Learning Genetic Representations for Classes of Real-Valued Optimization Problems

Eric O. Scott
Computer Science Department
George Mason University
Fairfax, Virginia USA
escott8@gmu.edu

Jeffrey K. Bassett
Krasnow Institute for Advanced Study
George Mason University
Fairfax, Virginia USA
jbasset1@gmu.edu

ABSTRACT

Applying evolutionary algorithms to new problem domains is an exercise in the art of parameter tuning and design decisions. A great deal of work has investigated ways to automate the tuning of various EA parameters such as population size, mutation options, etc. However, genotype-to-phenotype mappings have typically been considered too complex to adapt automatically. We demonstrate a genetic representation learning method that uses meta-evolution to adapt a bitstring encoding for a synthetic class of real-valued optimization problems. The genetic representation we learn performs as well or better than a Gray code both on new instances of the problem class it was trained on and on problem types that it was not trained on.

Categories and Subject Descriptors

I.2 [**Computing Methdologies**]: ARTIFICIAL INTELLIGENCE—*Problem Solving, Control Methods, and Search*

Keywords

Evolutionary Algorithms, Hyper-heuristics, Multitask Learning, Transfer Learning

1. INTRODUCTION

The successful application of evolutionary algorithms to difficult optimization and design problems in science and engineering is invariably an exercise in the art of parameter tuning. It is widely understood that design decisions such as the choice of operators, mutation rate, and population size must be determined for new problems in a domain-specific manner. A number of theoretical and empirical insights have been discovered in recent decades that can help practitioners navigate the design of evolutionary algorithms in a reasonably principled way [7]. In many cases, however, the application of EAs remains a time consuming process that requires a significant amount of domain knowledge and formal or informal experimentation.

GECCO '15, July 11 - 15, 2015, Madrid, Spain

 ISBN 978-1-4503-3488-4/15/07…$15.00
DOI: http://dx.doi.org/10.1145/2739482.2768460

We often think of evolutionary algorithms as stochastic processes that search a genotype space $\mathbb{G}$ for solutions that optimize some fitness function f. Genotypes, however, are often abstract representations that can be conveniently manipulated by evolutionary operators, but that have no obvious meaning by themselves as solutions to a problem. Importantly, then, in many applications, fitness functions aren't defined over genotype space, but instead over a *phenotype space*, where candidate solutions are described in a fashion that is natural to the problem domain. A fitness function for traveling salesman problems takes graph tours as input, for instance, while a fitness function for real-valued optimization accepts vectors in $\mathbb{R}^n$. Here, it is necessary to specify *genotype-to-phenotype mapping* $\mathcal{R}$ (a.k.a. *representation*, or *encoding*) as part of the EA, which transforms individuals from their genetic representation (such as a bitstring) into a corresponding phenotype (the input to the fitness function).

At a high level of abstraction, the design decisions that go into constructing a simple (μ, λ)- or $(\mu+\lambda)$-style EA include choosing the reproduction and selection operators and their parameters, the values of μ and λ, and a stopping condition. In particular, however, the search behavior of the algorithm depends heavily on the choice of representation. This is because the representation determines the *pattern of phenotypic change* that occurs when reproductive operators are applied to individuals' genotypes. The selection of a good representation for the problem domain is commonly seen as one of the most important components of EA design—a view that is consistent with biologists' understanding of the crucial role that genotype-phenotype relationships play in natural evolution [9].

Many methods have been devised to try and automate the choice of at least some design decisions for evolutionary algorithms, and to thereby reduce the manual effort needed to tune them to a problem or domain [6, 11]. Perhaps the most widely known of such methods are self-adaptive mutation operators which, modify parameters of a mutation operator while the EA is running [3]. These are often used in the evolutionary strategies family of algorithms. Other methods adjust the parameters of the EA in an offline way: the EA parameters remain fixed during an individual run, and many independent runs are conducted to search for parameters that lead to good performance. Meta-evolution, in which one evolutionary algorithm (the meta-EA) is used to tune the parameters of another evolutionary algorithm (the sub-EA), has historically been a popular approach to offline parameter tuning [8, 10].

Because they execute the sub-EA a large number of times, offline parameter tuning methods such as meta-evolution require a great deal of computational resources. If we wish merely to solve a single problem instance, meta-evolution is only advantageous in the case that exerting effort to find good parameter choices yields a substantial improvement in the quality of the final solution [12]. Meta-evolution may also be beneficial, however, if we wish to solve many instances of a *class* of related problems. In this case, the meta-EA may yield a set of parameters that can be reused to quickly solve a large or unlimited number of problem instances. Offline parameter tuning for a class of problems can be seen as a learning problem: the parameters are tuned on a training set of problem instances, and the ability of the resulting EA configuration to generalize can be evaluated by running the sub-EA on a set of test instances.

It is relatively straightforward to use a method such as meta-evolution to automate the selection of numeric design decisions such as population size, parameters for mutation, etc. The hyper-heuristics community, moreover, has shown that it is often possible to optimize more complicated parts of a heuristic algorithm's behavior by searching for combinations of operators or heuristics that play on each others' strengths and weaknesses [4, 5]. The design of elaborate, domain-specific reproductive operators or encodings, however, is often viewed as too complex and challenging to approach automatically [6]. As such, despite the large literature on automatic EA configuration, and despite the central importance of genetic representations to EA performance, very little work has attempted to adapt a representation to a problem or class of problems.

Simões et al. have recently suggested that multi-layer, feed-forward neural networks can be used for representing and adapting arbitrary genotype-to-phenotype encodings for real-valued optimization problems [14]. They conduct an exploratory study of how several such encodings radically alter the local neighborhood of a mutation operator that takes a fixed step size in a real-valued genotype space. Their goal is to understand how the selection of a fruitful genotype-to-phenotype mapping might be automated; however, they do not propose a mechanism for doing so, and it may be very difficult to learn complex neural network encodings that are useful on classes of problems.

In this work we present what is to our knowledge the first demonstration of representation learning. We propose a relatively simple *meta-representation*—a representation for representations—that allows us to specify a restricted class of genotype-to-phenotype maps that convert fixed-length bitstrings into real-valued vectors. The familiar binary encoding is a special case of this general class of bitstring encodings. We find that we are able to use meta-evolution to learn a representation that makes it easy to solve a synthetic set of training, validation and test problems. We also show that the resulting representation *generalizes* in the sense that it is useful on other problem instances from the same problem class and dimensionality it was trained on, and even on completely new problems from outside the class.

2. METHODOLOGY

A genotype-to-phenotype mapping defines the effect that each gene has on the phenotypic representation of a solution. In traditional binary representations for real-valued optimization, each bit in the genome has an independent additive effect on exactly one phenotypic trait. For instance, the bitstring $\vec{g} = 0101$ can be converted into the real-valued phenotype $p = 5.0$ by interpreting the two non-zero bits as $2^0 = 1$ and $2^2 = 4$ and collecting the sum. We observe that in binary code, A) since the effect of each bit is purely additive, there are no non-linear interactions among genes with respect to the phenotype, and B) the magnitude of each bit's impact varies exponentially, allowing both small and large jumps to occur in phenotype space when bit-flip mutation is applied.

We construct a general class of bitstring representations that preserve these two properties. We also allow our mappings to have the property of *pleiotropy*—genes may influence multiple traits. Let a phenotype consist of a vector in $\mathbb{R}^n$. Each element of the phenotype vector is a *phenotypic trait*. We define a mapping $\mathcal{R} : \mathbb{B}^m \mapsto \mathbb{R}^n$ by assigning a weight w_{ij} to every possible gene-trait interaction. Each gene is further assigned a factor $s_i \in \mathbb{R}$, which serves as an exponent that scales the magnitude of all the gene's phenotypic effects. Specifically, let $\vec{g} \in \mathbb{B}^m$ be a bitstring genome corresponding to an individual in the sub-EA. Then each trait in the corresponding phenotype vector $\vec{p} \in \mathbb{R}^n$ is determined by the equation

$$p_j = \sum_{i=1}^{m} 2^{s_i} w_{ij} g_i. \quad (1)$$

So when g_i is 1, a value proportional to w_{ij} is contributed to each trait p_j for all $j \in \{1..n\}$. When g_i is 0, the w_{ij}'s contribute nothing to $\vec{p}$. If the s_i's are uniformly distributed, then the magnitude of each gene's effect will be exponentially distributed. We find it helpful to think of the parameters in Equation 1 as a set of m vectors of length n. The bits in the sub-EA bitstring $\vec{g}$ *select* which of the vectors $2^{s_i}\vec{w}_i$ are added together in the phenotype space $\mathbb{R}^n$ to collectively form a candidate solution:

$$\vec{p} = \mathcal{R}(\vec{g}) = \sum_{i=1}^{m} 2^{s_i} \vec{w}_i g_i. \quad (2)$$

Each mapping of this type can be seen as a linear transformation from the space of bitstrings to $\mathbb{R}^n$, and any linear transformation between these spaces can be encoded by an appropriate choice of weights. Accordingly, we refer to mappings of this type as *linear pleiotropic encodings*.

Since we have allowed gene effects to be pleiotropic, these mappings bear some similarity to a representation once explored by Altenberg [1]. Equation 1 can also be interpreted as a feed-forward neural network with linear activation functions (Figure 1). Since we are limiting our choice of mappings to linear transformations that take binary inputs, however, the expressive power of this class of representations is more limited than the complex neural network mappings studied by Simões et al. [14].

2.1 Meta-Evolution of Representations

Fully $m \cdot (n+1)$ real-valued constants are required to specify the parameters that make up a single linear pleiotropic mapping $\mathcal{R}$. The question we pose here is whether it is feasible to automatically search this high-dimensional parameter space for a mapping that is effective on a class a problems.

For the purpose of meta-evolution, we group the parameters of a linear pleiotropic encoding into a sequence of m

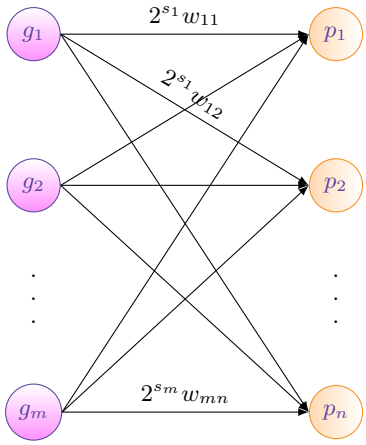

Figure 1: A linear pleiotropic encoding can be interpreted as a feed-forward neural network, in which genes have an additive effect on phenotypes.

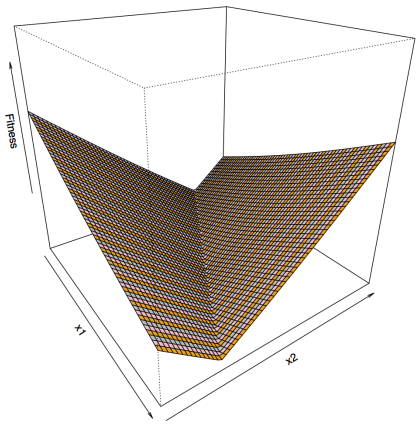

Figure 2: The valley objective.

tuples of the form $(s_i, \vec{w}_i) \in \mathbb{R} \times \mathbb{R}^n$. As is made clear from Equation 2, the values in the ith tuple of a meta-individual completely define the effect that the ith bit of a sub-EA individual has on the phenotype. This sequence of tuples forms the meta-level genome, and we treat each tuple as a meta-level gene.

We use a generational evolutionary algorithm with two-point crossover to tune the values of the tuples $(s, \vec{w})$. Each tuple in a child has a chance of being selected for mutation. When the ith tuple is selected for mutation, we add values drawn i.i.d. from a normal distribution to both s_i and to each element of $\vec{w}_i$, unless otherwise specified. The number of genes m that the sub-EA uses to represent solutions in $\mathbb{R}^n$ is a free parameter that must be chosen *a priori*. We opt to use 20 genes per dimension, simply because traditional bitstring encodings typically need about that many in order to have a sufficiently fine-grained ability to cover the phenotype space. The remaining design decisions we use to implement our meta-EA are detailed in Table 1.

The exponential scaling factors s_i do not add to the expressive power of the meta-representation scheme. Any linear pleiotropic mapping defined by a sequence of tuples $(s_i, \vec{w}_i)$ can be represented by an equivalent mapping $(1, \vec{v}_i)$, where $v_{ij} = 2^{s_i} w_{ij}$. The merit of explicitly including the s_i's is that it changes how the meta-EA's mutation operator affects vectors that differ exponentially in length. When s_i is large, a small Gaussian perturbation of the values in $\vec{w}_i$ has a large effect on the vector $2^{s_i}\vec{w}_i$. When s_i is small, perturbing the values of $\vec{w}_i$ has a small effect, allowing the shorter vectors to be fine-tuned.

The fitness we assign to a candidate mapping $\mathcal{R}$ should reflect $\mathcal{R}$'s ability to serve as a useful component of a sub-EA's search heuristic. One way to measure this is to plug $\mathcal{R}$ into a sub-EA, and then run the sub-EA several times on an objective function $f : \mathbb{R}^n \mapsto \mathbb{R}$ and see how it does. This raises the concern of over-fitting, however: if we succeed in adapting an encoding $\mathcal{R}$ that can easily solve f every time, have we found a good search heuristic, or have we simply designed an algorithm that has 'memorized' the location of f's global optimum by encoding it into $\mathcal{R}$? If our goal is only to optimize a single difficult function f, then we don't care about over-fitting. If we wish to reuse $\mathcal{R}$ to solve new instances from a class of problems, however, we need to encourage the meta-EA to find a more general solution.

Here we assume that we have at our disposal a *training set* composed of several examples from a class of problems, all of the same dimensionality. Such classes arise frequently in algorithmic applications – traveling salesmen problems with 20 cities, for instance, or room-scheduling problems with 20 rooms. To assign fitness to a mapping $\mathcal{R}$, we plug $\mathcal{R}$ into a sub-EA, and run the sub-EA once on each problem in the training set. The average best fitness the sub-EA achieves on the training problems becomes $\mathcal{R}$'s fitness.

2.2 Problem Class

For this study, we synthesize a problem domain that is diverse enough to demonstrate the ability for the meta-EA to learn, but that is still relatively simple to analyze. We define the *translation class of* f, $\mathcal{T}(f)$, as the set of all functions created by applying an arbitrary offset to f in its input space within some bounds. Every element of $\mathcal{T}(f)$ has the same shape, but the location of the global optimum varies. As such, it is not sufficient to construct a representation that memorizes the location of the optimum.

For the objective f we choose the "valley objective" defined in [2]:

$$f(\vec{x}) = 10\delta(\vec{x}, L) + \|\vec{x} - \vec{o}\|, \tag{3}$$

where L is some line that passes through the optimum $\vec{o}$, and $\delta(\vec{x}, L)$ denotes the distance between $\vec{x}$ and the nearest point on the line. In two dimensions, this function defines a valley with linearly sloping sides (via the first term) and a slight conical gradient that prevents its floor from being flat (via the second term) – see Figure 2. We define our translation class $\mathcal{T}(\texttt{Valley})$ such that random translations are applied within the bounds $[-15, 15]$ along each dimension.

2.3 Predictions

We have defined our meta-evolutionary scheme for representation learning and a synthetic problem class to exercise it on – namely the translation class $\mathcal{T}(\texttt{Valley})$. We now investigate our meta-EA's ability to learn a genotype-phenotype mapping on a set of training instances that is useful for solving new problem instances.

Component	Meta-EA	Sub-EA
Type	$(\mu + \mu)$	$(\mu + \mu)$
Pop. Size (μ)	50	50
Gene Type	$(s, \vec{w}) \in \mathbb{R} \times \mathbb{R}^n$	$b \in \mathbb{B}$
Genes/dimension	20	20
Initialization Bounds	$s \in [-4, 4]$, $w_i \in [-1, 1]$	n/a
Parent Selection	Binary tournament	Binary tournament
Reproduction	2-point crossover	2-point crossover
Mutation	Gaussian perturbation of all values ($\sigma = 1$)	Bit-flip
Mutation Rate	$1/L$ chance per gene	$1/L$
Mutation Bounds	Soft	n/a
Objective	Mean best fitness of 10 sub-EA runs	$\mathcal{T}$(`Valley`) with no rotation
Stopping Condition	500 generations	40 generations without improvement

Table 1: Default configuration used in our meta-EA experiments, except where otherwise specified.

First, we have made a number of design decisions in the implementation of the meta-EA itself, not all of which may be optimal. In particular, we went out of our way to include an extra scale parameter s_i in each gene of our meta-representation. Is this useful? Or is it superfluous? This is our first hypothesis:

> **Hypothesis 1:** It will be easier to improve the training fitness of a mapping $\mathcal{R}$ on the translation class $\mathcal{T}$(`Valley`) if we mutate both the magnitude and the elements of each vector in the mapping than if we only mutate one or the other.

Next, we predict that we ought to be able to learn a genotype-to-phenotype map that allows us to effectively solve arbitrary instances of $\mathcal{T}$(`Valley`). This is serves as a proof of concept for this approach to learning representations:

> **Hypothesis 2:** Let $\mathcal{R}$ be a linear pleiotropic encoding evolved to solve instances of the translation class $\mathcal{T}$(`Valley`). Then $\mathcal{R}$ will perform competitively against traditional bitstring encodings when applied to new instances of $\mathcal{T}$(`Valley`).

Provided that we succeed in learning a good mapping for $\mathcal{T}$(`Valley`), it would be useful if we could say something about *why* the learned linear pleiotropic representation works, rather than just *whether* it works. We may be able to determine a pattern in the resulting map that corresponds to features of the landscapes it was trained on:

> **Hypothesis 3:** The linear pleiotropic encoding we learn to solve $\mathcal{T}$(`Valley`) will contain a concentration of vectors that are aligned with the bottom of the valley.

Now, one might expect that since we are tailoring the map to a specific problem class, that it will only be useful for solving instances of that class. In some cases, however, the implicit search heuristic encoded by $\mathcal{R}$ may be of more *general* use, much like Gray code is useful on a wide diversity of problems. Stated differently, the information we learn about how to solve instances of $\mathcal{T}(f)$ may *transfer* to other problem classes:

> **Hypothesis 4:** Let $\mathcal{R}$ be a linear pleiotropic encoding evolved to solve instances of the translation class $\mathcal{T}$(`Valley`). Then $\mathcal{R}$ will perform comparably to traditional bitstring encodings on new problems that do not belong to $\mathcal{T}$(`Valley`).

3. RESULTS

3.1 Learning

As specified in Table 1, we ran 50 independent runs of a meta-EA with a population size of 50 for 500 generations. Each run is initialized with an independent sample of 10 training problems from $\mathcal{T}$(`Valley`). The fitness of a mapping $\mathcal{R}$ is defined as the average best fitness that the sub-EA achieves on the 10 training problems. The sub-EA stops when it has gone 40 generations with no improvement in its best fitness. We found that setting the meta-level population much lower than 50 caused it to converge prematurely, while increasing it beyond 50 did not measurably improve the final best training fitness (not shown).

It is not clear *a priori* whether it ought to be beneficial to mutate the s_i's and weights w_{ij}, or whether only one or the other need to vary for learning to occur. We ran experiments for four different adaptation schemes (Figure 3): A) one in which the scale factors (which control the magnitudes of the vectors) were all fixed at a value of 1, while the weights w_{ij} were mutated, B) the s_i's were randomly initialized and then held constant while the weights were mutated, C) the s_i's were mutated while the weights were held constant at their initial random values, and finally D) both the s_i's and the w_{ij}'s were allowed to mutate. The results confirm our **Hypothesis 1**: mutating both is more effective than holding one fixed. For the remainder of the paper all experiments were conducted with both types of mutation enabled (D).

To determine whether we begin to over-fit as evolution proceeds, an additional 20 instances of $\mathcal{T}$(`Valley`) are reserved as a validation set. Each run of the meta-EA took several hours to execute on a sequential processor, even though each sub-EA run took less than one second – so the meta-EA represents a substantial investment of resources. Because measuring the validation fitness of the population is expensive and does not affect evolution, we only take validation measurements every 20 generations.

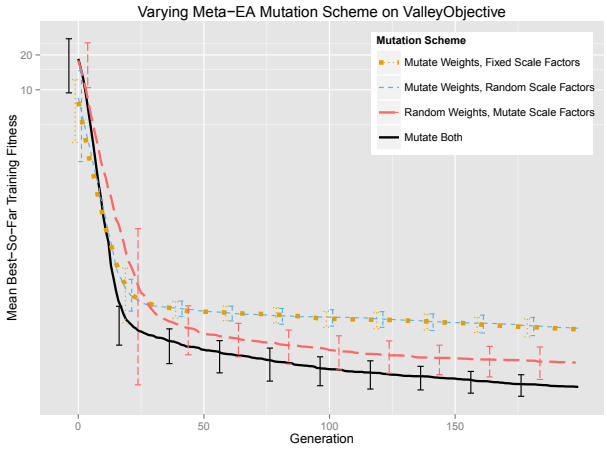

Figure 3: Meta-level fitness trajectories for different mutation methods. Error bars denote standard deviation.

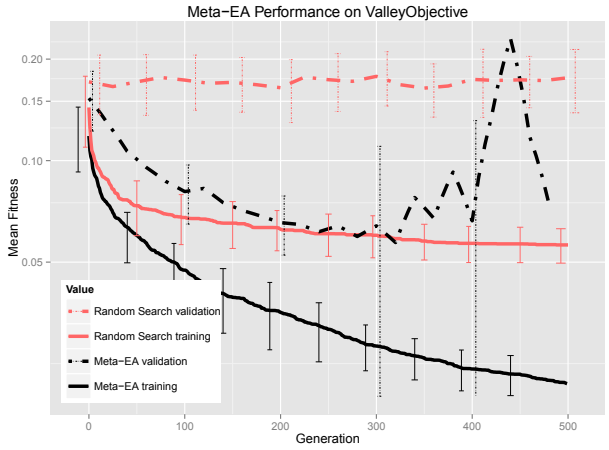

Figure 4: Training and validation fitness for 50 independent runs of the meta-EA. Error bars denote standard deviation on the mean. Reported is the mean best-so-far fitness on the training set and the mean best-of-generation fitness on the validation set.

Figure 4 shows the improvement in the mean training fitness across the 50 runs for the meta-EA, compared an experiment where the meta-EA was replaced by a random search algorithm. The validation curve shows the mean fitness of the *best-of-generation* individual with on the validation set. In general, this is a different individual than the individual with the best-so-far training fitness. We found that the individual with the best-so-far training fitness tended to be very over-fit to the training set (not shown). The best-of-generation validation fitness, however, consistently improves for about the first two hundred generations before becoming wildly unpredictable late in the run.

3.2 Testing on New Problem Instances

Our goal in the experiment in Figure 4 is to select a mapping that will perform well on instances it hasn't been trained on. We obtained a high-quality mapping that was not over-fit by selecting the individual that had the best *validation* fitness of the run among the times validation fitness was measured. So, we do not use information from the validation set during evolution, but at the end of the run we use the validation set to choose which individual to select as our trained genotype-to-phenotype mapping $\mathcal{R}^*$. To evaluate the performance of the mapping chosen from the validation results, we construct a tertiary *test set* by taking 100 more random instances of the translation class $\mathcal{T}(\texttt{Valley})$. We chose a learned encoding $\mathcal{R}^*$ from a typical run of the meta-EA, plugged it into a sub-EA, and ran the sub-EA once on each function in the test set.

The left-hand side of Figure 5 shows the results, compared against genetic algorithms that use Gray code and a standard binary encoding. The only difference between the three algorithms is the encoding method used. We find that the learned encoding's average performance is statistically indistinguishable from Gray code's performance at solving new instances of $\mathcal{T}(\texttt{Valley})$, and that it performs better than the standard encoding.

Now, it is well known that most common forms of recombination struggle with rotated problems if the rotation introduces interactions between variables that aren't present when the problem is aligned with the axes [13]. This is known as epistasis, and is a well studied problem in EAs. To see how our learned encoding $\mathcal{R}^*$ handles epistasis, we applied a $\pi/6$ radian rotation to every problem in our tertiary test set. The result is shown in right-hand side of Figure 5. The performance of the traditional encodings are very poor on the rotated version of the valley landscape. Because the mappings our meta-EA learns are pleiotropic, however, the sub-EA often alters more than one trait at a time when a single bit is flipped. As a result,it can deal with epistasis much better than the traditional encodings are able to. So we have confirmed **Hypothesis 2** (the learned encoding performs competitively), and we have demonstrated a simple form of transfer: the learned encoding is able to generalize to solve problems with a rotation that it was not trained on.

3.3 Analysis

Our third prediction was that the mappings we trained on the instances of $\mathcal{T}(\texttt{Valley})$ would hold a particular signature: We anticipate that the weight vectors $2^{s_i}\vec{w}_i$ will be pointed in a direction that aligns with the floor of the valley (the line L in Equation 3).

We applied a $\pi/6$ radian rotation to the valley objective, and created a translation class of rotated valleys $\mathcal{T}_{\pi/6}(\texttt{Valley})$. We ran 50 independent runs of the meta-EA with training and validation sets drawn from $\mathcal{T}_{\pi/6}(\texttt{Valley})$. We took the mappings $\mathcal{R}^*$ with the best validation fitness from each of the 50 runs, and analyzed the 40 vectors that made up each mapping, for a total of 2,000 vectors.

The results (Figure 6) indicate that a large proportion of the vectors are indeed aligned with the valley floor (indicated by the dashed red line). This confirms **Hypothesis 3**: a good pleiotropic mapping is one that permits mutations that take it along the valley floor. No such rule seems to apply to the larger vectors, however. This could indicate that the learned bias is more important during the exploitation phase of the search process than exploration.

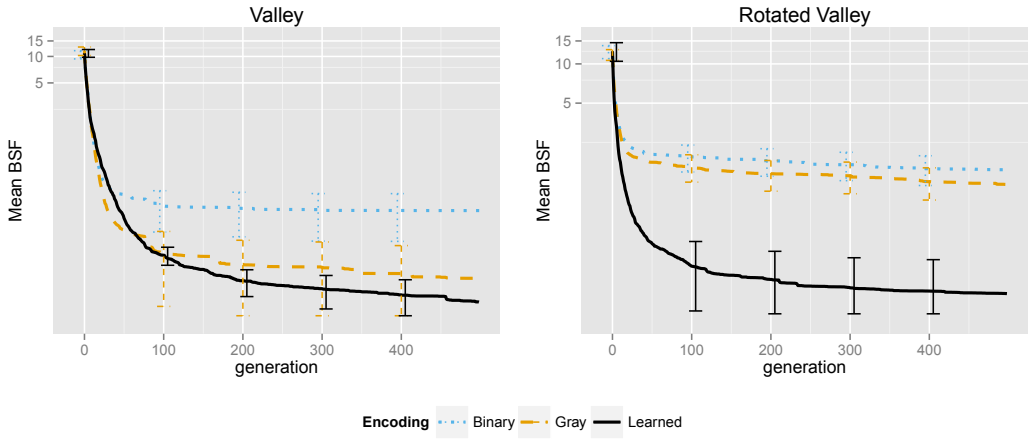

Figure 5: Comparing the performance of a learned representation to Gray code and standard binary encoding on instances of the non-rotated (Left) and rotated (Right) valley function. Error bars indicate the 95% confidence interval on the mean over the 100 test problems.

3.4 Robustness Across Problem Classes

We have shown that the mappings we were able to learn on the translation class of the valley objective perform as well as Gray code on other instances of the same problem class. But how does a learned encoding perform on problems unlike anything it has seen before, such as multi-modal landscapes?

We took the same mapping $\mathcal{R}^*$ that we trained on $\mathcal{T}(\texttt{Valley})$ in Section 3.2, and applied it to new test sets of 100 instances each from the sphere, Rastrigin, Rosenbrock and Ackley functions. We then applied a $\pi/6$ radian rotation to those instances, and ran our learned encoding on those too, for a total of 8 new translation classes. The results show that the learned encoding performed as well or better than Gray code on all 8 classes (Figure 7), even though $\mathcal{R}^*$ was not trained on functions of this sort.

4. CONCLUSION

We have introduced a new 'representation for representations' – the linear pleiotropic encodings – to facilitate learning genetic representations for classes of real-valued optimization problems. Although this brief study was confined to simple, synthetic problem classes, we have shown a proof of concept that learned pleiotropic representations can preform competitively with traditional bitstring encodings, that they are robust to rotations of the landscape, and that in some cases they may even be useful on problem classes that they were not trained for.

The results of our final experiments in particular indicate that not only does our representation learning scheme avoid over-fitting to the specific problems we train it on, but it also displays a remarkably general-purpose problem-solving ability, akin to Gray code. We conjecture that this robustness may be a result of the limited expressive power of linear pleiotropic encodings, which may prevent them from being over-fit to the environment they evolved for.

Overall, our results suggest that, while meta-evolution remains a costly way to design algorithms, representation learning may not be as intractable as is commonly believed. The general approach we have presented here is not necessarily specific to real-valued problems, either – pleiotropy is a versatile concept, and a similar scheme could be adapted to, for instance, pseudo-Boolean functions.

A limitation of the present work is that our meta-EA requires a fixed number of genes m and number of phenotypic dimensions n. In contrast to binary or Gray codes, which scale easily, a linear pleiotropic mapping learned for one dimensionality n cannot be directly used on a problem with a different number of dimensions, nor can the number of bits be adjusted as needed without learning a new encoding from scratch. Additionally, a possible threat to the validity of our conclusions is that all of our experiments on tertiary test sets of the various problem classes were conducted with one learned mapping that was the result of a single run of the meta-EA. While we believe these results are representative of the meta-EA's typical learning behavior, future work will need to confirm these conclusions with statistical rigor.

On a more general level, part of the reason that meta-evolution is so expensive is because there is no obvious way to do credit assignment. If one mapping is better than another, which tuples in the meta-representation are responsible for the difference in performance? Alcaraz et al. have demonstrated the usefulness of using measures of 'effective fitness' to predict which components of hyper-heuristics are the most promising as evolution progresses [15]. One could conceive of incorporating a similar credit-assignment mechanism into the evolution of EA representations.

5. REFERENCES

[1] L. Altenberg. Evolving better representations through selective genome growth. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 182–187. IEEE, 1994.

[2] J. K. Bassett. *Methods for improving the Design and Performance of Evolutionary Algorithms*. PhD thesis, George Mason University, Fairfax, VA, 2012.

[3] H.-G. Beyer and K. Deb. On self-adaptive features in real-parameter evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 5(3):250–270, 2001.

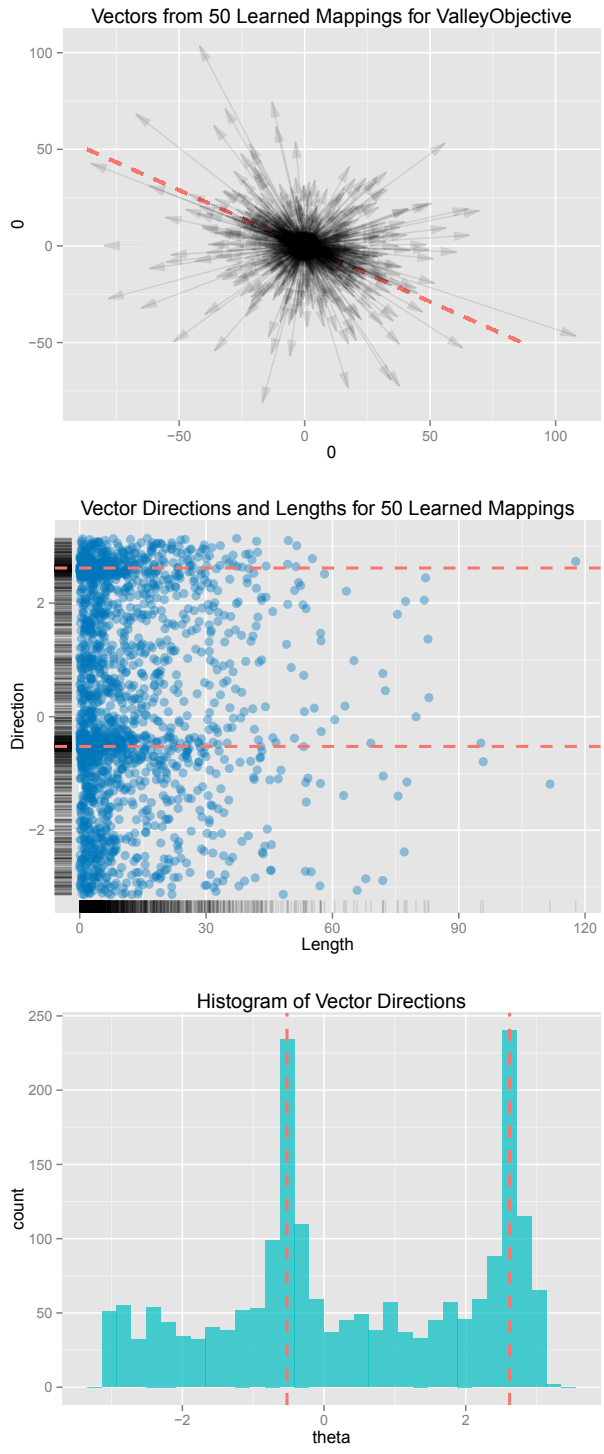

Figure 6: Three ways of summarizing the 2,000 vectors that make up 50 learned mappings that were trained on $\mathcal{T}_{\pi/6}$(`Valley`). The dashed line indicates the angle of the valley floor, which lies at a $-\pi/6$ rotation from the axis in these figures.

[4] E. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, and S. Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. In *Handbook of metaheuristics*, pages 457–474. Springer, 2003.

[5] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J. R. Woodward. A classification of hyper-heuristic approaches. In *Handbook of metaheuristics*, pages 449–468. Springer, 2010.

[6] K. De Jong. Parameter setting in EAs: a 30 year perspective. In *Parameter Setting in Evolutionary Algorithms*, pages 1–18. Springer, 2007.

[7] K. A. De Jong. *Evolutionary Computation: A Unified Approach*. MIT Press, Cambridge, Mass, 2006.

[8] W. De Landgraaf, A. Eiben, and V. Nannen. Parameter calibration using meta-algorithms. In *CEC 2007: IEEE Congress on Evolutionary Computation*, pages 71–78. IEEE, 2007.

[9] J. Gerhart and M. Kirschner. The theory of facilitated variation. *Proceedings of the National Academy of Sciences*, 104(suppl. 1):8582–8589, May 2007.

[10] J. J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics*, 16(1):122–128, 1986.

[11] G. Karafotias, M. Hoogendoorn, and A. Eiben. Parameter control in evolutionary algorithms: Trends and challenges. *IEEE Transactions on Evolutionary Computation*, 19(2):167–187, April 2015.

[12] S. Luke and A. Talukder. Is the meta-EA a viable optimization method? In *GECCO '13: Proceedings of the 15th annual conference on Genetic and Evolutionary Computation*, pages 1533–1540. ACM, 2013.

[13] R. Salomon. Re-evaluating genetic algorithm performance under coordinate rotation of benchmark functions. A survey of some theoretical and practical aspects of genetic algorithms. *BioSystems*, 39(3):263–278, 1996.

[14] L. F. Simões, D. Izzo, E. Haasdijk, and A. E. Eiben. Self-adaptive genotype-phenotype maps: Neural networks as a meta-representation. In T. Bartz-Beielstein, J. Branke, B. Filipič, and J. Smith, editors, *Parallel Problem Solving from Nature – PPSN XIII*, volume 8672 of *Lecture Notes in Computer Science*, pages 110–119. Springer, 2014.

[15] J. A. Soria Alcaraz, G. Ochoa, M. Carpio, and H. Puga. Evolvability metrics in adaptive operator selection. In *GECCO '14: Proceedings of the 2014 Conference on Genetic and Evolutionary Computation*, pages 1327–1334. ACM, 2014.

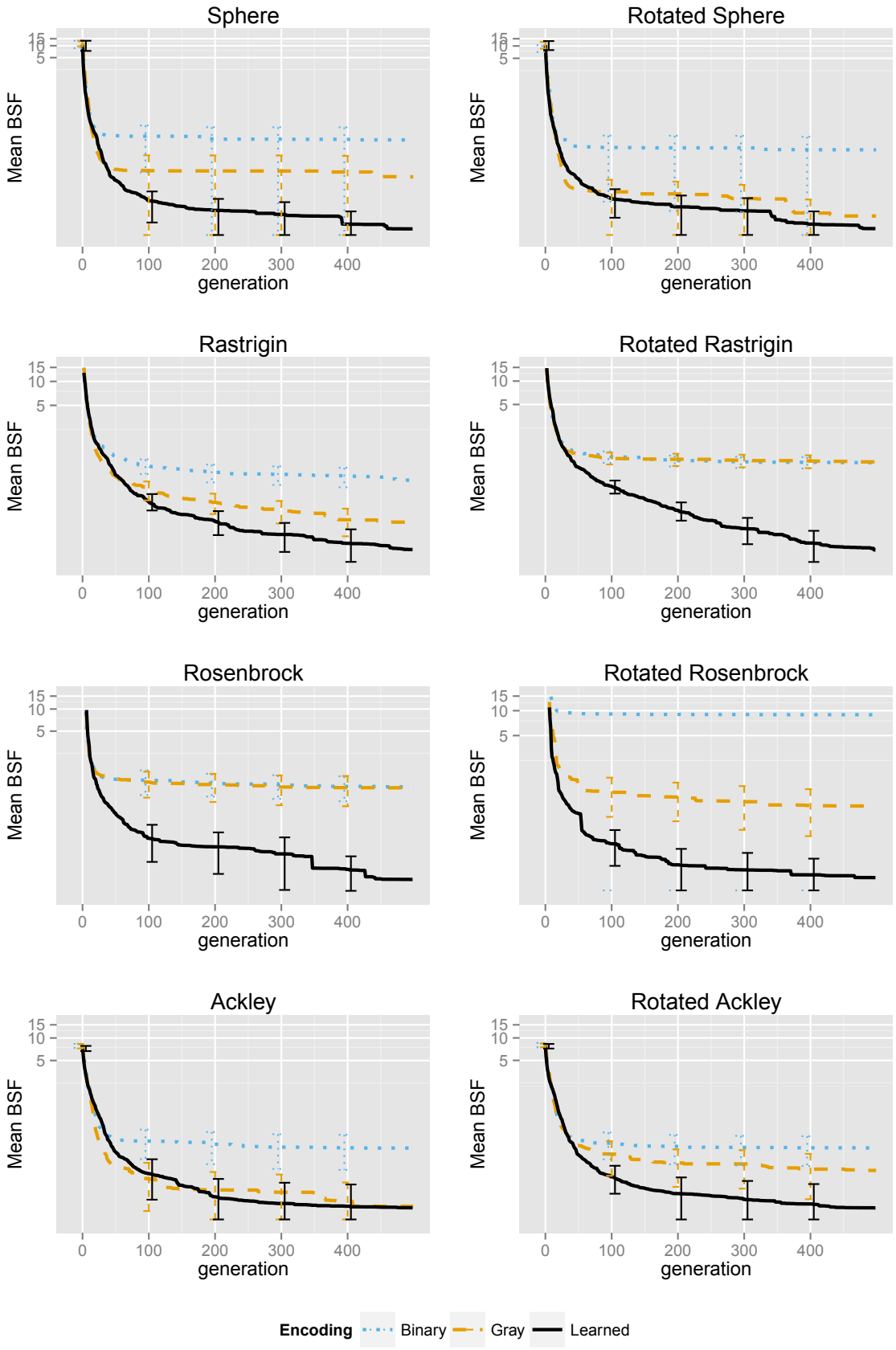

Figure 7: When we train a pleiotropic encoding on the class of valley objectives and then use it on instances of other functions, it performs as well or better than a traditional binary encoding. Shown is the sub-EA BSF averaged over 50 instances of each problem class. Error bars denote 95% confident intervals on the mean.