Biomimetic Representation with Enzyme Genetic Programming

Michael A. Lones (michael.lones@bioinspired.com)
Department of Electronics, University of York, York, YO10 5DD, UK

Andy M. Tyrrell (andy.tyrrell@bioinspired.com)
Department of Electronics, University of York, York, YO10 5DD, UK

Abstract. The standard parse tree representation of genetic programming, whilst a good choice from a generative viewpoint, does not capture the variational demands of evolution. This paper addresses the issue of whether representations in genetic programming might be improved by mimicry of biological behaviours, particularly those thought to be important in the evolution of metabolic pathways, the 'computational' structures of the cell. This issue is broached through a presentation of enzyme genetic programming, a form of genetic programming which uses a biomimetic representation. Evaluation upon problems in combinational logic design does not show any significant performance advantage over other approaches, though does demonstrate a number of interesting behaviours including the preclusion of bloat.

Keywords: genetic programming, biomimetic representation

1. Introduction

Solution representation is an important issue in virtually every domain of machine learning and artificial intelligence. Evolutionary computation is no exception. Representations in evolutionary computation are subject to two demands: generation and variation (Altenberg, 1994b); requiring that they both express a solution and react to changes in the encoding of this solution in a meaningful way. It is natural to choose a representation that reflects generative demands, since these are the same representations found in conventional, often manual, approaches to the same problem. However, ignoring variational demands can have profound effect upon navigation of the problem's search space; to the extent of making an easy problem difficult or a difficult problem intractable. For genetic programming (GP), the conventional parse tree representation is generative in nature and does not respond well to genetic operators. In this sense, it lacks evolvability.

A representation that captures both generative and variational demands can make solving a hard problem easier. This paper presents a genetic programming system, enzyme genetic programming, that uses biomimetic representations in an attempt to capture these variational demands and improve the evolution of executable structures.

The paper is organised as follows. Section 2 describes the representation problem in GP. Section 3 reviews biological concepts and

© 2002 *Kluwer Academic Publishers. Printed in the Netherlands.*

discusses how the mechanisms and representations of biology support effective evolution. Section 4 introduces enzyme genetic programming and section 5 presents the implementations of enzyme GP used in this study. Section 6 evaluates the method upon a selection of problems from combinational logic design. Section 7 discusses the behaviour of enzyme GP, and section 8 concludes.

2. The Representation Problem

The difficulty of a problem in evolutionary computation is a result of many factors (Jones, 1995). Chief among these is the fitness function, which defines the search space. However, the fitness function, the objective difficulty of the problem, is usually invariant. The effective difficulty, rather, is a result of the implementation, and in particular, the solution representation and the variational operators. The representation records the state of a particular search within the search space, and the operators decide how this representation can be transformed — and hence how the search space can be traversed. Moreover, representation and operators are dependent variables. The behaviour of the operators is constrained by the adaptability of the representation; such that a good representation enables appropriate transformations enacted by meaningful operators.

However, there are good reasons to believe that the standard representation of genetic programming, the parse tree, is not a good representation for an evolutionary algorithm. One reason is philosophical. Parse trees are not designed to be evolvable, and therefore we would not expect them to be evolvable. Evolvability would not appear to be a common property of representations, and it seems likely that it is especially uncommon amongst those used by humans to solve problems. Manual representations, after all, are designed to support human cognition; a process typified by limited memory, linearity and necessary decomposition. Evolution does not require limitations since it does not require understanding of the domain. Indeed, common wisdom in evolutionary computation records that placing unsuitable constraints upon a search space can reduce performance of evolutionary algorithms and even restrict search from traversing sub-spaces containing global optima.

This argument alone does not prove that parse trees are an unsuitable representation for evolution. Further evidence is given by analysis of the behaviour of crossover in tree-based genetic programming (Nordin et al., 1996; Angeline, 1997). Sub-tree crossover, the swapping of randomly selected sub-trees between individuals, is the nat-

ural recombination operator for a parse tree representation (Koza, 1992). Following from the argument presented earlier, an ineffective recombinative operator would suggest a representation sub-optimal for evolution. There are a number of empirical and theoretical reasons to believe that sub-tree crossover is not an effective operator. First, empirical analysis (Nordin and Banzhaf, 1995) shows that for most crossover events, offspring are less fit than their parents. Less frequently, the effect is neutral and infrequently it is beneficial. This suggests that rather than being a constructive operator, sub-tree crossover is primarily disruptive; and it has been argued that a major cause of solution bloat is protection from the disruptive behaviour of crossover (Nordin et al., 1996). A second observation (Angeline, 1997) is that the performance of sub-tree crossover can be equalled or bettered by mutative operators such as headless-chicken crossover; an operator which resembles crossover but is actually a form of macro-mutation. This supports the notion that sub-tree crossover is primarily a source of disruption rather than a meaningful recombinative operator.

The problems with sub-tree crossover derive from the explicit, positionally sensitive nature of parse trees. Parse trees have a globally defined structure and functional components within the parse tree have explicit context defined by this structure. This means that information regarding connections between components is recorded in terms of position; and this information is neither recognised nor preserved by sub-tree crossover. In effect, most of the behaviours open to sub-tree crossover involve the exchange of unrelated sub-trees, generating child solutions with functionality, and therefore fitness, very different (and most likely lower) than their parents.

One solution to this problem is to adapt crossover so that it actively searches for homologous regions between parent solutions (Langdon, 2000b). This has proved successful in reducing the incidence of bloat and can lead to improved performance, but comes at the expense of a less-natural operator with a higher computational overhead. Whilst homologous crossovers may have the potential to improve search in GP, there will always be the problem of annealing an essentially alien operator to a parse tree representation.

3. A Biomimetic Solution

An alternative approach is to remove the parse tree and replace it with a more supportive representation. Examples of this are grammatical evolution (Ryan et al., 1998), which uses a linear representation, and Parallel Distributed Genetic Programming (Poli, 1997), which evolves

control flow graphs. A less acute approach is to retain parse trees, but use them as a component within a more evolvable representation. Examples are MIPs (Multiple Interacting Programs) nets (Angeline, 1998) and gene expression programming (Ferreira, 2001); both of which represent programs as distributed collections of interacting parse trees.

This paper is about representations that mimic biology. Existing examples of this approach in GP and other areas of evolutionary computation include (Luke et al., 1999; Kennedy and Osborn, 2000; Keller and Banzhaf, 1999; Banzhaf, 1998; Kargupta, 1999; Kargupta, 2001; Goldberg et al., 1993); although some of those mentioned above (Angeline, 1998; Ferreira, 2001) also feature biological aspects. Before returning to this subject, however, it is useful to take a look at the representations of biology and discuss how these, in association with the mechanisms of biology, encourage the evolution of genetic content.

3.1. Biological Representations

Biological organisms, except for those with no body, have two forms of representation; the genotype and the phenotype. The genotype is the evolutionary form of an organism, derived from the genotype of its parents and contributing to the genotypes of its offspring. The phenotype is the physical form of an organism, constructed from the genotype through a process of development. It is this form which determines an organism's fitness, and therefore the perceived fitness of its genotype.

3.1.1. *Genetic Representation*

The genome is the genetic representation, yet despite the recent interest in genome-mapping projects, is widely viewed as a collection of genes; each of which determines a phenotypic trait. In practice, genomes are far more complex than this (Lewin, 2000). They do encode a collection of genes, but most of the genome does not encode genes (Brown and Brown, 1999). Genes do describe phenotypic traits but, given pleiotropic and polygenic effects, not usually according to a one-to-one mapping (Lewin, 2000).

It is probably true that a proportion of non-coding DNA constitutes 'junk' since there is little selective pressure towards compact genomes in large eukaryotes. However, non-coding DNA performs both functional and informational roles both within and between the lifetimes of organisms (Moore, 1996). Within an active genome, base sequences found in non-coding DNA provide binding sites for transcription factors. During recombination, non-coding DNA separates coding sequences, making it less likely that these will be targets for division during crossover. Transposons, sequences of (mostly) non-coding DNA which are capa-

ble of being moved or copied within the genome, may have significant evolutionary roles (Brown and Brown, 1999).

However, perhaps the most interesting property of non-coding DNA is that it contains copies and fragments of extinct genes, extant genes and possibly future genes. The retention of extinct genes provides a source of backtracking. Copies of extant genes are subject to evolution without having an effect upon fitness. Any of these inactive sequences might become incorporated into future genes. Consequently, genomes appear to have multiple roles: phenotype encoding, fossil record and evolutionary scratchpad. Considering all these roles, perhaps the genome is best considered as not recording merely a search point, but a search state.

3.1.2. *Phenotypic Representation*

Each gene provides a specification for a protein. This specification is realised through the processes of transcription and translation (Lewin, 2000; Lones and Tyrrell, 2001a). Once expressed, proteins interact with other chemicals and other proteins to form emergent structures called biochemical pathways (Michal, 1999); which form networks of computation and communication that implement the functionality of the organism.

If a cell is viewed as a computational system, then that computation takes the form of the manipulation, the metabolism, of the cell's chemical state; the concentrations of different chemical species within the cell. Manipulation of this state involves transformations, implemented by constructive and destructive chemical reactions, between chemical species; increasing the concentrations of some and reducing the concentrations of others. However, temperatures in biological systems are relatively low and many of these reactions will not occur without the help of enzymes, catalytic proteins that bind to specific chemicals and mediate their interaction and subsequent transformation. The chemical species to which an enzyme binds, its substrates, are determined by a property called its specificity; a result of the spatial arrangement of amino acids found at the enzyme's binding sites. Substrate binding occurs through inexact matching. The closer the match is between the substrate shape and the binding site shape, the stronger is the bond between enzyme and substrate; and consequently, the higher the likelihood of recognition.

The presence of enzymes activates transformative paths within the metabolism. This forms a network, a metabolic network, where chemical species are nodes and enzyme-mediated reactions are the connections between nodes. Metabolic networks are composed of metabolic pathways. A metabolic pathway is an assemblage of enzymes which,

roughly speaking, carries out a unified task involving a shared group of chemical species. This cooperation emerges from the sharing of substrates between enzymes, where the product of one enzyme becomes the substrate of another. Pathways can be linear, forked or iterative; iteration resulting where a product feeds back to an earlier stage in the pathway.

Proteins, including enzymes, have computational characteristics similar to artificial computational elements like transistors and logic gates (Capstick et al., 1992; Bray, 1995; Fisher et al., 1999) (and also (Conrad, 1992) some that are quite unrelated). The computation provided by metabolic pathways has been modelled using Petri nets (Reddy et al., 1993) and, in other work, artificial analogues of enzymes have been used to solve an information-processing task (Shackleton and Winter, 1997).

3.2. Neutral Evolution

When Kimura published his Neutral Theory of Evolution (Kimura, 1983), the dominant view of evolution was the selectionist school, which states that genetic change derives from the selective advantage of rare positive mutations which gradually replace less-fit alleles within the population. The neutral theory, however, proposes that genetic change is driven not by selection but by the random fixation of neutral mutations; mutations that do not cause fitness change, and in many cases do not cause phenotypic change. Accordingly, most mutations that proliferate are those that are neutral. Those that go extinct are either neutral or deleterious. Advantageous alleles, by comparison, are rare and have the least influence on population genetics.

There are several sources of neutral mutation. Mutation in non-coding DNA (unless it changes a protein binding site) leads to no phenotypic change. Likewise, synonymous mutation in coding DNA; which changes a codon, but not the amino acid it codes for; has no phenotypic effect. Also, change in a protein's amino acid sequence does not necessarily change the protein's behaviour. Proteins with the same behaviour but different amino acid sequences are called allozymes.

Recently, there has been considerable interest in the role of neutral evolution in evolutionary computation (Ebner et al., 2001; Barnett, 2001); where it is thought that neutrality increases an algorithm's ability to bypass local optima in many search spaces. Neutrality is defined as redundancy in the genotype-phenotype mapping, meaning that a single phenotype can be described by multiple different genotypes. Neutral mutation is then the transformation between genotypes that map to the same phenotype. These genotypes are said to be connected together by

a neutral network. Movement within this network, a sequence of neutral transformations, constitutes a neutral walk and is thought to allow the circumnavigation of local optima. To a certain extent, this hypothesis has been validated. See, for example, Shipman et al. (2000).

4. Enzyme Genetic Programming

Enzyme genetic programming is based upon the following simple model of biology. A genotype defines a collection of functional components, not all of which are necessarily expressed. When expressed, functional components interact with other functional components to form useful computational structures. These interactions occur according to each component's own interaction preferences and generally independently of genetic position. Whether a component is expressed depends in part upon which other components are present.

The principle behind enzyme GP is that the structure of a program is not given explicitly but is derived from connection choices made by each component of the program in a bottom-up fashion. A program is defined as an 'enzyme system', the format of which is described in section 4.1. During evaluation, each enzyme system carries out a developmental process which leads to a static executional structure comparable to a conventional GP program expression. This process is described in section 4.2. Enzyme systems are evolved using a genetic algorithm, defined in section 4.3.

4.1. Enzyme Representation

An enzyme system is a set of computational elements called enzymes. From a non-biological perspective, each of these elements can be seen as a functional component wrapped by an interface that determines how it interacts with the rest of the system. However, since these elements mimic biological enzymes, for the remainder of this paper they are described by shared biological terms.

The enzymes of enzyme GP are defined by three attributes: shape, activity and specificity (depicted in figure 1). Activity is the enzyme's role within the enzyme system and is either a function instance, an input terminal instance or an output terminal. Specificity describes the shape of an enzyme's preferred inputs. It is analogous to the binding domains in biological enzymes that determine which substrates will be bound by the enzyme. Unlike in biology, specificity is defined directly upon other enzymes rather than upon their products and hence an enzyme's specificity determines which other enzymes it will receive its

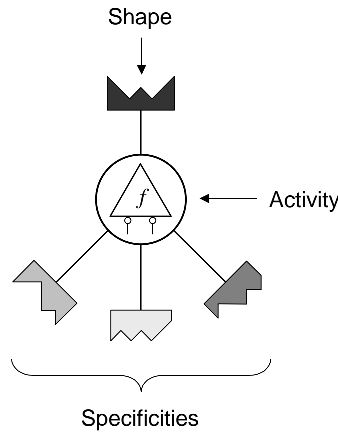


Figure 1. Enzyme attributes. Specificity strengths are shown by grayscale value.

inputs from. An enzyme has at least as many specificities as its activity has inputs so each input will be determined by a different specificity. Since there may be more specificities than there are activity inputs, some specificities remain unused. Each specificity has a strength and the strongest specificities determine inputs. The remainder are considered recessive. A specificity, like a binding region, chooses an input through pattern matching between its own shape and the shape of its substrate. An enzyme's shape is an identifier that describes how it is seen by other enzymes. In effect, a specificity declares the shape of the enzyme it would most prefer to receive input from.

There are three different kinds of enzyme within a system. A functional enzyme (or just enzyme) has a functional activity, a set of specificities and a shape. A gland has an input (terminal) activity and a shape. Since it receives no input from within the system, it has no specificites. A receptor has an output activity, which takes one input, and a set of specificities. It has no shape since it produces no output within the system. An enzyme system is a set of glands, enzymes and receptors containing at least one gland and one receptor.

4.2. Program Development

The executional structure of an enzyme program must reflect the constraints of the domain. The domain used in this study is combinational logic design — the design of non-recurrent digital circuits. The activities carried out by enzymes correlate with logic gates, input terminals and output terminals. Input terminal enzymes (glands) deliver data to the enzyme system, logic gate enzymes transform this data, and output terminal enzymes (receptors) sink data from the system. Execution

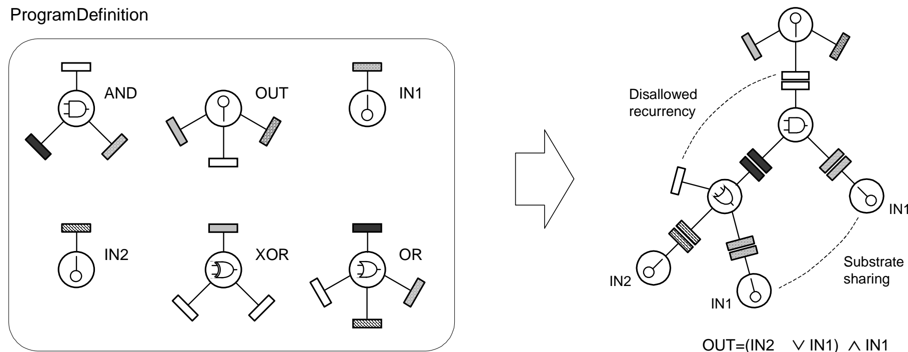


Figure 2. Development of an enzyme system. Shape is indicated by pattern and specificity strengths are not shown for clarity. Enzyme 'OUT' selects enzyme 'AND' as its input according to its strongest specificity. Enzyme 'AND' then selects its inputs, followed by enzyme 'OR'. However, due to the non-recurrency constraint, 'OR' may not choose 'AND' as an input. Enzyme 'IN1' is bound to more than one enzyme, demonstrating a capacity for reuse. Notice that enzyme 'XOR' is not expressed during execution.

requires valid combinational circuits, requiring that output terminals must have a source, logic functions must have a complete set of inputs and data must not feed-back to an earlier stage in the circuit.

The executional structure of a program emerges from interactions between enzymes in a developmental process analogous to the formation of metabolic pathways in biology. Enzymes attempt to bind substrates that match their strongest specificities. This occurs in a hierarchical fashion with receptors binding their substrates first. These substrates then attempt to bind their own substrates (unless they are glands, in which case they do nothing) with the process completing when all active enzymes have bound the appropriate number of substrates. However, given the non-recurrency constraint, enzymes will not always be able to satisfy their strongest specificities, in which case they will bind substrates for which they have lesser specificity. Choice of substrates is deterministic, meaning that a given enzyme system will always develop into the same executional structure. An example of development is shown in figure 2.

Note that enzymes will only be involved in the developmental process if they are chosen as substrates by other enzymes. Consequently, some enzymes may not be expressed in the program's executional structure. Alternatively, enzymes may be expressed more than once as the substrates of multiple enzymes; introducing a capacity for reuse of sub-circuits.

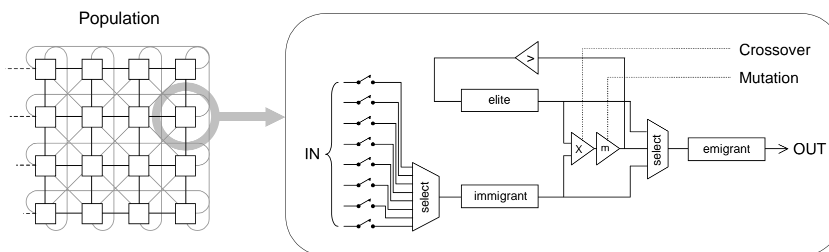


Figure 3. Genetic algorithm structure, showing the processing which occurs in each cell of the distributed population during each generation.

4.3. Program Evolution

Evolution of enzyme systems occurs within the framework of a diffusion model distributed genetic algorithm (shown in figure 3). The population is organised into a spatially-distributed network of cells, each of which carries out an evolution strategy upon local state and inputs from surrounding cells. The network topology determines the processing behaviour of the population. For all experiments reported in this paper, the network is a two dimensional edge-connected matrix (toroidal). A cell's evolution strategy selects the fittest individual from the emigrants of those cells designated as inputs by the network. This immigrant then undergoes recombination with the local 'elite'; the fittest individual created so far within this cell. If the fittest child is fitter than the elite, then the elite is replaced with this child. The cell's emigrant is the fittest individual out of the parents and the children. By implementing elitism, the algorithm retains fit solutions. By making this elitism local to each cell, diversity is preserved and new solutions are encouraged. Whilst each cell contains three individuals, only the elite is considered resident and only the children are evaluated during each generation.

Genotypes, enzyme system definitions, are stored in linear structures; the format of which is shown in figure 4. Genotypes are generated randomly to fill the initial population, though a particular implementation may place different constraints upon the size and constitution of programs (see below).

5. Implementation

This paper presents two implementations of enzyme GP. The first, the activity model, uses a simple definition of shape and is used to evolve the connectivity of fixed-length programs containing pre-defined

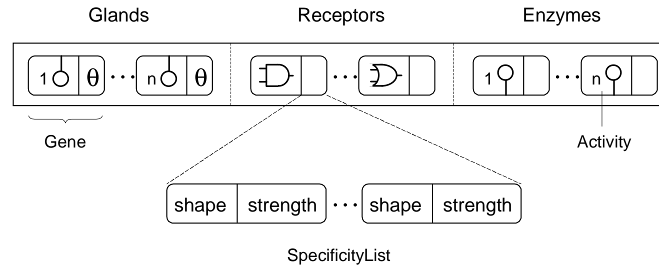


Figure 4. Genotype layout. The genotype defines which enzymes are present in an enzyme system. All genotypes declare a full set of terminal enzymes (glands and receptors). Note that glands do not have specificities, for they receive no input from other enzymes.

components. The second, the functionality model, uses a more advanced definition of shape which automatically supports variable-length programs and attempts to preserve local context during crossover. The functionality model is a refinement of the activity model. Descriptions of the activity model can also be found in (Lones and Tyrrell, 2001a; Lones and Tyrrell, 2001b), with significant details repeated here for completeness and to allow comparison with the functionality model.

5.1. Activity Model

For the activity model, the shape of an enzyme is its activity and the activity of an enzyme is an instance of a function; not the function itself. This allows there to be multiple identifiable instances of a function, each recognised as a separate activity.

Since shape is defined upon activity, so too is specificity. Moreover, within the activity model, each inputting enzyme has specificity for every outputting activity present within the enzyme system.

An enzyme system based upon the activity model can be visualised as a fully-connected weighted network where the weight of a particular edge (the strength of a particular specificity) defines a relative preference for this edge being realised as a wire within the circuit. This idea is depicted in figure 5.

Programs evolved by the activity model are fixed-length and contain only pre-defined instances of components. Recombination is implemented by uniform crossover. Concern that the activity model contains excessive redundancy (Lones and Tyrrell, 2001b) and that it can not easily be extended to support variable-length solutions has lead to the development, in this paper, of the functionality model.

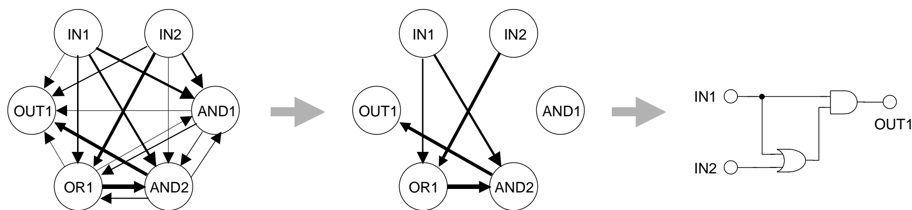


Figure 5. Visualising the activity model. Shape is equivalent to activity. Connections show the specificity of one activity for input from another. Specificity strengths are shown by line weight. During development, the strongest specificities are realised as wires (within the bounds of the non-recurrency constraint).

5.2. Functionality Model

5.2.1. *Requirements of functionality*

The purpose of shape is two-fold; to identify enzymes and to describe enzymes. Identification allows an enzyme to be referenced by other enzymes, whereas description allows an enzyme to be classified in relation to other enzymes. Activity is a basic definition of shape which segments all enzymes into discrete equivalence classes. However, the enzymes within an activity class are not normally equivalent and, given that specificity has a major influence in deciding an enzyme's role within a program, their behaviour is not necessarily related. Furthermore, the activity model provides no metric defining quantitative similarity between enzymes either within or between classes.

Functionality is a definition of shape that attempts to both describe an enzyme's role within a program and provide a quantitative distance metric for comparing enzymes. A distance metric is required because, during development, an enzyme will not always be able to bind its preferred substrate; either because that substrate is not present within the program, or to satisfy the non-recurrency constraint. In the activity model, the enzyme would, in this situation, bind a substrate activity for which it has a lesser preference. This second choice substrate is not necessarily related to the most preferred substrate, meaning that an enzyme may change its function considerably (and non-uniformly) depending upon its environment. Where each program contains an instance of every activity, this is a relatively small problem. However, in a system with different activities in each program, it is quite likely that an enzyme's most preferred substrate will not be available; and in this circumstance it would make more sense to bind as a substrate whichever enzyme has the closest role.

Ideally, the shape of an enzyme would describe the exact expression formed by the enzyme during development, but there are a number of reasons why this definition would not work in practice. First, generation

and comparison of complete expressions would be computationally expensive. Second, the expression formed by an enzyme is dependent upon which other enzymes are present within a program, making the shape of an enzyme variant with respect to its environment; and possibly causing evolutionary instability. Finally, an enzyme's complete expression is not available until after development.

5.2.2. *Definition of functionality*

Functionalities, by comparison, are available prior to development, are invariant, and are computationally inexpensive to generate and compare. This is possible because a functionality does not capture the exact form of an expression, but rather a functional profile of the expected expression. For now, an expected expression can be considered as the expression developed when each active enzyme is able to bind shapes that exactly matches its specificities. An enzyme's functionality is the relative incidence of each member of the function set within this expression. Formally, a functionality is a point within an n-dimensional reference space, where n is the size of the function set, and $\forall m \bullet m < n$, the magnitude in dimension m indicates the relative incidence of function m (weighted by depth) in the corresponding enzyme's expected expression. A functionality is recorded as a vector. The functionality, f, for a particular enzyme is a weighted vector sum of the functionality of the enzyme's own activity and the functionality of its inputs.

$$f(enzyme) = (1-k).f(activity) + k.f(inputs) \tag{1}$$

The functionality of an activity is simply a vector with a zero in all positions except that corresponding to the dimension of the activity's function; for which there is a value of one. The functionality of an enzyme's inputs is a weighted sum of the functionalities defined in its specificities; the weights being given by the strength of each specificity.

$$f(inputs) = \frac{\sum_{i=1}^{n} f(specificity_i).strength(specificity_i)}{\sum_{i=1}^{n} strength(specificity_i)} \tag{2}$$

The constant k decides how much bias is given to the functionality of the inputs rather than the enzyme's own activity in the calculation of an enzyme's functionality. If the matching distances between specificity and functionality remain small, a specificity captures the functionality of its substrate enzyme. Therefore, it also captures the substrate enzyme's specificities and, recursively, the activities of all enzymes below itself in the enzyme's expression. For this reason, the larger the value of k, the more bias is given to capturing functional information deeper in the expression. However, the matching distance between specificity

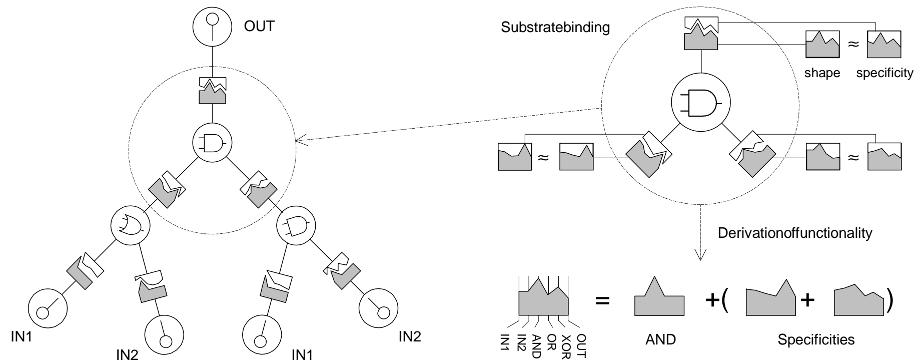


Figure 6. Development in the functionality model. Enzyme interactions emerge from inexact matching between specificities and shapes, both of which are defined as functionalities. Functionality vectors are plotted graphically. Some specificities are shown inverted to emphasise the matching process. An enzyme's shape is derived from its activity and the shape of its specificities according to equation 1.

and functionality will not always be small and so the distance between the expected expression and the actual expression will increase with depth. Accordingly, choosing a value for k is a tradeoff between a fuller description of an enzyme's role and accuracy of this description during development. Figure 6 shows expression development within the functionality model and also depicts visually the derivation of an enzyme's functionality.

Note that an enzyme's input functionality, and hence the enzyme's overall functionality, is derived from all specificities, including any that are recessive. The benefit of capturing recessive specificities is that a functionality gives a more complete description of the enzyme than would be available otherwise. It also makes functionality less variant in response to changes in dominance over evolutionary time; giving the enzyme a more constant character. Consequently, if mutation leads to a dominance change, the enzyme has a greater chance of retaining its previous role, which may lead to more effective behavioural exploration during evolution. It also means that the definition of expected expression given above is slightly inaccurate, since this expression also captures recessive sub-expressions to some degree.

The functionality reference space defines a point for every possible enzyme. However, it does not assign each enzyme a unique point and consequently enzymes describing different expressions but with the same functional profiles will be assigned identical functionalities. Nevertheless, this reference space is continuous and it seems very unlikely that two different enzymes with the same functionality would be present in the same population. A similar phenomenon is found in

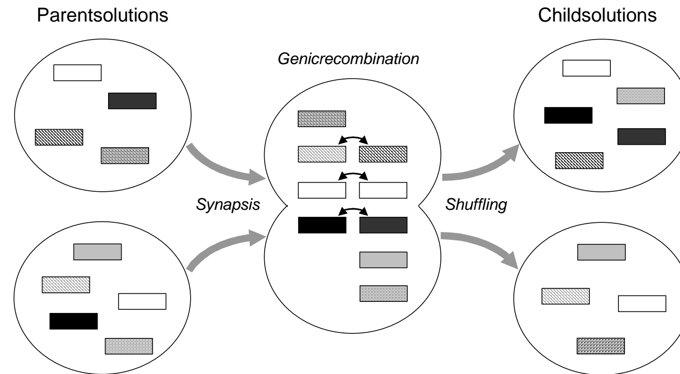


Figure 7. Recombination of variable length genotypes. Genes are shown as rectangles. Shade indicates functionality. Crossover is a two-stage process of genic recombination and gene shuffling. These operations preserve linear genotype structure, but are drawn in a way which emphasises their resemblance to meiosis.

biological systems, where recognition is also based upon a shape which reflects, but only captures in part, a protein's functional role within the system. In biological systems, the occurrence of two proteins with the same shape (or more exactly, the same markers) can cause interesting behaviours.

5.2.3. *Evolution of genotypes*

Enzyme systems generated for the initial population are variable length and composed of enzymes with varying activities. The number of enzymes within a single genotype is limited by upper and lower bounds. Enzymes are given activities chosen non-deterministically from the function set and a random number of specificities (also bounded). Specificity functionalities and strengths are generated randomly. After initialisation, the population contains a selection of enzymes which randomly sample the functionality reference space and are sorted into random enzyme system groupings.

Crossover must recombine heterogeneous genotypes. This is carried out using a two-stage process that bares some similarity to biological processes (see figure 7). To begin with, genes which are similar between the two parents align. Similarity is measured by distance between the functionalities of the genes' respective enzyme products and does not necessarily imply that they have the same activity but does suggest that they carry out similar roles. This alignment resembles synapsis where, during meiosis metaphase, chromosomes pair up and exchange genetic material. However, in this implementation only a fraction of the gene-pairs are recombined in order to limit disruption.

The second stage of crossover is gene shuffling, where the recombined genes are desegregated non-deterministically between the child genotypes. This is implemented using a uniform crossover and results in child genotypes with lengths intermediate to the lengths of their parents.

Following recombination, mutation is applied to the child genotypes. This involves randomly changing the magnitude of a small fraction of specificity strengths and dimensions of functionalities.

6. Evaluation

Enzyme GP has been applied to a range of problems in the domain of combinational logic design. The problems, listed in table I, have been chosen with the aim of exercising the method upon a range of tasks and to allow comparison with other methods. The 1-bit full adder and 2-bit multiplier problems have been given particular attention and their truth tables and standard solutions are shown in figure 8. Standard circuits for the other problems can be found in (Miller et al., 2000) or (Coello Coello et al., 2000). Parameter settings used for all experiments can be found in table II.

6.1. Comparison of activity and functionality models

The performance of the activity and functionality models has been compared upon the 1-bit full adder and 2-bit multiplier problems. Figures 9 and 10 show average solution times and success rates for both problems over a range of population sizes. For the functionality model, results for two different sets of solution size bounds are shown. Standard deviations have been measured and t-tests performed to de-

Table I. Test problems.

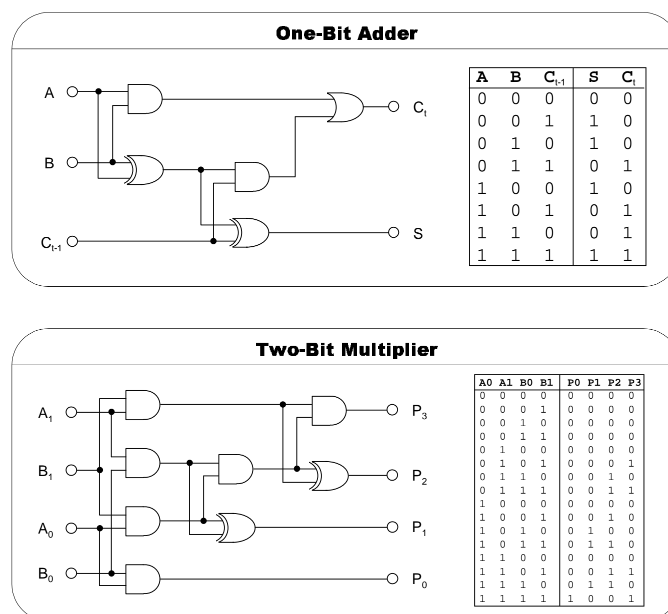
Name	Inputs	Outputs	Function set
1-bit full adder	3	2	AND,OR,XOR
2-bit full adder	5	3	XOR,MUX
2-bit multiplier	4	4	AND,XOR
3-bit multiplier	6	6	AND,XOR
even-3-parity	3	1	AND,OR,NAND,NOR
even-4-parity	4	1	AND,OR,NAND,NOR
Coello's 'example 5'	4	3	AND,OR,NOT,XOR

Table II. Parameter settings.

Parameter	Value
Input bias (constant 'k' in equation 1)	0.3
Number of specificities per enzyme	3
Distance limit for gene alignment	1
Proportion of gene pairs recombined	15%
Rate of specificity strength mutation	2%
Rate of functionality dimension mutation	15%

termine whether there is any significant difference between the mean solution times of different approaches. For both problems, Standard deviation is about 20–30 generations for the activity model and 30–40 for the functionality model. For results of t-tests see figure captions.

Both models are able to evolve solutions to both problems. Unsurprisingly, increasing population size leads to higher success rates and generally lower solution times. On both problems, the activity model performs better than the functionality model with lower-bound solution size and worse than the functionality model with upper-bound solution



A	B	C_{t-1}	S	C_t
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

A0	A1	B0	B1	P0	P1	P2	P3
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

Figure 8. Standard solutions and truth tables for one-bit full adder and two-bit multiplier problems.

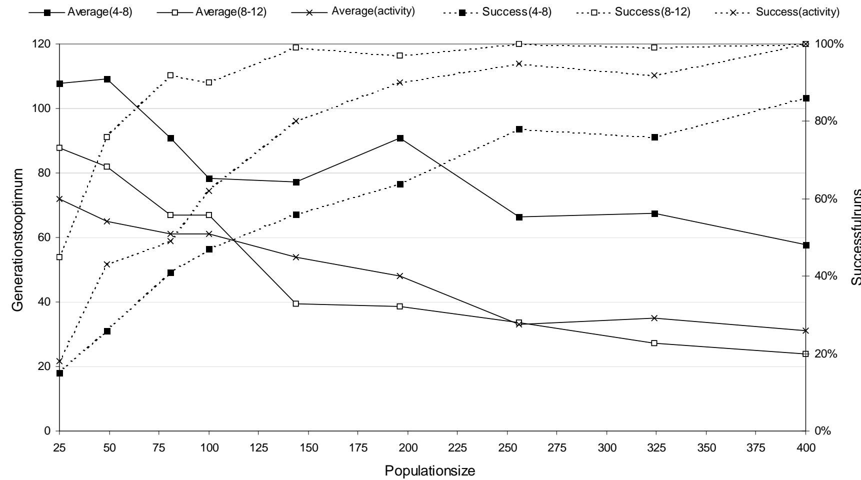


Figure 9. Results for evolution of the full adder. Averages and success rates are taken over 100 trials. Generation limit is 200 generations. For the functionality model, results are shown for solution length bounds of 4–8 gates and 8–12 gates. T-tests indicate no significant difference between the means of the activity model and the functionality model with bounds 8–12 for most population sizes.

size. On the whole the difference between the performance of the two models is not great. This is revealing since the functionality model must discover the correct activities in addition to the correct connections found by the activity model, implying that the functionality model does more useful work than the activity model in a given time.

6.2. Performance of enzyme GP

The adder, multiplier and even-4-parity problems have been solved by Miller (2000) using Cartesian GP (Miller and Thomson, 2000), a graph-based genetic programming system. Results from Koza (1992; 1994), using tree-based GP, are available for both parity problems. Koza (1992) has also attempted the 2-bit adder problem, but quantitative results are not available. Further comparison can be made with work on graph-based GP by Coello (2000), whose test problems include the 2-bit multiplier and a harder problem which he refers to as 'example 5'. Results for these problems for enzyme GP with the functionality model are shown in table III.

Enzyme GP is able to solve all problems apart from the 3-bit multiplier. This is a difficult problem and Miller required in the order of 100 million evaluations to find an optimal solution. Enzyme GP was able to find a 95% correct solution in four runs of 5,000 generations for a population size of 1,600.

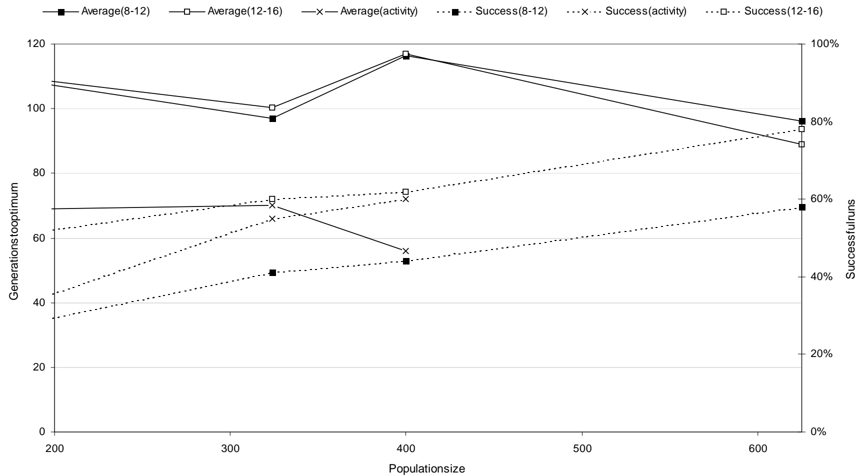


Figure 10. Results for evolution of the two-bit multiplier. For the functionality model, results are shown for solution length bounds of 8–12 gates and 12–16 gates. T-tests indicate no significant difference between the mean solution times of the functionality model for different solution size bounds.

Computational effort is a measure first used by Koza (1992) which gives the number of evaluations required for a 99% confidence of finding an optimal solution to a problem. Miller records minimum computational effort of between 210,015 and 585,045 for the 2-bit multiplier problem. Coello does not calculate computational effort though it appears to take about 150 generations to find a correct multiplier circuit within a population of 2,000. It should be noted that Coello's work is directed towards minimising circuit size rather than just finding correct solutions. Enzyme GP, requiring a minimum computational effort of 136,080 for a population of 324, compares favourably with these results.

Table III. Results using functionality model.

Problem name	Pop. size	Bounds	Average	Success	Effort
1-bit full adder	196	5–10	32	82%	39,200
2-bit full adder	324	10–20	113	74%	244,620
2-bit multiplier	324	12–16	118	77%	136,080
3-bit multiplier	1600	25–35	—	0%	—
even-3-parity	100	5–10	54	43%	79,000
even-4-parity	625	10–25	178	20%	—
Coello's 'example 5'	1,225	12–25	400	75%	—

For the 2-bit adder problem, Miller cites a minimum computational effort of 385,110. For enzyme GP, using the same functions as Miller, effort is 244,620. For 'example 5', Coello's results suggest a correct solution is found at about 900 generations for a population of 2,800. Enzyme GP takes on average 400 generations for a population of 1,225. Again, a favourable comparison.

Koza has evolved even-n-parity circuits using populations of size 4,000 (Koza, 1992) and 16,000 (Koza, 1994). For the even-3-parity problem (and without using ADFs), this gives minimum computational efforts of 80,000 and 96,000 respectively. For the even-4-parity problem, minimum computational efforts are 1,276,000 and 384,000 respectively. For enzyme GP, minimum computational effort has been calculated at 79,000 for the even-3-parity problem but has not yet been calculated for the even-4-parity problem. Early results suggest enzyme GP cannot easily evolve even-n-parity circuits where $n > 3$. This agrees with Miller's findings, where only 15 correct even-4-parity circuits were found in the course of 400 million evaluations. Langdon (1998) has suggested that parity circuits are unsuitable benchmarks for GP. Furthermore, parity circuits involve a high degree of re-use of sub-structures and it seems plausible that graph-based and enzyme GPs may be less able to take advantage of this fact than tree-based GPs where structural duplication is encouraged by non-deterministic subtree exchanges.

6.3. Solution size evolution

An issue of interest to GP practitioners is bloat. Bloat occurs when programs get larger and larger without a corresponding increase in functionality. Standard GP exhibits near quadratic growth if left unchecked (Langdon, 2000a). The exact causes of bloat are not known, though a number of theories have been proposed. These include hitchhiking (Tackett, 1994), protection from disruptive operators (Blickle and Thiele, 1994), operator biases (Altenberg, 1994a), removal biases (Soule et al., 1996) and search space bias (Langdon and Poli, 1997).

Figure 11 shows the evolution of 2-bit multiplier solution size for a selection of genotype size bounds using the functionality model. These graphs show that whilst individual runs freely explore between the size bounds, on average there is no bias towards either an increase or decrease in genotype size. This is especially pronounced for the higher size range of 10–30 where the average starting size of about 20 is well above the minimum correct phenotype size of 7 gates.

Phenotype size, the size of developed expressions, does appear biased. For the lowest size range where the average starting size of the phenotype is about 5.5, there is an average increase in size which levels

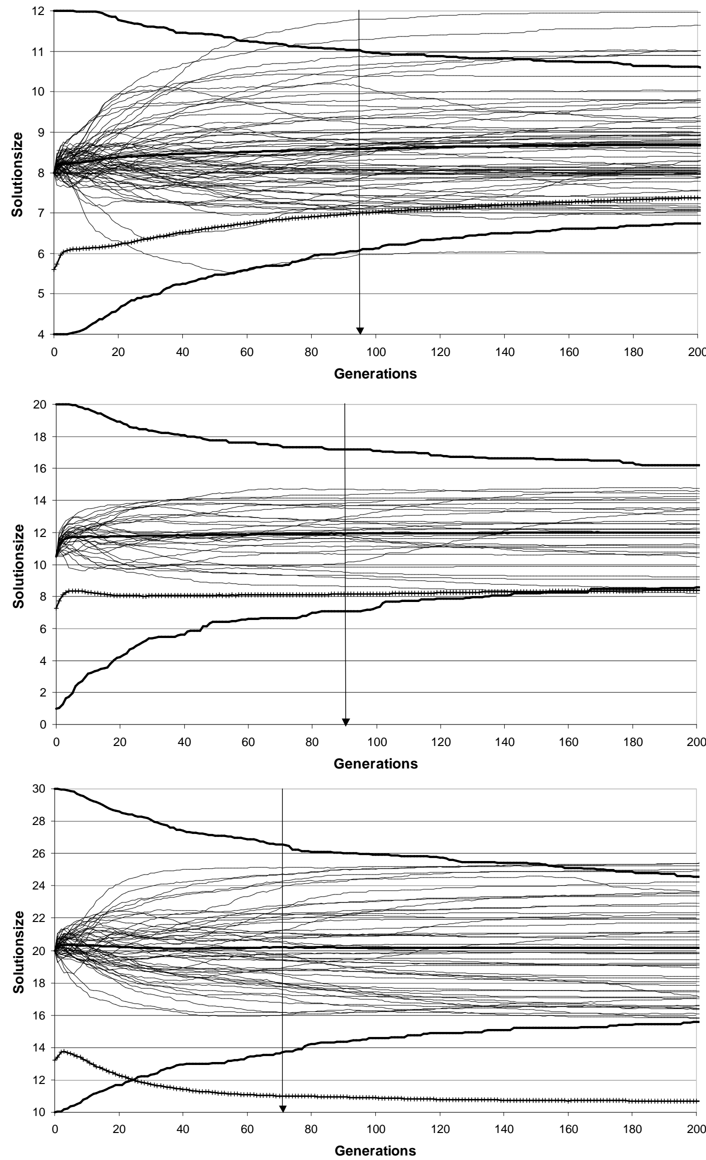


Figure 11. Solution size evolution for 2-bit multiplier problem for different size bounds. Faint lines show average genotype sizes for each run. Heavy un-crossed lines show average minimum, average and maximum solution sizes across all runs. Heavy crossed line shows average phenotype size across all runs. Arrow indicates average time to find a correct solution.

off once the size is above the minimum correct size. Where the starting size is close to this minimum there is little average change in phenotype size, as shown in the second graph. Where the starting size is above

the minimum correct size, as shown in the third graph where average starting size is about 13, there is negative growth towards shorter solution sizes. This is interesting and suggests that there is either an evolutionary advantage afforded to shorter solutions or an evolutionary disadvantage for longer solutions.

More recently, solution size analysis has been applied to a form of enzyme GP which places no bounds upon solution size (Lones and Tyrrell, 2002). This form of enzyme GP also demonstrates a bias towards exploring shorter solutions where the starting size is above the minimum viable size, and the paper speculates that this may be due to the greater presence of 'weak links' between specificities and shapes in larger solutions. This hypothesis is yet to be tested.

Nevertheless, enzyme GP does not suffer from bloat and, unlike tree-based GP, does not require size or shape limits to achieve this.

7. Discussion

7.1. Recombination

One of the goals behind enzyme GP is to provide a representation that does not suffer from the context-preservation problems found with tree-based representations. A key feature of enzyme GP is the use of a distributed genetic representation. Distributed representations seem appropriate for evolution for they allow context to be defined locally, within a component, rather than globally, with respect to the whole system. In enzyme GP, each component defines its context using terms independent of the particular program it finds itself in. This context is preserved by recombination. Of course, this does not imply that the component's definition of context will be meaningful in every program; but it will be invariant. This allows evolution to recognise the component's apparent fitness in a way that is not possible when its context is constantly changing; as would be the case for sub-trees using a parse tree representation and sub-tree crossover.

Whether recombination in enzyme genetic programming is in practice more meaningful, and therefore more successful, than for other representations will not be known until further experiments are carried out. However, it is possible to identify the factors that will determine whether or not it is successful. First, development should be sufficiently accurate. This depends on how often the context defined for a component, its specificity, matches the context it actually finds itself in, its bound substrates. If this incidence is sufficiently high, evolution should be able to determine a component's worth. If it is too low, there will not be enough consistent information available to measure its fitness.

Second, a component's definition of context should be sufficiently precise. For a component to have a role, its context must define some meaning. If this meaning is not precise enough, then the component's role will be ill-defined and opaque to evolution.

Third, context must be sufficiently specific. It should not define too many actual contexts as being equivalent. This could lead to the routine development of unrelated expressions in different programs, again making it hard to characterise the component's role.

Accuracy, precision and specificity are related concepts, but they place different requirements upon how context is defined. When functionality is used to define context, accuracy is how well the developed expression reflects the expected expression; precision is how well a functionality describes an expression; and specificity is how many expressions are assigned the same functionality. Functionality sacrifices exactness in all of these areas in order to be efficient, always available and invariant. The question is how much exactness is required in order for evolution to be effective. The results collated so far indicate that evolution is working effectively; though further research will be necessary to measure how effectively it is working. An interesting realisation is that biology also has imperfect exactness (perhaps due to efficiency reasons) of the kinds described above.

7.2. Neutrality

Neutrality is prevalent in biological representations, and likewise in the representations of enzyme GP. Sources of neutrality, other than that provided by the problem search space, include changes to enzymes that are not expressed, changes to recessive specificities and certain changes to dominant specificities. Neutrality is particularly ubiquitous in the activity model, where most specificities are recessive, and changes to recessive components are always neutral. The high incidence of these neutral mutations is shown in figure 12. For the functionality model, an enzyme's shape captures its specificities and change to either recessive or dominant specificities can change how it is bound by other enzymes. This also applies to recessive enzymes, which may become dominant due to a change in shape. Point mutations are unlikely to lead to a significant change in shape; though they may significantly affect the enzyme's choice of substrates. When mutation leads to a change in specificity yet no change in behaviour, the new enzyme has a similar role to an allozyme in biology.

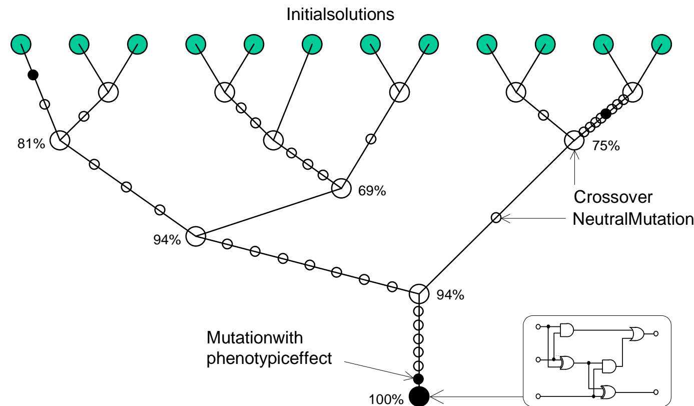


Figure 12. Evolution of a full adder with the activity model. Both crossover and mutation are used to evolve an optimal solution. Most mutations are neutral. Note the neutral walk that leads to the optimum after the final recombination.

7.3. Introns

In conventional GP, introns are sections of code that have no functional role but are generated by bloat (Smith and Harries, 1998). Enzyme GP also has introns, but these are present by intent and, unlike in conventional GP, may have a functional role. Introns in enzyme GP are components that are present in a program's genetic definition, but not in the developed expression. They are inactive either because they are redundant, or more generally, no other enzyme has chosen to bind them as a substrate; either because they do not match a specificity or to preserve non-recurrency. Redundant introns are enzymes that duplicate the function of other enzymes.

However, an intron in one program is not necessarily an intron in another program. An enzyme's expression depends upon whether it is compatible with the other enzymes in a program; and what is unused in one program may be used in another following recombination or mutation. Of particular interest is where a program contains a redundant copy of an enzyme. This copy is still subject to evolution, but any change does not affect fitness. This is similar to gene duplication in biology; where duplicated genes experience evolution without selection, leading to change that sometimes improves their function. More generally, this effect can also occur where enzymes have unused copies in other programs.

Introns could also be useful for backtracking. An enzyme that become unused may still be present as an intron within the population. If the enzyme that replaced it then proves to have important limitations, the saved copy may be used to restore an earlier point in search.

There is of course a balance between sustaining introns and preventing hitchhiking. If too large a proportion of a program definition consists of introns, the fitness value given to the program will not reflect the value of these introns. If the program is fit and some of the introns are deleterious, these deleterious introns may hitchhike with the program and disrupt evolution.

In a sense, an enzyme program definition describes more than one developed expression. If the developmental process were non-deterministic, then this would be literally so. If enzymes bound substrates probabilistically based upon specificity, most of the time they would bind the substrates for which they had highest specificity. Less frequently, they would bind substrates for which they had lower specificity. The expressions that developed over the course of time would be related, but variant. Whilst development is deterministic, only one expression is developed for each program; yet the amount of genetic change needed to generate a related expression is small and can be enabled by a single mutation. Less similar expressions can be accessed through several mutations. In effect, through mutation, a deterministic implementation can access the same expressions generated by a non-deterministic process. Mutation in enzyme GP is quite different to conventional GP. In conventional GP, mutation introduces new components to a solution whereas in enzyme GP, mutation changes the internal organisation of the program; changing the ordering and expressed status of enzymes. This allows programs to capture a range of solutions with mutation selecting which one is active. Whether this is advantageous or not is yet to be tested, but it could allow programs to group related expressions, structuring search and reducing evaluation cost.

8. Conclusions

A number of properties of biological representations have been identified which are thought supportive of the evolution of genetic material. An important realisation is that linkage between genes is not determined by genetic position but rather by the properties of the genes' products. It is also evident that biological genomes are more than a sum of their genes since they contain considerable quantities of non-coding genetic material that is informational and subject to evolution. Both coding and non-coding components of DNA are subject to neutral evolution, and this process is encouraged by a high degree of neutrality in the genetic encoding.

The program representation used in enzyme GP mimics those found in biology and captures each of the properties outlined in the previous

paragraph. In enzyme GP, the structure of a program is not given explicitly, as in conventional GP, but is derived from connection choices made by each component of the program in a bottom-up emergent fashion. Whether or not a component is expressed is determined by the connection preferences of other components. Furthermore, the connection preferences, or context, of a component are recorded in the component's genetic definition. This local definition of context, and the fact it is defined using a reference system independent of the program it is found within, means that it can not be lost or altered by crossover.

Enzyme GP has been evaluated upon a number of problems in the domain of combinational circuit design and its performance compared with that of other GP approaches in this domain. Enzyme GP does show some performance advantage over some other methods, yet it is uncertain whether this advantage is significant. In particular, there is some danger that matching between a program component's preferred context and its actual context may not be close enough on average to make recombination meaningful. However, this remains the focus of further investigation. Meanwhile, it can be concluded that enzyme GP does demonstrate some interesting behaviours, especially with regards to lack of bloat, and it seems fair to speculate that some of the ideas introduced in this paper could prove useful in the evolution of genetic programming.

References

Altenberg, L.: 1994a, 'Emergent phenomena in genetic programming'. In: A. V. Sebald and L. J. Fogel (eds.): *Evolutionary Programming – Proceedings of the Third Annual Conference*. pp. 233–241, World Scientific Publishing.

Altenberg, L.: 1994b, 'The evolution of evolvability in genetic programming'. In: K. Kinnear, Jr (ed.): *Advances in Genetic Programming*. MIT Press.

Angeline, P.: 1997, 'Subtree Crossover: Building block engine or macromutation?'. In: J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (eds.): *Genetic Programming 1997: Proceedings of the Second Annual Conference, GP97*. pp. 240–248, Morgan Kaufmann.

Angeline, P.: 1998, 'Multiple Interacting Programs: A Representation for Evolving Complex Behaviors'. *Cybernetics and Systems* **29**(8), 779–806.

Banzhaf, W.: 1998, 'Genotype-phenotype mapping and neutral variation—A case study in Genetic Programming'. In: Y. Davidor, H.-P. Schwefel, and R. Manner (eds.): *Proceedings of Parallel Problem Solving from Nature III*. Springer-Verlag.

Barnett, L.: 2001, 'Netcrawling — Optimal Evolutionary Search with Neutral Networks'. In: J.-H. Kim, B.-T. Zhang, G. Fogel, and I. Kuscu (eds.): *Proceedings of the 2001 Congress on Evolutionary Computation*. pp. 30–37, IEEE Press.

Blickle, T. and L. Thiele: 1994, 'Genetic programming and redundancy'. In: J. Hopf (ed.): *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*. pp. 33–38, Max-Planck-Institut für Informatik.

Bray, D.: 1995, 'Protein molecules as computational elements in living cells'. *Nature* **376**, 307–312.

Brown, T. A. and A. Brown: 1999, *Genomes*. Wiley.

Capstick, M., W. P. L. Marnane, and R. Pethig: 1992, 'Biological Computational Building Blocks'. *IEEE Computer* **25**(11), 22–29.

Coello Coello, C. A., A. D. Christiansen, and A. Hernández Aguirre: 2000, 'Use of Evolutionary Techniques to Automate the Design of Combinational Circuits'. *International Journal of Smart Engineering System Design* **2**(4), 299–314.

Conrad, M.: 1992, 'Molecular Computing: The Lock-Key Paradigm'. *IEEE Computer* **25**(11), 11–20.

Ebner, M., P. Langguth, J. Albert, M. Shackleton, and R. Shipman: 2001, 'On Neutral Networks and Evolvability'. In: J.-H. Kim, B.-T. Zhang, G. Fogel, and I. Kuscu (eds.): *Proceedings of the 2001 Congress on Evolutionary Computation*. pp. 1–8, IEEE Press.

Ferreira, C.: 2001, 'Gene Expression Programming: A New Adaptive Algorithm for Solving Problems'. *Complex Systems* **13**(2), 87–129.

Fisher, M. J., R. C. Paton, and K. Matsuno: 1999, 'Intracellular signalling proteins as 'smart' agents in parallel distributed processes'. *BioSystems* **50**, 159–171.

Goldberg, D. E., K. Deb, H. Kargupta, and H. George: 1993, 'Rapid Accurate Optimization of Difficult Problems Using Fast Messy Genetic Algorithms'. In: S. Forrest (ed.): *Proceedings of The Fifth International Conference On Genetic Algorithms*. Morgan Kaufmann.

Jones, T.: 1995, 'Evolutionary Algorithms, Fitness Landscapes and Search'. Ph.D. thesis, The University of New Mexico.

Kargupta, H.: 1999, 'SEARCH, Computational Processes in Evolution, and Preliminary Development of the Gene Expression Messy Genetic Algorithm'. *Journal of Complex Systems* **11**(4), 233–287.

Kargupta, H.: 2001, 'A striking property of genetic code-like transformations'. *Complex Systems Journal* **13**(1), 1–32.

Keller, R. and W. Banzhaf: 1999, 'The evolution of genetic code in genetic programming'. In: W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith (eds.): *Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann.

Kennedy, P. J. and T. R. Osborn: 2000, 'Operon Expression and Regulation with Spiders'. In: D. Whitley, D. Goldberg, and E. Cantu-Paz (eds.): *Proceedings of the 2000 Genetic and Evolutionary Computation Conference, Workshop Program*. pp. 161–166.

Kimura, M.: 1983, *The Neutral Theory of Molecular Evolution*. Cambridge University Press.

Koza, J.: 1992, *Genetic programming: on the programming of computers by means of natural selection*. MIT Press.

Koza, J.: 1994, *Genetic programming II: automatic discovery of reusable programs*. MIT Press.

Langdon, W. and R. Poli: 1998, 'Why "Building Blocks" Don't Work on Parity Problems'. Technical Report CSRP-98-17, School of Computer Science, University of Birmingham.

Langdon, W. B.: 2000a, 'Quadratic Bloat in Genetic Programming'. In: D. Whitley, D. Goldberg, and E. Cantu-Paz (eds.): *Proceedings of the 2000 Genetic and Evolutionary Computation Conference*. pp. 451–458, Morgan Kaufmann.

Langdon, W. B.: 2000b, 'Size fair and homologous tree genetic programming crossovers'. *Genetic programming and evolvable machines* **1**(1/2), 95–119.

Langdon, W. B. and R. Poli: 1997, 'Fitness causes bloat'. In: P. K. Chawdhry, R. Roy, and R. K. Pant (eds.): *Soft Computing in Engineering Design and Manufacturing*. pp. 13–22, Springer.

Lewin, B.: 2000, *Genes VII*. Oxford University Press.

Lones, M. A. and A. M. Tyrrell: 2001a, 'Biomimetic Representation in Genetic Programming'. In: J.-H. Kim, B.-T. Zhang, G. Fogel, and I. Kuscu (eds.): *Proceedings of the 2001 Genetic and Evolutionary Computation Conference, Workshop Program*. pp. 199–204.

Lones, M. A. and A. M. Tyrrell: 2001b, 'Enzyme Genetic Programming'. In: J.-H. Kim, B.-T. Zhang, G. Fogel, and I. Kuscu (eds.): *Proceedings of the 2001 Congress on Evolutionary Computation*, Vol. 2. pp. 1183–1190, IEEE Press.

Lones, M. A. and A. M. Tyrrell: 2002, 'Crossover and Bloat in the Functionality Model of Enzyme Genetic Programming'. To appear in the proceedings of the Congress on Evolutionary Computation 2002 (CEC2002).

Luke, S., S. Hamahashi, and H. Kitano: 1999, '"Genetic" Programming'. In: W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith (eds.): *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'99)*. Morgan Kaufmann.

Michal, G.: 1999, *Biochemical pathways*. John Wiley and Sons, Inc.

Miller, J. and P. Thomson: 2000, 'Cartesian Genetic Programming'. In: R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty (eds.): *Third European Conference on Genetic Programming*, Vol. 1802 of *Lecture Notes in Computer Science*. Springer.

Miller, J. F., D. Job, and V. K. Vassilev: 2000, 'Principles in the evolutionary design of digital circuits — part I'. *Genetic Programming and Evolvable Machines* **1**, 7–36.

Moore, M.: 1996, 'When the junk isn't junk'. *Nature* **379**, 402–403.

Nordin, P. and W. Banzhaf: 1995, 'Complexity compression and evolution'. In: L. Eshelman (ed.): *Genetic Algorithms: proceedings of the sixth international conference (ICGA95)*. pp. 310–317, Morgan Kaufmann, San Francisco.

Nordin, P., F. Francone, and W. Banzhaf: 1996, 'Explicitly defined introns and destructive crossover in genetic programming'. In: P. Angeline and K. Kinnear, Jr. (eds.): *Advances in Genetic Programming 2*. MIT Press, Cambridge, Chapt. 6, pp. 111–134.

Poli, R.: 1997, 'Evolution of Graph-Like Programs with Parallel Distributed Genetic Programming'. In: T. Bäck (ed.): *Proceedings of Seventh International Conference on Genetic Algorithms*. pp. 346–353, Morgan Kaufmann.

Reddy, V., M. Mavrovouniotis, and M. Liebman: 1993, 'Petri Net Representations in Metabolic Pathways'. In: L. Hunter (ed.): *Proceedings of the first international conference on intelligent systems for molecular biology*. MIT Press.

Ryan, C., J. J. Collins, and M. O'Neill: 1998, 'Grammatical Evolution: Evolving Programs for an Arbitrary Language'. In: W. Banzhaf (ed.): *First European Workshop on Genetic Programming*, Vol. 1391 of *Lecture Notes in Computer Science*. Springer.

Shackleton, M. and C. Winter: 1997, 'A Computational Architecture based on Cellular Processing'. In: M. Holcombe and R. Paton (eds.): *Proceedings of the International conference on Information Processing in Cells and Tissues (IPCAT'97)*. Plenum Press.

Shipman, R., M. Shackleton, and I. Harvey: 2000, 'The Use of Neutral Genotype-Phenotype Mappings for Improved Evolutionary Search'. *BT Technology Journal* **18**(4), 103–111.

Smith, P. and K. Harries: 1998, 'Code Growth, Explicitly Defined Introns and Alternative Selection Schemes'. *Evolutionary Computation* **6**(4), 339–360.

Soule, T., J. A. Foster, and J. Dickinson: 1996, 'Code growth in genetic programming'. In: J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (eds.): *Genetic Programming 1996: Proceedings of the First Annual Conference*. pp. 215–213, MIT Press.

Tackett, W. A.: 1994, 'Recombination, Selection, and the Genetic Construction of Computer Programs'. Ph.D. thesis, University of Southern California, Electrical Engineering Systems.