OMNIREP: Originating Meaning by Coevolving Encodings and Representations

Moshe Sipper and Jason H. Moore*

March 31, 2019

Abstract

A major effort in the practice of evolutionary computation (EC) goes into deciding how to represent individuals in the evolving population. This task is actually composed of two subtasks: defining a data structure that is the representation and defining the encoding that enables to interpret the representation. In this paper we employ a coevolutionary algorithm—dubbed *OMNIREP*—to discover *both* a representation and an encoding that solve a particular problem of interest. We describe four experiments that provide a *proof-of-concept* of OMNIREP's essential merit. We think that the proposed methodology holds potential as a problem solver and also as an exploratory medium when scouting for good representations.

Keywords: evolutionary algorithms; cooperative coevolution; interpretation

1 Representations, Encodings, and Coevolution

One of the basic tasks, indeed a sine qua non, of the evolutionary computation (EC) practitioner is to decide how to represent individuals in the (evolving) population, i.e., precisely specify the genetic makeup of the artificial entity under consideration. In his seminal 1989 book, [9] noted that, "Genetic algorithms require the natural parameter set of the optimization problem to be coded as a finite-length string over some finite alphabet." Three decades later the field of EC has seen problems other than optimization and representations other than strings, yet the essential necessity of specifying a *representation* and an *encoding* remains firm. As stated by [18], "the way in which candidate solutions are encoded is a central, if not *the* central, factor in the success of a genetic

*The authors are with the Institute for Biomedical Informatics (IBI), Perelman School of Medicine, University of Pennsylvania, Philadelphia, PA 19104. M. Sipper is also with the Department of Computer Science, Ben-Gurion University, Beer-Sheva 8410501, Israel. Emails: {sipper,jhmoore}@upenn.edu. This is a post-peer-review, pre-copyedit version of an article published in *Memetic Computing*. The final authenticated version is available on the journal's website: https://link.springer.com/journal/12293.

algorithm." And, more recently, [7] pointed out that, "Technically, a given representation might be preferable over others if it matches the given problem better, that is, it makes the encoding of candidate solutions easier or more natural."

The EC practitioner's foremost task is thus to identify a representation—a data structure—and its encoding, or *interpretation*. These can be viewed, in fact, as two distinct tasks, though they are usually dealt with simultaneously. To wit, one might define the representation as a bitstring and in the same breath go on to state the encoding, e.g., "the 120-bit bitstring represents 4 numerical values, each encoded by 30 bits, which are treated as signed floating-point values" (we will revisit this encoding in Section 2.2). As another example of a representation and its encoding consider the following: "A floating point number has 64 bits that encode a number of the form $\pm p \times 2^e$. The first bit encodes the sign, 0 for positive numbers and 1 for negative numbers. The next 11 bits encode the exponent e, and the last 52 bits encode the precision p"[1] (we experiment with a floating-point encoding in Section 2.3.)

In this paper we consider the two tasks—discovering a representation and discovering an encoding—as distinct yet tightly coupled: *A representation is meaningless without an encoding; an encoding is useless without a representation.* Our basic idea herein is to employ a *coevolutionary* algorithm to discover *both* a representation and an encoding that solve a particular problem of interest.

Coevolution refers to the simultaneous evolution of two or more species with coupled fitness [23]. Coevolving species can either compete (e.g., to obtain exclusivity on a limited resource) or cooperate (e.g., to gain access to some hard-to-attain resource). In a competitive coevolutionary algorithm the fitness of an individual is based on direct competition with individuals of other species, which in turn evolve separately in their own populations. Increased fitness of one of the species implies a reduction in the fitness of the other species.

A cooperative coevolutionary algorithm involves a number of independently evolving species, which come together to obtain problem solutions. The fitness of an individual depends on its ability to collaborate with individuals from other species [23].Interestingly, though the idea of coevolution originates (at least) with Darwin—who spoke of "coadaptations of organic beings to each other" in *Origin of Species*—it is arguably somewhat less pervasive in the field of EC than one might expect.

Our idea can be stated simply: Rather than specify a specific representation along with a specific encoding in advance, we shall set up a cooperative coevolutionary algorithm to coevolve the two, with a population of representations coevolving alongside a population of encodings.

We dubbed our algorithmic framework *OMNIREP*. Consistent with the focal point of this paper, the "encoding" of OMNIREP is somewhat fluid, referring to 'omni'—universal, and 'rep'—representation; and also denoting an acronym: **o**riginating **m**ea**n**ing by coevolving encod**i**ngs and **rep**resentations.

Of importance to note is OMNIREP's not being a specific algorithm but

[1] www.johndcook.com/blog/2009/04/06/anatomy-of-a-floating-point-number

rather an algorithmic framework, which can hopefully be of use in settings other than those exemplified herein. We believe that the OMNIREP methodology can aid researchers not only in solving *specific* problems but also as an *exploratory* tool when one is seeking out a good representation.

In a literature review of the field of memetic computing (MC), Neri and Cotta [19] stated that, "an MC approach is a linked collection of operators without any prefixed structure but with the only aim of solving the problem" (see also [20]). OMNIREP fits perfectly within this definition, given our desire to evolve a framework with less prefixed structure regarding the specifics of the encodings and representations.

An objection that might be raised is that the distinction between "representation" and "encoding" is a malleable one. Indeed, they can be two sets of loci in a single individual, which together map to a single element in the search space. We counter-argue by noting that in computer science one often makes the distinction between "inert" data and "active" programs. Moreover, the real value of OMNIREP comes from the higher-order epistatic interactions produced by representing the points in the search space as a "representation" and "encoding". The coevolutionary dynamics thus engendered create new ways of moving through the search space that could potentially increase or decrease evolvability.

This paper describes four basic examples of the OMNIREP idea, the intent being to provide a *proof-of-concept* of its essential merit. The setup and the results are described in the next section, after which—in light of our experiments—we discuss related research in Section 3. We surmise that some of the latter might benefit from OMNIREP thinking. We offer concluding remarks in Section 4.

Though the experiments described herein might be regarded as somewhat simplistic, we believe the significance of the results presented lies beyond their preliminary nature, in that we show how an essential part of an evolutionary computation, and in particular a genetic programming practitioner's job—finding a good representation—might be tackled through automated means. We therefore wish to share this idea with the community at large, hoping to see it used to tackle other problems.

2 OMNIREP: Setup, Experiments, and Results

2.1 Cooperative coevolution

We begin by describing the basic setup that is common to all experiments.[2] OMNIREP uses cooperative coevolution with two coevolving populations, one of representations, the other of encodings. The evolution of each population is identical to a single-population evolutionary algorithm—except where fitness is concerned (Figure 1). Below we describe the various components and parame-

[2]The OMNIREP code is available at https://github.com/EpistasisLab/.

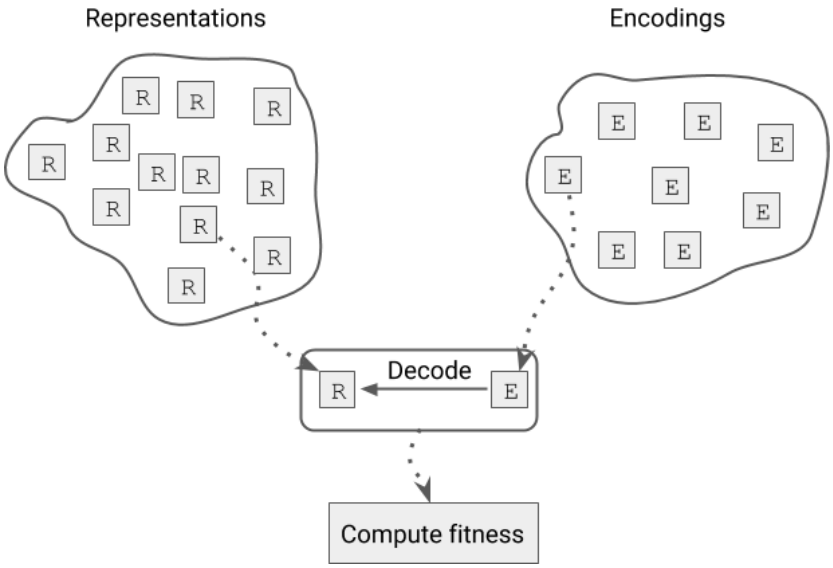

Figure 1: Fitness computation in OMNIREP, where two populations coevolve, one comprising representations, the other encodings. Fitness is computed by combining a representation individual (R) with an encoding individual (E).

ters of the system (see Table 1 for a summary[3]).

Selection. Tournament selection with tournament size 4, i.e., choose 4 individuals at random from the population and return the individual with the best fitness as the selected one.

Crossover. Single-point crossover, i.e., select a random crossover point and swap two parent genomes beyond this point to create two offspring.

Mutation is problem-specific and is described below per experiment.

Fitness. To compute fitness the two coevolving populations cooperate. Specifically, to compute the fitness of a single individual in one population, we use *representatives* from the other population [23]. The representatives (also called cooperators) are selected via a greedy strategy as the 4 fittest individuals from the previous generation. When evaluating the fitness of a particular representation individual, we combine it 4 times with the top 4 encoding individuals, compute 4 fitness values, and use the average fitness over these 4 evaluations as the final fitness value of the representation individual. In a similar manner we use the average of 4 representatives from the representations population when computing the fitness of an encoding individual. (Other possibilities include using the best fitness of the 4, the worst fitness, and selecting a different mix of representatives, e.g., best, median, and worst.) The specific manner by which a single fitness value is computed per representation-encoding pair is described below for each experiment.

[3]Some parameters may seem arbitrary but our recent findings provide some justification for this [25].

Table 1: Evolutionary parameters. Shown first are common parameters, followed by experiment-specific ones.

Description	Value
Common	
Number of evolutionary runs	1000
Maximal number of generations	1000
Stop if fitness $<$	0.001
Size of representations population	100
Size of encodings population	50
Type of selection	Tournament
Tournament size	4
Type of crossover	single-point
Probability of mutation (representations)	0.3
Probability of mutation (encodings)	0.3
Evolve encodings population every	3 generations
Number of representatives used for fitness	4
Number of top individuals copied (elitism)	2
Size of training and test sets	200
Experiment 1 (bitstring and bit count)	
Size of representation individual	120 (bits)
Size of encoding individual	4 (integers)
Minimal number of bits per coefficient	10
Maximal number of bits per coefficient	30
Experiment 2 (floating point and precision)	
Number of evolutionary runs	100
Size of representation individual	50 (floats)
Size of encoding individual	50 (integers)
Minimal precision	1 digit
Maximal precision	8 digits
Experiment 3 (program and instructions)	
Size of representation individual (program)	10 (integers)
Size of encoding individual	5 (integers)
Experiment 4 (image and blocks)	
Number of evolutionary runs per image	10
Number of images	9
Maximal number of generations	20000
Size of representations population	20
Size of encodings population	10
Size of representation individual	5000 (integers)
Size of encoding individual	5000 (tuples)
Minimal block size	1 pixel
Maximal block size	10 pixels

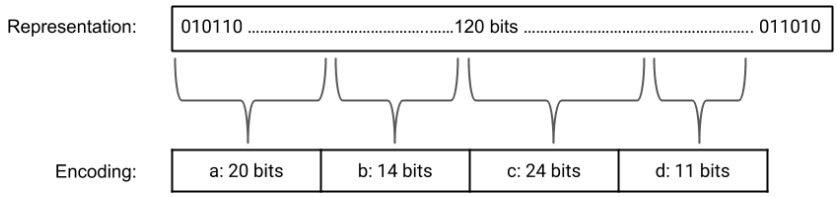

Figure 2: Experiment 1: Sample representation and encoding individuals.

Elitism. The 2 individuals with the highest fitness in a generation are copied ("cloned") into the next generation unchanged.

Evolutionary rates differ between the two populations, with the encoding population evolving more slowly, specifically, every 3 generations.

2.2 Experiment 1: Bitstring and bit count

Problem. Cubic polynomial regression:

$$y = ax^3 + bx^2 + cx + d\,,$$

where $a, b, c, d, x \in \mathbb{R} \cap [0, 1]$. Given 200 pairs of (x_i, y_i) values, $i = 1, \ldots, 200$, where x_i is the independent variable and y_i is the dependent variable, find the coefficients, a, b, c, d.

Populations. Our first experiment involves the simple yet quintessentially "classic" genetic-algorithm (GA) bitstring setup, but with the added twist of evolving the encoding. An individual in the representations population is a bitstring of length 120. An individual in the encodings population is a list of 4 integer values, or genes, each of which specifies the number of bits allocated (left to right) to the respective parameter (a, b, c, or d) in the representation individual (Figure 2). We set the minimum bit count per coefficient to 10 and the maximum to 30. Note that the 4 genes do not have to add up to 120.

Initialization. For every coevolutionary run: both populations are initialized randomly (random bits or random values in the appropriate range, respectively); target coefficients $a, b, c, d \in [0, 1]$ are chosen at random; a table of 200 (x_i, y_i) training pairs is generated using the target coefficients, with each $x_i \in [0, 1]$ randomly chosen (these are used for fitness evaluation during evolution); a table of 200 (x_i, y_i) test pairs is generated using the target coefficients, with each $x_i \in [0, 1]$ randomly chosen (these are used for post-evolution evaluation of the best solution).

Parameters. The full list of evolutionary parameters is given in Table 1.

Mutation. Mutation in a representation individual is bitwise, namely, flip each bit with probability 0.3. Mutation in an encoding individual is done with probability 0.3 (per individual) by selecting a random gene (of the 4) and replacing it with a new random (integer) value in the appropriate range.

Fitness. As noted above, a representation individual and an encoding individual are combined for fitness purposes. The 4 integer values of the encoding

individual define the number of bits that make up the respective coefficients in the representation individual. Each coefficient is a simple signed float, where the first bit represents the sign (0 for positive numbers and 1 for negative numbers), and the rest represent the fraction of the maximal value. For example, 0110, represents the value $+6/7$ and 1110 represents the value $-6/7$.

The combination of an encoding individual and a representation individual thus yields 4 real-valued coefficients, which can thereupon be used in conjunction with the data table to compute 200 output values. A single fitness value then equals the mean absolute error with respect to the known target values. As explained above in Section 2.1, the final fitness is computed as the average over 4 representation-encoding pairs.

Results. We performed 1000 evolutionary runs, each with a maximum of 1000 generations. A run terminated if a fitness threshold of 0.001 was attained (the perfect fitness score is 0).

The results are shown in Table 2 (note: this table summarizes all results in the paper, including the experiments described below). In addition to attaining good fitness and test scores, we note that, interestingly, OMNIREP chose similar bit counts for a, b, and c, while d received fewer bits. The total time for a batch of runs in this and the following experiments (i.e., one line of Table 2) was between 1-3 days on a node in our cluster (Intel® Xeon® E5-2650L).

Using the the same data as in the original experiment, we performed a comparative, fixed-encoding experiment, where no encoding evolution took place, only single-population evolution of the 120-bit bitstring, where each coefficient was allotted 30 bits. Evolution took slightly less time and fitness was similar. The OMNIREP version was far more compact, using significantly less bits.

2.3 Experiment 2: Floating point and precision

Problem. Regression:

$$y = \sum_{j=0}^{49} a_j x^{e_j} \ ,$$

where $a_j, x \in \mathbb{R} \cap [0,1]$, $e_j \in \{0,\ldots,4\}$, $j = 0,\ldots,49$. Given 200 pairs of (x_i, y_i) values, $i = 1,\ldots,200$, where x_i is the independent variable and y_i is the dependent variable, find the 50 coefficients, $\{a_0,\ldots,a_{49}\}$.

Populations. Driven by the example given in Section 1 involving floating-point precision, an individual in the representations population is a list of 50 real values $\in [0,1]$ (the coefficients a_i). An individual in the encodings population is a list of 50 integer values, each specifying the precision of the respective coefficient, namely, the number of digits $d \in \{1,\ldots,8\}$ after the decimal point.

Initialization. For every coevolutionary run: both populations are initialized randomly (random real values or random integer values in the appropriate range, respectively); target $\{a_i\}_{i=0}^{49} \in [0,1]$, $\{e_i\}_{i=0}^{49} \in \{0,\ldots,4\}$ are chosen at random; a table of 200 (x_i, y_i) training pairs is generated using the target coefficients and exponents, with each $x_i \in [0,1]$ randomly chosen; a table of 200

Table 2: Results. Values reported below are medians over all runs in the particular experiment (the choice of median rather than average is explained in Section 2.4). Shown: generation (gen) at which best-of-run fitness was attained, fitness, test score, and additional statistics where applicable (fitness and test scores pertain to the solution obtained by combining a representation individual with an encoding individual.)

(1a) Experiment 1 with OMNIREP: coevolution of bitstrings and bit counts. Shown also: bit counts per coefficients a, b, c, d.

(1b) Experiment 1 with a fixed encoding: no encoding evolution, each coefficient allotted a fixed 30 bits.

(2a) Experiment 2 with OMNIREP: coevolution of floating-point values and their precision (number of digits after decimal point). Shown also: median of medians, i.e., the overall median precision of each run's median evolved precision.

(2b) Experiment 2 with a fixed encoding: no encoding evolution, each coefficient assigned the maximal 8-digit precision.

(3a) Experiment 3 with OMNIREP: coevolution of programs and instructions.

(3b) Experiment 3 with a fixed random encoding.

(3c) Experiment 3 with a fixed encoding, part random, part taken from the target encoding.

(4) Experiment 4 with OMNIREP: coevolution of image blocks and sizes plus colors.

experiment		gen	fitness	test	additional statistics
(1a)	OMNIREP	606	0.005	0.005	a, b, c, d: 20, 21, 20, 11
(1b)	fixed	528	0.006	0.006	
(2a)	OMNIREP	996	0.011	0.011	precision: 3
(2b)	fixed	979	0.01	0.01	
(3a)	OMNIREP	15	0.028	0.026	
(3b)	fixed	7	0.438	0.44	
(3c)	fixed	7	0.3	0.308	
(4)	OMNIREP	19995	0.48	—	

(x_i, y_i) test pairs is generated using the target coefficients and exponents, with each $x_i \in [0, 1]$ randomly chosen.

Parameters. The full list of evolutionary parameters is given in Table 1.

Mutation. Mutation is done with probability 0.3 (per individual) by selecting a random gene (of the 50) and replacing it with a new random value (real-valued or integer, respectively) in the appropriate range ($[0, 1]$ for representation individuals, $\{1, \ldots, 8\}$ for encoding individuals).

Fitness. A representation individual and an encoding individual are combined for fitness purposes. The 50 integer values of the encoding individual define the number of digits after the decimal point of the respective coefficient in the representation individual. The combination of an encoding individual and a representation individual thus yields 50 real-valued coefficients, which can thereupon be used in conjunction with the data table to compute 200 output values. A single fitness value then equals the mean absolute error with respect to the known target values (4 such fitness values are averaged, see Section 2.1).

Results. We performed 100 evolutionary runs, each with a maximum of 1000 generations. A run terminated if a fitness threshold of 0.001 was attained. The results are shown in Table 2. Good fitness and test scores were attained, interestingly without evolving to use the maximal precision but less than half that.

Using the the same data as in the original experiment, we performed a comparative, fixed-encoding experiment, where each parameter was given the maximal precision of 8 digits (a plausible choice that would be made by an EC practitioner). Results were similar to OMNIREP.

2.4 Experiment 3: Program and instructions

Problem. Find a program that is able to emulate the output of an unknown target program.

We consider the evolution of a program composed of 10 lines, each line executing a mathematical, real-valued, univariate function, or instruction. There are 28 possible instructions, listed in Table 3.

Populations. The representation individual is a program comprising 10 lines, each one executing a *generic* instruction of the form `x=fi(x)`, where `fi` $\in$ {`f1,...,f5`}. The program has one variable, `x`, which is set to a specific value `v` at the outset, i.e., to each (10-line) program, the instruction `x=v` is added as the first line. `v` is thus the program's input. After a program finishes execution, its output is taken as the value of `x`.

To run a program one needs to couple it with an encoding individual, which provides the specifics of what each `fi` performs, i.e., which of the 28 functions it represents. The encoding individual is a list of 5 integer values, each in the range $\{1, \ldots, 28\}$ (Figure 3). Note that we do not disallow the encoding individual to have duplicate instructions, as this was deemed more general, allowing evolution to come up with solutions that use less than 5 instruction types.

Initialization. For every coevolutionary run: both populations are initialized to random values in the appropriate range; a random target representation

Table 3: Mathematical instructions that make up the programs of experiment 3.

Instruction	Returns	Instruction	Returns
`plus1(x)`	x+1	`mul10(x)`	x*10
`plus2(x)`	x+2	`div2(x)`	x/2
`plus3(x)`	x+3	`div3(x)`	x/3
`plus4(x)`	x+4	`div4(x)`	x/4
`plus5(x)`	x+5	`div5(x)`	x/5
`minus1(x)`	x-1	`div10(x)`	x/10
`minus2(x)`	x-2	`sin(x)`	sin(x)
`minus3(x)`	x-3	`cos(x)`	cos(x)
`minus4(x)`	x-4	`tan(x)`	tan(x)
`minus5(x)`	x-5	`floor(x)`	floor of x
`mul2(x)`	x*2	`ceil(x)`	ceiling of x
`mul3(x)`	x*3	`degrees(x)`	x converted from radians to degrees
`mul4(x)`	x*4	`radians(x)`	x converted from degrees to radians
`mul5(x)`	x*5	`fabs(x)`	absolute value of x

Representation	Encoding
`x=v`	`f1: mul10`
`x=f1(x)`	`f2: fabs`
`x=f2(x)`	`f3: tan`
`x=f3(x)`	`f4: mul10`
`x=f4(x)`	`f5: minus2`
`x=f2(x)`	
`x=f2(x)`	
`x=f5(x)`	
`x=f2(x)`	
`x=f1(x)`	
`x=f5(x)`	

Figure 3: Experiment 3 (programs): Sample representation and encoding individuals, the former being a 10-line program with generic instructions, and the latter being the instruction meanings, i.e., indexes into Table 3.

and random target encoding are generated; a table of 200 (x_i, y_i) training pairs is generated using the target representation and encoding, with each $x_i \in [0, 1]$ randomly chosen; a table of 200 (x_i, y_i) test pairs is generated using the target representation and encoding, with each $x_i \in [0, 1]$ randomly chosen.

Parameters. The evolutionary parameters are given in Table 1.

Mutation. Mutation is done with probability 0.3 (per individual) by selecting a random gene (of the 10 or 5, respectively) and replacing it with a new random (integer) value in the appropriate range ($\{1, \ldots, 5\}$ for representation individuals, $\{1, \ldots, 28\}$ for encoding individuals).

Fitness. A representation individual and an encoding individual are combined for fitness purposes. The program represented by the former, decoded using the latter, is run on each of the 200 x_i values from the data table, generating 200 output values. A single fitness value then equals the mean absolute error with respect to the known target values (4 such fitness values are averaged, see Section 2.1).

Results. We performed 1000 evolutionary runs, each with a maximum of 1000 generations. A run terminated if a fitness threshold of 0.001 was attained. As shown in Table 2 the results were good.

Using the the same data as in the original experiment, we performed two comparative, fixed-encoding experiments, where no encoding evolution took place, only single-population evolution of the programs: (1) a random encoding was generated per each of the 1000 runs, emulating a state of "no knowledge" about the program encoding; and (2) for each run, the encoding used 2 of the functions from the known target encoding and 3 random functions, emulating a state of "partial knowledge" about the program encoding. Results were worse than OMNIREP in both cases.

There is an interesting reason behind our choice of reporting on median rather than average values in Table 2. Because of the unconstrained nature of the programs some runs ended up with huge (i.e., bad) fitness values (e.g., when sequences of multiplications arose). We did not deem it necessary to report averages by removing outliers (e.g., by using rules such as $3 \times$ standard deviation or $1.5 \times$ interquartile range), because our interest lay in overall algorithmic performance. A good median value means that at least 1 in 2 evolutionary runs produces satisfactory results. In many cases, EC practitioners would be content with 1-in-10 runs being good. Likely, through judicious constraints to the individuals, and fixes to selection, crossover, and mutation, we could avoid programs that produce large values, but we did not feel this was cardinal in our current study.

2.5 Experiment 4: Image and blocks

Problem. Evolutionary Art is a branch of EC wherein artwork is generated through an evolutionary algorithm; it is a growing domain, which has boasted a specialized conference over the past few years [6]. Our goal herein was to evolve artistic renderings of given images. We were inspired by the work of Johansson, who used what is essentially a $1 + 1$ evolution strategy—single parent, single

child, both competing against each other—to evolve a replica of the Mona Lisa using 50 semi-transparent polygons.[4]

Populations. A two-dimensional image of dimensions $\{width, height\}$ is treated as a one-dimensional list of pixels of size $width \times height$ (ranging, in our case, from 10848 pixels to 68816). The representation individual's genome is a list of pixel indexes, $p_i \in \{0, \ldots, width * height - 1\}$, $i = 1, \ldots, 5000$, where p_i is the start of a same-color block of pixels. The encoding individual is a list equal in length to the representation individual, consisting of tuples (b_i, c_i), where b_i is block i's length, and c_i is block i's color. If a pixel is uncolored by any block it is assigned a default base color.

An encoding individual combines with a representation individual to paint a picture, made up of same-color blocks of length and color indicated by the former and start positions indicated by the latter.

Initialization. For every coevolutionary run: both populations are initialized to random values in the appropriate range: a single representation-individual value (gene) is an integer in the range $\{0, \ldots, width * height - 1\}$, and a single encoding-individual gene is a tuple $(block\_size, block\_color)$, where $block\_size \in \{1, \ldots, 10\}$ and $block\_color \in \{0, \ldots, 3\}$; a target painting is chosen and its list of pixels generated.

Parameters. The evolutionary parameters are given in Table 1.

Mutation. Mutation is done with probability 0.3 (per individual) by selecting a random gene and replacing it with a new random value of the appropiate type.

Fitness. A representation individual and an encoding individual are combined for fitness purposes to produce a list of pixels that define an image. A single fitness value then equals the mean absolute error with respect to the known target pixels (4 such fitness values are averaged, see Section 2.1).

Results. We performed 90 evolutionary runs, each with a maximum of 20000 generations. A run terminated if a fitness threshold of 0.001 was attained. Table 2 shows that we achieved good fitness and Figure 4 presents a gallery of evolved pictures.

3 Related Research

Having (hopefully) demonstrated a basic proof-of-concept of the OMNIREP idea, we now present a brief discussion of related work. This is by no means an in-depth review but only meant to note other works that bear mentioning. While none of them employ coevolution of representations and encodings, perhaps they might benefit from an OMNIREP approach.

Generative and Developmental Encoding is a branch of EC concerned with genetic encodings motivated by biology. A structure that repeats multiple times can be represented by a single set of genes that is reused in a genotype-to-phenotype mapping [27].

[4] www.rogerjohansson.blog/2008/12/07/genetic-programming-evolution-of-mona-lisa/

Figure 4: Experiment 4: Sample results. Each pair of images shows an original (left) and evolved (right) picture.

[12] argued in favor of using developmental mechanisms in genetic algorithms, providing a framework that distinguishes between two developmental mechanisms—learning and maturation. They observed that in some contexts, maturation and local search can be incorporated into the fitness evaluation, but illustrated reasons for considering them separately.

Early work by [11] compared the efficiency of two encoding schemes for Artificial Neural Networks (ANNs) optimized by evolutionary algorithms. Direct encoding encoded the weights for an a-priori fixed neural network architecture while cellular encoding encoded both weights and the architecture of the neural network. The authors noted that, "The advantage of cellular encoding is that it could automatically find small architectures whose structure and complexity fit the specificity of the problem." Indeed, a similar argument could be made for OMNIREP with respect to finding good encodings and representations.

[4] explored the use of growth processes, or embryogenies, to map genotypes to phenotypes within evolutionary systems, identifying three main types of embryogenies in EC: external (non-evolved), explicit (evolved), and implicit (evolved, indirect). For a problem of tessellating tiles they showed that implicit embryogeny outperformed the other methods.

[16] presented developmental genetic programming, wherein a fully developed electrical circuit is produced by progressively executing circuit-constructing functions from an individual program tree. Thus, the trees did not directly represent a problem solution—in this case an electrical circuit—but rather instructions on how to *grow* a full-blown circuit from a simple embryo.

In Gene Expression Programming the individuals in the population are encoded as linear strings of fixed length, which are afterwards expressed as nonlinear entities of different sizes and shapes (i.e., simple diagram representations or expression trees) [8].

[13] presented an example of a generative representation for the concurrent evolution of the morphology and neural controller of simulated robots. Each robot was constructed from a sequence of construction commands that specified how to assemble both the morphology and the neural controller.

[1] presented Genetic Library Builder (GLiB), a genetic programming-based system that introduced two novel mutation operators: 1) *compression* extracts environment-specific additions to the primitive language from the genetic material of the population, defining new modules; and 2) *expansion* replaces a compressed module by its original definition.

Hypercube-based NeuroEvolution of Augmenting Topologies (HyperNEAT) is a form of neuroevolution, i.e., evolving artificial neural networks through evolutionary algorithms [26]. HyperNEAT employs an indirect encoding that can produce connectivity patterns with symmetries and repeating motifs by interpreting spatial patterns generated within a hypercube as connectivity patterns in a lower-dimensional space.

Though not used extensively, variable-length genomes have been around for quite some time.[5] Early work by [10] presented messy genetic algorithms

[5]Of course, some representations, such as trees in genetic programming, are inherently

(mGA), which process variable-length strings that may be either under- or over-specified with respect to the problem being solved. Again, while different than what we have done here, there is an interesting common thread, that of allowing evolution to discover the underlying representation needed: "Prior to this time, no GA had ever solved a provably difficult problem to optimality without prior knowledge of good string arrangements. The mGA presented herein repeatedly achieves globally optimal results without such knowledge ..." [10]

[17] presented a general variable length genome, called exG, to address the problems of fixed-length representations in canonical evolutionary algorithms. They presented some preliminary results and a discussion of the proposed method's usage.

Grammatical Evolution (GE) was introduced by [24] as a variation on genetic programming. Here, a Backus-Naur Form (BNF) grammar is specified that allows a computer program or model to be constructed by a simple genetic algorithm operating on an array of bits. The GE approach is appealing because only the specification of the grammar needs to be altered for different applications. One might consider subjecting the grammar encoding to evolution in an OMNIREP manner (as done, e.g., by [2]). [21] combines GE with the developmental notion of complex genotype-to-phenotype mapping.

Linear genetic programming is a form of genetic programming wherein computer programs in a population are represented as sequences of instructions [3]. This is similar to our second experiment (Section 2.4), except that the encoding is fixed.

Within a memetic computing framework, Iacca et al. [15] proposed, "a bottom-up approach which starts constructing the algorithm from scratch and, most importantly, allows an understanding of functioning and potentials of each search operator composing the algorithm." Caraffini et al. [5] proposed a computational prototype for the automatic design of optimization algorithms, consisting of two phases: a problem analyzer first detects the features of the problem, which are then used to select the operators and their links, thus performing the algorithmic design automatically. Both these works share the desire to tackle basic algorithmic design issues in a (more) automatic manner.

Within the context of membrane systems (also called P systems), Zhang et al. [29] proposed a novel way to design a P system for directly obtaining the approximate solutions of combinatorial optimization problems without the aid of evolutionary operators. Iacca et al. [14] presented a coevolutionary algorithm, proposing a memetic computing structure wherein a population of candidate solutions, termed coevolving aging particles, are perturbed, independently, along each dimension. Both these works address basic representational issues.

Tangentially related to our work herein is the extensive research on parameters and hyper-parameters in EC, some of which has focused on self-adaptive algorithms, wherein the parameters to be adapted are encoded into the chromosomes and undergo crossover and mutation. The reader is referred to [25] for a comprehensive discussion of this area. Another tangential connection is to

variable-length. Herein, we simply refer to the literature on "variable-length genomes".

"smart" crossover and mutation operators, wherein, interestingly, coevolution has also been applied [28].

4 Concluding Remarks

We presented four experiments that provide a proof-of-concept of the idea of coevolving representations and encodings. We perceive OMNIREP not as a particular algorithm but rather as a meta-algorithm, which might hopefully be suitable for other settings. Essentially, any scenario where some form of representation may be interpreted in several ways, or where the representation and encoding can be rendered "fluid" rather than fixed, might be a candidate for an OMNIREP approach.

We think that the framework expounded here can aid researchers not only in solving specific problems but also as an exploratory route when one is seeking out a good representation. One could use OMNIREP to select a "winning" encoding and then continue with fixed-encoding evolution.

Future exploration might look into more applications and application areas for OMNIREP. For example, experiment 3 (programs) might be expanded to full-blown coevolution of programs in common programming languages such as Java and Python, along with their interpreters or compilers [22].

Various tricks of the EC trade can be applied to OMNIREP, e.g., lexicase selection, smart variation operators (e.g., [28]), hall-of-fame, novelty search, and more.

Many real-life problems are ones that exhibit constraints, multiple objectives, or both. Such problems would be well worth exploring. Last, but not least, we plan to address issues relating to computational costs incurred when running OMNIREP-like algorithms.

Thinking wider, once one embraces the "fluidity" of representations, other setups come to mind. We have considered here two populations, one of representations and one of encodings, which we might rename *OMNIREP-e*, the 'e' referring not only to 'encodings' but also to 'explicit', since the encodings are explicitly embodied in their own population.

Another version we suggest is implicit OMNIREP, *OMNIREP-i* (Figure 5). Consider the cubic polynomial example, wherein we sought four coefficients, a, b, c, d. We could set up *four* coevolving populations, which are identical at the outset in that they all contain individuals that represent numerical parameters. However, when combined to asses fitness, the individuals from the first population are used for coefficient a, those from the second population for coefficient b, and so forth. Thus, in time, each population will implicitly come to represent, or specialize, in one coefficient.

Yet another ominrep version, *OMNIREP-l*, might introduce the idea of *levels* (Figure 6): an individual in the first population is decoded by an individual in the second population, which in turn is decoded by an individual in the third population, and so forth (hopefully not ad infinitum). This might produce better and more compact solutions (as we saw in our experiments, the current

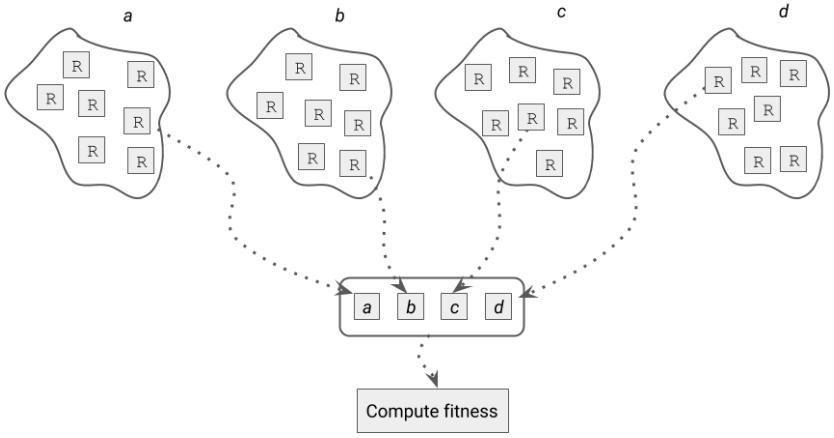

Figure 5: In OMNIREP-i each coevolving population evolves to specialize in a specific task.

OMNIREP-e can evolve compact encodings).

A major goal of artificial intelligence is that of mimicking humans by aiming to jointly discover interpretations *and* representations, ultimately resulting in meaningful insight and understanding. While this is a grander goal than that dealt with in this paper, we believe we may have taken a small step towards it.

5 Acknowledgements

This work was supported by National Institutes of Health grants AI116794, DK112217, ES013508, HL134015, LM010098, LM011360, LM012601, and TR001263.

References

[1] P. J. Angeline and J. B. Pollack. Coevolving high-level representations. In C. G. Langton, editor, *Artificial Life III*, volume XVII of *SFI Studies in the Sciences of Complexity*, pages 55–71, Santa Fe, New Mexico, 1994. Addison-Wesley.

[2] R. M. A. Azad and C. Ryan. An examination of simultaneous evolution of grammars and solutions. In T. Yu, R. Riolo, and B. Worzel, editors, *Genetic Programming Theory and Practice III*, pages 141–158, Boston, MA, 2006. Springer US.

[3] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA, 1998.

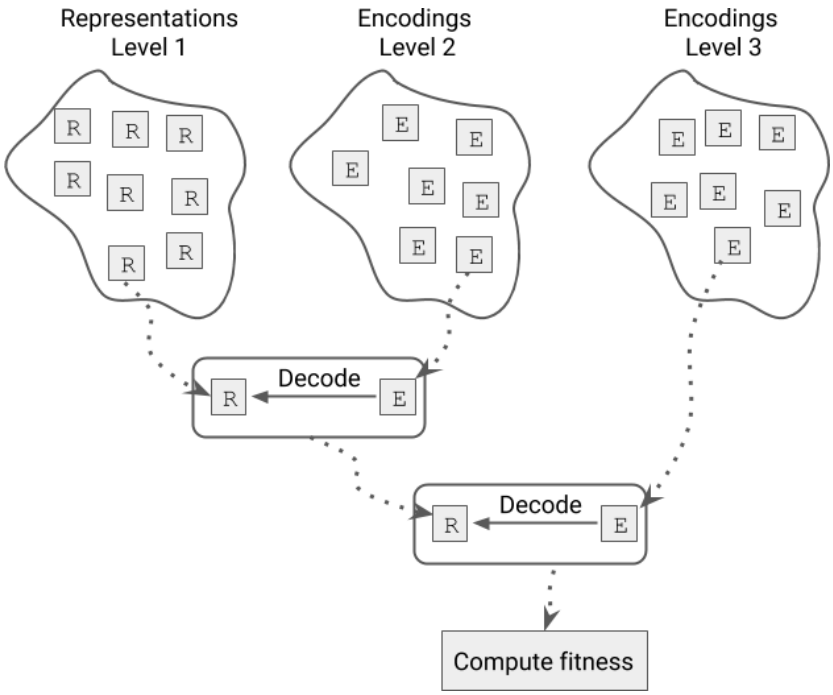

Figure 6: In OMNIREP-l there is a hierarchy of coevolving populations.

[4] P. Bentley and S. Kumar. Three ways to grow designs: A comparison of embryogenies for an evolutionary design problem. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1*, GECCO'99, pages 35–43, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[5] F. Caraffini, F. Neri, and L. Picinali. An analysis on separability for memetic computing automatic design. *Information Sciences*, 265:1–22, 2014.

[6] J. Correia, V. Ciesielski, and A. Liapis. *Proceedings of Computational Intelligence in Music, Sound, Art and Design: 6th International Conference*. Springer, Berlin, 2017.

[7] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer-Verlag, Berlin, 2003.

[8] C. Ferreira. Gene expression programming: A new adaptive algorithm for solving problems. *Complex Systems*, 13(2):87–129, 2001.

[9] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[10] D. E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3:493–530, 1989.

[11] F. Gruau, D. Whitley, and L. Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In *Proceedings of the 1st Annual Conference on Genetic Programming*, pages 81–89, Cambridge, MA, USA, 1996. MIT Press.

[12] W. E. Hart, T. E. Kammeyer, and R. K. Belew. The role of development in genetic algorithms. In L. D. Whitley and M. D. Vose, editors, *Foundations of Genetic Algorithms*, volume 3, pages 315 – 332. Elsevier, 1995.

[13] G. S. Hornby and J. B. Pollack. Creating high-level components with a generative representation for body-brain evolution. *Artificial Life*, 8(3):223–246, 2002.

[14] G. Iacca, F. Caraffini, and F. Neri. Multi-strategy coevolving aging particle optimization. *International journal of neural systems*, 24(01):1450008, 2014.

[15] G. Iacca, F. Neri, E. Mininno, Y.-S. Ong, and M.-H. Lim. Ockham's razor in memetic computing: three stage optimal memetic exploration. *Information Sciences*, 188:17–43, 2012.

[16] J. R. Koza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[17] C. Y. Lee and E. K. Antonsson. Variable length genomes for evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann, 2000.

[18] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT press, Cambridge, MA, 1998.

[19] F. Neri and C. Cotta. Memetic algorithms and memetic computing optimization: A literature review. *Swarm and Evolutionary Computation*, 2:1–14, 2012.

[20] F. Neri, C. Cotta, and P. Moscato. *Handbook of memetic algorithms*, volume 379. Springer, 2012.

[21] M. Nicolau and C. Ryan. LINKGAUGE: Tackling hard deceptive problems with a new linkage learning genetic algorithm. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 488–494. Morgan Kaufmann Publishers Inc., 2002.

[22] M. Orlov and M. Sipper. Flight of the FINCH through the Java wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–182, 2011.

[23] C. A. Pena-Reyes and M. Sipper. Fuzzy CoCo: A cooperative-coevolutionary approach to fuzzy modeling. *IEEE Transactions on Fuzzy Systems*, 9(5):727–737, 2001.

[24] C. Ryan, J. J. Collins, and M. O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Genetic Programming, First European Workshop, EuroGP'98, Paris, France, April 14-15, 1998, Proceedings*, pages 83–96, 1998.

[25] M. Sipper, W. Fu, K. Ahuja, and J. H. Moore. Investigating the parameter space of evolutionary algorithms. *BioData Mining*, 11(2):1–14, 2018.

[26] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, 15(2):185–212, 2009.

[27] K. O. Stanley and R. Miikkulainen. A taxonomy for artificial embryogeny. *Artificial Life*, 9(2):93–130, 2003.

[28] A. Zaritsky and M. Sipper. The preservation of favored building blocks in the struggle for fitness: The puzzle algorithm. *IEEE Transactions on Evolutionary Computation*, 8(5):443–455, 2004.

[29] G. Zhang, H. Rong, F. Neri, and M. J. Pérez-Jiménez. An optimization spiking neural p system for approximately solving combinatorial optimization problems. *International Journal of Neural Systems*, 24(05):1440006, 2014.