Incorporating Expert Knowledge in Evolutionary Search: A Study of Seeding Methods

Michael D. Schmidt
Computational Synthesis Laboratory
Cornell University
Ithaca, NY 14853, USA
mds47@cornell.edu

Hod Lipson
Computational Synthesis Laboratory
Cornell University
Ithaca, NY 14853, USA
hod.lipson@cornell.edu

ABSTRACT
We investigated several methods for utilizing expert knowledge in evolutionary search, and compared their impact on performance and scalability into increasingly complex problems. We collected data over one thousand randomly generated problems. We then simulated collecting expert knowledge for each problem by optimizing an approximated version of the exact solution. We then compared six different methods of seeding the approximate model in to the genetic program, such as using the entire approximate model at once or breaking it into pieces. Contrary to common intuition, we found that inserting the complete expert solution into the population is not the best way to utilize that information; using parts of that solution is often more effective. Additionally, we found that each method scaled differently based on the complexity and accuracy of the approximate solution. Inserting randomized pieces of the approximate solution into the population scaled the best into high complexity problems and was the most invariant to the accuracy of the approximate solution. Furthermore, this method produced the least bloated solutions of all methods. In general, methods that used randomized parameter coefficients scaled best with the approximate error, and methods that inserted entire approximate solutions scaled worst with the problem complexity.

Categories and Subject Descriptors
I.2.6 [**Symbolic and Algebraic Manipulation**]: Learning – *Knowledge acquisition*

General Terms
Algorithms

Keywords
Symbolic Regression, Prior Knowledge

1. INTRODUCTION
A common challenge in genetic programming is how to take advantage of prior knowledge and expert knowledge. Utilizing expert knowledge could be used to find solutions that are more interpretable or reliable in their applications [1, 2]. Perhaps most importantly however, expert knowledge could be used to scale genetic programs to solve increasingly complex problems [3] – freeing new evolutionary runs from having to reinvent all past knowledge from scratch over and over.

In this paper, we explore one of the more general forms of expert knowledge: reusing established or prior solutions to solve a related problem at hand. For example, if we had a model of the metabolic network in a yeast cell, how could we reuse this model to find the metabolic network of a mammalian cell using an evolutionary search? We can generalize this task in genetic programming as the problem of reusing any previous solution that has the same basic problem structure or tree encoding for a new problem.

We define *seeding* as the reuse of a prior knowledge solution by introducing all or any part of its encoding into the population during a new evolutionary run. By injecting genes from a prior knowledge solution, seeding is effectively biasing the evolutionary search toward solutions that use ubiquitous features of the related solution [4], even though solutions to the new problems may look very different at a higher level.

There are many potential approaches to seed the solutions in an evolutionary search. Here, we examine six general seeding approaches: injecting prior solutions in their entirety into the population, injecting pieces of the prior solution, injecting entire solutions but with randomly rearranged and shuffled versions of the prior solution, and finally each of these the methods again with either the optimized parameter values from the prior solution or randomized parameters.

We measure the impact of each method on randomly generated problems over one thousand evolutionary runs each. We simulated expert knowledge for these random problems by simplifying and approximating their exact solutions.

In the remaining sections, we overview back ground information in symbolic regression and seeding, describe each seeding method in greater detail, compare their results in fitness, convergence, and bloat, and end with discussion and conclusions.

2. BACKGROUND

2.1 Symbolic Regression
Symbolic regression [5] is a type of genetic program for searching the space of expressions computationally by minimizing various error metrics. Both the *parameters* and the *form* of the equation are subject to search. In symbolic regression, many initially

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'09, July 8–12, 2009, Montréal Québec, Canada.
Copyright 2009 ACM 1-58113-000-0/00/0004…$5.00.

random symbolic equations compete to model experimental data in the most parsimonious way. It forms new equations by recombining previous equations and probabilistically varying their sub-expressions. The algorithm retains equations that model the experimental data well while abandoning unpromising solutions. After an equation reaches a desired level of accuracy, the algorithm terminates, returning its most parsimonious equation that is most likely to correspond to the intrinsic mechanisms of the observed system.

In symbolic regression, the *genotype* or *encoding* represents symbolic expressions in computer memory. Often, the genotype is a binary tree of algebraic operations with numerical constants and symbolic variables at its leaves [6, 7] (for example, a binary parse tree). Other encodings include acyclic graphs [8] and tree-adjunct grammars [9].

The *fitness* of a particular genotype (a candidate equation) is a numerical measure of how well it fits the data, such as the equation's correlation or squared-error with respect to the experimental data. We use normalized mean absolute error in our experiments. We normalize by the standard deviation of the data so that we can compare fitness trends between different target equations and datasets.

A *point mutation* can randomly change the type of the floating-point operation (for example, flipping an add operation to a multiply or an add to a system variable), or randomly change the parameter constant associated with that operation (if it is used). The *crossover* operation recombines two existing equations to form a new equation. To perform crossover, we select a random location in the genome, and copy all operation and parameter values to the left of this point from the first parent and remaining operations and parameters to the right from the second parent.

2.2 Equation Complexity

We define the *complexity* of an equation to be the number of nodes in the equation's binary parse tree. More complex equations are more difficult to find because the evolutionary search must build and optimize a larger solution.

Past results show that the performance of symbolic regression depends heavily on the complexity of the exact target equation [10-12]. Therefore, we consider the complexity of the problems in our experiments and how the performances of different methods change as target complexity increases.

2.3 Convergence

We define convergence in symbolic regression as when the evolutionary search identifies the exact target solution as the top ranked solution in the population without overfitting.

We test for convergence when generating our final results using a cross validation dataset. The validation dataset has a much wider range of input values than the training dataset used for fitness calculations. This helps distinguish between overfit solutions and exact fits. If the error on the wider cross validation dataset is near zero, we consider the equation to be converged.

The concept of convergence assumes that there is an exact and general equation underlying the system producing the experimental data. There may be cases however, where no underlying equation exists.

3. OUR APPROACH

There are many different forms of expert knowledge and ways of incorporating it into an evolutionary search [1, 2]. Here, we consider one general form of prior knowledge where we have a prior solution to a simpler problem, or an approximate solution to a more complex problem.

We consider six different policies for using a prior approximate solution: seeding the population with the full solution, seeding with random shuffles of the full solution, a mutation operator for injecting building-blocks of the approximate solution into the population, and finally, using either randomized or optimized parameters for each of these methods (see Fig. 1).

3.1 No seeding

In the no seeding case, we use and ordinary evolutionary search with a completely random initial population and operators. Variance is introduced solely through random mutation and crossover, and individuals are selected based only on their fitness.

3.2 Approximate Equation Seed

In the approximate equation seed, we introduce exact copies of the approximate into the initial population. Only a few equations

	Optimized Coefficients	Randomized Coefficients
Whole Equation Seed	$f(\boldsymbol{x}) = 1.3335\ \boldsymbol{x}^2 \cos(\boldsymbol{x} - 0.4232)$	$f(\boldsymbol{x}) = \alpha\ \boldsymbol{x}^2 \cos(\boldsymbol{x} - \beta)$
Shuffled Equation Seed	$f(\boldsymbol{x}) = (x - 0.4232) \cos(1.3335\ \boldsymbol{x}^2)$	$f(\boldsymbol{x}) = (\boldsymbol{x} - \alpha) \cos(\beta\ \boldsymbol{x}^2)$
Building Block Seed	$(\boldsymbol{x} - 0.4232)$, $\cos(1.3335\ \boldsymbol{x}^2)$, $1.0\ \boldsymbol{x}^2$	$(\boldsymbol{x} - \alpha)$, $\cos(\alpha\ \boldsymbol{x}^2)$, $\alpha\ \boldsymbol{x}^2$

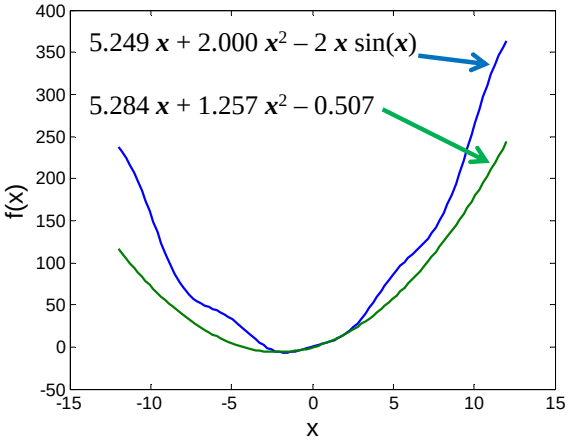


Figure 1. Example seed equations for each method (left) and an example randomly generated target equation plotted next to the automatically generated approximate equation (right).

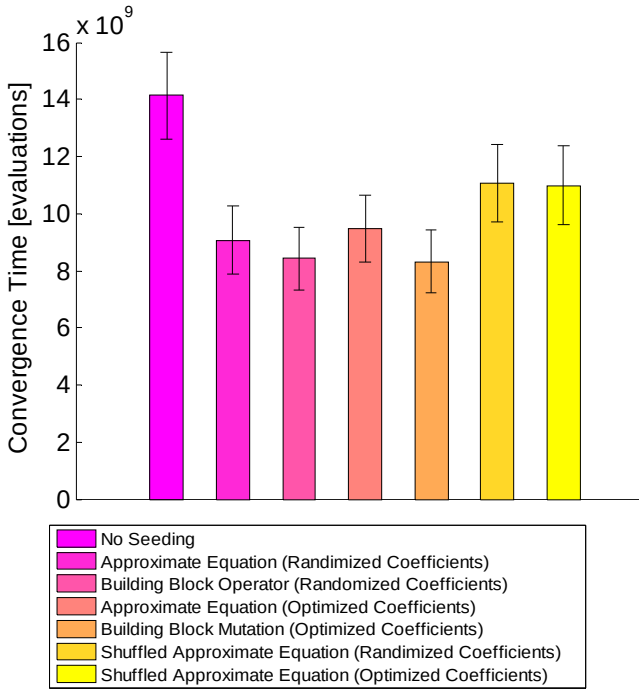


Figure 2. The expected time for the evolutionary search to converge to the exact target equation for each seeding method measured in function evaluations (does not include runs that did not converge). Error bars show the standard error.

are seeded to maintain the initial population diversity. In our experiments, we introduce one approximate equation copy for every 10 random initial solutions.

This is the most straight-forward method for using a prior model. The idea is that evolution will use the seeded equations if it likes and will adapt it to the exact model of the system.

There is a potential danger to this method however, in that the approximate solution may trap the evolutionary search in local optima; particularly if the seeded equation is a local optima itself. In the worst case, the evolution fixates on the seeded solution, losing diversity, and is unable to improve upon it.

3.3 Shuffled Equation

Instead of seeding with the exact approximate equation, we could instead introduce slightly randomized and rearranged version of the approximation. In the case where the approximate equation is a local optima solution, randomly shuffling its sub-expressions would effectively produce random solutions; but random solutions composed of the same parts of the approximate solution.

Random shuffles of the approximate solution should have roughly the same fitness distribution as ordinary random solutions, but will still introduce all parts of the approximate equation into the initial population. The idea is that the evolutionary search can recompose these shuffled solutions if beneficial, but will not be immediately placed into a local optima solution.

We implement the shuffling by performing two random shuffles of the approximate solution (or until the fitness changes since shuffles could be neutral). A shuffle consists of picking two random sub-trees of the equation's binary parse tree, and exchanging them.

Though shuffled equations are less likely to push the evolution into local optima, it may not be the best use of the approximate equation. The random shuffles could destroy important parts of the solution, or may be deleterious to the other shuffled components making them difficult to evolve from.

3.4 Building Block Mutation

The third method we consider is injecting only individual parts of the approximate equation into the population. We call these parts the building blocks defined by the approximate equation.

We define the set of building blocks for a particular equation to be all sub-trees (sub-expressions) of the equation's binary parse tree.

We define a new type of mutation operator using the set of building blocks defined by the approximate equation. In addition to typical genetic programming mutations, the algorithm can now replace a sub-expression with one of the building blocks at random.

The idea behind this method is that it may be easier to reuse individual pieces of an expert model rather than adapt the entire equation at once. This operation provides a more granular method for the evolutionary search to pick and choose the useful components of the approximate equation.

One possible danger of this approach is it could produce more bloated solutions, thereby inhibiting finding a general and parsimonious solution.

3.5 Parameter Constants

Finally, for each of the three seeding methods, we can choose to keep the exact coefficient values used in the approximate solution, or randomize these coefficients.

Randomizing the coefficients is one way to deter or delay the possibility of falling into a local optima based on the seed, such as in the whole equation and shuffled equation seed.

The downside of randomizing the constants, however, is that the evolutionary search must always refit them if used. So, randomizing the parameter coefficients does discard some of the prior information contained in the approximate equation.

4. EXPERIMENTS

4.1 Generating Random Problems

We used randomly generated problems to evaluate the performance of each seeding method. While random equations do not always resemble real-world applications of genetic programming and symbolic regression, they do provide a base or average case for comparison. Additionally, we can vary and control the complexity of the equations and effectively the difficulty of the evolutionary search.

For symbolic regression, we can produce a random equation in the same way we generate initially random population of equations. We generated one-dimensional equations and then sampled them over the range [-2,2] to produce synthetic experimental data as would ordinarily be used with symbolic regression. Additionally, we generated a larger test data set over the range [-10,10]. We use this data set for reporting the performance and convergence rates of each method in our results.

We generated 100 random symbolic equations and corresponding datasets. We then ran each method on the same random problems ten times for each equation.

We also generated the random target equations such that their complexities were evenly distributed. We measure the complexity of an equation as the number of nodes in its binary parse tree. We also perform symbolic simplification of the equation before hand so that redundant or cancelling terms do not exaggerate the complexity measure.

The random target functions are then evenly distributed between complexities 5 to 35 (or 5 to 35 nodes). Therefore, each seeding method evolves to solve each complexity of target equation approximately 30 times.

4.2 Expert Knowledge in Random Problems

We are using random target equations to generate random problems for testing our seeding methods. Therefore, we need a method for producing expert knowledge for each randomly generated problem. Since we are generating the random problems with a random equation, we know the exact solution to each problem. This allows us to generate approximate models that are equivalent to an expert-derived approximate model, or perhaps an expert derived model of a slightly simpler problem.

We generate the expert knowledge model based on the randomly generated target equation. We first want to approximate this equation so that we aren't giving the exact answer for every problem. To do this we take a randomly generated target equation, and select a random sub-expression that contains at least one operation and is not a leaf node. We then set this sub-expression equal to a random constant.

This creates a simpler and distorted version of the exact target function; however, the output of this function may be drastically different. To be considered an expert knowledge equation, the equation should at least also mimic the general features in the output of the exact target equation.

To mimic an expert derived approximation, we take this simpler equation and refit all of its parameters via nonlinear regression so that it fits the more complex target equation as closely as possible.

The end result is a simpler, but useful approximate model that resembles the target equation that should still have a good initial fitness during evolution. An example is plotted in Fig. 1. This equation still contains much of the exact structure of the target equation, and is potentially useful for the evolutionary search.

4.3 Experimental Setup

We used the fitness prediction symbolic regression algorithm described in [10-12] to search the space of symbolic equations. We use the deterministic crowding selection method [13], with 1% mutation probability and 75% crossover probability. The encoding is an operation list acyclic graph with a maximum of 64 operations/nodes [8]. Single-point crossover exchanges operations in the genome at a random split. The operation set contains addition, subtraction, multiply, sine, and cosine operations.

We limit the size of the equation to narrow our search to human-interpretable equations (equations we could fit on a piece of paper). We allowed a maximum of 64 nodes, each possibly representing five types of mathematical operations, two to four variables, or a parameter constant. Ignoring the infinite parameter space, we are effectively searching a space of roughly 10^{54} parameterized genotypes.

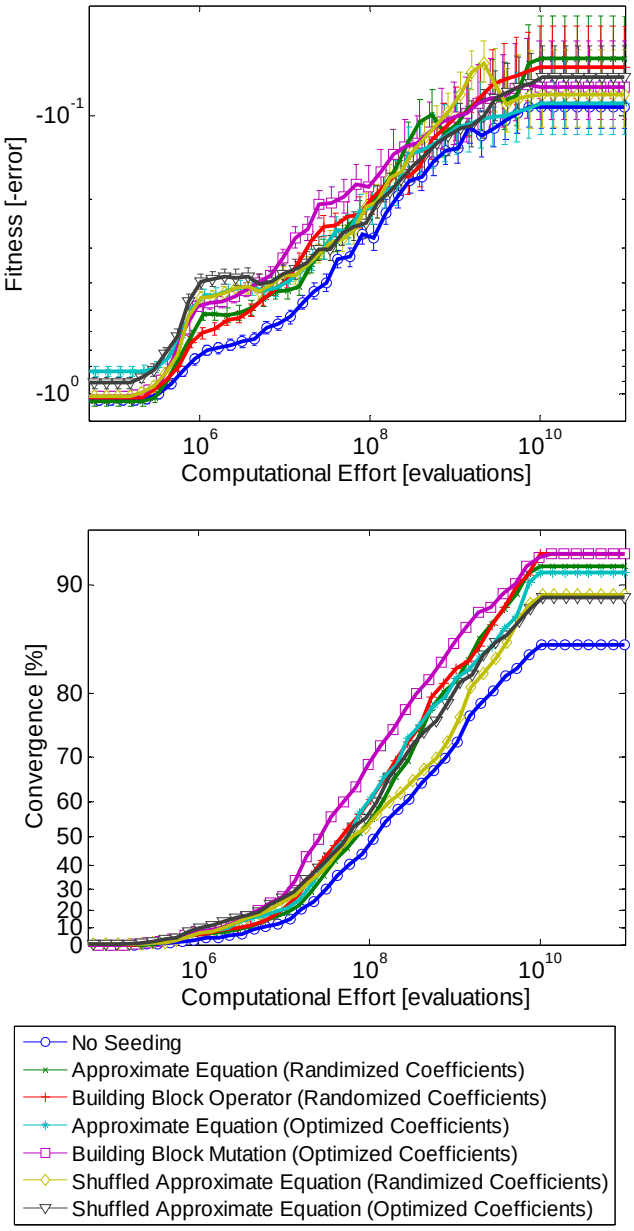


Figure 3. The mean fitness (top) and convergence rate (bottom) for each method measured over each evolutionary trial. Error bars show the standard error.

The fitness predictor population contains 1280 predictors, distributed over 80 cores. The fitness predictor subset size is 128 indices to the full training data set. Predictors are also evolved using deterministic crowding, but with 10% mutation and 50% crossover. For predicted fitness values, we only calculate over the smaller subset of the fitness predictor rather than the entire data set.

5. RESULTS

We executed 1000 trials per seeding method over 100 randomly generated target equations. We tracked the best solution in each

generation, measuring its fitness, convergence, and bloat over the evolutionary run. Fitness and convergence were calculated using a withheld test dataset that spanned a larger input range than the training data set.

5.1 Time to Convergence

The time to convergence is the total computational effort for each method to find the exact target solution in the evolutionary search. Fig. 2 compares the convergence time for each seeding method, averaged over all target equations and evolutionary runs.

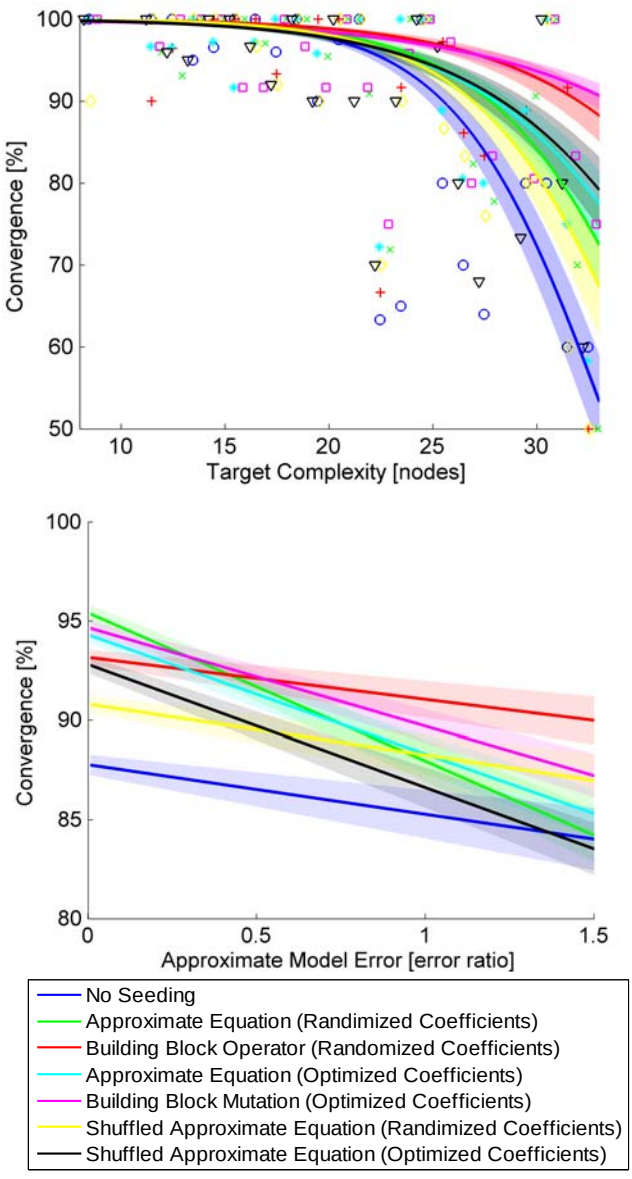


Figure 4. The logistic trends of each seeding method in convergence rate versus target equation complexity (top), and linear trends in convergence versus the error of the approximate seed equation from the target equation (bottom). Error bars show the range based on the standard errors of the trend fit parameters.

Time to convergence measures only the runs that did indeed converge. Therefore, it is a measure of the best cases for each method; comparing, potentially, how much the evolution can be sped up with each seeding method. It is important to note however that fast convergence is not always good; but, it is a measure of the evolvability.

The ordinary evolutionary runs without seeding were the slowest to converge (Fig. 2) on average. This suggests that all of the seeding methods can speed up the convergence. The next slowest are the shuffled equation seeding methods. This indicates that evolving the randomly shuffled seed equations is the most difficult, but still faster than no seeding at all.

The fastest method to converge is the building block seeding, followed closely by the whole equation seeding. This suggests whole equations and the equation building blocks are easier to evolve than equations from scratch or randomly shuffled equation seeds.

The time to convergence appears to be invariant to using either randomized or optimized parameter constants in the seed. This is particularly interesting because randomized coefficients must always be re-learned or refit. The invariance to the coefficient method indicates that the evolvability and convergence times depend primarily on finding the structure of the equation in the average case.

5.2 Fitness over Time

We also tracked the fitness of the top ranked equation over all runs for each method on the withheld cross validation dataset.

We can see that the methods that use whole equations for seeding (the approximate equation and shuffled equation seeding methods with optimized constants) have higher initial fitness as should be expected (Fig. 3). However, these methods are overcome by the randomized versions later in evolution.

The standard error in fitness increases over time, making it difficult to discriminate between the methods. However, we can pick out some additional general trends. All seeding methods appear to strictly dominate the no seeding method. Also, the most fit solutions at the end tend to be the least fit solutions early on.

5.3 Convergence over Time

The convergence rates over time are more stable than the fitness, making it easier for comparing between each seeding method. The convergence rate shows the percentage of runs that found the exact target solution versus the time (or computational effort) into the evolutionary run (Fig. 3).

All runs start with zero convergence and increase gradually on a sigmoid trend to their maximum convergence performance.

Again, all seeding methods dominate the ordinary non-seeding method. The next worst is the shuffled approximate equations. The highest convergence methods are the building block and equation seed methods.

The building block seeding method with optimized constants stands out the most in Fig. 3. It converges the soonest, and is tied for the highest convergence rate at the end of each trial with the building block seeding with randomized constants. This suggests that the optimized constants can improve convergence early on, but may not help in the long run.

5.4 Scaling with Complexity

So far we have only looked at the average performances of each seeding method over all equations. However, the impact of seeding may depend on the different traits of the target functions. Here we break down the performance of each method based on the complexity of the target equation.

Breaking the performances up by the target equation complexity makes the performance trends noisier. Therefore, we use a trend fit to help visualize the differences between each method.

For the convergence versus the target equation complexity, we fit a sigmoid trend curve to each method (Fig. 4). A sigmoid trend is appropriate for this data since the convergence rate ranges between 0 and 100% depending on the problem difficulty (such as complexity). The sigmoid trend curve has two parameters, the origin slope and the origin offset, making it a low variance trend model.

Based on the sigmoid trends, we want to see which methods drop off in convergence the latest with increasingly complex target equations.

Shown in Fig. 4, the non-seeding method drops of the fastest. The best performing methods are the building block seeding methods. The remaining methods fall in-between. This result suggests that building block seeding scales the best with the problem complexity; solving the most complex problems more reliably on average.

5.5 Scaling with Seed Equation Error

Next, we look at the convergence rates plotted against the error of the approximate model that is used for seeding. We can view this as the dependence on the confidence or quality of our expert knowledge equation – for example, how does the performance vary between highly accurate approximate seed equations and inaccurate seed equations.

For this data we fit a linear trend to help visualize the differences between each method (Fig. 4). This is the most appropriate trend to fit because the seeding equation error does have a dominating influence over the convergence rates. So, we can only pick out the local general trends.

We first notice that there are three methods that appear to be invariant to the approximate model error: no seeding, shuffled equation with random coefficients, and building block with random coefficients (Fig. 4). This is not surprising for the non-seeding method since it does not use the seed equation. The performance of the other two has the same slope, but higher convergence.

This suggests that the randomized coefficient building block seed and shuffled equation can still use parts of the seeding equation even when it is a poor approximation.

It is interesting to note that even the non-seeding method has a slight decreasing trend in convergence with the error of the seed equation, despite not using the seed equation. We generate the seed equation by approximating the exact equation. Therefore, there is a secondary trend in this figure, which is the target equation's sensitivity to approximations. An equation that is difficult to approximate accurately may contain more complex features, thereby making the target equation more difficult to fit in general.

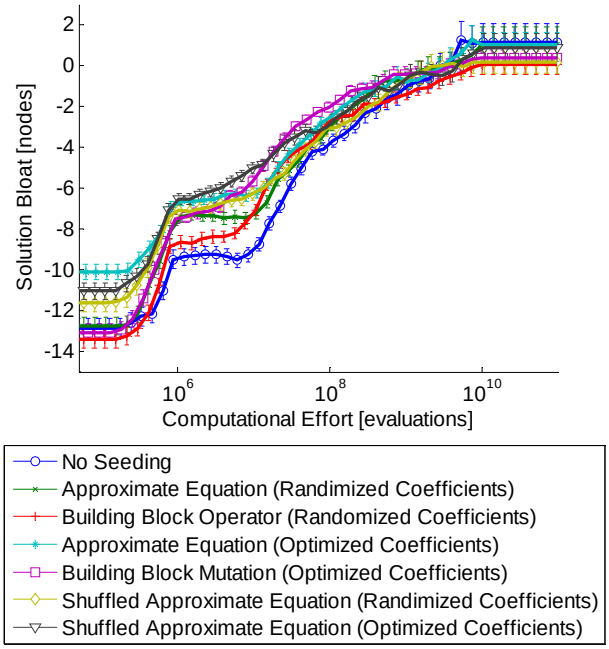


Figure 5. The solution bloat of the top ranked solution over the evolutionary runs. Bloat is measured as the top ranked equation's complexity minus the target equation complexity. Error bars show the standard error.

5.6 Solution Bloat over Time

Finally, we examined the bloat of the top ranked solution of each method in each evolutionary run. We define the bloat as the complexity of the equation (the number of nodes in the equations binary parse tree) minus the complexity of the target equation. Equations with positive bloat are larger than they need to be while negative bloat means the equation is too small.

We can see that the whole equation seeding methods start off with higher bloat on average (Fig. 5). This means the seed equations tend to be more complex than the average randomly generated equations. However, all methods converge in complexity toward the target equation complexity over time.

Overall, none of the methods experienced an excessive amount of bloat over time. However, we can pick out some general trends.

The non-seeding method has the most bloated solutions, and the highest variance in bloat – particularly near the end of the evolution.

The building block seeding, in comparison, has the least amount of bloat. This is surprising because the mutation operator with the building blocks provides a means to create additional bloat. Therefore, we suspect that the benefits of the seeding itself dominate this metric, resulting in more exact results on the target solution.

6. CONCLUSIONS

We have explored the effects of incorporating expert knowledge into evolutionary search. We considered a general expert knowledge case, where the expert knowledge consists of an approximate solution or a related solution to the problem at hand. We investigated six seeding methods for utilizing this type of

prior expert knowledge: seeding with the whole solution, the randomly shuffled solution, pieces of the solution, and using random or optimized parameter coefficients in each of these three methods.

Our results show that each seeding method can substantially improve the convergence and fitness performance over not seeding. However, different methods scaled differently based on the different traits of the target function.

We found that the building block mutation seeding method converged the fastest among all methods and achieved the highest convergence rates on average for all problems. It also maintained the highest convergence rates for the most complex target equations, and was the most invariant to the error and quality of the seeding equation.

We also found that the seeding methods that used whole equations (no seeding, whole equation seeding, and shuffled equation seeding) scaled the worst with the target equation complexity. Additionally, the methods that used the optimized parameters (rather than randomized parameters) of the seed equation were the most sensitive to decreasing quality and accuracy of the seeding equation.

While many other possible types of expert knowledge may exist for genetic programming, we conclude that in the case of seeding with a prior solution, it is best to seed with the building blocks of the prior solution, and to randomize the parameter coefficients before seeding.

7. ACKNOWLEDGMENTS

This research is supported by the U.S. National Science Foundation Graduate Research Fellowship Program, and U.S. National Science Foundation Grant CMMI 0547376.

8. REFERENCES

[1] J. Moore and B. White, "Exploiting Expert Knowledge in Genetic Programming for Genome-Wide Genetic Analysis," in *Parallel Problem Solving from Nature - PPSN IX*, 2006, pp. 969-977.

[2] S. G. Casey, C. W. Bill, and H. M. Jason, "Using expert knowledge in initialization for genome-wide analysis of epistasis using genetic programming," in *Proceedings of the 10th annual conference on Genetic and evolutionary computation* Atlanta, GA, USA: ACM, 2008.

[3] W. Banzhaf and J. Miller, "The Challenge of Complexity," in *Frontiers of Evolutionary Computation*, 2004, pp. 243-260.

[4] A.-T. Mohammad-Reza and J. Mohammad, "Incorporating A-Priori Expert Knowledge in Genetic Algorithms," in *Proceedings of the 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation*: IEEE Computer Society, 1997.

[5] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.

[6] M. Ben, J. W. Mark, and W. B. Geoffrey, "Using a Tree Structured Genetic Algorithm to Perform Symbolic Regression," in *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*. vol. 414, A. M. S. Zalzala, Ed. Sheffield, UK: IEE London, UK, 1995, pp. 487--492.

[7] D. Edwin and B. P. Jordan, "Multi-Objective Methods for Tree Size Control," in *Genetic Programming and Evolvable Machines*. vol. 4, 2003, pp. 211--233.

[8] M. Schmidt and H. Lipson, "Comparison of tree and graph encodings as function of problem complexity," in *Proceedings of the Genetic and Evolutionary Computation Conference*, London, 2007, pp. 1674--1679.

[9] X. H. Nguyen, R. I. McKay, and D. L. Essam, "Solving the Symbolic Regression Problem with Tree-Adjunct Grammar Guided Genetic Programming: The Comparative Results," in *The Australian Journal of Intelligent Information Processing Systems*. vol. 7, 2001, pp. 114--121.

[10] M. D. Schmidt and H. Lipson, "Coevolution of Fitness Maximizers and Fitness Predictors" in *Proceedings of the Genetic and Evolutionary Computation Conference, Late Breaking Paper*, 2005.

[11] M. D. Schmidt and H. Lipson, "Co-evolving Fitness Predictors for Accelerating and Reducing Evaluations," in *Genetic Programming Theory and Practice IV*. vol. 5, L. R. Rick, S. Terence, and W. Bill, Eds. Ann Arbor: Springer, 2006, pp. 113-130.

[12] M. D. Schmidt and H. Lipson, "Coevolution of Fitness Predictors," *IEEE Transactions on Evolutionary Computation,* vol. 12, pp. 736-749, Dec 2008.

[13] S. W. Mahfoud, "Niching methods for genetic algorithms," University of Illinois at Urbana-Champaign, 1995.