Evolving Graphs and Networks with Edge Encoding: Preliminary Report

Sean Luke*
seanl@cs.umd.edu
http://www.cs.umd.edu/~seanl/

Lee Spector*†
lspector@hampshire.edu
http://hampshire.edu/~lasCCS/

*Department of Computer Science
University of Maryland
College Park, MD 20742

†School of Cognitive Science and Cultural Studies
Hampshire College
Amherst, MA 01002

ABSTRACT

We present an alternative to the cellular encoding technique [Gruau 1992] for evolving graph and network structures via genetic programming. The new technique, called edge encoding, uses edge operators rather than the node operators of cellular encoding. While both cellular encoding and edge encoding can produce all possible graphs, the two encodings bias the genetic search process in different ways; each may therefore be most useful for a different set of problems. The problems for which these techniques may be used, and for which we think edge encoding may be particularly useful, include the evolution of recurrent neural networks, finite automata, and graph-based queries to symbolic knowledge bases. In this preliminary report we present a technical description of edge encoding and an initial comparison to cellular encoding. Experimental investigation of the relative merits of these encoding schemes is currently in progress.

1 Introduction

Several previous studies have examined the use of genetic algorithms to produce graph-based and network-based computational mechanisms. A major thrust in this area has involved using genetic and evolutionary algorithms to evolve the structure and weights of neural networks. One approach is to fix the network topology and evolve the weights as values in the chromosome [Collins and Jefferson 1991]. Other approaches seek to vary the number of nodes while explicitly describing the entire network within the chromosome [Fullmer and Miikkulainen 1991] [Angeline, Saunders, and Pollack 1994] [Lindgren et al.]. A third approach tries to evolve a network not as a set of explicit connections but as a set of rules which "grow" into a network. [Kitano 1990] evolved networks using context-free grammars which operated on network matrices. [Boers et al. 1993] "grew" networks using L-grammars operating on nodes and edges.

[1]To appear in the Late-breaking Papers of the Genetic Programming 96 (GP96) conference, Stanford, July 1996

Genetic programming (GP) techniques have also been used to evolve a variety of graph structures, including push-down automata [Zomorodian 1995] and novel graph-based programs [Teller 1996]. [Koza and Rice 1991] used GP to directly encode the edges and nodes of a neural network. Gruau's rule-based *cellular encoding* technique [Gruau 1992] uses tree-like chromosomes and other elements of GP technique to evolve neural networks.

2 Cellular Encoding

Cellular encoding [Gruau 1992] uses chromosomes consisting of trees of *node operators* to evolve a graph, commonly for use as a neural network. Each operator accepts from its parent a node in the graph, and modifies that node, possibly creating new nodes and edges. Operators are executed in a breadth-first, left-to-right traversal of the tree. Operators can be *nonterminals*, meaning that they have children in the tree, or *terminals*, which have no children. When finished modifying the graph node they were given, nonterminal operators pass this node on to one of their children for future modification. In the process of modifying a graph node, some nonterminals create additional nodes in the graph; each additional graph node is passed to an additional child of the operator. For example, if an operator modifies a graph node and in the process creates two new nodes, it will pass the original node to a child and the other nodes to its other two children. Terminal operators have no children, and represent the end of modification for a graph node — it becomes a permanent fixture in the graph. Typically, a graph begins with a single node, which is passed to the root operator in the encoding tree to start the graph-building process.

Cellular encoding is particularly interesting for several reasons. First, cellular encoding can describe all possible neural networks — in fact, Gruau presents a compiler which transforms Pascal functions (with a few limitations) into neural networks which compute the same functions. Second, cellular encoding is modular in the sense that it has special

operators which attempt to reuse groups of rules, thereby generating very large networks from reasonably small chromosomes. Additionally, cellular encoding takes advantage of GP-like tree genotypes to permit network phenotypes of any size, and to provide a straightforward mechanism for crossover between networks of widely differing topology.

Although cellular encoding is a powerful technique, it nonetheless has weaknesses. First, cellular encoding's chromosome traversal (breadth-first) and highly execution-order-dependent operators can result in subtrees within the individual which, if crossed over to other individuals, would result in very different phenotypes than they expressed in the original individual. For many domains it may be more appropriate to use an encoding mechanism which better preserves phenotypes through crossover. Other disadvantages come from cellular encoding's use of graph nodes as the target of its operations: as cellular encoding modifies nodes, the edges multiply rapidly, but cellular encoding provides only a very limited mechanism, *link registers*, to label and modify individual edges. Additionally, the graphs cellular encoding produces tend to consist of highly interconnected nodes. This is useful for cellular encoding's primary focus, namely fully-connected graphs such as those found in feedforward networks or Hopfield networks. However, it may be less desirable in other domains.

In this preliminary report we present *edge encoding*, an alternative to cellular encoding that addresses many of these concerns. Our technique uses *edge operators* rather than the *node operators* of cellular encoding. We believe that the result is a more elegant formalism with improved recombinative dynamics. Further, we believe edge operators may be more successful than node operators in certain domains. Because both cellular encoding and edge encoding are "complete" in the sense that they can produce all possible graphs, it is not clear how claims about the ultimate superiority of one technique over the other could be supported; still, it may be the case that one technique is clearly superior over some range of practical problems.

We are currently conducting experiments on a range of problems to assess the relative merits of cellular encoding and edge encoding. In this preliminary report, however, we confine ourselves to a technical description of edge encoding and an initial comparison between the two. Our current experiments include problems in which evolved graphs are interpreted as recurrent neural networks, as finite automata, and as graph-based queries to large symbolic knowledge bases. We will present the results of these experiments in subsequent reports as they are completed.

3 Edge Encodings

Edge encodings were originally developed to examine new recurrent neural network structures, but are equally applicable to developing a wide range of graph domains. Like a cellular encoding, an edge encoding is a tree-structured chromosome whose phenotype is a directed graph, optionally with labels or functions associated with its edges and nodes. Edge encodings differ from cellular encoding in several ways:

- Cellular encoding grows a graph by modifying the nodes in the graph. An edge encoding grows a graph by modifying the *edges* in the graph.
- Both encodings typically traverse their respective trees in preorder, leftmost-children-first. But while cellular encoding traverses its chromosome tree breadth-first, an edge encoding usually traverses trees depth-first. Further, simple edge encodings are independent of execution order as long as parents in the tree are executed before their children.
- Because an edge encoding operates on edges, the leaf nodes in an edge encoding chromosome represent unique edges in the resultant graph. In cellular encoding, leaf-nodes represent *nodes* in the resultant graph.

3.1 Building an edge encoding

Edge encodings are stored as forests of trees, similar to Genetic Programming's chromosomes. The nodes of each tree are operators which act on edges in the graph phenotype. An edge operator accepts from its parent operator a single edge in the graph, occasionally with some optional data (often a stack of nodes). It modifies or deletes the edge and its head and tail nodes as it pleases. In the process it may create zero or more new edges and nodes. The operator then passes the original edge (unless it has been deleted) and any new edges and optional data to its children in the encoding tree, one edge per child.

This is different from Genetic Programming in that edge encoding operators are executed in *preorder*, that is, a parent is executed before its children. As a result, child operators are in no sense "arguments" of the parents as they are in the symbolic expression trees commonly used in GP. In fact, just the reverse is true. Each operator modifies a graph edge and passes that edge (and any newly created edges) to its children for future modification.

For example, consider the Double operator shown in Figure 1. This operator has two children in the encoding tree. It receives from its parent a single edge $E(a,b)$ in the graph (where a is the tail of the edge E, and b is E's head). From $E(a,b)$, Double "grows" an additional edge $F(a,b)$. These two edges are each passed to child operators for additional modifications; E is passed to the Double's left child, and F is passed to its right child.

Edge encoding's graph-generation process begins with a graph consisting of a single edge. This edge is passed to the root-node operator in the edge encoding tree, which modifies the edge and passes resultant edges to its children, and so on. Terminal (leaf) nodes in the edge encoding tree have no children, and so stop the modification process for a particular

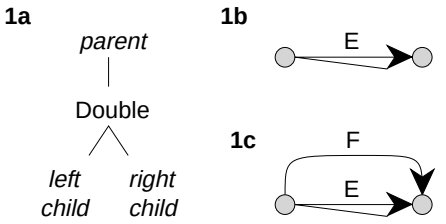

Figure 1. The Double operator. 1a shows the operator relative to its parent and children operators in the encoding tree. 1b shows the initial edge *E* passed to Double from its parent. 1c shows the edges *E* and *F* after Double's execution. *E* and *F* are then passed to Double's left and right children, respectively.

edge. After all nodes in the tree have made their modifications, the resultant graph is returned as the chromosome's phenotype.

The operators in a particular edge encoding are commonly of two forms. First, there are operators which change the topology of the graph, by adding or deleting edges or nodes. Second, there are operators which add semantics to the edges or nodes: labelling edges, assigning transfer functions to nodes, etc.

4 A Simple Edge Encoding

To demonstrate an edge encoding, we begin with a simple set of basic operators which cannot describe all directed graphs, but are sufficient to build all nondeterministic finite-state automata (NFA). These operators each take an edge and no optional data from their parents, and pass on to children at most two resultant edges. A nice property of these operators is that they can be executed in any order, so long as parent operators are executed prior to child operators.

In this simple encoding, each individual consists of a single tree of operators. Assume that each operator is passed some edge $E(a,b)$, which after processing is passed to the left child. The operators which describe the topology of the graph are:

Table 1. Simple topological operators.

Operator	Children	Description
Double	2	Create an edge $F(a,b)$.
Bud	2	Create a node c. Create an edge $F(b,c)$.
Split	2	Create a node c. Modify E to be $E(a,c)$. Create an edge $F(c,b)$.
Loop	2	Create a self-loop edge $F(b,b)$.
Reverse	1	Reverse E to be $E(b,a)$.

These operators are sufficient to develop the topology of an NFA which recognizes any regular expression. To develop the full NFA, some custom semantic operators are necessary

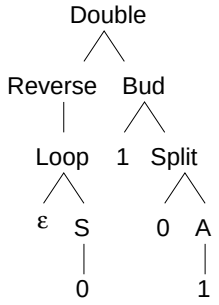

Figure 2. An edge encoding chromosome which describes an NFA that reads the regular expression ((0|1)*101).

to define the starting and accepting states of the NFA and label the edges with tokens. For example, suppose one were trying to develop an NFA that matched the regular expression ((0|1)*101), from a language consisting only of 1's and 0's. Using edge encoding, five more operators are necessary:

Table 2. NFA semantic operators.

Operator	Children	Description
S	1	Assign the head of $E(a,b)$ (node b) to be a starting state.
A	1	Assign the head of $E(a,b)$ (node b) to be an accepting state. It's valid for a state to be a starting state and an accepting state at the same time.
1	0	Label an edge with a "1", that is, define it to be an edge which can be traversed only on reading a 1.
0	0	Label an edge with a "0", that is, define it to be an edge which can be traversed only on reading a 0.
ϵ	0	Label an edge with an "ϵ", that is, define it to be an edge which may be traversed without reading any token.

Figure 2 shows an edge encoding chromosome using these operators whose phenotype is an NFA that reads the regular expression ((0|1)*101). Using a Lisp-like syntax, Figure 2 can be written as (Double (Reverse (Loop ϵ (S 0))) (Bud 1 (Split 0 (A 1)))). Figure 3 shows the development of the NFA from this chromosome.

4.1 An Informal Proof

Here we informally demonstrate that the above operators (in fact, just Reverse, Split, and Double) are sufficient to build NFAs which parse all regular expressions. To do this we will in effect perform the inverse of Thompson's construction

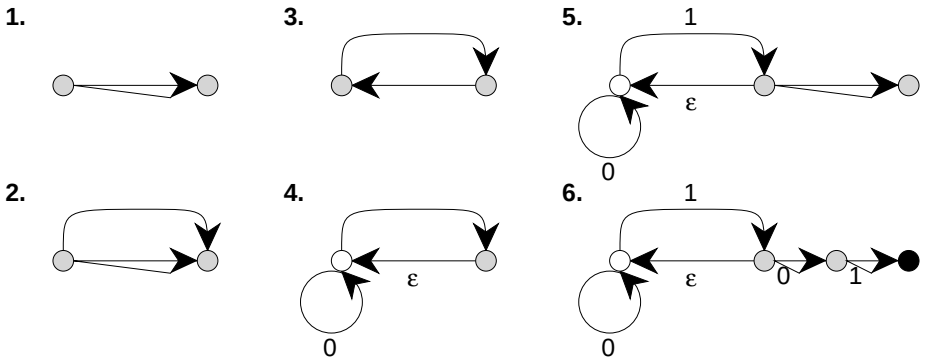

Figure 3. The growth of the NFA from the encoding in Figure 2. The steps are: 1. The initial edge. 2. After applying Double. 3. After applying Reverse. 4. After applying Loop, S, ϵ, and 0. The white circle is a starting state. 5. After applying Bud and 1. 6. After applying Split, 0, A, and 1. The black circle is an accepting state.

as described in [Aho, Sethi, and Ullman 1986], [Thompson 1968]. Our NFA will have exactly one starting and one accepting state. We begin the edge encoding with a single edge whose tail is a starting state and whose head is an accepting state. To do this we begin with an edge encoding of the form (S (Reverse (A (Reverse *E*)))), where *E* is the terminal position in the encoding that defines our initial edge in the NFA. We associate edge *E* with our initial regular expression *e*. The NFA is created by using edge operators to expand *E* into NFA constructions equivalent to subexpressions of *e*. This process continues until *e* has been broken down into nothing but atomic subexpression tokens, at which time all that remains to do is to label the equivalent edges with those tokens.

Thompson's construction first parses a regular expression into its subexpressions, then builds an NFA bottom-up by grouping smaller NFAs that represent those subexpressions. We form the NFA in reverse (top-down) by splicing the subexpressions into the main NFA, "growing" it into its full form. Each growth step is equivalent to one step of breaking an expression into one or two subexpressions; each of those subexpressions is associated with a new edge created in the growth step.

To begin, recognize that given any regular expression *r* over an alphabet Σ, there exist regular expressions *s* and *t* such that at least one of the following is true:

- *r* is ϵ.
- *r* is a single token in Σ.
- *r* can be broken down into *st*.
- *r* can be broken down into *s*|*t*.
- *r* can be broken down into *s**.
- *r* can be broken down into (*s*).

Let *R* be the edge in the NFA associated with the regular expression *r*, as shown in Figure 4. We want to break *r* down into *s* and *t*, and at the same time build from *R* a sub-NFA structure which contains the edges *S* and *T* associated with *s* and *t*. If *r* is ϵ, or a token in Σ, then we can label *R* as appropriate, and we are finished with it. If *r* is of the form *st*, we replace *R* with the construction in Figure 4a by replacing *R*'s terminal node with (Split *S T*). If *r* is of the form *s*|*t*, we replace *R*'s terminal node with (Double (Split (Split ϵ *T*) ϵ) (Split (Split ϵ *S*) ϵ)), resulting in Figure 4b. If *r* is of the form *s**, we replace *R*'s terminal node with (Double (Split (Split ϵ (Double *S* (Reverse ϵ))) ϵ) ϵ), resulting in Figure 4c. If *r* is of the form (*s*), we replace *R*'s terminal node with *S*.

Assuming that *S* and *T* will later expand into the appropriate NFA's to recognize *s* and *t*, then clearly the above constructions will compute *st*, *s*|*t*, *s**, and (*s*) respectively — these are exactly the constructions given in [Aho, Sethi, and Ullman 1986].

Once we've broken *r* down and built sub-NFA structures out of *R*, we associate the edges *S* and *T* with *s* and *t* as appropriate, and repeat this process on *S* and *T*. Since the regular expressions are finite, we are guaranteed that at some point all expressions will break down to ϵ or a token in Σ, at which time we're done expanding the NFA.

5 Additional Operators

The operators presented above develop enough planar graphs to describe all NFAs, among other things. They do not describe more sophisticated planar graphs, however: for example, the complete graph of four nodes (K_4) cannot be generated using the above operators. One way to generate additional planar graphs is to use an operator borrowed from cellular encoding: SplitNode. This operator acts on the node *b* at the head of *E*(*a*,*b*). First, SplitNode creates a new node *n*. Then for each edge *F*(*b*,*c*) for some node *c*, this opera-

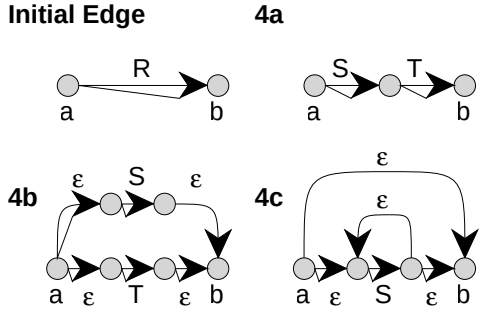

Figure 4. Edge encoding NFA constructions after breaking down the regular expression *r* into subexpressions. 4a shows the construction for the subexpression *st*. 4b shows the construction for *s*|*t*. 4c shows the construction for *s**.

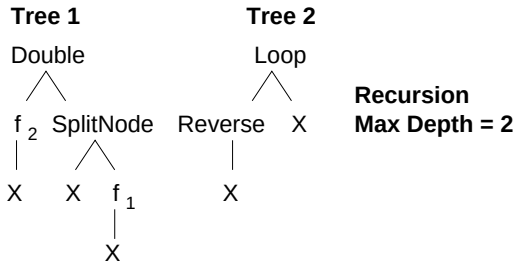

Figure 5. A chromosome using recursion with two trees and a maximum recursion depth of two. The X operator is a boring terminal operator which does nothing at all to its edge. Execution begins with tree **1**.

tor modifies *F* to be *F*(*n*,*c*). Lastly, the operator adds a new edge *G*(*b*,*n*). In essence, SplitNode splits node *b* into two nodes *b* and *n*, moving to *n* all formerly outgoing edges of *b*, and adding single edge between *b* and *n*. Using this powerful operator comes at a price, however. Because SplitNode modifies nodes associated with edges other than the operator's official edge *E*, we must now ensure a total execution order throughout the chromosome, or else the interpretation of the chromosome is ambiguous. One way to do this is to always traverse the tree in depth-first, left-to-right order.

An important feature of edge encoding is its use of *modular operators*. Modular operators in edge encoding are similar to Automatically Defined Functions found in GP [Koza 1994]: the chromosome is no longer a tree but a forest of *n* trees, each of which can be "called" by operators in other trees. For each tree **M**, we add to our collection of operators a special operator f_m, which has a single child. Additionally, we define a maximum recursion depth of *r* modular operator calls. When f_m is executed, it passes the edge it was provided to the root operator of **M**, which begins execution for that tree. Hence, when f_m is executed, it is as if **M** was grafted in at f_m's position. The recursion depth is incremented by one until all operators in the tree **M** have completed execution. If and only if the recursion depth exceeded *r* at the time that f_m was executed, f_m does not pass the edge to **M**'s root operator but instead passes it to f_m's sole child. Initially, execution begins with the first tree in the forest.

This scheme makes it possible for a tree in the chromosome to repeatedly call other trees or recursively call itself to develop many repetitive sections of a graph. This can grow large, sophisticated graphs from relatively small chromosomes. As an example, Figure 5 shows a small two-tree chromosome which, even with a low recursion depth (2) still manages to grow a complex graph by using SplitNode and recursive operators. Figure 6 shows the growth pattern of the phenotype graph.

There are many other possible operators. For example, Cut is a useful terminal operator which simply eliminates its edge from the graph. Export is a two-child operator which creates a new edge separated from the graph entirely, thereby making possible disconnected graphs. When passed edge *E*(*a*,*b*), MergeNode could merge *a* and *b* into a single node, eliminating *E*. In addition to other uses, this operator can create K_1 with optional self-loops. And of course, there are many variants of existing operators. For example, LoopTail might add a self loop to the *tail* of the edge rather than its head, an abbreviation for (Reverse (Loop (Reverse ...) ...)).

6 Creating All Graphs

Although edge encodings using the operators above can describe many interesting graphs, they cannot describe *all* graphs, particularly ones that have a large number of non-planar interconnections. It appears that to create all possible graphs one must either impose an unusual traversal order (like cellular encoding's breadth-first traversal with "Wait" operators), or allow operators to pass additional information to one another, aside from a single edge. Because we feel that it will better help to preserve the semantics of building blocks after crossover, we have been experimenting with the second of these options.

In our scheme, operators pass to their children not only an edge but a (possibly empty) stack of graph nodes. This permits earlier operators to create new nodes to which later operators may attach edges. An operator may modify this stack by making a copy of it, modifying the copy, and passing the copy to its children. Operators which do not modify the stack (including all operators discussed so far) simply pass it through to their children. The initial stack passed to the root operator is empty.

Assume that each operator is passed an edge *E*(*a*,*b*), which after processing is passed to its left child. The following op-

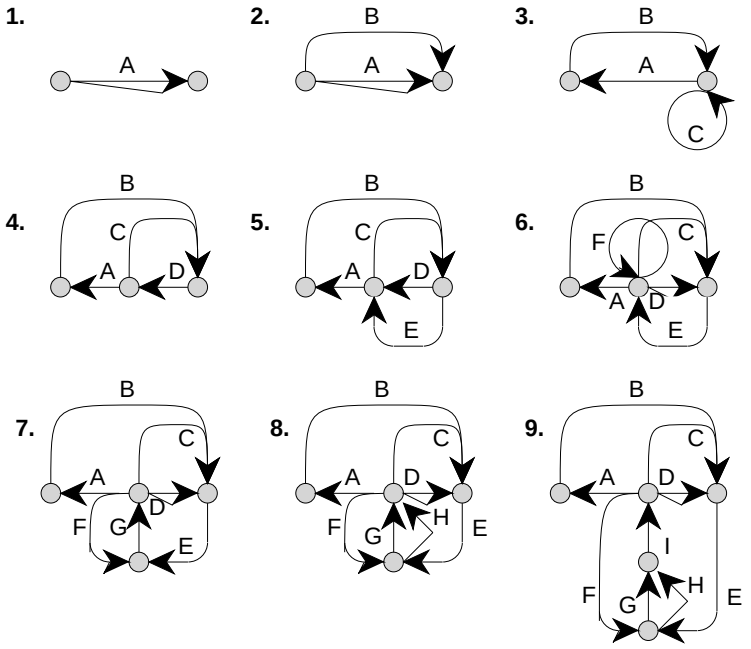

Figure 6. The growth of the graph structure defined in Figure 4. Edges are labeled to make it easier to keep track of them in this example. 1. The initial edge. 2. After Double. 3. After tree **2** is called, which performs Loop and Reverse. 4. After SplitNode. 5. After tree **1** is recursively called, performing Double. 6. After tree **2** is called a second time, which performs Loop and Reverse. 7. After Splitnode. 8. After tree **1** is recursively called a second time, performing Double. Tree **2** cannot be called any more (the recursion depth is at maximum). 9. After Splitnode. Tree **1** can no longer be recursively called (maximum recursion depth), so the graph is finished.

erators are sufficient to describe the topology of all connected graphs of two or more nodes:

Table 3. Operators for creating general graphs.

Operator	Children	Effect
Double	2	Create an additional edge $F(a,b)$.
Loop	2	Create an additional edge $F(b,b)$.
Reverse	1	Modify E to be $E(b,a)$.
Cut	0	Eliminate edge E.
Push	1	Create a new node c. Make a copy of the stack. Push c onto the copy, and give the copy to Push's child.
Attach	3	Make a copy of the stack. If the stack is empty, create and push a new node onto the copy. Let the top node on the stack copy be c. Create two new edges $F(a,c)$ and $G(b,c)$. Pop c off the stack copy, then pass the copy to Attach's children. This in effect "attaches" edges to the top node on the stack, forming a triangle EFG.

We will not present our proof here that these operators are sufficient to create all connected graphs of two or more nodes. Instead, we present only the following outline: to create an arbitrary connected graph of n nodes, first push n-2 nodes onto the stack in a series of Push operations. Then create the complete graph K_n of n nodes, with some multi-edges, by performing Attaches from the original edge to nodes, and from these newly attached edges to other nodes. Add multi-edges and self-loops as necessary using Loop and Double. Reverse edges into their proper direction. Then Cut out unwanted edges to form the resultant graph. Presumably there are more efficient ways to construct any particular graph, but this construction shows that the encodings can describe all possible graphs. Figures 7 and 8 give an example of an edge encoding which uses Push and Attach to produce a non-planar graph (K_5).

There are a large number of other possible stack-manipulation operators. For example, Pop might pop a graph node off its stack without making any attachments. Or Push-Tail might push the tail node of an edge onto the stack. Variations on Attach might attach only a single edge or attach edges but not pop the node off the stack afterwards.

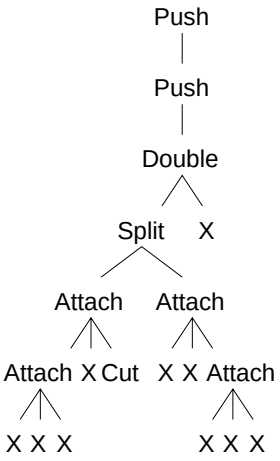

Figure 7. One of many chromosomes which produce one version of K_5 with no multi-edges or self-loops. The X operator is a boring terminal operator which does nothing at all to its edge.

Although these kinds of operators can produce all graphs in theory, our preliminary results indicate that graphs with a high number of interconnections demand a convoluted combination of operators. This may make some common graphs difficult to evolve. We have been experimenting with a number of ways to flatten the space of possible graphs for genetic search purposes. One observation is that useful Pushes always appear earlier in the tree than Attaches. We can take advantage of this by eliminating Pushes entirely, assuming instead that the initial stack passed to the root node automatically holds as many graph nodes as will be needed by later Attaches.

7 Initial Comparison with Cellular Encoding

Both edge encoding and cellular encoding are strong enough to describe all graphs with two or more nodes. However, these encodings differ greatly in the types of graphs their evolutionary landscapes favor. Cellular encoding favors graphs with high numbers of interconnections, that is, those with a high edge/node ratio. To remove many interconnections, cellular encodings go through a complicated routine which increments "node registers" and then performs "Cut" operations. In contrast, edge encoding favors graphs with low numbers of interconnections, that is, those with a low edge/node ratio. To create dense edge connections other than multi-edges, edge encodings likewise require a complicated combination of operators.

Another interesting comparison is the size of encodings. When simply evolving network topologies, edge encodings are often larger than cellular encodings. This is because an edge encoding operates on edges, and so there is always at least one operator per edge in an edge encoding. However, if edge labels need to be encoded distinctly from each other (such as continuous-valued weights in a neural network), the two encodings are often similar in size, because cellular encoding must use one or more distinct operators to label each edge. Additionally, it appears that while the encodings may differ in size, both encodings' phenotype-creation time is roughly bounded by the size of the graph, not the encoding size.

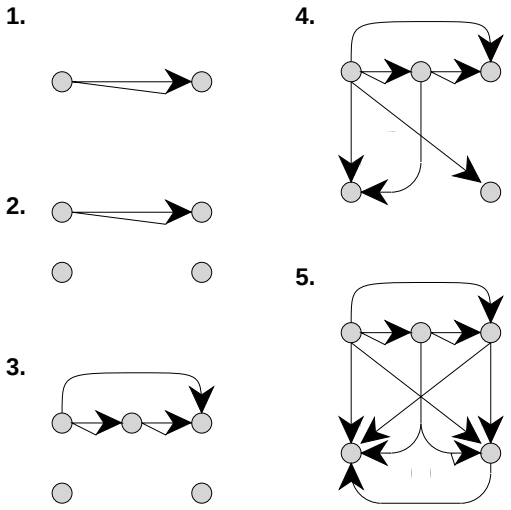

Figure 8. The growth of the graph structure described in Figure 7. 1. The initial edge. 2. After the two Pushes. 3. After Double and Split. 4. After the leftmost Attaches and Cut. 5. After the remaining Attaches.

A final but important comparison is the degree to which each scheme takes advantage of building blocks in its graph construction. In many domains, a highly fit chromosome often consists of subparts which in and of themselves are reasonably fit, or which may serve as useful components for other individuals. In these domains, the genetic landscape favors the use of building blocks, so a chromosomal encoding which promotes the development of building blocks is very valuable. Cellular encoding attempts to promote building blocks through modular reuse and a GP-like crossover mechanism. However, we are not convinced that this promotes building blocks as much as one might expect, since crossover between individuals can often result in dramatic, unexpected, non-local changes in both individuals. This is because cellular encoding's breadth-first traversal and execution-order-dependent operators make it possible for a subtree's *position* in an individual to have significant effects on the meaning of the subtree and on its effects on the phenotype. The lack of good building-block promotion may not be so serious given

the primary domain for cellular encoding, namely the evolution of standard feedforward or fully-recurrent neural networks with large numbers of connections. [Angeline, Saunders, and Pollack 1994] make a strong argument that not only do basic blocks not help in designing such neural networks, but that these domains are *GA-deceptive* [Goldberg 1989], meaning that the development of building blocks actively *inhibits* finding a solution. Nonetheless building blocks will probably prove useful for a great many other graph domains. In these domains, we think that edge encodings may have an advantage because the effects of crossover are highly localized, and subnetwork characteristics are mostly preserved.

8 Conclusion

Edge encoding, like cellular encoding, allows one to use standard S-expression-based Genetic Programming techniques to evolve arbitrary graph structures. The resulting graphs may be employed in various ways, for example as neural networks, as automata, or as knowledge-base queries. Each encoding scheme biases genetic search in a different way; for example, cellular encoding favors graphs with high edge/node ratios while edge encoding favors graphs with low edge/node ratios. For this reason, we believe that certain domains will be much better served by one scheme than by the other.

We also believe that edge encoding has greater advantages than those conferred by the mere *difference* of search bias. For example, we believe that edge encoding allows for "cleaner," more intuitive control over edges and nodes in graphs for many applications. We also believe that edge encoding allows for better use of genetic building blocks that maintain their utility after crossover. In this preliminary report, however, we focused on the technical description of edge encoding, and we therefore presented little evidence for our beliefs about the advantages of the technique. The evidence for (or against) these beliefs will result from experiments that compare the two encoding schemes for a range of problems; such experiments are currently in progress.

Acknowledgements

This research was supported in part by grants from ONR (N00014-J-91-1451), AFOSR (F49620-93-1-0065), and ARPA contract DAST-95-C0037.

Bibliography

Aho, A.V., R. Sethi, J.D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Reading: Addison-Wesley. 121–125.

Angeline, P.J., G.M. Saunders, J.B. Pollack. 1994. An Evolutionary Algorithm that Constructs Recurrent Neural Networks. In *IEEE Transactions on Neural Networks.* 54–65. IEEE.

Boers, E.J.W., H. Kuiper, B.L.M. Happel, and I.G. Sprinkhuizen-Kuyper. 1993. Designing Modular Artificial Neural Networks. In *Proceedings of Computing Science in The Netherlands*, H.A. Wijshoff , editor. 87–96. Amsterdam: SION, Stichting Mathematisch Centrum.

Collins, R. and D. Jefferson. 1991. An Artifical Neural Network Representation for Artificial Organisms. *Parallel Problem Solving from Nature*. H.P. Schwefel and R. Männer, editors. 269-263. Berlin: Springer-Verlag.

Fullmer, B. and R. Miikkulainen. 1991. Using Marker-based Genetic Encoding of Neural Networks to Evolve Finite-state Behavior. In *Proceedings of the First european Conference on Artificial Life (ECAL-91).* Paris.

Goldberg, D. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Reading: Addison-Weley. 41–54.

Gruau, F. 1992. Genetic Synthesis of Boolean Neural Networks with a Cell Rewriting Developmental Process. In *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92),* L. Darrell Whitley and J. David Schaffer, editors. 55–74. Los Alamitos: IEEE Press.

Kitano, H. 1990. Designing Neural Networks Using a Genetic Algorithm with a Graph Generation System. In *Complex Systems* 4. 461–476.

Koza, J.R. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: The MIT Press.

Koza, J.R. and J.P. Rice. 1991. Genetic Generation of Both the Weights and Architecture for a Neural Network. In *IEEE International Joint Conference on Neural Networks.* II-397–II-404. Seattle: IEEE.

Lindgren, K., A. Nilsson, M.G. Nordahl, I. Råde. 1992. Regular Language Inference Using Evolving Neural Networks. In *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92),* L. Darrell Whitley and J. David Schaffer, editors. 55–74, Los Alamitos: IEEE Press.

Teller, A. 1996. Evolving Programmers: The Co-evolution of Intelligent Recombination Operators. In *Advances in Genetic Programming II,* P. Angeline and K. Kinnear, editors. Cambridge: MIT Press.

Thompson, K. 1968. Regular Expression Search Algorithm. In *Communications of the ACM.* 11:6. 419–422.

Zomorodian, A. 1995. Context-free Language Induction by Evolution of Deterministic Push-down Automata Using Genetic Programming. In *Working Notes of the Genetic Programming Symposium, AAAI-95.* Eric Siegel and John Koza, chairs. AAAI Press.