Why Evolution Is Not a Good Paradigm For Program Induction; A Critique of Genetic Programming

John R. Woodward
University of Nottingham
199, Taikang East Road, University Park
Ningbo, 315100, People's Republic of China
John.Woodward @Nottingham.edu.cn

Ruibin Bai
University of Nottingham
199, Taikang East Road, University Park
Ningbo, 315100, People's Republic of China
Ruibin.Bai @Nottingham.edu.cn

ABSTRACT

We revisit the roots of Genetic Programming (i.e. Natural Evolution), and conclude that the mechanisms of the process of evolution (i.e. selection, inheritance and variation) are highly suited to the process; genetic code is an effective transmitter of information and crossover is an effective way to search through the viable combinations. Evolution is not without its limitations, which are pointed out, and it appears to be a highly effective problem solver; however we over-estimate the problem solving ability of evolution, as it is often trying to solve "self-imposed" survival problems.

We are concerned with the evolution of Turing Equivalent programs (i.e. those with iteration and memory). Each of the mechanisms which make evolution work so well are examined from the perspective of program induction. Computer code is not as robust as genetic code, and therefore poorly suited to the process of evolution, resulting in a insurmountable landscape which cannot be navigated effectively with current syntax based genetic operators. Crossover, has problems being adopted in a computational setting, primarily due to a lack of context of exchanged code. A review of the literature reveals that evolved programs contain at most two nested loops, indicating that a glass ceiling to what can currently be accomplished.

Categories and Subject Descriptors

I [**Computing Methodologies**]: Artificial Intelligence—*Automatic Programming*

General Terms

Algorithms

Keywords

Genetic Programming

1. INTRODUCTION

Koza poses the question ([1], page 1, attributed to Arthur Samuel in the 1950's) *"How can computers learn to solve problems without being explicitly programmed? ..., how can computers be made to do what is needed to be done, without being told exactly how to do it."* Inspired by evolution, Genetic Programming (GP) goes some way to addressing this issue, however, ironically, the programs evolved are typically a sub-set of Turing Equivalent (TE) programs (i.e. logical or arithmetic expressions rather than computer programs involving iteration and memory [2]). This paper states a case for the unsuitability of GP to evolve TE programs.

Evolution is a process which acts on biological entities, and causes the frequency of genes to vary over time. Three factors are necessary for evolution to take place; variation, inheritance and selection. Variation is caused by mutation and crossover. In either case, mutation of a gene, or the crossing-over of genetic material from two parents, will result in an individual which has different traits to the parents. Selection is provided by the environment, and inheritance is supported by the propagation of genetic material. GP is a machine learning method inspired by evolution.

In machine learning, the concept of search is a very general "test-and-generate" methodology, in which a set of potential solutions is evaluated. Evolutionary computation searches the set of solutions in a style inspired by evolution. If the solution set contains computer programs, then we call this approach GP [1, 3]. Thus GP is the search of the space of executable data structures. The programs could represent logical expressions or arithmetic expressions (as is usually done in GP [1]), however it can also include computer programs in the most general sense (i.e. effective procedures or algorithms) which include capabilities of iteration/recursion and memory. It is the evolution of algorithms which is our concern in this paper. Our goal in GP is to find a program defined by a set of test cases, and not to study aspects of the evolutionary process.

There are a number of motivations for studying program induction using TE systems. Angeline [4] states, *'A poor representation limits what behaviors can be created while a good representation permits the induction of any potentially useful sequences of actions'*. Teller ([5], page 133) states, *'given that programs written in a Turing Complete language are more difficult to evolve than programs written in less expressive languages... why bother evolving them?'*. He has two answers; Firstly, TE programs are more expressive than other reactive functions (which have no memory) which are usually evolved in GP [1]. Secondly, computer programs can express a function in less space than less expressive representations. This also aids generalization; almost all of the solution evolved with TE programs are general (see section 2). While there are a number of important motivations for wanting to induce algorithms, the central message of this paper is that evolution is not an effective methodology for automatically synthesizing algorithms.

The line of the argument of this article is as follows. In section 2 we review the literature concerning the application of GP to TE program spaces. We review some aspects of evolution in section 3, and argue that genetic code is a highly suitable representation to carry genetic information, and crossover is an effective way to search this space. We also explore some of the limits of evolution and argue that it is illusively successful. In section 4, we examine some of the problems of applying the principles of evolution to the task of automatically generating computer programs. These problems stem largely from the highly discontinuous mapping between program space and function space, an issue which is compounded by the extra flexibility available if we are evolving TE computer programs rather than less expressive executable data structures. In the remaining sections we summarize these issues and conclude the article.

2. LITERATURE REVIEW

Various computer models have been used, including Turing Machines (TM), Universal Register Machines, and some model designed especially for evolution. Often the solution evolved is completely general, solving all instances of the problem exactly. The majority of authors comment of the unsuitability of the genetic operator, crossover, to search the space of programs. Almost all of the authors avoid the problem of non-halting programs by enforcing a fixed upper limit on number of instructions executed. Turing Machines are possibly the most well known models of computation, yet it is interesting that little work has been done using them as a representation for evolution [6, 7, 8].

Cramer [9] evolves a multiplication function. He takes the PL language (which is TE) containing instructions: `INC`, `ZERO`, `LOOP`, and `GOTO`. Programs consist of lists of instructions and have an arbitrary number of globally scoped integer variables. He studies a less expressive language (PL-:GOTO) which consists of the instructions of PL without the instruction `GOTO`. Programs written in PL-:GOTO are guaranteed to halt and correspond to the set of primitive recursive functions. He also adds a `SET` instruction and a `BLOCK` operator that accepts two statements as arguments and evaluates them sequentially, which is a grouping operation and calls this the JB language. In the JB language, a program is represented as a list of integers. To interpret a list, it is split into sections of three integers, the first integer is the instruction and the remaining two integers are the arguments. Each set of three integers therefore corresponds to a program statement. He comments that this language is unsuitable for evolution as it is very sensitive to changes because *'a single unfortunate mutation .... would destroy any useful features of the program'* and the language TB is introduced which has a tree like structure. Genetic operators act near the leaves of the tree in order to avoid the *'catastrophic minor change'* problems JB suffered from.

The key to designing an effective evaluation function is one that rewards multiplication-like behavior (i.e. reflects the fact we are moving nearer the global optima). Cramer [9] used the following *'after much experimentation'* (indicating the difficulty of designing a good fitness function). Programs with the following types of behavior were given successively more credit; 1. the output variables changed from their initial values, 2. simple dependency of the output on the input, 3. the value of the input is a factor of the output, 4. the output is the product of the two inputs.

Tanomaru et. al. [6] argues that *'since automata are typically represented by their state transition tables, this seems to be the most natural representation to be adopted'*. Crossover exchanges randomly selected contiguous rows in the state transition table, and the children therefore maybe of different length to their parents. Mutation changes the contents of a percentage of randomly selected cells of the state transition table. They tackle two problems of sorting tapes of two symbols and proper subtraction.

Tanomaru [7] argues that crossover is not an effective procedure as it can result in meaningless automata (as they refer to non existent states) and thus valuable processing time is wasted. Crossover is entirely dropped as *'it is based on the building block hypothesis, which is very unlikely to hold in the case of automata generation'* (he offers no explanation of this point of view is offered). Three problems were tackled; the recognition of a regular, context free and a context sensitive language. Results indicate that the enhanced approach outperforms the simpler approach detailed in [6].

Vallejo et. al. [8] evolved Turing Machines with the capabilities of bio-sequence recognition. The experiments use a machine with 32 states and a tape alphabet consisting of 8 symbols. It is not stated how many of these states are actually active in the final solution. The GA found a Turing Machine which correctly classified all *training sequences* and accepted *several sequences* not included in the training set.

Huelsbergen presents a series of papers [10, 11, 12] in which he evolves programs. His register machine consists of a number of registers which hold integer values, and a flag which can have one of three values (*lessthan*, *greaterthan* or *equal*). Examples of instructions included in this language are `compare` and `jump`. There are a collection of conditional `jump` instructions which jump depending on the value of the flag. A number of other instructions include `increment`, `decrement` and `clear`. In this system (unlike Cramer's [9]) an explicit looping structure is not used which makes the problem harder. In [10] Huelsbergen evolves a multiplication function using a primitive set including an addition instruction. In [11] Huelsbergen evolves programs which generate recursive sequences (squares, cubes, factorial and Fibonacci numbers). In this paper the instruction set has additional instructions (subtraction, multiply and division). In a similar vein to his previous paper, he demonstrates that recursive sequence generating programs can be evolved without using an instruction set which includes an explicit recursion operator. In [12] Huelsbergen evolves solutions to the parity problem. He claims that no domain specific operators are used, however he does include a number of logical operators not included in the previous papers [10, 11]. Two point crossover and single point mutation are employed. Headless chicken crossover is also used, where a randomly generated contiguous group of instructions is inserted into a program, rather than being inherited from a parent. Interestingly on this problem macro-mutation significantly outperforms crossover. All of the solutions obtained in these papers [10, 11, 12] could be described as containing a single loop. There is a looping structure which executes a section of code repeatedly however, no nested loops were evolved (but were not needed for these problems). As the solutions only contain one loop, there is no obvious building block, so it is perhaps not surprising that crossover did not perform well.

Schmidhuber et. al. produced a series of papers [13, 14, 15]. These papers use a probability distribution over instructions which are part of programs written in as assembly type language. These papers are not concerned with a population based evolutionary approach but rather learning to learn where an individual program alters its own learning bias as it receives reward from the environment. In these works the agent can potentially use any learning algorithm as the instruction set is TE (this therefore has the advantage over other self adaptation approaches which can only alter the way they learn in a very restricted sense). There are a number of "normal primitives" which perform standard mathematical operations on the registers. There are also a number of special primitives which can alter the probability distributions of the instructions in the program (and therefore alter how the program alters itself). Another special primitive signals to the program when to assess its own learning. Schmidhuber et. al. [14], tackle a simple maze navigation task. The solution produced consists of two explicit nested loops [14] (page 9) and [15] (page 116). In a second implementation, a function regression problem is tackled and a single looping program is evolved [14] (page 17) and [15] (page 127).

Nordin et al. extend the Compiling GP System (CGPS) [16], which uses instructions of fixed length, to Automatic Induction of Machine Code GP (AIM-GP) [17], which uses instructions of variable length. The original motivation was to directly evolve machine code avoiding the interpretation process high level languages have to go through, achieving an impressive improvement in speed of around 1000 times compared to LISP implementations. In CGPS, crossover points are readily identifiable as instructions are of fixed length. To avoid this issue in AIM-GP, instructions are grouped together in fixed length instruction blocks (the size of the blocks being a globally defined parameter). The block size must be set large enough to accommodate the largest instruction. Crossover exchanges these instruction blocks. Instruction blocks are like an implicit glue (compare this with Cramer's `BLOCK` instruction) and may assist in the protection of building blocks against possible disruption by crossover. As more than one instruction may appear in an instruction block, crossover cannot separate the instructions. Mutation can do one of 3 things, either affect a whole instruction block, an instruction or the operand of an instruction. Homologous crossover preserves the context in tree based GP by selecting the same crossover point in both parents, thus preserving location. This is not easy to achieve in a linear system, but one way of achieving it is to use ADFs [17].

Teller [2] states that standard GP is not capable of evolving TE structures. He adds iteration and memory to make the representation TE. Some problems require only current state information for them to be solved, while other problems need historical state information in addition, so Teller [18] introduces a new problem to GP. An agent is in a grid world with a number of boxes. The aim is for the agent to push all the boxes to the edges of the environment. The agent has no way of telling which way it is pointing unless it keeps track of that information using Indexed Memory. He evolves the agent both with and without ADFs and, although ADFs are not required for Turing Completeness, their presence does improve performance. An agent's exploration of the grid is terminated after 80 time steps (a little less than 2 tours of the board).

Langdon [19, 20] evolves a number of data structures (including stacks, queues and lists) using Teller's Indexed Memory [18]. Langdon [19] generates stack and queue data structures. An individual is five trees, one for each operation of the stack (push, pop, makenull, empty, and top). Only trees representing the same operation are crossed over. The problems only require solutions to implement a stack of 10 integers, however the solutions scale up to stacks of any depth. The fitness function is the number of fitness cases passed. Langdon [20] again uses Indexed Memory to evolve a list data structure, a generalization of a stack and a queue which were previously evolved [19]. He evolves ten list operations including insert, delete and locate. A `for-while` loop was used and a limit was placed on the number of iterations. This limit was set as low as possible but still allowed loops to span all the available memory.

3. EVOLUTION

In this section we look at the suitability of genetic code for the task it performs (i.e. transmitting inheritable information to the next generation), and also the effectiveness of crossover (the major genetic operator compared to mutation) to search for new individuals. We also consider some of the limits of evolution, and ask why does evolution appear to be so successful (e.g. it has created the complexity of human brains and many different versions of eyes).

3.1 Genetic code

Genetic code is highly suited to its purpose [21, 22, 23, 24]. The process of translating through all the stages from the genetic bases (abbreviated to A, C, G, T) on a string of DNA, through to a functioning protein is proof of principle that genetic code is suitable for the task of inheritance. Contiguous sections of three bases along a DNA molecule, called codons, code for one of 21 amino acids and a STOP symbol. Genetic code has some properties which make it particularly suitable as the transmitter of genetic information. We make the following points;

1. Codons, which are groups of three bases, code for an amino acid. This is the minimal number. As there are four types of bases, a codon can code for up to 64 different symbols. If only 2 bases were used, 16 amino acids could be coded for, while 4 bases could code for 256 amino acids, making the 4th redundant.
2. The amino acids coded for by codons are not mapped to randomly, but are grouped together in clusters. This means that often changing (or mutating) the third base in a codon does not change the amino acid represented. Hence, in terms of coding theory this is a perfect code as it is as robust to mutation as it can be, given the number of bases, the size of codons and the alphabet of amino acids to represent.
3. Even if a base is mutated, so the codon containing it codes for a different amino acid, the properties of the resulting protein are still very similar (e.g. in terms of their properties; non-polar, polar, basic, acidic). Thus, while the resulting protein may contain a different amino acid, it properties, and therefore function, may have only slightly changed.

These three points mean that genetic code is efficient (i.e.

it uses the shortest number of bases), and is reasonably robust against mutation. And it is this robustness which may make it so suitable for the carrier of hereditary information (i.e. it can effectively transmit information to the next generation), but is also open to the possibility of slight corruption during transmission, and thus allows for variation in the resulting population. In contrast, computer code is very brittle; often one instruction out of place causes either a syntactic error, or results in a program with completely different semantic properties to the original program.

3.2 Crossover and Sexual Reproduction

In evolution, variation is necessary. In asexual reproduction, this happens only via mutation, where new genetic material may be introduced into the gene pool. In sexual reproduction, this variation happens chiefly via crossover, where new genes are not introduced into the gene pool, but new individuals are created. Importantly, mutation may result in non-viable biological entities which perish before becoming reproductively active, whereas crossover is much more likely to produce individuals which are viable but phenotypically different to the parents. Crossover is a much safer bet compared to mutation, in this sense of being able to create original off-spring (i.e. different from off-spring), but also have a high chance of being viable.

Sexual reproduction has undoubtedly been responsible for the evolution of more complex organisms than asexual reproduction (i.e. eukaryotic, compared to prokaryotic). In evolution, when two individual reproduce, the resulting individuals are a combination of the parent in some sense. In a simplistic example, the eye color of the off-spring is related to the eye color of the parents. The situation is similar for skin color and hair color. The exact relationship of the color of the said phenotypic feature is related to which genes for which colors are present in the parents. The mixing or blending of phenotypic characteristics occurs at the genetic level, in what is called genetic recombination (crossover). What does not happen is interference between the colors of different features. For example, we do not crossover the eye color of the father, with the hair color of the mother, to give the resultant skin color of the child. In general, like-genes are exchanged with like-genes. Of course, mutation may cause some interference effects, but in general the phenotypic features are inherited independently (i.e. the hair color of the off-spring is determined by the hair color of the parents, independently of the genes for eye color and skin color). In genetics, contiguous sections of DNA are decoded into contiguous chains of amino acids which go on to form proteins. That is a pair of codons that appear on a string of DNA, will code for a pair of amino acids, and this pair of amino acids will appear together in the resulting protein. It therefore makes sense that a genetic operator acts on contiguous sections of DNA. That is, closely neighboring bases on a string of DNA are highly likely to be inherited together, rather than separated. Thus, crossover appears to be a very effective way of producing new individuals, which are some sort of combination of the parents.

3.3 Re-evaluation

A species is more likely to be successful in future generations if it produces more individuals in the current generation. Once the number of individuals in a population hits zero, that species becomes extinct, and will no longer successful. Increasing the numbers of a species could be thought of as the implicit aim of evolution. While evolution does not have an intention behind it, it is a little like thinking a ball rolling down a slope has the intention of getting to the bottom, or a ensemble of particles has the aim of becoming disordered (when in actual fact it is just laws of nature at work). An outside observer may therefore interpret evolution as having the goal of increasing numbers of a species. In other words, evolution does not care about re-evaluation. When the population does saturate, then chaos may start to play a role, throwing the dynamics into a complex behavior. In GP we are usually concerned with the earlier stages of evolution before the population saturates in any sense.

3.4 Limits of evolution

Evolution has produced some interesting solutions to the problems of survival. One such example is the way some bacteria reproduce. Some bacteria can reproduce twice as fast as it takes to copy its genetic material. Nature's solution to this problem is for a bacteria to effectively give birth to a bacteria which is already pregnant, thus halving the time it would take to reproduce otherwise [21].

While evolution has yielded some intriguing solutions, we should not fall into a false sense of security of thinking that it is without its limitations. One example of the limits of what cannot be achieved by evolution is illustrated by the artificial selection of bulldogs, where the preferred phenotype is a large head in proportion to the rest of the body. In fact, the head of an unborn puppy has become so large it can no longer pass through the pelvic girdle of the mother during birth unaided. As a consequence, the majority of bulldogs are born by cesarean section. This is therefore a phenotype which is highly unlikely to have evolved if evolution was left to its own devices.

Another example of the limits of evolution is the difficulty for natural selection to produce something which works on the principle of the wheel. From an engineering perspective, the wheel is simple; it is a freely rotating object connected to a spindle. In biology, we see structures similar to wheels, but are typically only able to rotate through an angle of less than one rotation. For example, the skulls of mammals can typically rotate on a bone in the neck, but in a restricted sense. The reason for this restricted movement is that blood vessels, nerve cells and other connective tissues must remain attached to the two objects (e.g. the body and the head). Perhaps the only example where nature has evolved freely rotating structures is in the flagellum of some bacteria, which is in sharp contrast to the number of examples of evolution of the eye. One may protest that evolution does not need to produce mechanisms similar to wheels, and works perfectly well without in the absence of rotating mechanisms, so this is a pointless question. However, in an artificial scenario, it is hard to see how wheel-like structures could evolve. For example in [25] the aim was to evolve agents which could move away from the origin, and here wheels would be advantageous.

3.5 Why does evolution seem so successful?

Perhaps one of the reasons why biological evolution has given rise to such vast complexity is that different species are competing both for resources (e.g. light, water, nutrition), but more importantly against each other (e.g. infection, predator-prey relationships, parasitic relationships and symbiotic relationships and niches in intricate food-webs). This effectively results in an "arms-race". Many of the factors influencing survival are concerned with, not so much the physical environment, but the threat from biological environments (i.e. there is a bigger threat to one species from other organisms rather than physical factors). Probably the best way to compete with biological evolution is to use biological evolution itself. The point is that evolution may seem highly successful in terms of the vast diversity it has produced, as it is competing with itself. That is, many of the problems a biological entity encounter will have biological solutions. For example, if a predator species evolves to run faster, then the prey species must respond by evolving faster off-spring if it is to remain successful. Similarly, companies competing with each other are well matched as they have the same set of resources and limitations (e.g. what is economically, technologically and legally possible).

3.6 Conclusion

Evolution is a remarkably process. Genetic code is a suitable representation to carry information to future generations. Crossover is a suitable way to explore combinations of different traits within a given species. However, this suitability of evolution to explore the space of biological entities, does not necessarily transfer to searching the space of computer programs. We also examined some of the limitations of evolution, and that we may be over generous in estimating the creativity of evolution as it is often solving self imposed problems in a biological arms-race. Evolution is a myopic process, which only cares about the next generation, and can be lead down blind alleys, having only a local view of the landscape. GP, if it is to be successful, needs to be long-sighted and have the end goal in mind. The take-away message of this section is that we should not overestimate the potential of evolution when applying the paradigm to searching the space of computer programs, which we examine in the next section.

4. GENETIC PROGRAMMING

4.1 Non-biological description of GP

GP is usually described using the vocabulary from genetics and evolutionary biology. GP can be completely described in standard mathematical terms, without any reference to biological terminology. For example, a population is a multi-set of programs, and a crossover operator is a stochastic binary operator mapping two programs to two programs. There is often confusion over some terminology (e.g. chromosome and its GP analogue) and a "?" is entered in the table 1. We have taken genotype to mean a program (syntax), and phenotype to mean the function that the program expresses (semantics).

One advantage of a mathematical terminology is that it allows specialists in the field of biologically related areas to communicate with people with the common interest of program induction (i.e. computer scientists) without the biological background. If we are trying to model biology, we want our computer model to be isomorphic to actual biology. In GP, we are not trying to model evolution so we do not require biological concepts have a corresponding interpretation in GP. Adopting a non-biological terminology may also help distance GP from, for example, the computational modeling of biology, which has different aims and objectives. It also frees the research from being too faithful to the underlying biology (as we are interesting in finding programs rather than modeling biology). As Freeland [26] (2003, p 310) notes: *"A luxury of EC is to bend fundamental rules of evolution beyond anything biologically plausible, and thus to answer questions where biologists have assumptions"*. We now give an example where, using biological terminology has limited the way researcher think.

Table 1: Biological terminology and their mathematical equivalent used in GP. A "?" means there is not a clear analogy.

biological	mathematical
population	multi-set of programs
individual or off-spring	program
mutation operator	unary operator
crossover operator	binary operator
selection	n-ary function
gene	instruction
chromosome	ADF?
genotype	program
phenotype	function
fitness	objective value
allele	?
species	?

In biological crossover the amount of genetic material is conserved, and this is often the case with crossover operators proposed in GP. Crossover is a function mapping from the cartesian product of programs to the cartesian product of programs; $XO : p_1 \text{ x } p_2 \rightarrow p_3 \text{ x } p_4$ where p_i is a program, subject to the constrain $S(p_1)+S(p_2) = S(p_3)+S(p_4)$, where $S(p_i)$ is the size of the ith program. A description of a more general crossover operator which does not conform to its biological counterpart just involves dropping the constrain on the sizes of the resultant pair of programs. Thus genetic material can be thought of as being copied between programs, rather than being exchanged. There are two points concerning this broader class of crossover operators. Firstly, conventional crossover is a special case of this exchange crossover. Secondly, removing a good sub-program from one program, while benefiting the recipient program, will be detrimental to the donor program. This example is a case where GP researchers have been over-faithful to the biological origins. Adopting an unbiased mathematical terminology may reduce this temptation.

4.2 Crossover

Crossover is a feature which really differentiates evolutionary methods from other search methods. In section 3.2 we gave the hair-eye-skin color example of biological crossover, where like-features are exchanged for like-features. Here, we give the analogous computational example, however this is not what occurs in GP. Imagine a pair of word-processing programs. Each contains a set of fonts, a set of drawing objects, a set of icons. A new word-processing program could be obtained by exchanging (i.e. crossing over) the fonts, drawing objects and icons. In other words like-code is exchanged for like-code. The resulting program will be a new word-processing program (i.e. it still retains the functionality of a word processing package) with distinct appearance to the participating parent programs. This is not how crossover is currently performed in GP.

It would appear that crossover is good for exchanging "like for like". But some sort of structure has to be in place beforehand, defining what "like" means (i.e. giving the genetic code or computer code a context in which to operate and produce meaning. Code in a different context has different meaning). For example, given the overall structure of a word-processing program, crossover can happily exchange like-code for like-code, but the question is, where does this overall structure first come from in the first place. GP does not currently have a good answer to this dilemma.

4.3 Number of loops

In table 2 we reference papers and the problem they applied GP to. We also list the number of loops that were found in the solutions. In some cases loop programming constructs were explicit in the primitive set (e.g `BLOCK` in [9], `forwhile` [27]). In other cases, an explicit loop construct was not included and had to be synthesized from other primitives. For example, with a TM loops are constructed from state transitions, and with a Universal Register Machine, loops are constructed from the following instructions; `compare`, `jump`, `increment`, `decrement` and `clear`.

We choose a loop to illustrate the limitations of GP for the following reasons. Firstly, loops are very important from a programming perspective as they are one of the methods we can write more general and compact code. Secondly, the contents of a loop (i.e. the body of a loop) is potentially hard to evolve, as a small change is likely to accumulate into a large error as the code is iterated over numerous times. Thirdly, there is not a gradual way to construct a loop from nothing ; you either have a loop or you do not. One missing instruction in the case of a Universal Register Machine, or a missing state in a TM , completely destroys the loop's behavior. This is similar to the difficultly of evolution attempting to evolve a wheel, you either have a wheel with the functionality of a wheel, or you have nothing, there is no in between; half a wheel is useless.

While the number of loops in a program is not a full measure of how difficult a problem is or how complex an evolved program is, we assume it as an indication. We can see a worrying trend that in the case of GP which includes explicit loop constructs; only programs containing pairs of nested loops have been evolved. When no explicit loop primitive was available, only programs containing a single loop were evolved. This perhaps indicates a glass-ceiling to the program complexity that can be achieved with GP in its current state. To our knowledge, a nested implicit loop has not been evolved. In some senses, the number of loops is a very poor measure of program complexity, as all programs can be written with the application of a single loop structure [2]. Hence in all of Teller's work with Indexed Memory, only a single loop which contains a program is ever evident.

Table 2: Table of different problems tackled, and the number of loops found in the program code of the evolved solutions. In cases where explicit loop instructions were available 2 nested loops are seen. In cases where loops had to be constructed from simpler instructions, only 1 loop was seen. A "?" is inserted where we could not establish the number of loops from the published work.

loops	problem	reference
1	multiplication	[10]
1	squares, cubes, factorial, Fibonacci	[11]
1	even parity	[12]
1	sorting, proper subtraction	[6]
1	language recognition	[7]
?	HIV data-set	[8]
2	multiplication	[9]
2	maze navigation, function regression	[14]
2	evolving data structures	[27, 28]

4.4 Manipulating syntax

How can making random changes on syntax bring about meaningful changes in semantics? By blindly bombarding syntax data structures and hoping that improvements in semantics will result is effectively "shooting in the dark". There is very little correlation between any syntactic metric (defined across program space) and any semantic metric (defined across function space), which makes the space hard to search [2]. That is, we could be a single instruction away from a program which represents the required function, but the function of that program bears no semantic relationship to a desired function. Consider of all the difficulties Cramer [9] had hand-crafting a fitness function which rewarded multiplication like behavior.

Imagine the following thought experiment. We have a GP algorithm, but instead of an algorithm performing the genetic operations of crossover and mutation, a human programmer replaces the genetic operator. Also, imagine that the programmer is not aware of the meaning of the primitives (function set and terminal set). The human programmer is only aware of feedback received in terms of how well programs perform with respect to the fitness function. The human programmer would not blindly exchange code without regard for their semantic content. Instead he would look at the functional interpretation of a sub-program and hopefully combine promising parts. For example, current crossover operators can exchange two sub-programs which, while syntactically different are semantically equivalent. Clearly such an exchange would be rather pointless. As the human programmer, we would never make such an exchange. This thought experiment is analogous to the Chinese-room thought experiment put forward by John Searle. In other words, our search through the space of programs, should be guided by the semantics of the programs, and not by their syntax.

Some attempts have been made to give context to code [17]. For example, crossover operators which transfer code between the same positions in two program trees. However, it is not the position of the code that gives it its context (i.e. meaning), but its relationship relative to the other code in the program. For example, a font library positioned at line

145 in one program, does not imply that the font library in a second word processing package is positioned also at line 145 (see the word processing package example earlier).

4.5 Stochastic solution, deterministic problem

In many instances the target problem does not have a stochastic element (e.g. the traveling salesman problem). It seems inappropriate to want to tackle such a problem with a stochastic search process. That is, given an instance of a problem, an evolutionary computation approach will almost certainly give a different answer every time due to the probabilistic way the search is conducted. There are other machine learning methods which would not introduce this probabilistic uncertainty into the solution. For example, the C4.5 [29] is an algorithm for generating decision-trees by reducing the entropy at each node in the decision-tree. If the algorithm is run a second time, it will give the same answer as it is deterministic algorithm.

GP is not the only culprit of attempting to solve a deterministic problem with a probabilistic approach. Many current search methods are stochastic e.g. simulated annealing, and artificial neural networks (ANNs). Support vector machines (SVM) have gone some way to address this issue in ANNs. While an ANN effectively moves a decision boundary around by adjusting connection weights, with SVM, a decision boundary is calculated directly from the given class points. The ANN approach may return a whole range of possible solutions, but the SVM approach returns a single unique solution. Thus, what is needed is an analogous method in GP, which removes the randomness. One may argue that Levin search is such a solution.

5. SUMMARY

A literature review has revealed that only a small fraction of the papers on GP deal with evolving TE computer programs, with the ability to iterate and utilize memory, while the majority of papers deal with evolving less expressive logical or arithmetic functions. This suggests a reluctance in the field to tackle these more challenging search spaces, and the aim of this paper is to unravel why.

We revisit evolution, which is the inspiration for GP. We find that genetic code is suited to the task it performs (i.e. robustly carrying information from one generation to the next), and that genetic recombination is an effective way of producing viable off-spring which are phenotypically different to the parents, thus providing variation. It is brought to the reader's attention, however, that evolution does have its limits, and while it has given rise to an incredible array of diversity, and therefore apparent potential creativity which we would like to harness in GP, this is largely due to having to solve "self-imposed" problems. Evolution is in an arms-race with itself; competing species are largely hill-climbing in the space of viable biological entities, and small changes in one species mutually drives small changes in another species (either cooperatively or competitively), giving the impression of some sort of evolutionary progress.

Returning to GP, while the goal of GP is not to replicate evolutionary processes, we believe the biological nomenclature should be super-seeded by an unbiased abstract mathematical language. This will free researchers of the emotional baggage of being unnecessarily faithful to evolution. A broader class of crossover operator was proposed to illustrate this misplaced trust. Crossover in GP fails to be as effective as its biological counterpart, as it does not exchange like-code for like-code (i.e. the semantics of the exchanged code is not comparable) as there is currently no information about context available. How can a crossover operator "know" if code performing a certain function at some location in one program will perform the same function at a different location in a different program? We examine what has been achieved in the literature, and find a worrying trend that largely small toy-problems been attempted which require only a few line of code to solve by hand. Also, GP is a stochastic answer to a deterministic question, and therefore such a methodology is questionable. Surely a deterministic problem requires a deterministic solution method.

6. CONCLUSIONS

If we are to work toward a general theory of Artificial Intelligence (AI), it should certainly be able to make claims concerning program induction using TE representations. Langdon et. al. [30] (page IX) illustrate current AI approaches as separate islands in a sea, and the foundations of AI existing below the sea. As a TE language is the most expressive representation, we suggest that an understanding of the induction of functions using TE representations must lie deep within such a theory. A theory of AI based on TMs should encompass a theory of induction based on finite-state automata or push-down automata, as what these automata can express are subsets of what a Turing Machine can express. Vallejo et. al. [8] say, *'a GP based on the evolution of Turing Machines using genetic algorithms provides a convenient framework for understanding the fundamental theoretical capabilities and limitations of GP'*. We believe, not only should a theory of AI include induction using TE representations, it should be central.

A paradigm shift in program induction away from GP is well overdue. There is a distinct lack of ambitious results in the area of TE evolution in the literature (i.e. most applications are toy problems). We conclude that GP in its current form is fundamentally flawed, when applied to the space of TE programs. This paper could be interpreted as an attack on GP. Rather, it is intended to be viewed as identifying the weaknesses in the approach which need to be addressed if the field is to progress from its current stagnating phase. If we are seeking a program based on its sematic merits, then we should use a search method which is guided by semantics, rather than dominant syntactic gradient-decent methods which are currently employed.

7. REFERENCES

[1] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[2] Astro 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 June 1994. IEEE Press.

[3] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann, dpunkt.verlag, January 1998.

[4] Peter J. Angeline. A historical perspective on the evolution of executable structures. *Fundamenta Informaticae*, 35(1–4):179–195, August 1998.

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

[6] Julio Tanomaru and Akio Azuma. Automatic generation of turing machines by a genetic approach. In Daniel Borrajo and Pedro Isasi, editors, *The First International Workshop on Machine Learning, Forecasting, and Optimization (MALFO96)*, pages 173–184, Gatafe, Spain, 10–12 July 1996.

[7] Julio Tanomaru. Evolving turing machines from examples. In J.-K. Hao, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers, editors, *Artificial Evolution*, volume 1363 of *LNCS*, Nimes, France, October 1993. Springer-Verlag.

[8] Edgar E. Vallejo and Fernando Ramos. Evolving turing machines for biosequences recognition and analysis. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 192–203, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.

[9] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In John 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.

[10] Lorenz Huelsbergen. Toward simulated evolution of machine language iteration. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 315–320, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[11] Lorenz Huelsbergen. Learning recursive sequences via evolution of machine-language programs. In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 186–194, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[12] Lorenz Huelsbergen. Finding general solutions to the parity problem by evolving machine-language representations. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 158–166, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[13] Jurgen Schmidhuber and Fakultat Fur Informatik. On learning how to learn learning strategies, February 01 1995.

[14] Juergen Schmidhuber, Jieyu Zhao, and Marco Wiering. Simple principles of metalearning. Technical Report IDSIA-69-96, IDSIA, Lugano, Switzerland, Corso Elvezia 36, CH-6900, Switzerland, June 27 1996.

[15] Jurgen Schmidhuber, Jieyu Zhao, and Marco Wiering. Shifting inductive bias with success-story algorithm, adaptive levin search, and incremental self-improvement. *Machine Learning*, 28:105, 1997.

[16] Peter Nordin, Wolfgang Banzhaf, Fachbereich Informatik, Fachbereich Informatik, Lehrstuhl Fur Systemanalyse, and Lehrstuhl Fur Systemanalyse. Evolving turing-complete programs for a register machine with self-modifying code. In *Genetic algorithms: proceedings of the sixth international conference (ICGA95*, pages 318–325. Morgan Kaufmann, 1995.

[17] Peter Nordin, Wolfgang Banzhaf, and Frank D. Francone. Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 12, pages 275–299. MIT Press, Cambridge, MA, USA, June 1999.

[18] Astro Teller. The evolution of mental models. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 9, pages 199–219. MIT Press, 1994.

[19] W. B. Langdon. Evolving data structures using genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 295–302, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[20] William B. Langdon. Data structures and genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 20, pages 395–414. MIT Press, Cambridge, MA, USA, 1996.

[21] Franklin M Harold. *The way of the cell : molecules, organisms, and the order of life*. Oxford University Press, Oxford, 2001.

[22] Guy Sella and David H. Ardell. The coevolution of genes and genetic codes: CrickŠs frozen accident revisited. *Journal of Molecular Evolution*, pages 297–313.

[23] Freeland S.J; Wu T; Keulmann N. The case for an error minimizing standard genetic code. pages 457–477.

[24] G Trinquier and YH Sanejouand. Which effective property of amino acids is best preserved by the genetic code? *Protein Engineering*, pages 153–169, 1998.

[25] K. Sims. Evolving 3d morphology and behavior by competition. *Artificial Life*, 1(4):353–372, 1994.

[26] Stephen Freeland. Three fundamentals of the biological genetic algorithm. In Rick L. Riolo and Bill Worzel, editors, *Genetic Programming Theory and Practice*, chapter 19, pages 303–312. Kluwer, 2003.

[27] William B. Langdon. Evolving data structures with genetic programming. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 295–302. Morgan Kaufmann, 1995.

[28] W. B. Langdon. Data structures and genetic programming. Research Note RN/95/70, University College London, Gower Street, London WC1E 6BT, UK, September 1995.

[29] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[30] W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.