Queue-based Genetic Programming

Elko Tchernev
CSEE Dept., University of Maryland Baltimore County (UMBC)
1000 Hilltop Circle
Baltimore, MD 21250, USA
+1 410 455 3762
etcher1@umbc.edu

Dhananjay S. Phatak
CSEE Dept., University of Maryland Baltimore County (UMBC)
1000 Hilltop Circle
Baltimore, MD 21250, USA
+1 410 455 3624
phatak@umbc.edu

ABSTRACT
In this paper, we describe the use of a queue instead of a stack or a parse tree for the internal representation and genetic operations of a Genetic Programming system. Specifically, implementation issues and application areas are discussed.

Categories and Subject Descriptors
E.1 [**Data**]: Data Structures – *graphs and networks, lists, stacks and queues, trees.* F.1.1 [**Theory of Computation**]: Models of Computation – *push-down automata, self-modifying machines, probabilistic computation.* I.2.6 [**Artificial Intelligence**]: Learning – *connectionism and neural nets.*

General Terms
Algorithms, Experimentation, Theory.

Keywords
Genetic Programming, Tree, Stack, Queue.

1. STACK-BASED GENETIC PROGRAMMING
Genetic Programming, or GP, has traditionally used tree-based techniques for representation and reproduction. The most widely used crossover method is subtree crossover, and the majority of the alternatives in published literature are variants thereof. However, if the trees are manipulated in their prefix or postfix form, other approaches exist that preserve the syntactic integrity of the participating parent trees. Stack based Genetic programming, introduced by Perkis in [3], represents programs as lists of nodes of functions or terminals that consume their inputs from a stack and place their outputs on a stack. These implementations, including the early work of Bruce, Stoffel and Spector, [1], [5] and later [4], do not try to preserve the stack correctness of the individuals in the population, but rather rely on the evaluation framework to identify any stack underflow or overflow. In contrast, in GP with stack-correct (Forth) crossover, introduced by Tchernev in [6] and [7], the crossover operators manipulate the post order representation of the program tree. Because the crossover points are chosen to have compatible stack depths, no malformation is possible. If the initial population is stack-correct (no individuals have underflow, and the final stack depth equals the desired number of outputs), it is guaranteed that all individuals produced by using stack-correct crossover will be stack-correct.

2. QUEUE-BASED GENETIC PROGRAMMING
A stack is a LIFO structure – the last item placed into it (on top) is the first to be removed by subsequent operations. On the other hand, the queue, a FIFO structure, has almost the same properties. Like the stack, it can accept any number of items (up to its maximum capacity, of course), and they remain ordered the way they were received. The only difference is that the extraction order is reversed – the first item placed into the queue, is the first to be removed. Therefore, replacing the stack of a stack-based GP system with a queue requires very little change in order to produce a queue-based system. The functions and terminals of the system need to be modified such that values are read from the queue, where they were popped from the stack before, and results are written into the queue, where they were pushed on the stack before. Analogous to stack-correctness in stack-based GP, we can define queue-correctness for queue-based GP. An evolved individual is queue-correct, if at no point in it is there queue underflow (reading an empty queue), and at the end the queue contains the exact number of output values that the problem specifies (there are no extra values produced).

3. PROPERTIES OF QUEUE-BASED GP
3.1 Parse tree equivalence
As shown in [6] and [7], in stack-based GP, the individuals with their corresponding stack-based execution sequences, represent postorder traversals of parse trees, and the stack depth diagrams preserve the topology of these trees. Because of the stack machine's locality of scope, subtrees in stack-based representations are contained contiguously in the node list; subtree crossover can be performed by simple two-point cuts, provided the points are identified. This is not the case in queue-based GP: except when trivially only one subtree exists, the locations of the subtree nodes in the node list are not contiguous. In general, it is not possible to perform a two-point crossover operation whose effect is to swap subtrees in the corresponding infix expressions. As can be seen on Figure 1, the semantic composition of a queue-based expression is highly context-dependent and non-local. The operands to any function have been placed into the queue by a function or terminal at a distant location in the node list, and the result from the function's evaluation will be used by another function again further down the list. The distance between nodes acting on re-

lated data, is, of course, the queue length; only after all earlier items are acted on, can a particular data item be modified.

3.2 Crossover possibilities

Since the queue, from a black-box perspective is equivalent to a stack (ignoring the order of items for a moment), it is possible to perform the same crossover operations as in stack-based GP, and obtain queue-correct individuals. Using queue length instead of stack depth for crossover point choice, all the stack-correct crossover methods from [7] can be applied. (They are One-Point, Two-Point, Subtree and Delta). Their effects on the participating individuals' parse trees, however, are very different from the effects in Stack-based GP.

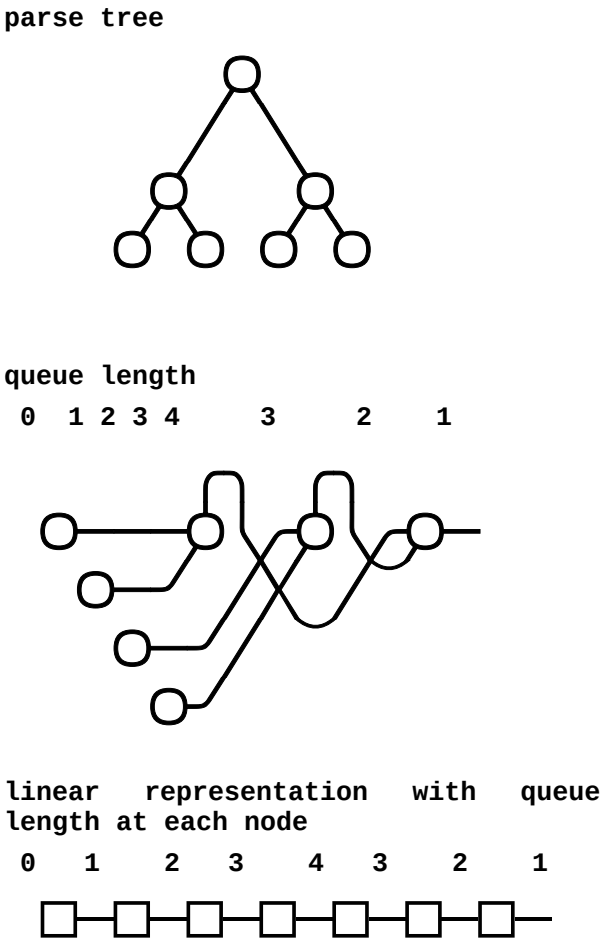

Figure 1. A sequence of nodes, their corresponding queue picture and parse tree

The rule for subtree crossover in stack-based GP can be mechanically applied in queue-based GP, and will produce queue-correct individuals. The effect on the parse-tree equivalents will definitely be different from exchanging subtrees. Similarly, the effects of the other crossovers will be different from their effects in stack-based GP, as can be seen in Figure 2, Figure 3 and Figure 4.

4. APPLICATION DOMAINS

Crossover in queue-based GP is particularly disruptive. In traditional tree- or stack-based GP, a sequence of nodes taken from one individual and placed into another, preserves its data cohesion to some extent. Its function nodes can operate on data from leaf nodes that are contained in the same node sequence. In queue-based GP, the likelihood of this happening is much lower.

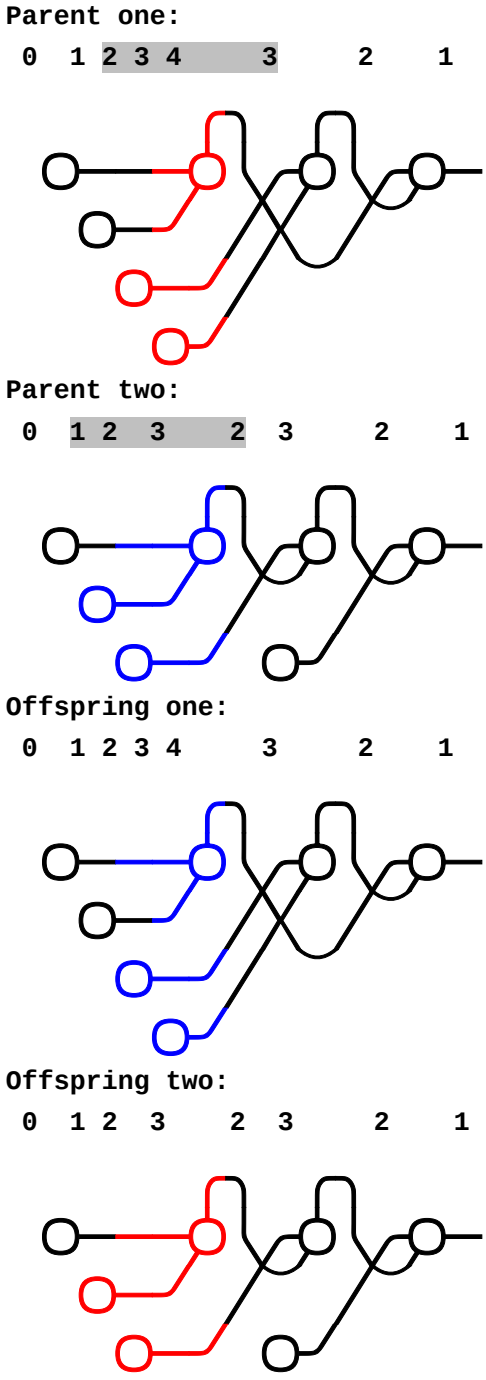

Figure 2. Crossover point selection and result using the "subtree" rule

Guest functions operate on items in the queue that were most likely placed there by nodes in the host individual, and guest leaf nodes produce data to be used away from them, most likely by host functions. In order to maintain some sort of data cohesion,

the crossover segment needs to be sufficiently long, and to be placed in a context with the same overall queue length. Therefore, the application of queue-based GP to evolve highly modular programs with short useful building blocks, for example any of the Parity problems, is not recommended.

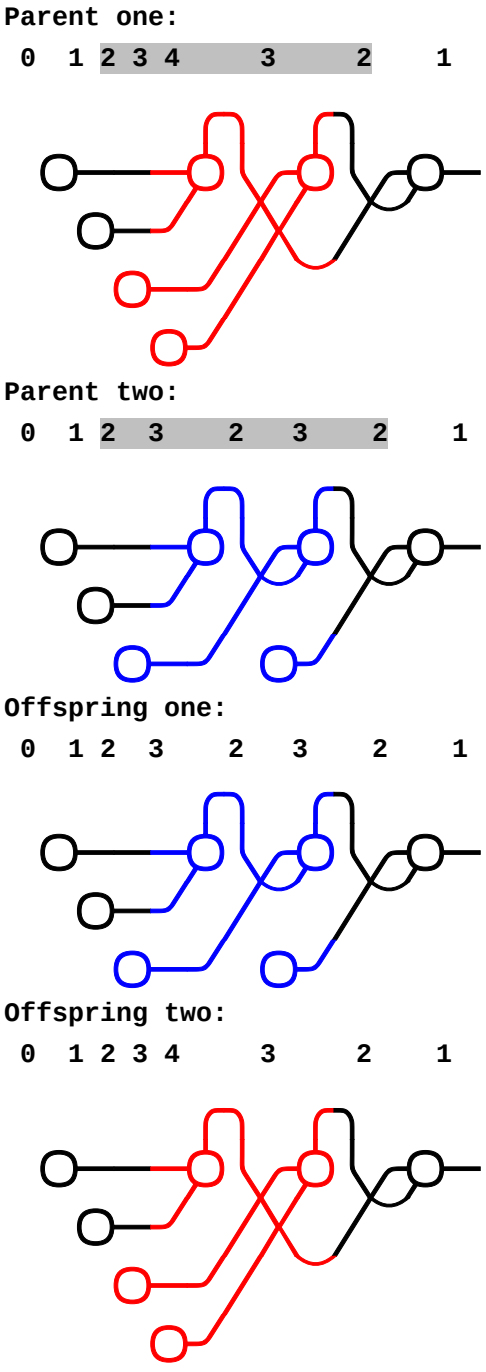

Figure 3. Crossover point selection and result using the "two-point" rule

A more suitable domain would be evolving complex functions with no repeated internal structure, or situations where the entire contents of the queue must be processed in some manner that is not local. The second case is particularly matched to the structure of layered feed-forward neural networks. If the items in the queue are activations from a previous layer, and a sequence of functions represents neural network nodes, then the processed results go to the other end of the queue, out of the way of the results from the previous layer that are still being processed.

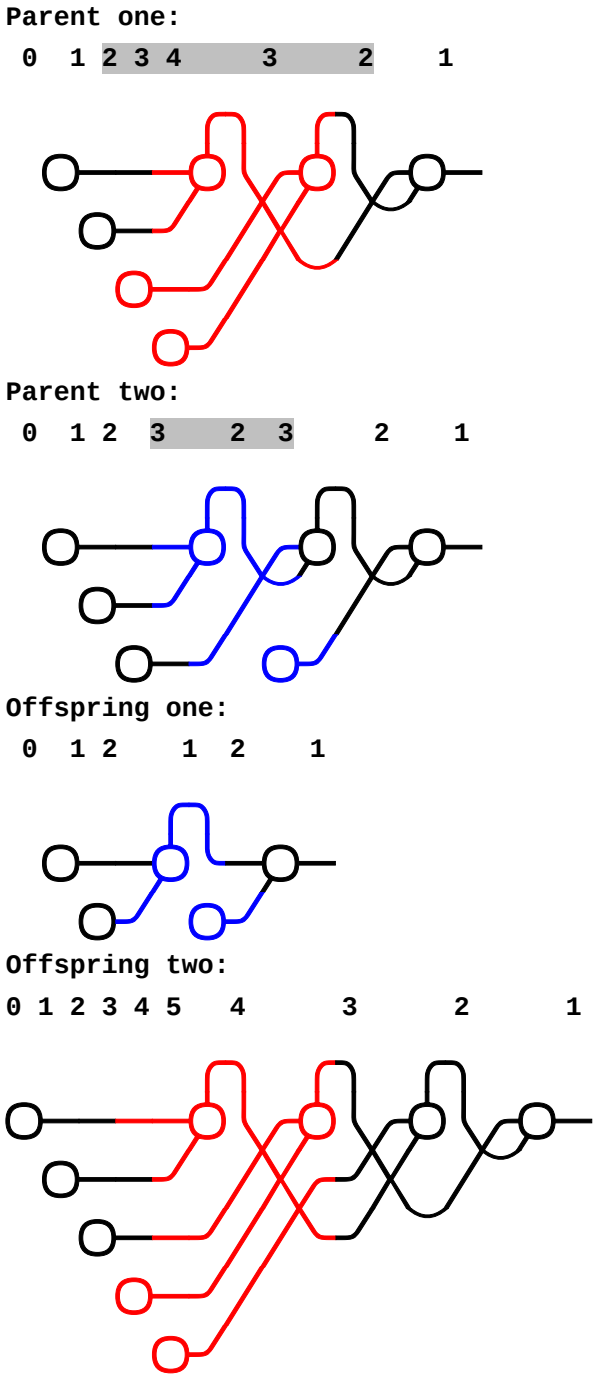

Figure 4. Crossover point selection and result using the "delta" rule

This is much better than in stack-based GP for evolving neural networks, where the results from a node go to the top of the stack

and prevent easy access to older results. A sample neural network realized with the queue paradigm can be seen in Figure 5.

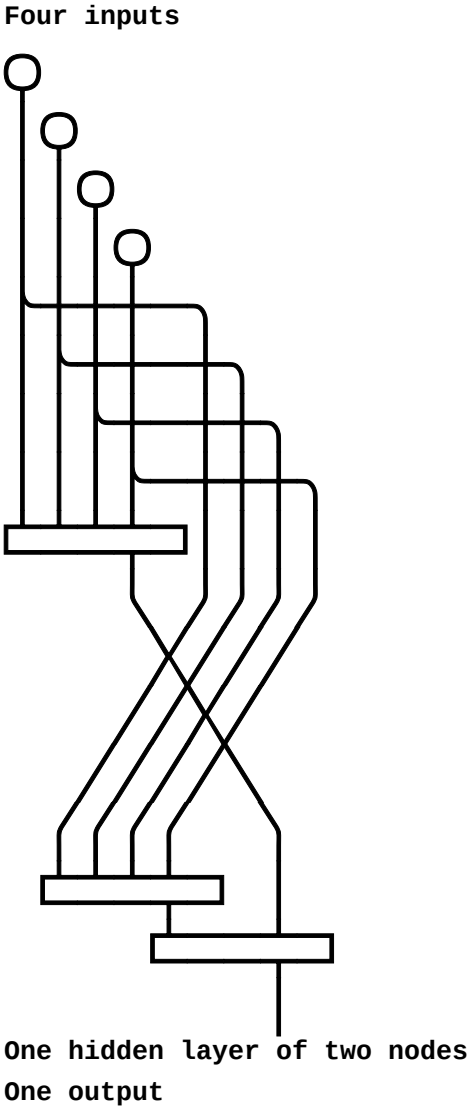

Figure 5. Feedforward Neural Network implemented using a queue

5. CONCLUSION AND FUTURE WORK

At the time of writing, trying to evolve small, structured and easy problems (the two boxes problem from [2]) with conservative population sizes were not successful, as expected. Experiments are underway in applying queue-based GP to evolve layered neural networks; the outcome will show what crossover methods and settings (if any) produce results with specific desired properties.

6. ACKNOWLEDGMENTS

This work was supported in part by NSF grants ECS-9875705 and ECS-0196362.

7. REFERENCES

1. Wilker Shane Bruce. "The Lawnmower Problem Revisited: Stack-Based Genetic Programming and Automatically Defined Functions", In Genetic Programming 1997: Proceedings of the Second Annual Conference, Stanford University, CA, USA, San Francisco, CA, USA, pp. 52-57, 1997.

2. John R. Koza. "Genetic Programming: On the Programming of Computers by Means of Natural Selection", Cambridge, MA, USA, MIT Press, 1992.

3. Tim Perkis. "Stack-Based Genetic Programming", In Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Orlando, Florida, USA, IEEE Press, pp. 148-153, 1994.

4. Lee Spector and Alan Robinson. "Genetic Programming and Autoconstructive Evolution with the Push Programming Language", In *Genetic Programming and Evolvable Machines*, pp. 7-40, 2002.

5. Kilian Stoffel and Lee Spector. "High-Performance, Parallel, Stack-Based Genetic Programming", In Genetic Programming 1996: Proceedings of the First Annual Conference, Stanford University, CA, USA, MIT Press, pp. 224-229, 1996.

6. Elko Tchernev. "Forth Crossover Is Not a Macromutation?", In Genetic Programming 1998: Proceedings of the Third Annual Conference, University of Wisconsin, Madison, Wisconsin, USA, San Francisco, CA, USA, pp. 381-386, 1998.

7. Elko Tchernev. "Stack-Correct Crossover Methods in Genetic Programming", In Late Breaking Papers at the Genetic and Evolutionary Computation Conference (GECCO-2002), New York, NY, AAAI, pp. 443-449, 2002.