GA or GP? That is *not* the question.

John R. Woodward
School of Computer Science
The University of Birmingham
B15 2TT UK
J.R.Woodward@cs.bham.ac.uk

Abstract-

Genetic Algorithms (GAs) and Genetic Programming (GP) are often considered as seperate but related fields. Typically, GAs use a fixed length linear representation, whereas GP uses a variable size tree representation. This paper argues that the differences are unimportant. Firstly, variable length actually means variable length up to some *fixed* limit, so can really be considered as fixed length. Secondly, the representations and genetic operators of GA and GP appear different, however ultimately it is a population of bit strings in the computers memory which is being manipulated whether it is GA or GP which is being run on the computer.

The important difference lies in the interpretation of the representation; if there is a one to one mapping between the description of an object and the object itself (as is the case with the representation of numbers), or a many to one mapping (as is the case with the representation of programs). This has ramifications for the validity of the No Free Lunch theorem, which is valid in the first case but not in the second. It is argued that due to the highly related nature of GAs and GP, that many of the empirical results discovered in one field will apply to the other field, for example maintaining high diversity in a population to improve performance.

1 Introduction

1.1 Evolutionary Computation

Evolutionary Computation (EC) is an umbrella term covering GAs, GP, Evolutionary Strategies and Evolutionary Programming. While each of these areas are related, there are differences [BNKF98, DFS97]. The arguments presented in this paper can be applied to each of these areas in an attempt to unify the field rather than have it fragment. We will be concerned with GAs and GP.

Standard GAs use a fixed length linear representation and makes heavy use of crossover. Usually the representation is binary but higher arity alphabets have been used. In contrast, GP uses a variable length tree structure, however like GAs, GP also makes heavy use of crossover. Here only linear and tree structures are considered, but other representations have been used including graph based representations.

Recently there has been some debate within the GAs and GP communities regarding the relationship between GAs and GP. Mitchell [Mit96] (page 269) states "GP is a variant of GAs in which the hypothesis being manipulated are computer programs rather than bit strings". Ultimately computer programs are essentially bit strings so this distinction is not clear. Langdon [LP02] (in the preface) state "since GP is more expressive than GAs, it (GP theory) can be viewed as a generalization of GAs theory". Whether GP is a generalization of GAs or not will depend on how the bit strings are interpreted. Reeves and Rowe [RR02] (page 2) are more cautious in their introduction stating that they do not want to give the impression that other areas of EC are unimportant to GAs theory. Banzhaf et. al. [BNKF98] also describe some of the differences. What is certain is that it is unclear what the exact differences are.

In order for a field to advance a framework is required with which to work. Part of a framework is the terminology used, and getting the terminology correct is an essential part of the process. It is clear that GAs and GP are related as they are both inspired by Darwinian evolution. It is also clear that GAs and GP are regarded as seperate fields. They have seperate conferences and seperate text books [BNKF98, Gol89, Koz92, Mit96]. (Though many researchers in one field may attend conferences in the other field and publish in both fields). While accurate classification is important, over fragmentation can mean that related areas become separated, and this will have a detrimental effect on the progress of both fields.

1.2 The Terms GAs and GP

In a sense the terms GAs and GP are misleading. GAs do not necessarily represent the evolution of algorithms, but rather a subset of all possible algorithms. GP typically does not evolve computer programs as we would expect, but typically logical or mathematical expressions are evolved, or `if then` type rules of a classifier system. Real world human produced programs often involve the use of iteration and memory which has not been fully explored in GP [Woo03].

1.3 Is it GA or is it GP?

Consider the following general scenario. A population of fixed length bit strings are evolved using the process of selection followed by mutation and crossover (one point crossover and uniform crossover). The fitness of an individual bit string in the population is given by some cost function which, given a bit string, returns a real value. This test and generate cycle is iterated until some termination condition is met. This general scenario could even be considered as the definition of GAs: the individuals in the population are fixed length bit strings and (for the sake of argument) the crossover rate is high compared to the mutation rate.

Now, if the cost function is interpreting the bit string as a computer program and the value returned reflects the performance of the program at a particular task, one would be inclined to say we have been describing a GP system. (For example the bit strings could be fed into a Universal Turing Machine). This senario would appear to blur the distinction between GAs and GP.

In a seminal paper, Cramer [Cra85] evolved programs first as bit strings but comments on the epistatic nature and moves to a tree based representation. The question is perhaps not what the representation is (i.e. a bit string or a tree) but rather the interpretation of the representation.

1.4 Outline of Paper

The outline of the paper is as follows. In sec. 2 the apparent differences between GA and GP are discussed. In sec. 3 a fundamental difference is pointed out between the representation of number and programs is considered, and in sec. 4 the implications of this regarding No Free Lunch are considered. Following this there is a discussion in sec. 5 and a summary in sec 6.

2 Apparent Differences

While there are no strictly agreed upon definition about what GAs and GP are, there are some fundamental differences. Firstly we look at the issue of fixed or variable length representation, and then the issue of the representation (bit strings or trees). Finally the operators used to move around the corresponding search spaces are considered.

2.1 Fixed and Variable Length

Possibly the greatest distinction between GAs and GP is that of fixed or variable length. In some cases, the size of the required solution sought may be known beforehand, for example the traveling salesman problem, where the problem is to find the shortest route listing all cities. Clearly any proposed path must include all cities once and only once. However, there are many problems where it is difficult to prespecify the size of the solution. Clearly, if we know the size of the solution we do not need to use a variable length representation as this would make the search space larger.

Usually a maximum depth or size of tree is imposed in GP to avoid memory overflow (caused by bloat). During a run, the size of trees tends to increase and this needs to be controlled. Even if an upper limit on the size of a tree is not imposed, there is an effective upper limit which is dictated by the finite memory of the machine on which the GP is being executed. So in reality, GP has variable size up to some limit.

There is no reason why the size of a bit string in GAs cannot vary during the evolution. Both crossover and mutation operators, which operate on fixed length structures, can be engineered into operators which produce variable length bit strings.

Conversely, with GP there is no reason why fixed size GP cannot be implemented. For example, when crossover selects points in different parents, crossover can only take place if the subtrees that are to be exchanged are of the same size. This guarantees that each program retains its original size (thought its shape will change). This could be called 'fixed size GP' This is very similar to one point crossover (described in [LP02]), except that the same crossover point does not need to be chosen in both of the parents.

It may in some cases be confusing to talk about fixed or variable length representations. Consider a binary representation of integers. All 01, 001, 0001 represent the value one. This is a variable length representation, however all the leading 0's are effectively redundant. But if we consider a fixed length of e.g. 4 bits, this encompasses all the bit strings just mentioned. Hence when evolving binary representations of integers, it is effectively a variable length representation up to some fixed limit as leading 0's do not contribute and can be ignored.

Huelsburgen [Hue96] evolves an assembly type language where the individuals are of fixed length, however he includes a 'NOP' operation ('no operation') which effectively does nothing. This makes an apparently fixed length representation into a variable length one (with some fixed upper bound). If we look at 'fixed size GP' (defined above) there will in general be subbranches which will not contribute to the overall output of the tree, whether or not there is an explicit 'NOP' operation. A subtree may never be executed, for example, we could have and `if then else` statement which is always true so the `else` branch is never executed. Subbraches which do not contribute are called introns and are unavoidable in evolution. Hence an apparently fixed length representation is effectively variable length up to some fixed limit. There is no truly variable length representation, they are only variable up to some fixed limit.

2.2 Representation

The representation and the genetic operators used to manipulate the representation are central to the success of a search algorithm. The landscape metaphor, often used to explain the search process, is defined by which points in the search space are connected to which other points. The connectivity is defined by the representation and the genetic operators (see chapter 2 [LP02]).

In GAs, the mutation of bit strings takes the form of flipping a single bit. Two main crossover operators have been used: uniform crossover uniformly selects a bit from each position from one of the two parents, one point crossover selects a point at random along the genome of two parents and takes the left part of parent one and combines this with the right part parent two. In GP, mutation takes the form of replacing a randomly selected sub tree in a program with a randomly generated sub tree. Crossover swaps randomly selected sub trees of two programs. There are of course other more sophisticated operators, each of which has their success and failures.

The operators used to move around these two search spaces seem very different, but ultimately it is bit strings being manipulated in the computers memory (see figure 1). GAs can be thought of as manipulating bit strings in their primitive form. GP however manipulates tree structures, and while it may not be initially obvious what is happening at the bit level, this is indeed what is happening. It is possible to design GP operators which perform the same operations on bit strings and are therefore equivalent to GAs. (These operators will look very different from the traditional operators but there is no reason why we should restrict ourselves to the traditional operators). Tree structures map to bit strings, so we can apply operations at the tree level and map these to bit strings and can therefore interpret operations on trees as operations on bit strings.

A good example of this is the work by Banzhaf et. al. [NB95], who evolves computer programs by directly manipulating the bits at the machine level. The motivation for this work is to avoid the interpretation stage standard GP has to go through.

2.3 Genetic Operators

One important potential difference between GA and GP is the effect of crossover. In GA, the crossover operators can move genetic material from either of the parents, but usually places it in the same location in the child (i.e. the position of the gene in the genotype is not altered by crossover). Thus crossover does not move the location of a bit within a bit string. (There is of course no reason why this should be the case unless the interpretation of the bit string does not allow this). The crossover operator in GP typically moves a subtree from one parent to a different location in the child. In GP, subtrees can be interpreted anywhere in the overall tree. A GP crossover operator has been proposed which preserves the location of any subtree crossed over. The same randomly selected crossover point in two parents is chosen and therefore the crossed over subtrees will have the same location in the offspring as they did in the parents [LP02].

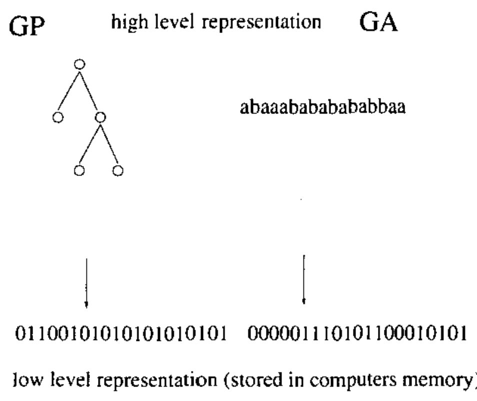
GP
high level representation
GA
abaaababababababbaa
01100101010101010101
000001110101100010101
low level representation (stored in computers memory)

Figure 1: GA (on the right) uses a string representation which is stored in a computer as a bit string. In general, the actual bit string stored in memory will be different to the actual string (which may be constructed from an alphabet with more symbols) which the GA is manipulating. GP (on the left) uses a tree based representation which is ultimately stored in memory as a bit string. GP can be thought of as GAs as GP is the evolution of bit strings in the computers memory.

3 Numbers or Programs

In this section a fundamental difference between the representation of numbers and programs in pointed out. There are many ways to represent numbers, however typically there is a one to one mapping between the representation and the number being represented. With programs (be they classifier systems, artificial neural networks, finite state machines or Turing Complete programs) there is a many to one mapping between the representation and the program being represented. This difference has consequences when searching the space. If the mapping between the representation and the object being represented is one to one, a uniform sampling of the representation will lead to a *uniform* sampling of the objects being represented. If the mapping between the representation and the object being represented is a many to one mapping, a uniform sampling of the representation will lead to a *non uniform* sampling of the objects being represented.

3.1 Numbers

If our aim is to optimize a function, we typically choose some way to represent numbers which are input to the function. (For the sake of argument let the function inputs be positive integers). In GA common representations are binary and grey scale. These are both one to one representations, a number has only one bit string corresponding to it and vice versa. Typically there is a one to one mapping between the bit strings GA evolves and what they represent (in this case numbers) [LP02] (page 194). In practice the distinction between genotype and phenotype is blurred due to this one to one mapping between description and object being described [RR02] (page 20). Note that an even sampling of bit strings translates to an even distribution of the integers which are represented.

One could use GP to optimise a function. A population of trees, which represent numbers, could be evolved to search the space. For example, given a function set { +, -, * } and a terminal set {1} the space of integers could be explored. (Typically in a GP the terminal set consists of variables from the problem, but there is no reason why numbers cannot be used). Unlike the case of GAs, each number can be represented many ways. For example, the value 2 could be + 1 1 or * + 1 1 1 (i.e. 1*1 + 1) among others. Hence there is a many to one mapping between the space of trees and the space of numbers. We also point out that this mapping is not uniform. If the space of trees is uniformly sampled this will lead to a non uniform sampling of the integers represented. (Note that it is not suggested that this is an efficient way to optimise a function, it is just used to illustrate one of the differences between GA and GP).

If we know nothing about the function we are trying to optimise then clearly we should choose a one to one representation of numbers. It would require a very specialist knowledge of the function to favour a many to one representation over a one to one representation.

3.2 Programs

The general problem of program induction involves finding a program on which the error score on a set of test cases is zero (or within some tolerance limit). The set of test cases defines the problem. Each program receives a score based on its performance on the test cases, hence each program maps to an error score. The representations used here could be of many forms, for example classifier systems, logical or mathematical expressions, artificial neural networks, finite state automata or computer programs (the list is not exhaustive).

For each of these representations GA could be used to evolve the representation. Fixed length bit strings could be used to represent any of these forms and evolved in the standard way. For example bit string could be used to represent the weights in an artificial neural network [Mit96, Yao99]. Computer programs could be represented as bit strings fed into a Universal Turing Machine. GP could be used to represent logical expressions, given a function set of logical operators. GP could also be used to represent computer programs as trees.

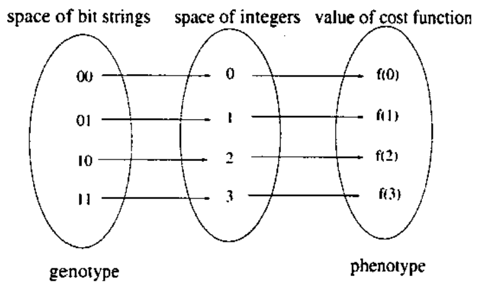

Figure 2: There is a problem independent one to one mapping between bit strings (left ellipse) and integers (middle ellipse). There is a mapping between the space of integers (middle ellipse) and the function values (right ellipse) which depends on the problem. This mapping may be one to one or many to one depending on the problem. Over all problems NFL.

However, irrespective of whether GA or GP is used there will always be a many to one mapping between the description and the object being described. In some cases it may be possible to determine if two instances of a representation are equivalent before they are executed. For example, an algorithm exists that determines if two finite state automata are equivalent [HU79] (page 64). An important question is to determine if it is better to execute the programs on the test cases or determine if they are equivalent using some analytic technique. If the programs being evolved are Turing Complete it is impossible to determine if two programs are equivalent (due to Rice's Theorem [HU79]page 185).

In general the mapping from programs to the functions they compute is a non-uniform many to one mapping. If a uniform sample of the representation is taken this will map to a non-uniform distribution over the functions the programs represent. (Essentially simple program will have many descriptions and more complicated programs will have fewer descriptions). Langdon [Lan99] has investigated the distribution of functionality with varying program size. Using either enumeration or sampling techniques, he plots the frequency that a function is represented. The conclusion is that above some threshold, the distribution of performance is independent of program length. Simple visual examination of these graphs reveals that these distributions are non-uniform.

4 No Free Lunch

The No Free Lunch theorem (NFL) is a central theorem in search which is often taken to mean all search algorithms perform equally over all problems. (see [SVW01, WN03]). This section first introduces the idea of the theorem, then goes on to examine it in the context of a one to one and a many to one mapping between representation and the objects being represented. NFL is valid in the case of a one to one mapping. However when there is a non uniform many to one mapping between representation and the objects being represented it is suggested that NFL is not valid. With GA, either situation can occur, however the second situation is *always* the case with the representations used in GP.

4.1 No Free Lunch

Let D and R be finite sets and $f : D \to R$ be a function where $r_i \equiv f(d_i)$. D is the domain and R is the range. The size of D is $|D|$ and the size of R is $|R|$. Each value in D maps to a single value in R. For a given D and R there are $|R|^{|D|}$ possible functions (see figure 4).

A search operator and a search algorithm are taken to be the same and represent any algorithm that produces a search vector. A search vector V is an ordered sequence of points in D; $V \equiv \langle d_1, d_2, \ldots, d_m \rangle$. It is assumed that no point is revisited. The ith element in this vector corresponds to the ith point visited. A complete search vector is any vector that lists all points in D once and only once and has length $|D|$. There are $|D|!$ distinct complete search vectors. We are not concerned with how the search vectors are generated.

A search vector corresponds to a path in D. A given search vector and function will produce a corresponding path in R. Let us call this sequence of points in R a performance vector. A search vector of length l corresponds to a performance vector of length l, given a specific function.

In [SVW01] four equivalent statements of NFL are discussed, the third is stated here;

Theorem NFL: Every search algorithm generates precisely the same collection of performance vectors when all functions are considered.

4.2 One to One Representations

There are a number of examples of representations where the mapping between the representation and the object being represented is one to one. Perhaps the most familiar is that of bit strings and numbers. A general situation is shown in figure 2. The ellipse on the left shows the space of bit strings (of length 2 in this case). Each of these bit strings maps onto a non-negative integer in the ellipse in the centre. This mapping is one to one and is problem independent. The mapping between the space of non-negative integers and the cost function f is problem dependent (in fact f is the definition of the problem). Here each point in the ellipse on the right is shown as a separate point, but in general these values could be the same or different. NFL is valid between the space of integers and the space of the values of the cost function if all cost functions are considered. NFL is valid between the space of bit strings and the space

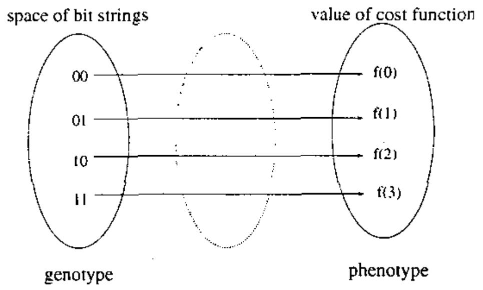

Figure 3: Often in GAs the mapping between genotype (bit strings) and phenotype is one to one. The distinction is blurred and the bit strings are directly mapped to function values. This is shown by removing the central ellipse which is the interpretation stage. NFL is valid between the space of bit strings (left ellipse) and the space of values of the cost function (right ellipse).

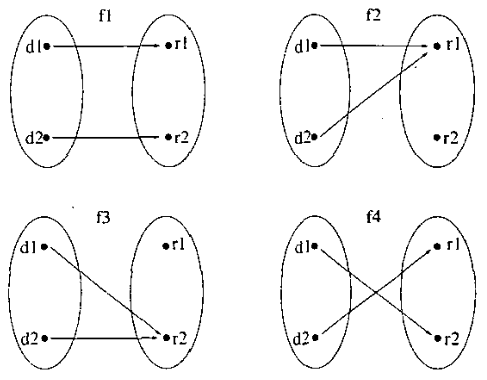

Figure 4: All possible functions between two sets which only contain two points are shown.

of the values of the cost function if the mapping between bit strings and integers is one to one and all cost functions are considered.

As the mapping between bit strings and integers is so trivial, (we could even call it transparent), it is often forgotten and this is where the blurring between genotype and phenotype occurs. This is shown in figure 3 where the bit strings are translated directly into values returned by the cost function. In fact in some cases the problems are even considered to be a function of the bit strings themselves ([RR02] Appendix A).

Figure 4 shows all possible functions between two sets which, in this case only contain two points each. Let us illustrate NFL by simply listing all the performance vectors for all of these functions. There are only two possible search vectors $V_1 \equiv \langle d_1, d_2 \rangle$ and $V_2 \equiv \langle d_2, d_1 \rangle$. Let us consider all of the functions f_1, f_2, f_3, f_4. V_1 produces the

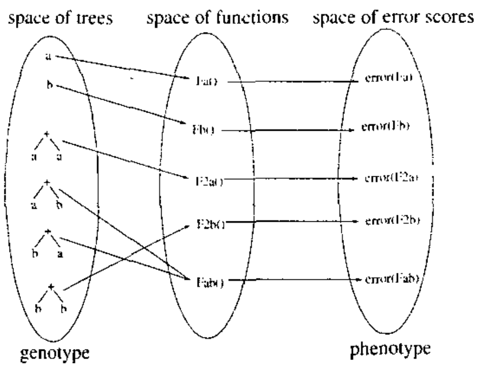

Figure 5: There is a problem independent many to one mapping between trees (left ellipse) and the functions they represent (middle ellipse). There is a mapping between the functions they represent (middle ellipse) and the function values (right ellipse) which depends on the problem. Due to the non uniform nature of the mapping between these spaces (the left ellipse and right ellipse), not all functions exist and NFL is not valid.

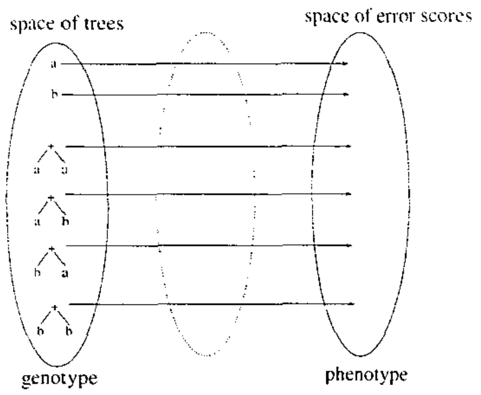

Figure 6: In GP there is a many to one mapping between the representation and what is being represented. A tree is interpreted as a function which recieves and error score. Due to this interpretation stage, not all functions between the space of trees and space of error scores exist. NFL is not valid.

performance vectors $\langle r_1, r_2\rangle$, $\langle r_1, r_1\rangle$, $\langle r_2, r_2\rangle$, and $\langle r_2, r_1\rangle$ respectively. V_2 produces the performance vectors $\langle r_2, r_1\rangle$, $\langle r_1, r_1\rangle$, $\langle r_2, r_2\rangle$, and $\langle r_1, r_2\rangle$ respectively. By inspection we can see that the same collection of performance vectors is produced by each of the search vectors.

4.3 Many to One Representations

In GP, the mapping between the genotype and phenotype is always many to one. A familiar example is function regression where, for a given function and terminal set, there are many ways to express the same function. Consider a function set given a function set { + } and a terminal set $\{a, b\}$ and trees up to a maximum size of 3. All possible trees (six in total) are shown in the left ellipse of figure 5. This set of trees maps onto a set of possible functions listed in the central ellipse. The mapping between the space of trees and the space of functions they represent is independent of the problem and depends only on the function and terminal set. The mapping between these two sets is many to one and is non-uniform. All functions are represented once except the function $a + b$ which is represented twice (by the trees $a + b$ and $b + a$). No test, based on the functionality of these two trees, will differentiate them as they are functionally equivalent, i.e. for all inputs they will produce the same output. The mapping between the space of functions (central ellipse) and the cost function $error(.)$ (right ellipse) is problem dependent. In fact $error(.)$ is the problem itself (defined by the test cases). Typically, the values this function returns is called the error score in GP. The actual values returned by the cost function will depend on the test cases. Here each point in the ellipse on the right is shown as separate points, but in general these values could be the same depending on the cost function $error(.)$. NFL is valid between the space of functions (central ellipse) and the space of the values of the cost function (right ellipse) if all cost functions are considered. However, NFL is not valid between the space of trees (left ellipse) and the space of the values of the cost function (right ellipse) due to the non-uniform many to one mapping between these spaces.

To illustrate this, let us consider an example. Imagine a search algorithm, P, which visits the trees in the search space in the following order: $+ba, a, b, +aa, +bb, +ab$, and a second search algorithm, Q, which visits the trees in the search space in the following order: $+ab, +ba, a, b, +aa, +bb,$. With P, each of the first five trees represent a different functionality, and only the last tree ($+ab$) represents a duplicate functionality (the same as the first tree visited). With Q, the first two trees it visits ($+ab, +ba$), both have the same functionality and cannot be distinguished. If the problem requires us to find a tree with functionality $+ab$, both P and Q visit such a tree with their first visit. However, for any other function, P will always visit a tree that performs a function before Q. Thus over all problems, P will outperform Q.

As the mapping between trees and functions is non-trivial, it requires an interpretation stage. Each function is mapped to by many different trees. An impossible situation is shown in figure 6 where each tree receives a different value by the cost function. The trees $+ab$ and $+ba$ must map to the same error score, which invalidates NFL.

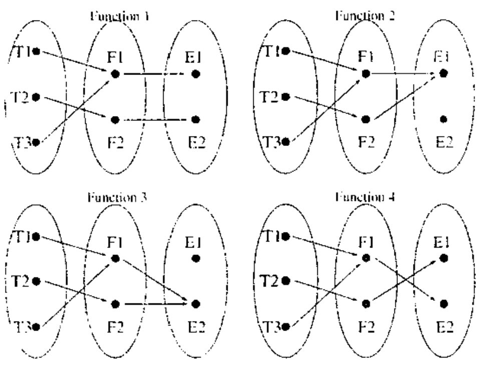

Figure 7: The space of trees T maps to the space of functions F which maps to the space of error scores E. The mapping between the space of trees is fixed. All possible functions between space of functions F and the space of error scores E are shown.

Figure 7 shows all the mappings that are possible between a set of trees to a set of functions then onto a set of error scores. In this example, tree T_1 has functionality F_1, tree T_2 has functionality F_2 and tree T_3 has functionality F_1. This is a fixed mapping. All possible mappings between the space of functions, F, and the space of error scores E is shown. Let us illustrate that NFL does not hold in this case by simply listing all the performance vectors for all of these cases. There are 6 possible search vectors but we only need to consider two to show that there are differences. Consider $V_1 \equiv \langle T_1, T_2, T_3 \rangle$ and $V_2 \equiv \langle T_1, T_3, T_2 \rangle$. V_1 produces the performance vectors $\langle E_1, E_2, E_1 \rangle$, $\langle E_1, E_1, E_1 \rangle$, $\langle E_2, E_2, E_2 \rangle$, $\langle E_2, E_1, E_2 \rangle$ respectively. V_2 produces the performance vectors $\langle E_1, E_1, E_2 \rangle$, $\langle E_1, E_1, E_1 \rangle$, $\langle E_2, E_2, E_2 \rangle$, $\langle E_2, E_2, E_1 \rangle$ respectively. By inspection we can see that different collections of performance vectors are produced by the two search vectors. This difference is due to the many to one mapping between the descriptions of functions (i.e. trees) and the functions they represent.

5 Discussion

NFL is a central theorem in search. This paper suggests that NFL is not valid for the representations used in GP due to the non-uniform many to one mapping between the description of an object and the object itself. The majority of representations used with GA are one to one (and therefore uniform) between the description of an object and the object itself. The many to one problem of the genotype to phenotype mapping is discussed in [Yao99] in the context of evolving artificial neural nets. Essentially any permutation of hidden nodes produces a neural net with the same functionality.

GA and GP are largely studied empirically. General observations made about one may apply to the other as they are so highly related. For example in [RR02] (page 59) a number of guidelines are given to improve the performance of a GA. These suggestions include using an incremental approach to selection rather than a generational approach and maintaining a high diversity in the population. Due to the highly related nature of GAs and GP, we suggest that most of these recommendations (if not all!) will apply to GP.

This paper has concentrated on GA and GP, however there are other paradigms also included within EC, namely Evolutionary Programming (EP) and Evolutionary Strategies (ES). It is suggested for similar reasons that separating EP and ES from other fields is detrimental to the development of EC as a whole.

6 Summary

GAs and GP are considered related but different algorithms. GAs use fixed length bit strings as their representation, GP uses variable length trees as theirs. The differences between GAs and GP are discussed; the representation, fixed or variable length genotype and the genetic operators. It is suggested that these differences are unimportant. Whether we are considering GAs or GP, whatever the high level representation is, it will ultimately be stored in the computer's memory as a bit string. GP uses a variable length representation, but an imposed finite upper limit essentially makes it a fixed length representation. The genetic operators used in GA and GP appear different, GA operators usually do not move the location of the gene within the genotype. However there is no reason why this should be the case. Conversely, GP operators may move the location of a subtree to a different location, but again there is no reason why this should be the case.

The important difference lies in the interpretation of the representation, whether it is a one to one mapping between the description and the object being represented, or a many to one mapping between the description and the object being represented. The difference between these two types of mappings is discussed in the context of number and programs. It is due to this many to one mapping that NFL is invalidated.

In GAs the mapping between the description and the object being represented may be one to one or many to one. In GP the mapping between the description and the object being represented is always many to one and non uniform. *GA or GP, that is not the question.* What is the question is what type of representation we are dealing with.

7 Acknowledgments

I wish to thank James Foster.

Bibliography

[BNKF98] 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.

[Cra85] N. L. Cramer. A representation for the adaptive generation of simple programs. In *International Conference on Genetic Algorithms and Their Applications.*, pages 183–187, July 1985.

[DFS97] Kenneth De Jong, David B. Fogel, and Hans-Paul Schwefel. A history of evolutionary computation. In Thomas Back, David B. Fogel, and Zbigniew Michalewicz, editors, *Handbook of Evolutionary Computation*, pages A2.3:1–12. Institute of Physics Publishing and Oxford University Press, Bristol, New York, 1997.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[HU79] JE Hopcroft and JD Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[Hue96] Lorenz Huelsbergen. Toward simulated evolution of machine language iteration. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 315–320, Stanford University, CA, USA, 28–31 1996. MIT Press.

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[Lan99] William B. Langdon. Scaling of program fitness spaces. *Evolutionary Computation*, 7(4), 1999.

[LP02] W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[Mit96] Melanie Mitchell. *An Introduction to Genetic Algorithms*. Complex Adaptive Systems. MIT-Press, Cambridge, 1996.

[NB95] Peter Nordin and Wolfgang Banzhaf. Evolving turing-complete programs for a register machine with self-modifying code. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 318–325, Pittsburgh, PA, USA, 15-19 1995. Morgan Kaufmann.

[RR02] Colin R. Reeves and Jonathan E. Rowe. *Genetic Algorithms - Principles and Perspectives A Guide to GA Theory*. Kluwer, 2002.

[SVW01] C. Schumacher, M. D. Vose, and L. D. Whitley. The no free lunch and problem description length. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 565–570, San Francisco, CA 94104, USA, 7-11 July 2001. Morgan Kaufmann.

[WN03] J. R. Woodward and J. R. Neil. No free lunch, program induction and combinatorial problems. In *Genetic Programming, Proceedings of EuroGP 2003*, Essex, UK, 14-16 apr 2003. Springer-Verlag.

[Woo03] Woodward. Evolving turing complete representations. In *Congress on Evolutionary Computation*, 2003.

[Yao99] X. Yao. Evolving artificial neural networks. *IEEE: Proceedings of the IEEE*, 87:1423–1447, 1999.