Prioritized Grammar Enumeration: Symbolic Regression by Dynamic Programming

Tony Worm
Binghamton University
worm@binghamton.edu

Kenneth Chiu
Binghamton University
kchiu@cs.binghamton.edu

ABSTRACT

We introduce Prioritized Grammar Enumeration (PGE), a deterministic Symbolic Regression (SR) algorithm using dynamic programming techniques. PGE maintains the tree-based representation and Pareto non-dominated sorting from Genetic Programming (GP), but replaces genetic operators and random number use with grammar production rules and systematic choices. PGE uses non-linear regression and abstract parameters to fit the coefficients of an equation, effectively separating the exploration for form, from the optimization of a form. Memoization enables PGE to evaluate each point of the search space only once, and a Pareto Priority Queue provides direction to the search. Sorting and simplification algorithms are used to transform candidate expressions into a canonical form, reducing the size of the search space. Our results show that PGE performs well on 22 benchmarks from the SR literature, returning exact formulas in many cases. As a deterministic algorithm, PGE offers reliability and reproducibility of results, a key aspect to any system used by scientists at large. We believe PGE is a capable SR implementation, following an alternative perspective we hope leads the community to new ideas.

Categories and Subject Descriptors

I.2.8 [**Problem Solving, Control Methods, and Search**]: [Dynamic Programming, Graph and tree search strategies]; G.2.1 [**Combinatorics**]: [Generating functions]; I.6.5 [**Model Development**]: [Modeling methodologies]

General Terms

Algorithms, Performance, Reliability

Keywords

Symbolic Regression, Genetic Programming, Representation, Grammar Enumeration

1. INTRODUCTION

Symbolic Regression (SR) is a subproblem of Genetic Programming (GP) which evolves equations to model observational data [19]. Despite two decades of research, SR still has not become a mainstream technology the way linear and non-linear regression have. GP, as the de facto algorithm for SR, suffers from two limitations. First, state-of-the-art GP implementations often fail to return the original formula from which the input data was generated [17]. Second, the results returned are inconsistent and difficult to reproduce. A user who is not an expert in GP will not likely trust an algorithm which cannot reliably reproduce the same results with each invocation and leaves them to decide whether or not to keep the results. GP's difficulties stem from its non-deterministic behavior and use of random number generation. To become a technology, SR will have to overcome the issues brought on by non-determinism while reliably returning correct answers to known problems.

This work introduces Prioritized Grammar Enumeration (PGE), a deterministic SR algorithm. PGE reconsiders the SR problem as an exploration within an abstract grammar. Working with a grammar's production rules, PGE prioritizes the enumeration of expressions in that language. Starting from simple basis functions, PGE searches by expanding the best candidates into increasingly complex equations. Candidates are selected from a Pareto Priority Queue to balance the trade-off between accuracy and parsimony. New equations are generated from the current best through recursive application of the grammar's production rules, resulting in all possible offspring within one production rule application. PGE extends the SR search tree without the use of any random number generation.

PGE drastically reduces the size of the search space, avoiding invalid expressions and merging isomorphic representations with simplification and partial ordering. It memoizes equations as it searches and discards equations which have been previously encountered. PGE fits and evaluates each unique equations once, fully optimizing each equation form the first time it is discovered. By fitting abstract parameters (placeholder coefficients without value) using non-linear regression, PGE finds the 'best' version of an equation. This effectively separates the search for form, from the optimization of that form.

PGE's foundations in dynamic programming and deterministic execution result in an algorithm which will traverse the same search path when given the same inputs. In doing so, PGE offers exactly reproducible results with the reliability of getting those results. We demonstrate PGE's capabilities on 22 benchmarks documented in[24]. PGE shows good accuracy and returns exact expressions in many cases. We postulate that PGE, in conjunction with standardized benchmark problems, can be a benchmark against which SR implementations can measure themselves.

2. RELATED WORK

Since Koza's initial formulation of GP and SR [19], there has been a plethora of research from generalized implementation enhancements to improvements addressing the issues of disruptiveness of crossover, bloat, population diversity, and premature convergence among many others. We cover material which has relevance to our discussion, though it will necessarily be abridged due to space limitations.

Selection is arguably the most important aspect of GP, determining which solutions continue contributing to the search and which results will ultimately be returned. The Pareto non-dominated sort, or Pareto front, balances the trade-off between opposing goals in multi-objective optimization [5, 9, 33, 21, 30]. Various methods improve the diversity along the Pareto front, and/or maintain archives of good solutions: NSGA-II [2], SPEA2 [35], SPEA2+ [12], and PESA-II [1]. These algorithms all aim to improve the quality of solutions in a population. Age Layered Population Structure (ALPS) [10] partition the population by age to restrict competition and breeding interactions, combating premature convergence.

At the individual genome level, changes to representation, crossover, and coefficient optimization have been explored. Tree-based representation is standard, others include linear, grammar-based [25], and graph or Cartesian GP [27]. Invalid mathematical operations are removed in [11, 18] and restricted by the grammar in TAG3P [8, 7, 6]. Additionally, TAG3P only allows crossover points to be of the same s-expression type. Context-aware crossover [22] selects a snip in one tree and substitutes it at all valid locations in the other parent. [15] extends this to include all possible valid snip replacements from both parents. Semantically Aware Crossover (SAC) [28] biases crossover to exchange semantically different material and Semantic Similarity-based Crossover (SSC) [32] extends this, by limiting the size of material to small, manageable snips. Real-valued coefficient optimization, a known difficulty for GP, has been improved by local gradient search, non-linear regression, and swarm intelligence [31, 29, 3].

Abstract Expression Grammar (AEG) [16], uses several concurrent searches with state-of-the-art GP implementations. AEG replaces functions, variables, and coefficients with abstract place-holders. These place-holders enable different optimization methods to focus on restricted subsections of the search space or parts of an expression. Parameters and restrictions to a SR method are encapsulated into a SQL-like language, providing high level specifications and fine-grained search control. Opening book rules allow each island to be tailored to search a reduced area of the search space, while closing book rules enable AEG to update the constraints on islands which have stagnated. The author states that designing closing book rules is time consuming, were arrived at empirically, and often need to be tailored to the SR system and problem at hand. AEG was shown to make many problems tractable with current techniques [17]. Additionally, almost any SR algorithm can be used within the AEG framework.

On a new front, Fast Function eXtraction (FFX) [23] is a recently purposed SR implementation which does not use genetic operators or a tree based representation. Instead, FFX uses a Generalized Linear Model (GLM) of the form:

$$y = F(\vec{x}, \vec{w}) = \sum_{b}^{B} w_b f_b(\vec{x})$$

FFX learns a linear combination of b basis functions, from $1 \rightarrow B$, by applying pathwise regularized learning. Basis functions are incrementally included and fit until the number of functional bases equals the desired model complexity. The GLM is linear in coefficients, w.r.t. the terms of the summation, though trigonometric, logarithmic, and other non-linear functions are permitted. FFX is deterministic, making no use of random number generation and is also computationally efficient. FFX, however, suffers from two significant limitations: (1) there are no coefficients or parameters within the bases, meaning more difficult, non-linear relationships are beyond its abilities. This issue could be addressed by using non-linear regression and abstract coefficients. (2) individual terms of the summation are limited in complexity to a pair-wise combination of uni-variate and bi-variate bases determined at initialization. Seeding with increased basis functions could become prohibitive as the number of terms grows through pair-wise basis combinations. In the 13 variable example provided, the initial number of GLM basis functions was 7100.

Candidate fitness metrics, methods for comparing implementations, and benchmark problems vary widely across the GP field. Last year, [24] surveyed three years of literature from EuroGP and GECCO GP track, bringing this issue to the forefront of the community. Their aim was to start a discussion on unifying and standardizing the evaluation process in GP. We agree with these ideals and use 22 of their SR target functions for the evaluation of PGE. Further, we believe PGE can contribute to this effort, as a deterministic, base-line algorithm, against which evolutionary methods can measure themselves. We do, however, disagree with the assumption in [24], that results should not be expected to be repeatable, and thus unverifiable by a third party. A non-GP practitioner will not likely use a tool which gives different answers each time it is used. This has been partially addressed by *rate of convergence* (how often an implementation finds an answer) and *cumulative probability of success* (the prob-

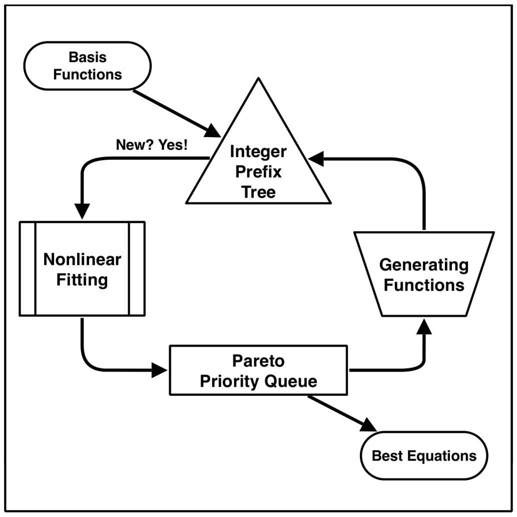

Figure 1: The PGE Search Process

Figure 2: Grammar for Mathematical Equations

```
START  →  E
E  →  E + T  |  E ∗ T  |  T
T  →  T − N  |  T / N  |  N
N  →  Cos(E)  |  Sin(E)  |  Tan(E)  |
       Log(E)  |  Exp(E)  |  Sqrt(E)  |  L
L  →  (E)  |  −(E)  |  (E)^(E)  |  TERM
TERM  →  Constant  |  Variable
```

ability that an ideal solution would be found on or prior to generation i). Both of these methods require many trials. Nevertheless, we agree that the optimum is less obtainable and that a consensus needs to be reached on unbiased methods for comparison between different implementations.

3. PRIORITIZED GRAMMAR ENUMERATION

Prioritized Grammar Enumeration (PGE) is a deterministic, dynamic programming algorithm for SR. Being largely influenced by GP, PGE maintains the parse tree representation for equations and incorporates the Pareto non-dominated sort. PGE diverges in how it organizes, explores, and processes the search space. PGE replaces the genetic operators with grammar production rules and random selections with well defined choices. In doing so, PGE becomes an algorithm offering reliability and reproducibility that a non-deterministic implementation cannot.

Figure 1 shows a pictorial representation of the PGE algorithm. It is presented now to serve as a reference point as the details are unveiled. Psuedocode is provided, at the end of the section, after the necessary material has been covered.

3.1 Organizing the Search Space

Figure 2 is a grammar for mathematical equations and the one we use in this work. A grammar, such as this one, defines the search space which a SR implementation will explore. The terminals, non-terminals, and production rules form the building blocks from which valid equations may be constructed. Equations are composed of these basic components to form the 'DNA' of an expression, a functional genome that mirrors the parse tree for an equation. Operators (+ - * / sin exp etc.) are the internal nodes to the tree. Operands are the terminals or leaves representing state variables, constants, and real-valued numbers. PGE, like AEG, uses indexed coefficients, enabling these abstract parameters to be optimized separately from the search for form.

PGE uses n-ary trees, where operands can have a variable number of children. In the n-ary tree representation, the associative operators can have n sub-trees, flattening and reducing the tree's size. This is a slight modification from the usual binary tree; only affecting the associative operators addition and multiplication. The n-ary tree does not change the modeling ability of the equation, but will effect the trade-off between parsimony and accuracy. This in turn effects the selection operation of any SR implementation, though we do not investigate this issue here.

In addition to the reductions in parse tree size, the n-ary representation eases the design of sorting and simplification algorithms. These algorithms, detailed next, work within the parse tree to reshape equivalent trees into a canonical form. This effectively merges isomorphic equations, reducing the size of the search space, while simultaneously adding structure to the search space.

Reducing the Search Space

The size of the space defined by a grammar is infinite, even when disregarding the increase brought by considering real valued coefficients. This is the result of a grammar's production rules being applicable recursively and indefinitely. Adding to the multiplicity, an equation will usually have several derivations for each parse tree and isomorphs through manipulations with basic algebra techniques. Consider the equation $a \cdot b \cdot c$ (Figure 3). This equation has 12 different binary tree representations, from six leaf permutations and two shape permutations. If we use both addition and multiplication, this number of trees is 48.

As the number of operations, operands, and tree complexity are increased, the size of the search space undergoes a combinatorial explosion. To consolidate this space, PGE imposes operator restrictions, simplifies equations, and partially orders sub-expressions. When combined, these methods only allow syntactically valid equations to be considered, merge isomorphic equations into a canonical form, reduce the overall size of the search space, and create a structural ordering to the search space.

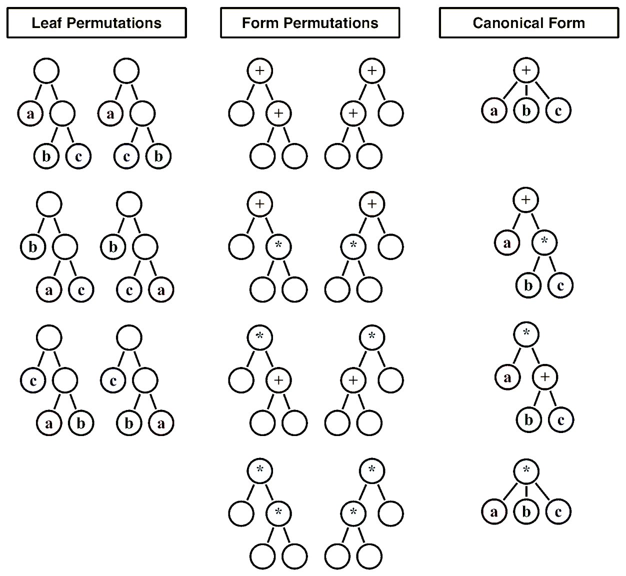

Figure 3: Equation Tree Permutations

Operator restrictions, or interval arithmetic [11], disallow invalid mathematics such as dividing by zero or taking the logarithm of a negative number. In general, we wish to only consider valid equations, however, there may be instances when an invalid equation is an intermediary to a valid equation ($log(x) \rightarrow log(abs(x))$). Though this issues is beyond the scope of this paper, it deserves further investigation.

Simplifications group like terms together $(x + x)$, replace sub-expressions which evaluate to a constant value ($\sin \pi$, $\frac{x}{x}$, or $x - x$), and reduce unnecessarily complex expressions such as $\frac{1}{\frac{1}{x}}$. The resulting equations are equivalent, simpler forms. There is debate as to how simplification effects the SR process [14, 13, 26]. Certainly, changing the tree effects the Pareto trade-off which in turn has consequences on selection and therefore search progress. Questions still remain as to how much and which simplifications should be applied, which forms of the equation should be kept, and the overall effects simplification has on the search process.

Despite these questions, PGE still employs the use of the aforementioned simplifications. This is done to eliminate those sub-expressions which unnecessarily complicate equation and the overall search space. If they were not replaced, many equivalent variations of a good equation could crowd out other potential solutions, as they would be selected more often. This would result in the algorithm stagnating in a local optima of the search space. Simplifications effectively trim out parts of the search space which are redundant, and also return results which are more comprehensible.

Partial ordering of sub-expressions provides the necessary machinery for comparing and thus sorting terms of associative operators. In PGE, partial ordering is created by placing a relative order on each building block type. Terminals are less than non-terminals, variables are less than constants, and unary functions are less than binary functions. Since addition and multiplication are the only associative operators, they are the only building blocks affected by sorting of sub-expressions. Figure 3, right column, shows the canonical form of $a \cdot b \cdot c$ with a n-ary tree and sorting. If $a < b < c$, then there is only one representation for each of the expressions, reducing the original 48 to just 4. The savings created by sorting become larger as equations become more complex.

Partial ordering, coupled with the n-ary tree representation and simplifications, yields many-fold reductions of the search space. Invalid and ineffectual expressions are removed, variations of associative forms are limited, and isomorphs are combined, shrinking the number of representable equations that need to be explored by a SR implementation. PGE takes reductionism one step further, fully optimizing an equation in the search space the first time it is encountered and remembering which equations it has discovered thus far. These features enable PGE to discard equations it has already seen and is described next.

3.2 Evaluating Forms Once

At its core, PGE is a dynamic programming algorithm which aims to evaluate each sub-problem once. In PGE and SR, a sub-problem is a particular equation form, namely the parse tree and its parameters or coefficients. The key to evaluating forms once is to fully optimize a form the first time it is seen and to remember the forms which have already been explored. PGE optimizes forms by fitting abstract parameters with non-linear regression and by recording which equations it has already seen with a lookup trie.

Non-Linear Regression

Non-linear regression is not unique to PGE, or GP for that matter, but it is a central component of PGE's ability to evaluate an equation form once. As PGE encounters new equation forms, it fits the abstract parameters using non-linear regression, resulting in the 'best' version of that form on the training data. PGE uses the Levmar C library implementation of the Levenberg-Marquardt optimization algorithm [20]. The analytical Jacobian version is used by symbolically calculating the derivations of an equation w.r.t. each coefficient. In cases were an equation is linear to the coefficients, the Levmar library returns in one iteration by using singular value decomposition (SVD). Training performance is used as the metric for fitness comparison, along with equation size, as is usual when using Pareto fronts. The fully optimized form is later evaluated on unseen testing data to provide an unbiased measure of accuracy.

PGE's treatment of coefficients is in direct contrast to the probabilistic optimization GP uses through its genetic operators. Abstract parameters and non-linear regression enables the separation of search for equation form, from optimization of a form's parameters. This separation, in turn, enables an equation form to be fully evaluated once, removing duplication of effort that GP experiences when using genetic operators to optimizing the coefficients of an equation form.

Memoization of Equation Forms

In PGE, a sub-problem is equivalent to a particular equation form. Sub-problems are encountered when an equation has several parse derivations and isomorphs to an equation exist through algebraic manipulations. This means there are multiple orderings of the production rules which result in the same equation and that each equation may appear in more than one form, as several equivalent points within a grammar's representable space.

Detecting previously encountered equations is the key to the dynamic programming approach taken by PGE. The memoization of form allows PGE to consider a form just once. PGE matches equations by comparing the serialized representation of the equations. Serialization transforms an equation into a sequence of integers by assigning a unique value to each node type. The resulting integer sequence is equivalent to the prefix notation of the equation. Also, since PGE uses abstract coefficients, they are all converted to the same value. This means PGE only memoizes their existence and location.

PGE uses a trie structure, implemented as an integer prefix tree (IPT), for the lookup table of currently explored equations. An equation serial is the key that the IPT uses to return a boolean value, indicating if the respective equation is new or not. The IPT was inspired by the suffix tree algorithm for string matching [34, 4]. The suffix tree algorithm gives log-time lookup for string existence in a piece of text. Similarly, the IPT provides log-time determination, w.r.t. the size of the equation alphabet (terminals & non-terminals), and linear-time to the length of the serial (a sequence of integers), as to whether an equation has been encountered before. If the IPT is reasonably balanced, then it has log-time w.r.t. the number of equations encountered thus far, with worst case being linear-time.

Listing 4 provides the psuedocode for the IPT. The IPT is a recursive structure which is iteratively built as new equations are encountered. To perform a lookup, and possible insertion, a serial S and the root *memoNode* are passed to the *Insert* function. At each recursive call, equivalently each *memoNode* of the IPT and position in the serial S, the first element of S (the current node type) is looked up in a balanced binary tree (*memoNode.next*). If the node does not exist, it implies that this serial has not been seen before, and thus, that this is a new equation. When this happens, a new *memoNode* is created and each successive recursive call will produce a new *memoNode*. Recursion continues until the serial has been completely traversed. If no new nodes have been inserted, then it means that this equation has been encountered before, and is subsequently discarded.

The IPT also tracks the number of total unique visits. This allows PGE to report the total number of unique equa-

Figure 4: Memoziation Tree

```
struct memoNode {
  int                curr_type
  map[int,memoNode]  next

  int unique
}

func Insert(S []int, N memoNode) bool {
  inserted = false
  in = N.next[S[0]]

  // does this branch exist?
  if in == nil
    in = new(memoNode)
    in.curr_type = S[0]
   N.next[S[0]] = in
    inserted = true

  // recursive call to insert
  if len(S) > 1
    inserted = Insert(S[1:],in) || inserted

  // visitation accounting
  if inserted == true
   N.unique++

  return inserted
}
```

Figure 5: PGE Expansion Functions

```
func AddTerm(E Expr) {
  E  →  E + c*TERM
  E  →  E + c*N(TERM)
}
func WidenTerm(T Expr) {
  T  →  T * TERM
  T  →  T * N(TERM)
  T  →  T / TERM
  T  →  T / N(TERM)
}
func DeepenTerm(N Expr) {
  N  →  N + c*TERM
  N  →  N + c*N(TERM)
  N  →  (c*TERM) / N
  N  →  N / (c*TERM)
}
```

tions evaluated, which is the same as the total number of equations evaluated in PGE. Due to the reductions of isomorphs to a canonical form, the reported amount of space explored by PGE is much larger in the original, unreduced search space. To our knowledge, no one has before reported the unique number of equations evaluated. The IPT is easily incorporated into almost any SR implementation, and we believe that it can improve the reporting of the extent to which the search space was explored, as well as the amount of effort expended on searching for form versus optimizing forms.

3.3 Removing Non-Determinism

Removing sources of non-determinism was a central theme to the development of PGE. The aim was to produce an algorithm which would give the same results with each invocation. To achieve this deterministic behavior, PGE makes no use of random number generation. It performs the exact same algorithmic steps given the same parameters and same training data. PGE replaces the initialization, breeding and selection mechanisms of GP. The first two are detailed here and the selection policy is described in Section 3.4.

To determine the SR starting points, GP uses methods like grow, full, and ramped half-and-half to randomly generate initial equations. In contrast, PGE initializes a search with a set of basis functions, such as $c_0 * x_i$, $c_0 + c_1 * x_i$, $\frac{c_0}{x_i}$, and $c_0 * f(x_i)$. These starting points are the simplest functions and are predetermined by the usable building blocks. Instead of growing equations at the beginning, PGE starts with simple functions, expanding them to reach new, unseen areas of the search space.

To expand a candidate equation, PGE uses generating functions derived from the grammar's production rules. Generating functions are the deterministic replacement for the non-deterministic genetic operators, crossover and mutation. Each generation function corresponds to one or more of the grammar's production rules. The generation functions are policies for how to expand and modify an equation's parse tree to obtain functions 'close' to the current one. New equations are produced by recursively applying the generating functions over the parse tree. From a single tree, this process produces a set of reachable equations within one step of the input equation. As the production rules are applied recursively, the current node's type determines the appropriate generation function(s) to apply. Listing 5 shows some example generation functions. *AddTerm* increases the number of terms in a summation, such as $aX + bY \Rightarrow aX + bY + cZ$. *WidenTerm* increases the number of terms in a multiplication, such as $aXY^2 \Rightarrow aX^2Y^2$ or $aXY^2 \Rightarrow aXY^2Z$. *DeepenTerm* increases the complexity of a term, such as $aXY \Rightarrow a(X + bZ)Y$ or $aSin(X) \Rightarrow aSin(X + Y)$.

Variations on the PGE expansion algorithm can be created by parameterizing where and which generation functions are applied to a candidate. By modifying the productions applied at each node, we can reduce or increase the set of equations generated at each node of the tree, and thus each point in the search space. In this paper, we used three different expansion methods. The basic method (PGE-1) restricts the grammar by removing productions which result in non-distributed expressions like $ax * (b + cx * (d + x^2))$. It only uses *AddTerm* and *WidenTerm* during the recursive expansion. The second method (PGE-2) adds back the previously mentioned restriction, using all three generation functions from Listing 5. This results in isomorphs which are too 'far' apart for simplification rules to consider the same. Despite being equivalent, these isomorphic forms can produce very different offspring, and represent separate areas of the search space. The third method (PGE-3) is FFX inspired, but starts with only a set of univariate bases. These bases are iteratively grown into a summation of multiplications by applying *AddTerm* and *WidenTerm*, creating candidates with more and increasingly complex bases. PGE-3 differs from the PGE-1 method by explicitly defining the root node to be addition and placing stricter limitations on the complexity of each term in the summation.

The initialization and generating functions, determine the set of reachable expressions in a SR search. As with the equations themselves, there is a trade-off between space and complexity; how wide and how deep a search can explore. The expansion methods discussed above enable PGE to remove non-determinism at the individual level. To fully remove non-determinism, and give direction to the search, the selection strategy still needs to be replaced.

3.4 Directing the Search

SR seeks to optimizes both the parsimony and accuracy of equations. PGE uses a priority queue to express which points in the search space to expand next. By incorporating the Pareto non-dominated sort into a priority queue, the PGE search can exploit and optimize the trade-off between competing objectives in a deterministic order.

The Pareto Priority Queue

The Pareto Priority Queue (PPQ) is the deterministic mechanism for controlling the search direction of PGE. The PPQ replaces selection for mating with prioritized equations for expansion (selection for survival is unnecessary since all equations are remembered). The PPQ is what its name implies, a priority queue built on the Pareto non-dominated sorting. The PPQ prioritizes expansion towards those branches of the search space which balance size and accuracy best. The PPQ removes the need for generations, survivors, and mating pools, storing all explored equations which have not been expanded. Our experimentations have not shown this to be prohibitive. Consumed memory never exceeded 500Mb, even in the face of hundreds of thousands of unique equations.

To construct the PPQ, successive Pareto frontiers are appended onto a linear structure. Thus, the smallest equation from the first Pareto front is at the head of the queue. The next elements are the remainder of the first Pareto front in increasing size. The remaining Pareto fronts follow a similar pattern. Priority is first given to the Pareto frontier and then to size within the frontier. This is the same as Pareto sorted array that results from the GP Pareto sort during selection.

During processing, PGE removes the top p equations from the PPQ in order to select the next areas to expand. By doing so, PGE selects the p smallest equations from the first Pareto frontier. This gives variation across the trade-offs for which equations to expand, exploiting multiple paths in the space simultaneously. If only the first (smallest) equation is removed, search would progress through the space by size. If p is too large, then overly complex equations are produced, over fitting the data and causing bloat to ensue. Bloat in PGE is the result of good solutions crowding the front of the PPQ. As the search progresses, extraneous material is added to candidates which has little effect on the accuracy of the equation. This has a compounding effect, whereby the extra material increases the number of expansion points of the tree, creating even more similarly accurate expressions with ineffectual sub-expressions.

PGE's selection policy is deterministic. With each independent PGE invocation, and each iteration, the same equations will be at the front of the queue. This means the same equations will be expanded, the same productions will be applied, and the same results will be returned.

Processing in PGE

In PGE, processing follows a deterministic execution path. It uses no random number generation. Given a parameter setting and training data, PGE will execute the same way every time. Further, the PGE search algorithm only has to be run once in order to obtain conclusive results. This is an advantage over GP implementations, which are run multiple times to produce statistically significant results for the number of successful trials. PGE has no analogue to the percentage of successful trials that GP has.

The PGE search proceeds as follows. Initialization generates a set of basis functions for each variable. These basis functions are then memoized by the IPT, fit to the training data, evaluated on the testing data, and pushed into the PPQ. The main PGE loop iteratively pops p equations from the top of the PPQ for processing. Each of these equations is expanded through recursive application of the generation functions. The resulting equations are memoized using the IPT. If the equation has been encountered before, it is discarded. The remaining unique equations are fit to the training data with non-linear regression, evaluated on the testing data, and pushed into the PPQ. PGE continues until a model of desired accuracy is discovered or a computational threshold is reached. Theoretically, PGE could explore the entire space of representable equations given infinite space and unlimited time.

Figure 6: The PGE Search Loop

```
PPQ = ∅
memoTree = ∅
func PGE_Search(int p, GenerationFunc gFuncs,
                Data trainData, Data testData ) {
  bases = createBasisFunctions()
  memoTree.insert(bases)
  bases.FitCoefficents(trainData)
  PPQ.push(bases)
  bases.Evaluate(testData)

  while (check_stopping_criteria() != stop)
    new_eqns = ∅
    for i=0; i < p; i++
      top_eqn = PPQ.pop()
      new_eqns += Expand(top,gFuncs)

    for e in new_eqns
      did_ins = memoTree.insert(e)
      if did_ins
        e.FitCoefficents(trainData)
        PPQ.push(e)
        e.Evaluate(testData)
}
```

Parameters

PGE is nearly a parameter free algorithm, requiring only that the building blocks, generating functions, the value of p, and termination criteria be established prior to beginning an equation search. Since building blocks are generally established by the benchmarks, and stopping criteria are relatively common across SR implementations, PGE has only two unique parameters: (1) p, the number equations to remove from the PPQ and (2) which generation functions to use. This is a substantial advantage over GP, which requires many parameters to be set, a notoriously difficult task.

4. EXPERIMENTAL RESULTS

We evaluate PGE on 22 benchmarks from [24] and compare results with the published results[17, 32]. We used 200 training points, 2000 testing points, and operator parameters established in [24]. We performed our model searches using a single core running on an Intel i5-2500k @ 3.30GHz with 8 GB of memory. The PGE results in Table 1 were obtained from 400 iterations with a $p = 4$. The PGE results in Table 2 were obtained from 200 iterations with a $p = 3$. Source code is available at github.com/verdverm/go-pge.

We compare PGE against published results of the SSC and AEG algorithm described in Section 2. The SSC-T6 and SSC-T7 errors in Table 1 are taken from Tables 6 and 7 in [32]. The numbers are the average absolute error, obtained

Table 1: Benchmarks results for Nguyen problems

Problem		SSC-T6	SSC-T7	PGE-1			PGE-2			PGE-3		
Name	Equation	error	error	error	time	eqns	error	time	eqns	error	time	eqns
Nguyen-01	$x^3 + x^2 + x$	0.0035	0.003	**0.000003**	0.1s	67	**0.000003**	0.2s	131	**0.000003**	0.1s	73
Nguyen-02	$x^4 + x^3 + x^2 + x$	0.0075	0.007	**0.000004**	1.2s	464	**0.000004**	0.6s	218	**0.000004**	0.3s	124
Nguyen-03	$x^5 + x^4 + x^3 + x^2 + x$	0.009	0.0095	**0.000003**	9.0s	2138	0.094944	9.8s	2240	**0.000003**	1.0s	386
Nguyen-04	$x^6 + x^5 + x^4 + x^3 + x^2 + x$	0.013	0.011	0.107365	12.5s	2428	0.000005	12.2s	2361	**0.000004**	3.8s	1308
Nguyen-05	$sin(x^2) * cos(x) - 1$	0.0045	0.0045	0.000068	46.5s	10164	0.000014	278.1s	10032	**0.000001**	450.1s	18399
Nguyen-06	$sin(x) + sin(x + x^2)$	0.0045	0.0035	0.011798	3.8s	2947	0.000446	805.5s	32079	**0.000036**	68.1s	6196
Nguyen-07	$ln(x + 1) + ln(x^2 + 1)$	0.003	0.0035	0.002571	0.2s	131	0.000351	329.4s	25597	**0.000251**	1078.4s	23499
Nguyen-08	$sqrt(x)$	0.0065	0.005	**0.000001**	0.1s	21	**0.000001**	0.1s	116	**0.000001**	0.1s	78
Nguyen-09	$sin(x) + sin(y^2)$	0.0264	0.0099	0.003898	1.0s	868	**0.000001**	60.0s	6519	0.000270	1461.5s	53721
Nguyen-10	$2 * sin(x) * cos(y)$	0.0122	0.0066	**0.000002**	1.5s	1233	0.000004	1.9s	461	0.000004	0.1s	174

Table 2: Benchmarks results for Korns problems

Problem		AEG-T2		AEG-T4		PGE-1		PGE-2		PGE-3		Testing
Name	Equation	NLSE	eqns	NLSE	eqns	error	eqns	error	eqns	error	eqns	StdDev
Korns-01	$1.57 + 24.3v$	0.00	.15K	0.00	.06K	0.000000	.17K	0.000000	.47K	0.000000	.35K	709
Korns-02	$0.23 + 14.2(v + y)/3w$	0.00	3.26K	0.00	113.00K	0.027277	25.05K	0.0055	34.07K	0.1135	2.30K	165
Korns-03	$-5.41 + 4.9(v - x + y/w)/3w$	0.00	804.49K	0.00	222.46K	0.498	0.36K	0.0065	29.00K	0.1245	1.96K	1810
Korns-04	$-2.3 + 0.13sin(z)$	0.00	.59K	0.00	.86K	0.000000	.17K	0.000000	10.95K	0.000000	28.06K	0.093
Korns-05	$3 + 2.13ln(w)$	0.00	.25K	0.00	.16K	0.000000	.17K	0.000000	.48K	0.000000	.35K	2.154
Korns-06	$1.3 + 0.13sqrt(x)$	0.00	.13K	0.00	.01K	0.000000	.17K	0.000000	.48K	0.000000	.35K	0.215
Korns-07	$213.809408(1 - e^{-0.547237x})$	0.00	187.26K	0.00	4.10K	0.031941	8.37K	0.0075	53.08K	0.058696	9.45K	11487
Korns-08	$6.87 + 11sqrt(7.23xvw)$	0.00	5.99K	0.00	11.00K	0.021827	19.34K	0.000000	9.86K	0.069829	54.18K	2034
Korns-09	$(sqrt(x)/ln(y)) * (e^z/v^2)$	0.00	97.24K	0.00	116.81K	0.1855	1.87K	0.000000	7.14K	0.0615	.70K	8735
Korns-10	$0.81 + 24.3\frac{2y+3z^2}{4v^3+5w^4}$	0.99	763.53K	0.00	1.34 M	0.055193	4.42K	0.008	78.19K	0.107	1.65K	2763
Korns-11	$6.87 + 11cos(7.23x^3)$	1.00	774.89K	0.00	4.7 M	0.493	.17K	0.0055	8.33K	0.1195	2.62K	7.794
Korns-12	$2 - 2.1cos(9.8x)sin(1.3w)$	1.04	812.79K	1.00	16.7 M	0.117404	34.34K	0.0065	44.27K	0.124	1.67K	1.056

by selecting the best summed absolute error and dividing by the number of sample points. This was done so that runs with different number of sample points could be compared. The errors in Table 2 here, for AEG-T2 and AEG-T4 are from [17], Tables 2 and 4 respectively. These numbers are the least squared errors normalized to the standard deviation of the output variable. Since the errors in Table 2 are not directly comparable due to differing error calculations, we included the standard deviation of the output variable from our testing data. Equations listed for AEG are total number of equations evaluated before a solution was found as reported in [17].

PGE-(1,2,3) are the three expansion variations discussed in Section 3.3 The reported error numbers for PGE in both tables are the average absolute error on the testing data. The times reported in Table 1 are the amount of elapsed time from the initial invocation of the PGE implementation until the minimal error is reached. Reported equation counts are the number unique equations evaluated when the minimum was reached.

PGE solved all cases were the error was less than 0.0001, meaning PGE returned the *exact* formula in the first Pareto front. Numerical accuracy limitations and minor variations in coefficient values preclude this error from reaching zero. The amount of improvement in error is several orders of magnitude over SSC, demonstrating PGE's ability to hone in on solutions. For the single and bimodal Nguyen problems, PGE solved all but one. For the five-variable Korns problems PGE solves 6 exactly, but did not solve the other 6. PGE had difficulty with nested addition and non-linear coefficients within non-linear functions. AEG required larger numbers of equations for the same problems, so is not surprising that PGE's error values were larger for these problems. Performance, however, is still good on the simple problems and shows they are tractable even with noisy channels.

Comparing number of equations examined is more difficult. AEG equations reported are the total evaluated, while PGE is the unique count. Due to PGE's use of sorting and canonical forms, the number of equations evaluated by PGE would be much larger in an equivalent, unreduced space. It is also likely that PGE overlooks portions of the expression space through its simplifications and search space ordering. We hope to determine the extent to which this happens and further characterize the search space in future work.

PGE is also efficient, not only from the space reduction but also in time. All searches ran to completion in under 30 minutes. Beyond this, the simple equations from both test suites were often found very early in the search, and in many cases finding the correct solution in just a few seconds. These times are comparable to AEG which was reported as taking 1-60 minutes to complete. In comparison, FFX runs very quickly, returning results on 13 variable problems in just 5-60 seconds. As was discussed in Section 2, its limitations render it unable to find many of these benchmarks, so we omitted further analysis. SSC times were not reported, however, the reported errors were averaged over 100 trials. Because PGE is deterministic, it does not have to be run multiple times in this manner, creating a relative savings in time.

5. CONCLUSIONS

As we aim for an algorithm that solves the SR problem we must consider alternative implementations to the usual GP. PGE is a deterministic SR implementation with exactly reproducible results, using no random number generation. PGE replaces evolutionary methods with well defined expansion rules and prioritizes the search with the Pareto Priority Queue. PGE consolidates the space of equations by merging isomorphs with sorting and simplification algorithms. PGE further avoids duplication of effort by considering form once, through the use of abstract parameters, non-linear regression, and memoization. This also allows PGE to separate the search for form, from the optimization of that form. PGE is effective, providing accurate results and returning many exact solutions in a short amount of time. Additionally, PGE only needs to be run once to provide statistically significant results.

As a deterministic algorithm, PGE may be useful for characterizing the difficulty of problems, and the IPT can provide a new metric for measuring the search coverage. PGE may also bring new insites on the nature of non-convexity in SR problems.

We believe further progress can be made by incorporating the use of all-valid s-expression crossover and embedding PGE into the AEG framework to make use of abstract variables and functions. Speedups may be realized, while maintaining deterministic behavior by parallelizing the evaluation of equations. We hope to provide validation on more of the benchmarks as well as differential equations, invariants, and other problems from domains beyond equations which can be represented by an abstract grammar.

6. REFERENCES

[1] D.W. Corne, N.R. Jerram, J.D. Knowles, M.J. Oates, et al. Pesa-ii: Region-based selection in evolutionary multiobjective optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'2001*. Citeseer, 2001.

[2] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. *Lecture notes in computer science*, 1917:849–858, 2000.

[3] R.C. Eberhart, Y. Shi, and J. Kennedy. *Swarm intelligence*. Morgan Kaufmann, 2001.

[4] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, FOCS '97, pages 137–, Washington, DC, USA, 1997. IEEE Computer Society.

[5] C.M. Fonseca, P.J. Fleming, et al. Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In *Proceedings of the fifth international conference on genetic algorithms*, volume 1, page 416. San Mateo, California, 1993.

[6] N. Hoai, R.I. McKay, and HA Abbass. Tree adjoining grammars, language bias, and genetic programming. *Genetic Programming*, pages 157–183, 2003.

[7] N.X. Hoai. Solving the symbolic regression problem with tree-adjunct grammar guided genetic programming: The preliminary results. In *In Proceedings of Congress on Evolutionary Computation (CEC-2002), Hawai*. Citeseer, 2002.

[8] N.X. Hoai and RI McKay. A framework for tree adjunct grammar guided genetic programming. In *Proceedings of the Post-graduate ADFA Conference on Computer Science (PACCS-2001)*, pages 93–99, 2001.

[9] J. Horn, N. Nafpliotis, and D.E. Goldberg. A niched pareto genetic algorithm for multiobjective optimization. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 82–87. IEEE, 1994.

[10] G.S. Hornby. Alps: the age-layered population structure for reducing the problem of premature convergence. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 815–822. ACM, 2006.

[11] Maarten Keijzer. Improving symbolic regression with interval arithmetic and linear scaling. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 70–82, Essex, 14-16 April 2003. Springer-Verlag.

[12] M. Kim, T. Hiroyasu, M. Miki, and S. Watanabe. Spea2+: Improving the performance of the strength pareto evolutionary algorithm 2. In *Parallel problem solving from nature-PPSN VIII*, pages 742–751. Springer, 2004.

[13] D. Kinzett, M. Johnston, and M. Zhang. How online simplification affects building blocks in genetic programming. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 979–986. ACM, 2009.

[14] D. Kinzett, M. Zhang, and M. Johnston. Using numerical simplification to control bloat in genetic programming. *Simulated Evolution and Learning*, pages 493–502, 2008.

[15] M.F. Korns. Large-scale, time-constrained symbolic regression-classification. *Genetic Programming Theory and Practice V*, pages 53–68, 2008.

[16] M.F. Korns. Abstract expression grammar symbolic regression. *Genetic Programming Theory and Practice VIII*, pages 109–128, 2011.

[17] M.F. Korns. Accuracy in symbolic regression. *Genetic Programming Theory and Practice IX*, pages 129–151, 2011.

[18] M. Kotanchek, G. Smits, and E. Vladislavleva. Trustable symbolic regression models: using ensembles, interval arithmetic and pareto fronts to develop robust and trust-aware models. *Genetic Programming Theory and Practice V*, pages 201–220, 2008.

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

[20] M.I.A. Lourakis. levmar: Levenberg-marquardt nonlinear least squares algorithms in C/C++. [web page] `http://www.ics.forth.gr/~lourakis/levmar/`, Jul. 2004. [Accessed on 13 July. 2012.].

[21] S. Luke, L. Panait, et al. Lexicographic parsimony pressure. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 829–836, 2002.

[22] H. Majeed and C. Ryan. Using context-aware crossover to improve the performance of gp. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 847–854. ACM, 2006.

[23] Trent McConaghy. Ffx: Fast, scalable, deterministic symbolic regression technology. In Rick Riolo, Ekaterina Vladislavleva, and Jason H. Moore, editors, *Genetic Programming Theory and Practice IX*, Genetic and Evolutionary Computation, pages 235–260. Springer New York, 2011. 10.1007/978-1-4614-1770-5-13.

[24] James McDermott, David R. White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth De Jong, and Una-May O'Reilly. Genetic programming needs better benchmarks. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, GECCO '12, pages 791–798, New York, NY, USA, 2012. ACM.

[25] R.I. Mckay, N.X. Hoai, P.A. Whigham, Y. Shan, and M. OâĂŹNeill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3):365–396, 2010.

[26] R.K. McRee. Symbolic regression using nearest neighbor indexing. In *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation*, pages 1983–1990. ACM, 2010.

[27] J.F. Miller and P. Thomson. Cartesian genetic programming. *Lecture Notes in Computer Science*, pages 121–132, 2000.

[28] Q. Nguyen, X. Nguyen, and M. OâĂŹNeill. Semantic aware crossover for genetic programming: the case for real-valued function regression. *Genetic Programming*, pages 292–302, 2009.

[29] G.R. Raidl. A hybrid gp approach for numerically robust symbolic regression. *Genetic Programming*, pages 323–328, 1998.

[30] G. Smits and M. Kotanchek. Pareto-front exploitation in symbolic regression. *Genetic programming theory and practice II*, pages 283–299, 2005.

[31] A. Topchy and W.F. Punch. Faster genetic programming based on local gradient search of numeric leaf values. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 155–162, 2001.

[32] N.Q. Uy, N.X. Hoai, M. O'Neill, RI McKay, and E. Galván-López. Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines*, 12(2):91–119, 2011.

[33] D.A. Van Veldhuizen and G.B. Lamont. Evolutionary computation and convergence to a pareto front. In *Late Breaking Papers at the Genetic Programming 1998 Conference*, pages 221–228, 1998.

[34] Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)*, SWAT '73, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society.

[35] E. Zitzler, M. Laumann, L. Thiele, E. Zitzler, E. Zitzler, L. Thiele, and L. Thiele. Spea2: Improving the strength pareto evolutionary algorithm, 2001.