Cooption and Catalysis in a Model of Technological Evolution

Eric Scott, Andrews University/Santa Fe Institute (Mentor: J. Doyne Farmer)

02 September, 2010

Abstract

Most evolutionary algorithms ultimately focus on optimizing solutions to a single target function, coevolution and related methods notwithstanding. Cooptive phenomena between organisms adapted to distinct environmental niches, however, lie at the heart of the evolution of complex functions in nature and technology, where solutions adapted for one problem are repurposed to solve another, related problem. Boolean functions have become a popular toy model for exploring the dynamics of such processes, and provide insight into new approaches to evolutionary computation. We implemented the basics of a model of combinatorially evolving logic circuits developed by Brian Arthur and Wolfgang Polak, and began to explore the sort of cooptive and catalytic phenomena its success depends upon. We observed a significant difference in the dynamics of evolution between when the full suite of fitness functions is present, versus when only a few pieces of the selective pressure are active at a time. Future work will examine these dynamics in more detail.

1 Introduction

Evolutionary computation (EC) aims to exploit the complexity-generating power of adaptive processes in nature to optimize solutions for real-world, human-defined problems. To date, the state of the art is notoriously limited when compared to the algorithms' natural counterparts. The impass is described by Yaneer Bar-Yam[3]:

> While the GA/EA [Genetic Algorithm/Evolutionary Algorithm] approach can help in specific cases, it is well known that evolution from scratch is slow. Thus it is helpful to take advantage of the capability of human beings to contribute to the design of the system... A better understanding is necessary in order to realize the promise of evolutionary methods. The objective revolves around mimicry of the processes that promote rapid innovation through competition. The creation of an effective «artificial ecology»or «artificial economy»requires design."

It is the present author's emphasis that part of the solution to these difficulties lies in Stephen Jay Gould's concept of *exaptation*, in which functions evolved for one purpose or need are coopted and further refined for another task or environment.[10] Exaptation exploits commonalities and chance relationships between ecological niches to learn and ultimately generalize from multiple fitness functions and develop higher quality and/or more complex solutions. In this paradigm, *intermediate functions* play a major role.

The importance of diverse fitness environments to evolution of complex and/or robust features has been observed in many computational models[13][14][15][17]. Logic circuits and other genetic programming paradigms[12] have often provided a sandbox for exploring the dynamics of evolution in part because of the ease with which they can be analyzed, but also because of the (limited) analogy they provide to genetic regulatory networks.[11] Macia and Solé[14] have used them to draw attention to the role degeneracy plays in organism robustness, while Parter, Kashtan and Alon[15] have modeled the theory of Facilitated Variation[8] to show the sponteanous development of modularity under exposure to multiple, distinct fitness environments. Similarly, Lenski et al.[13] have used the *Avida* artificial life simulator to highlight the importance of intermediate logic functions in the evolution of solutions to more complex functions.

The EC tools that come closest to utilizing the notion of multiple environments include optimization of dynamic environments[4] (fitness functions that change over time), coevolutionary algorithms[6] (in which multiple solutions compete or cooperate interactively), multiobjective optimization (where multiple conflicting objectives are pursued simulatenously)[18], and mixtures thereof (for example, [9]). None of these methods have the goal of utilizing exaptive analogies between distinct problems and, with the exception of coevolution, neither do they attempt to implement intermediate developmental

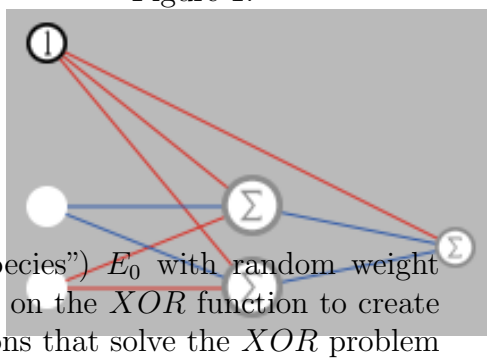
steps as the dominant solution mechanism.

We analyze the evolution of boolean functions in a model of technological evolution developed by Arthur and Polak[2], with the goal of identifying the potential of one logic function being coopted for the development of another, and to chart sequences through this path-dependent space of selection pressures analogous to the transformations undergone by reactants in a chemical network. Under this metaphor, we conceptualize selection pressures as enzymes, optimized solutions as educts and products, and cooption events as catalysis.

1.1 ANN Illustration

We illustrate this perspective with a toy example using an Artificial Neural Network (ANN). A simple ANN was trained via backpropagation to simulate the OR, XOR, and $COUNTONES$ (count the number of ones in the input signal) functions for a two-bit input string (See Figure 1).[16]

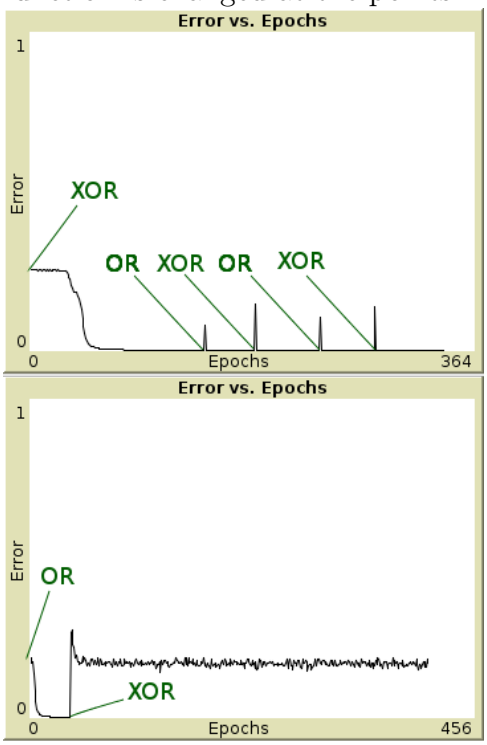
Figure 1:



The network is initialized to state ("species") E_0 with random weight values on $(-0.1, 0.1)$. We may then train it on the XOR function to create the "species" X_0. The X_i's represent solutions that solve the XOR problem fairly well, O_i's solve OR, C_i's solve $COUNTONES$, and D_i's fail to solve any of the three. Now, if we use X_0 as the initial condition to train a solution O_1 to OR, it takes only one training cycle of backpropagation. Since this is significantly shorter than the number of cycles required to train O_0 from E_0, we say that X_0 can be *exapted* by OR. Alternatively, one could say that XOR *catalyzes* the production of a solution to OR. Using O_1, then, as the initial condition to train a new solution to XOR, we get a cycle that may be repeated *ad infinitum* (See Figure 2a). On the other hand, if we train E_0 on OR first, and use the resulting solution O_0 as the initial condition to train XOR, we see that the algorithm never converges onto a solution to XOR. We may say, then, that OR has a *negative* catalytic effect on solving XOR (See Figure 2b).

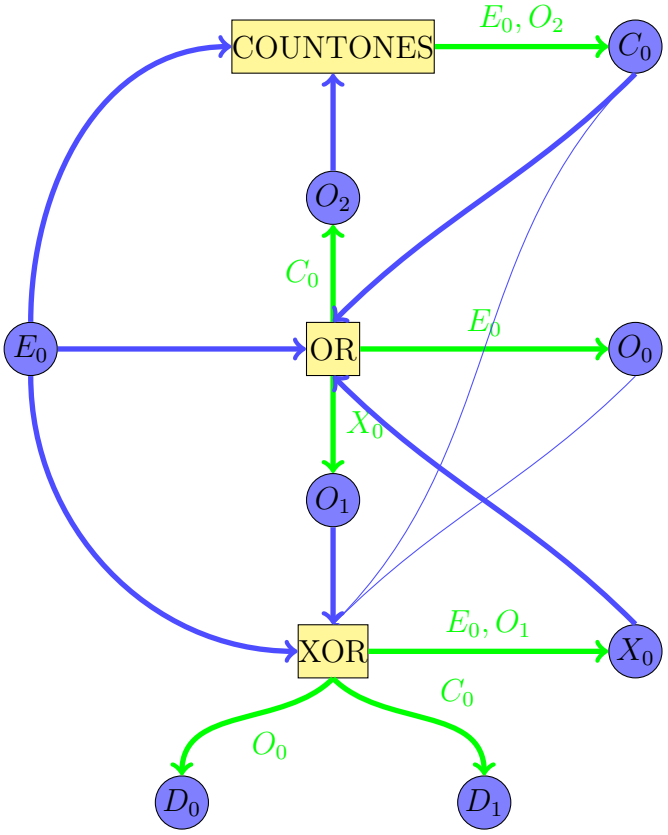
Exaptive relationships between the three objective functions are represented by the complicated graph in Figure 3, which imitates the visualization of catalytic networks.[7] The green edges point from the fitness function to

Figure 2: Error vs. optimization time for backpropagation. The fitness function is changed at the points marked by green labels.



the species that was optimized on it. The edge labels denote the initial condition used to create the species. Note that solutions to the same problem generated from different initial conditions (ex. $\{O_0, O_1, O_2\}$) display different exaptive (chemical) properties. Two or more solutions are represented by the same node in the graph if they display similar chemical properties.

Figure 3:



2 Model

The model developed by Arthur and Polak[2] is unique in that the only evolutionary operator is the combination of existing circuits (i.e. the composition of boolean functions). Since its primary motivation is to explore the path-dependent dynamics of technologies, as engineers synthesize tools and

components that were developed in different environments to meet separate needs, there is no mutation operator. This divergence from the biological paradigm comes with a benefit in that, by composing boolean functions of different dimensionality, solutions can pass through fitness functions of different dimensions, thus broadening the class of functions amongst which we can look for catalytic effects. For example, if the function $A : \mathbb{B}^m \mapsto \mathbb{B}^n$ has been evolved to solve fitness function α that maps m inputs to n outputs, it can be composed with some other function into a new circuit $B : \mathbb{B}^p \mapsto \mathbb{B}^q$ and tested against a function β with a different number of inputs and outputs.

The algorithm proceeds as follows:

- Initialize a set primitives to contain only NAND.
- Initialize a pool of circuits to be empty.
- Select from 2–12 components from the primitives set and/or pool with a weighted choice function.
- Randomly wire together n variations of the circuit
- For each variation:

  Evaluate the average fitness against a battery of fitness functions.

  If the variation's truth table completely matches one of the goals, add it to the primitives set.

Components were selected randomly from the primitives set with a probability of 0.8. Otherwise, a component was selected from the pool via tournament selection on average fitness.

We aim to answer questions like:

- Does the presence of an AND *circuit* in the primitives set facilitate the evolution of a HALF-ADDER circuit?
- Does the presence of both AND and HALF-ADDER *fitness functions* as a multiobjective selection pressure facilitate the evolution of a HALF-ADDER?

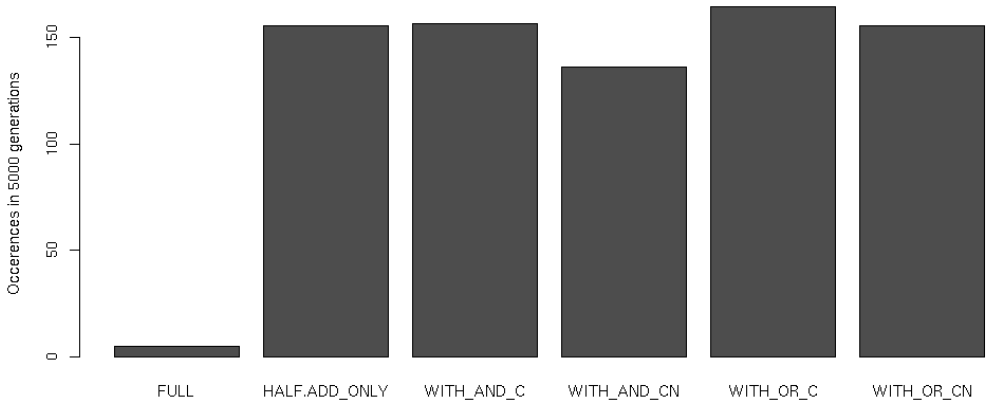
These two different types of events could be roughly (and somewhat artificially) distinguished as *cooptive* and *catalytic*, respectively.

3 Results

Several scenarios were constructed with different initial sets of primitives and suites of fitness functions. The rate at which desired functions were evolved was then measured over several thousand generations.

Scenario	*Initial Primitives*	*Fitness Functions*
HALF.ADD_ONLY	NAND	HALF-ADDER
WITH_AND_C	NAND, AND	HALF-ADDER
WITH_AND_CN	NAND, AND	HALF-ADDER, AND
WITH_OR_C	NAND, OR	HALF-ADDER
WITH_OR_CN	NAND, OR	HALF-ADDER, OR

Figure 4: The results of one trial of independent evolutions of a HALF_ADDER in 5,000 generations under various scenarios. With all fitness functions active and only NAND in the initial primitives set, very few occurances occured. Roughly 150 occurences appeared under a variety of simpler scenarios.



In all the scenarios tested besides FULL, the mean occurence rate for 5,000 generations was close to 150 with a standard deviation close to 10. No statistically significant differences between the scenarios were detected. A broader family of scenarios will need to be analyzed in the future to determine the sort of interactions that lead to the reduction of occurrences in the FULL scenario.

In general, our simulation does not converge on solutions as fast as Arthur and Polak's. This may have to do with differences in our selection mechanism, or with parameters, or it could be a bug in our code.

4 Discussion

We implemented the basics of Arthur and Polak's model, and began to explore the sort of exaptive phenomena its success depends upon. We observed a significant difference in the dynamics of evolution between when the full suite of fitness functions is present, versus when only a few pieces of the selective pressure are active at a time. We determined that there is need for further analysis of more complex scenarios before we can map the sort of relationships we observed in the Artificial Neural Network example.

In the future we would aim to map these dependencies in detail, and to answer the question:

- Are the catalytic effects of fitness functions greater than the sum of their parts?

Other items to consider include the usefulness of the combinatorial model of evolutionary processes, and how it might be extended. The work of Parter et al. in [15] also leads one to wonder about the significance of facilitated variation-inspired models for evolutionary computation, in which modularity emerges dynamically rather than being imposed *a priori*. And finally, an ambitious endeavour might seek to generalize catalytic evolution beyond boolean functions to more complex, arbitrary classes of objective functions.

5 Implementation

The simulation was implemented in Common Lisp. As with most algorithms, it is easier to describe qualitatively than to code, so a few pointers are offered here. The full code is available at `http://github.com/SigmaX/CircuitTech`.

5.1 Representation

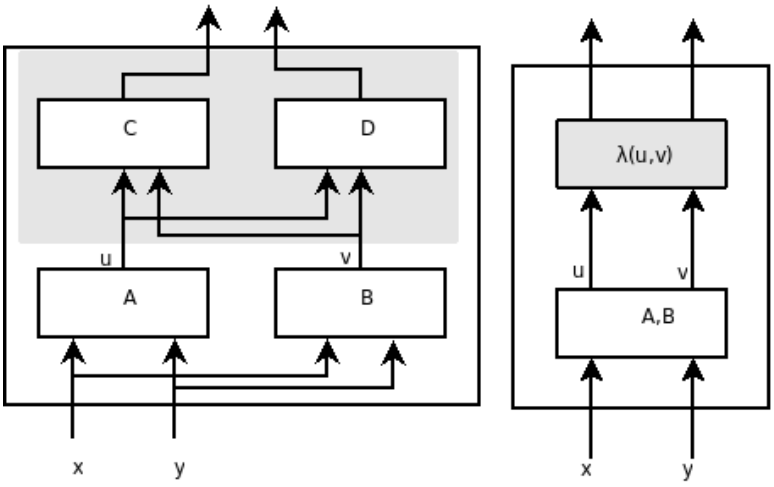
So that the substructure of a circuit can be easily retrieved for visualization, we represent it as an adjacency matrix. Since the components of a circuit

may have multiple inputs and outputs, a digraph of functional dependencies between components is insufficient, since it would not specify which output of a component fed into which input of the next. We thus map outputs onto inputs directly in our matrix. For example, the circuit shown in Figure 4 is represented as follows:

	O_1	O_2	A_1	A_2	B_1	B_2	C_1	C_2	D_1	D_2
x	0	0	1	0	1	0	0	0	0	0
y	0	0	0	1	0	1	0	0	0	0
A_O	0	0	0	0	0	0	1	0	1	0
B_O	0	0	0	0	0	0	0	1	0	1
C_O	1	0	0	0	0	0	0	0	0	0
D_O	0	1	0	0	0	0	0	0	0	0

where $\{x, y\}$ are the inputs to the circuit and $\{O_1, O_2\}$ are its outputs. The labels are omitted in memory: a corresponding *component vector* is defined from which they can be generated.

Figure 5:



Note that each component input can receive a signal from only one source, and that the corresponding functional dependency graph (in which each node is a component and edges denote *any* dependency) will always be acyclic in the circuits we are generating. We accomplish this as follows: A random ordering of components is selected, and both the rows and columns follow

this order. We begin with the zero matrix. For each column vector of the matrix corresponding to an input A_i of component A, a random element between the first row and the first output A_O of A is set to one. The columns corresponding to the circuit outputs O_i have any element set to one. This ensures that the corresponding functional dependency graph has an upper triangular matrix with a zero diagonal, i.e. that it is acyclic. Since the ordering of the components was selected randomly, the resulting circuit is also random.

Executing circuits is expensive, especially since each component may itself be a circuit with many subcomponents, and so on. It is thus to our benefit to store the truth table of a circuit, so that its output can be used more than once without executing the function repeatedly. We generate a graph representation of the circuit's truth table known as a Binary Decision Diagram (BDD), which represents the if-then-else normal form of a boolean function in a binary tree structure that can be easily walked for all input bit string permutations[1]. The BDD is compressed into a memory efficient Reduced Ordered Binary Decision Diagram (ROBDD), much like a prefix tree for a dictionary with an n-ary (as opposed to binary) alphabet can be compressed into a Directed Acyclic Word Graph[5]. ROBDDs have the added quality that two logic functions that are isomorphic (perform the same computation on a set of input variables) have ROBDDs that are not only isomorphic but identical. The ROBDD for functions with more than around sixteen variables can still become quite large, as the number of possible bit string permutations is 2^n. It is computationally intractable to generate the entire truth table of functions much larger than this.

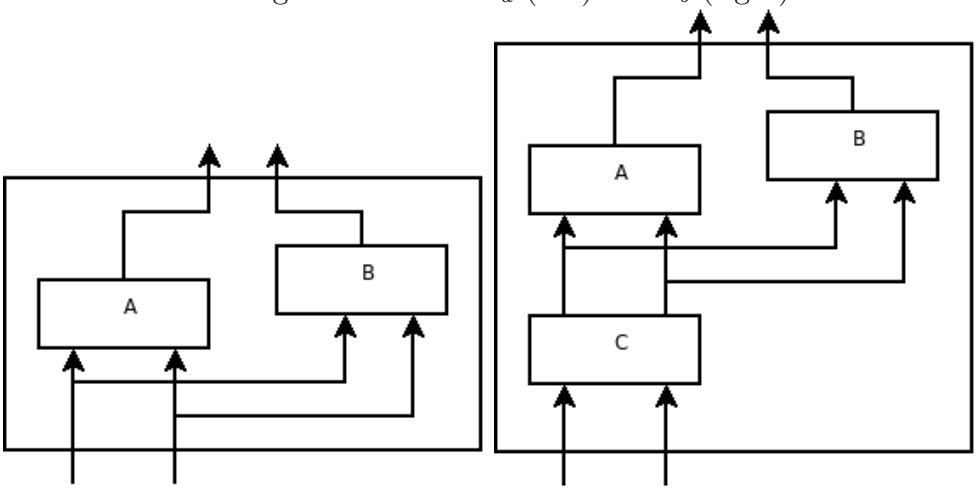
ROBDDs can be composed efficiently, and Arthur and Polak use such operators to generate new circuits. Our system composed functions via the generation of adjacency matrices and Lisp functions as described above.

5.2 Execution Tiers

To generate the ROBDDs for circuits, we need code that executes the function. To achieve this we automatically generate a Lisp function from the circuit's adjacency matrix.

```
(defun one_a(x y)
  (list (funcall A x y) (funcall B x y)))
```

Figure 6: Circuit 1_a (left) and 1_b (right).



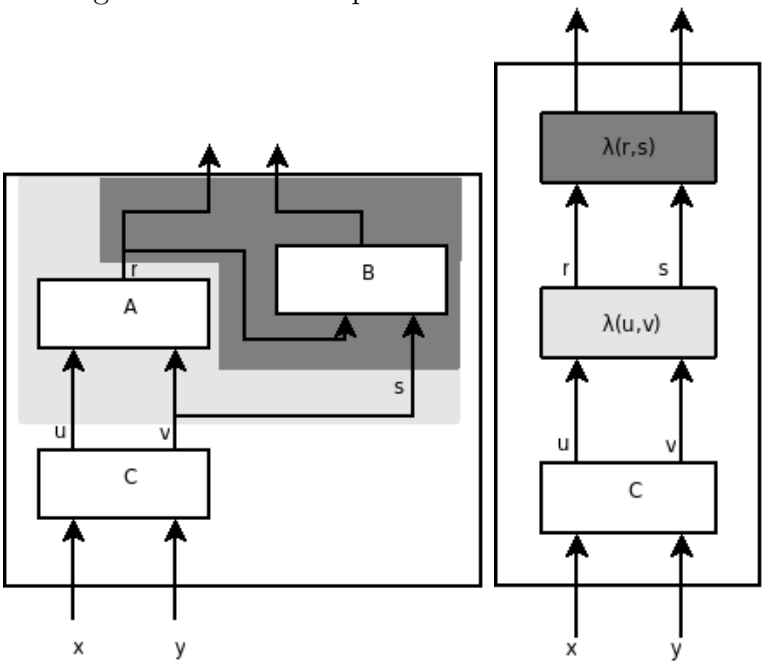

```
(defun one_b(x y)
  (apply
    #'(lambda (u v)
      (list (funcall B u v) (funcall C u v)))
    (funcall D x y)))
```

Note that we used a lambda expression in circuit 1_b so that the outputs of D could be used multiple times. More complex functions may require several nested lambda expressions, each of which must be executed in sequence, its outputs providing the inputs for the next function. We conceptualize each lambda as an *execution tier*, and consider each tier τ to be a set of component functions. Determining a circuit's execution tiers is the first step toward generating its Lisp expression, and is equivalent to performing a topological sort on the components (though, again, the circuits resist a straightforward graph representation).

Formally, we may define a tier as follows. Let $\mathbb{C}$ be the set of component functions used in the circuit, ex. $\mathbb{C} = \{A, B, C, D\}$, each of which in turn is a set of input variables X_i. Let ϕ_{X_i} be the column vector of the adjacency matrix corresponding to X_i, with $\phi_{X_i}(z)$ being the matrix element corresponding to output z (ex. In Figure 5, $\phi_{A_1}(x) = 1$, but $\phi_{A_1}(y) = 0$).

Definition 1 *A component function X is initially in tier τ_0^1 if and only if*

Figure 7: The decomposition of a circuit into three execution tiers.



X's outputs depend only on the circuit's inputs $x_i \in I_1$. That is, all the entries in each X_i's column vector are zero except those corresponding to inputs to the circuit, i.e.:

$$\tau_0^1 \equiv \{X \in \mathbb{C} | \forall X_j \in X, (\forall (z \in \phi_{X_j} | z \notin I_1), \neg \phi_{X_j}(z))\}. \quad (1)$$

Now, visualizing higher tiers as subcircuits with the outputs of the previous tier as inputs, if we define the outputs of the functions in τ_0^1 unioned with I_1 as I_2 and so on, we can define the ith tier:

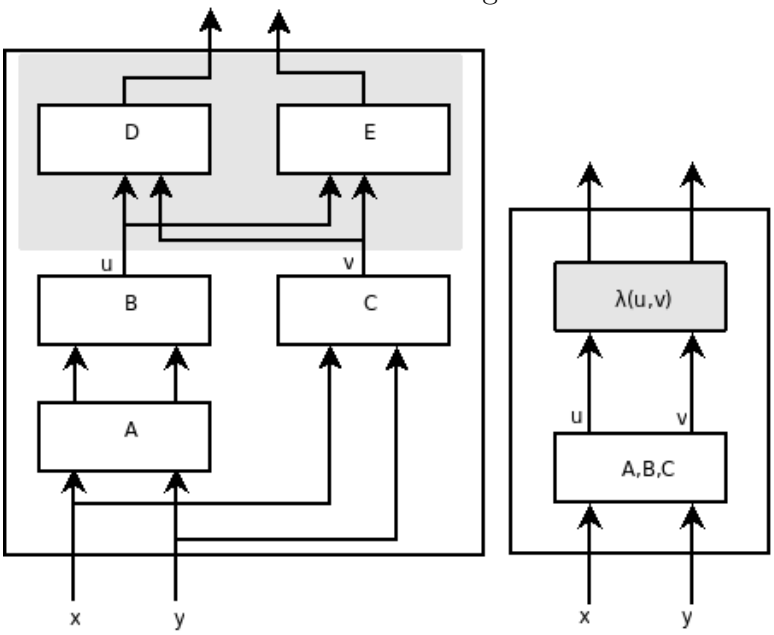
$$I_{i+1} \equiv I_i \cup \{Y_O | (Obj(Y_O) \in \tau^i) \wedge (\exists X_j \in X \notin \tau^i, (\phi_{X_j}(Y_O)))\} \quad (2)$$

$$\tau_0^i \equiv \{X \in \mathbb{C} | \forall X_j \in X, (\forall (z \in \phi_{X_j} | z \notin I_i), \neg \phi_{X_j}(z))\}. \quad (3)$$

The subscript 0 signifies that this is not our final definition.

Equation (3) works for the circuit in Figure 5, but it is not general. A special case must be compensated for, as demonstrated by the circuit in Figure 8. Component B in Figure 8 is a part of the first tier, but depends on the output of A, contrary to our definition in (1). Since the output of

Figure 8:



A is used only once, we do not need a lambda function (tier) to process it. To complete the definition of τ then, we define an iterative algorithm to add components akin to B. First we define three helper functions:

Definition 2 *Object and Usage functions:*

- *A component X is in the set $Obj(X_O)$ iff X_O is an output for X.*
- *$Usage(X_O)$ is the number of edges directed out of X_O, i.e.*

$$Usage(X_O) \equiv \sum_i \phi_i(X_O)$$

- *$ObUsage(X)$ is the number of distinct components the outputs of component X are used in.*

If a function's output is used more than once, we need a need a new tier to store the value for processing multiple times. Furthermore, if a component's outputs are directed into different child components, complex car and cdr arrangments can appear in the expression which are difficult to automatically

generate. Thus we also require a new tier if a component's $ObUsage$ is greater than one:

Definition 3 *Y is appended to τ^i iff for all signals X_{Oj} directed into Y:*

- $(Obj(X_{Oj}) \in \tau) \vee (X_{Oj} \in I_i)$
- *X_{Oj} is used only once, i.e. $Usage(X_i) = 1$.*
- *X has only one child components, i.e. $ObUsage(X) = 1$.*

Combining these yields the following recurrence relation, to be executed until no change in τ^i occurs:

$$\tau^i = \tau^i \cup \{Y \in \mathbb{C} | \forall Y_i \in Y, \forall (X_{Oj} | \phi_{Y_i}(X_{Oj})), \\ [(Obj(X_{Oj}) \in \tau^i) \vee (X_{Oj} \in I_i)] \wedge [Usage(X_{Oj}) = 1] \wedge [Usage(Obj(X_{Oj})) = 1]\} \tag{4}$$

References

[1] Henrik Reif Andersen. An introduction to binary decision diagrams. October 1997.

[2] W. Brian Arthur and Wolfgang Polak. The evolution of technology within a simple computer model. *Complexity*, 11(5):23–32, May/June 2006.

[3] Yaneer Bar-Yam. When systems engineering fails – toward complex systems engineering. In *International Conference on Systems, Man and Cybernetics*, volume 2, pages 2021–2028, 2003.

[4] Jürgen Branke. *Evolutionary Optimization in Dynamic Environments*. Kluwer, Norwell, MA, 2001.

[5] Maxime Crochemore and Renaud Vérin. Direct construction of compact directed acyclic word graphs. In *Lecture Notes in Computer Science*, volume 1264, pages 116–129, 1997.

[6] Andries P. Engelbrecht. *Computational Intelligence: An Introduction*. Wiley & Sons, Ltd, England, 2007.

[7] Cristoph Flamm, Alexander Ullrich, Heinz Ekker, Hartin Mann, and Daniel Högerl et al. Evolution of metabolic networks: A computational framework. Febuary 2010.

[8] J. Gerhart and M. Kirschner. The theory of facilitated variation. *PNAS*, 104, May 2007.

[9] Chi-Keong Goh and Kay Chen Tan. A competitive-cooperative coevolutionary paradigm for dynamic multiobjective optimization. *IEEE Transaction on Evolutionary Computation*, 13(1):103–127, February 2009.

[10] Stephen Jay Gould and Elisabeth S. Vrba. Exaptation–a missing term in the science of form. *Paleobiology*, 8(1):4–15, 1982.

[11] Stuart A. Kauffman. *The Origins of Order*. Oxford University Press, New York, NY, 1993.

[12] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, 1992.

[13] Richard E. Lenski, Charles Ofria, Robert T. Pennock, and Christoph Adami. The evolutionary origin of complex features. *Nature*, 423:139–144, May 2003.

[14] Javier Macia and Richard Solé. Distributed robustness in cellular networks: insights from synthetic evolved circuits. *J. R. Soc. Interface*, 6:393–400, September 2008.

[15] Merav Parter, Nadav Kashtan, and Uri Alon. Facilitated variation: How evolution learns from past environments to generalize to new environments. *PLoS Computational Biology*, 4(11), November 2008.

[16] W. Rand and U. Wilensky. Netlogo artificial neural net model.

[17] Thomas Ray. Evolution, ecology, and optimization of digital organisms. August 1992.

[18] David A. Van Veldhuizen and Gary B. Lamont. Multiobjective evolutionary algorithms: Analyzing the state-of-the-art. *Evolutionary Computation*, 8(2):125–147, 2000.