, California, USA, 9-11 July 2001.

[8] T. E. Lewis and G. D. Magoulas. TREAD: A new genetic programming representation aimed at research of long term complexity growth. In M. Keijzer, G. Antoniol, C. B. Congdon, K. Deb, B. Doerr, N. Hansen, J. H. Holmes, G. S. Hornby, D. Howard, J. Kennedy, S. Kumar, F. G. Lobo, J. F. Miller, J. Moore, F. Neumann, M. Pelikan, J. Pollack, K. Sastry, K. Stanley, A. Stoica, E.-G. Talbi, and I. Wegener, editors, *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1339–1340, Atlanta, GA, USA, 12-16 July 2008. ACM.

[9] J. F. Miller and P. Thomson. Cartesian genetic programming. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 121–132, Edinburgh, 15-16 Apr. 2000. Springer-Verlag.

[10] P. Nordin. A compiling genetic programming system that directly manipulates the machine code. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, 1994.

[11] R. Poli. Evolution of graph-like programs with parallel distributed genetic programming. In T. Back, editor, *Genetic Algorithms: Proceedings of the Seventh International Conference*, pages 346–353, Michigan State University, East Lansing, MI, USA, 19-23 July 1997. Morgan Kaufmann.

[12] S. Shirakawa, S. Ogino, and T. Nagao. Graph structured program evolution. In H. Lipson, editor, *Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, 2007*, pages 1686–1693. ACM, 2007.

[13] A. Teller. Turing completeness in the language of genetic programming with indexed memory. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 136–141, Orlando, Florida, USA, 27-29 1994. IEEE Press.

[14] A. Teller. *Algorithm Evolution with Internal Reinforcement for Signal Understanding*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, 5 Dec. 1998.

[15] A. Teller and M. Veloso. PADO: A new learning architecture for object recognition. In K. Ikeuchi and M. Veloso, editors, *Symbolic Visual Learning*, pages 81–116. Oxford University Press, 1996.

[16] J. Woodward. GA or GP? that is not the question. In R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 1056–1063, Canberra, 8-12 Dec. 2003. IEEE Press.

[17] T. Yabuki and H. Iba. Genetic programming using a Turing complete representation: recurrent network consisting of trees. In L. N. de Castro and F. J. Von Zuben, editors, *Recent Developments in Biologically Inspired Computing*, chapter 4, pages 61–81. Idea Group Publishing, 2004.