On the Architecture and Implementation of Tree-based Genetic Programming in HeuristicLab

Michael Kommenda, Gabriel Kronberger,
Stefan Wagner, Stephan Winkler, and Michael Affenzeller
University of Applied Sciences Upper Austria
School of Informatics, Communication and Media
Softwarepark 11, 4232 Hagenberg, AUSTRIA
{michael.kommenda, gabriel.kronberger,
stefan.wagner, stephan.winkler, michael.affenzeller}
@fh-hagenberg.at

ABSTRACT

This article describes the architecture and implementation of the genetic programming (GP) framework of HeuristicLab. In particular we focus on the core design goals, namely extensibility, usability, and performance optimization and explain our approach to reach these goals. The overall design, the encoding, interpretation, and evaluation of programs is described and code examples are given to explain core aspects of the framework. HeuristicLab is available as open source software at `http://dev.heuristiclab.com`.

Categories and Subject Descriptors

I.2.2 [**Automatic Programming**]: Program synthesis; I.2.8 [**Problem Solving, Control Methods, and Search**]: Heuristic Methods

General Terms

Design

Keywords

Genetic Programming, Symbolic Regression, HeuristicLab

1. INTRODUCTION

Genetic programming (GP) is a well-established heuristic method for solving optimization problems and an active research field. A simple variant of tree-based GP can be implemented in only 500 lines of code [11]. However, it is not straightforward to implement a generic and flexible GP framework in an efficient way. Several frameworks have been developed that provide components and out-of-the-box algorithm implementations for heuristic optimization in general and, in particular, GP. Prominent examples of heuristic optimization frameworks with GP support are ECJ [9], Evolving Objects [2], and Open BEAGLE [1].

Most frameworks are implemented as libraries and provide only a rudimentary graphical user interface (GUI). Before an algorithm can be used to solve a given optimization problem, users often have to build the source, setup the optimization environment, and manipulate settings in configuration files. HeuristicLab [12] tries to avoid this by providing as much functionality as possible as ready-to-use components within the GUI without sacrificing extensibility.

In this publication the design goals, architecture, and implementation of GP in HeuristicLab are described. Section 2 lists the design principles and gives an overview of the features of GP in HeuristicLab. Section 3 explains the architecture and implementation in detail, in particular the separation of problem and algorithm implementations. Furthermore, we describe the code for interpretation and fitness evaluation for symbolic regression solutions in Section 4. Finally, Section 5 concludes the paper with a summary.

1.1 HeuristicLab

The development of HeuristicLab started in 2002 to provide an environment for developing and testing heuristic optimization methods. The main motivation was to build a paradigm-independent, flexible, extensible, and user-friendly software framework using state-of-the-art programming techniques and concepts that can be used in research projects and for teaching. C# was chosen as development language as it provides a reasonable trade-off between development and execution efficiency. After several major releases the latest version HeuristicLab 3.3 has been made available under the GNU General Public License (GPL v3.0) and can be downloaded at `http://dev.heuristiclab.com`.

Since version 1.0 HeuristicLab has included tree-based GP, with a strong focus on symbolic regression and classification. To comply with the design and architecture of HeuristicLab 3.3 the previous GP implementation has been completely rewritten. This new implementation is described in the following sections.

2. OVERVIEW

The architecture of our GP implementation is based on the following design considerations which are also central design goals of the core framework of HeuristicLab.

GECCO'12 Companion, July 7–11, 2012, Philadelphia, PA, USA.

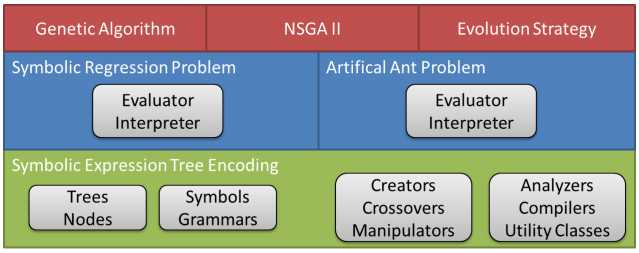

Figure 1: The architecture of the most important components of the implementation. The bottom layer displays encoding specific classes that can be used by problems or algorithms. Multiple algorithms can be used to solve GP problems. A problem implementation must provide an evaluator; the remaining operators are supplied by the encoding.

2.1 Design Considerations

Separation of Algorithms, Encodings, and Problems

The first design consideration is separation of the algorithm, the encoding, and the problem (as shown in Figure 1). The separation of these components allows the reuse of existing code in various locations. For example, the implementations of the artificial ant problem [5] and of symbolic regression both use the same operators provided by the symbolic expression tree encoding. Since crossover and mutation operators are provided by the encoding, both tasks can be solved by all population-based, evolutionary algorithms. Thus, implementing all combinations of algorithms and problems individually is not necessary.

Extensibility

Another important aspect is that the implementation should provide operators for tree creation, manipulation, and analysis described in the literature. However, the framework must be extensible as it is impossible to provide all features out of the box and researchers frequently want to use improved operator variants. In HeuristicLab extensibility is achieved through a plugin-based architecture [13] and algorithm modeling concept. Custom operators can be implemented in separate plugins which are discovered and loaded during program runtime to inject new functionality. Furthermore, algorithms are represented in a graphical way and can be modified to allow customization to specific needs.

Reproducibility of Results

Most heuristic algorithms are non-deterministic and rely on pseudo random number generators (RNG) to find solutions. Therefore, multiple executions of the same algorithm usually yield different results. To guarantee the reproducibility of results it should be possible to specify the seed value for the RNG. For the same seed value the algorithm must behave in exactly the same way and yield the same results. Even if parts of the algorithm are executed in parallel, this principle must be adhered to.

Efficiency

An important and well-known application of GP is symbolic regression, thus, special emphasis has been put on the efficiency of the code for symbolic regression in HeuristicLab. This may be counter-intuitive as a comprehensive GUI is provided, C# with .NET 4.0 is used as programming language, and a costly algorithm abstraction is used to provide a generic framework. Nevertheless, a large part of the execution time in symbolic regression is spent on fitness evaluation, which scales linearly with the number of data rows. We routinely work with rather large datasets comprised of more than $10,000$ data rows. Thus, the evaluation of symbolic regression solutions must be as efficient as possible (cf. Section 4). In particular, dynamic memory allocation should be reduced as much as possible to reduce memory pressure. In managed languages as C#, the overhead of garbage collection can be a performance killer if fitness evaluation is implemented in a naive way. This is especially important on multi-core machines where the memory bandwidth can become the bottleneck for fitness evaluation.

Multi-core Support

The last design principle is that operators should be implemented in a functional programming style and not save an internal state to ease multi-core parallelization. Using stateful operators would require locking mechanisms which would limit the possible performance improvement achieved by using multiple cores.

2.2 Features

Symbolic Tree Representation

As mentioned previously, the GP implementation in HeuristicLab is based on symbolic expression trees as representation for evolved programs. This has the advantage that operators for individual creation, crossover, and manipulation are straightforward and well described in the literature, for example in [5, 11, 8].

Syntactic Restrictions

The encoding is further enhanced by possibilities to restrict the space of possible solutions with grammars. A grammar defines the syntactical structure of trees by specifying which symbols are allowed at which position. Correctly configured grammars for problems are already provided and can be manipulated graphically in the user interface. Through this mechanism it is possible to imitate strongly typed GP [10], but it is more powerful, for example, it allows the incorporation of a-priori knowledge by fixing parts of the trees which cannot be modified. An exemplary grammar to create arithmetic expressions to solve a symbolic regression problem is shown in Figure 3.

Automatically Defined Functions

Automatically defined functions (ADFs) were first described by John Koza in [6] and allow genetic programming to evolve reusable subroutines dynamically. HeuristicLab provides *architecture manipulation operators* which create, invoke, and delete subroutines and also manipulate the arguments of the subroutines. Through these operators the evolutionary process can evolve ADFs if necessary to solve a given problem.

Default Operator Implementations

HeuristicLab provides several standard operators for tree-based GP, for example *full*, *grow*, *ramped half-and-half*, and *PTC2* for tree creation [11, 8], *subtree crossover* [5] for child creation, and *replace branch*, *point mutation*, and *tree shaker* for tree manipulation. The operators are implemented as

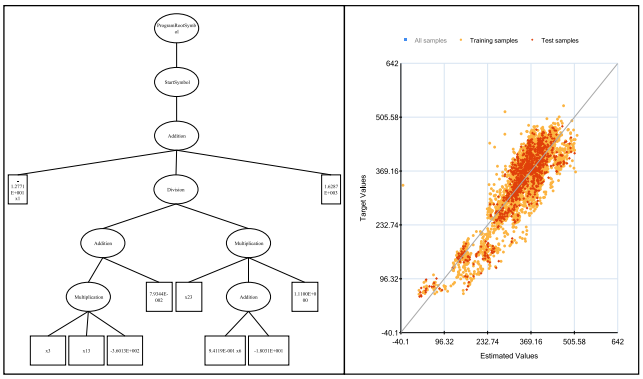

Figure 2: Screenshot of the visualization capabilities of HeuristicLab showing a symbolic expression model together with its evaluation.

described in the cited literature, but have been adapted to support the search space restrictions defined by grammars.

Comprehensive Visualization Capabilities

HeuristicLab provides a comprehensive GUI and visualization support, not only for GP. Figure 2 shows exemplary screenshots including the visualization of a symbolic expression tree and its evaluation result as a line chart.

Analysis Support

Furthermore the internal algorithm behavior can be assessed by so-called *analyzers*, that are executed every iteration. Analyzers extract information of the population and the algorithm during the runtime. Currently analyzers to extract the best solution, the distribution of qualities, tree sizes, and symbol frequencies are provided.

Benchmark Problems

HeuristicLab 3.3.6 includes implementations of the artificial ant problem, and symbolic regression and classification problems. To facilitate algorithm and operator comparison, HeuristicLab includes a number of difficult artificial and real-world benchmark problems for genetic programming as of version 3.3.7. These benchmark problems have been added in reaction to a recent discussion on the GP mailing list[1] where it was stated that GP has a toy problem problem, as many research papers publish results on simple toy problems only.

3. ENCODING OF SYMBOLIC EXPRESSION TREES

In the following the overall design and implementation of symbolic expression trees, symbols, and grammars are described. The core functionality for handling symbolic expression trees is provided by the encoding. The encoding defines the structure of individuals and combines problem-independent classes that can be reused. All standard methods for tree creation, manipulation, crossover and compilation are located in the encoding and therefore a newly implemented problem just has to provide concrete symbols and methods to evaluate solution candidates. For example, in the case of a symbolic regression problem, the evaluator calculates the error of the model predictions and uses an interpreter to calculate the output of the formula for each row of the dataset.

Any algorithm that uses recombination and mutation operators to generate new solution candidates, for instance a genetic algorithm (GA), can be used to solve any problem using the symbolic expression tree encoding for instance a symbolic regression problem. A specialized algorithm for genetic programming with reproduction and crossover probability is not yet provided but is planned to be added soon.

3.1 Symbolic Expression Trees

The GP system of HeuristicLab is based on a tree representation and the most important interfaces are: `ISymbolicExpressionTree`, `ISymbolicExpressionTreeNode`, and `ISymbol`.

The structure of a tree is defined by linked nodes, and the semantic is defined by symbols attached to these nodes. The `SymbolicExpressionTree` is the class for trees and has properties for accessing its root node, getting its length and depth, and iterating all tree nodes in a prefix or postfix manner. Starting from the root node, every other node can be reached, as an `ISymbolicExpressionTreeNode` has properties and methods to manipulate its parent, its subtrees, its symbol and helper methods for applying actions on all subtrees. An individual is therefore simply created, by linking tree nodes and setting the root node of the `ISymbolicExpressionTree`.

The root node of a tree is per default a node with a specialized symbol, the `ProgramRootSymbol`, whose child is a node with the `StartSymbol` that tells the interpreter where the program starts. This convention was introduced as the usage of ADFs makes it necessary to have multiple program fragments and ADFs are represented as additional nodes containing a `DefunSymbol`, next to the `StartSymbol` (cf. Section 3.3).

3.2 Symbols and Grammars

In addition to the structure of a tree, symbols and grammars are necessary for individual creation. A symbol defines the semantic of a tree node (how it is interpreted) and specifies a minimum and maximum arity; terminal symbols have a minimum and maximum arity of zero. All available symbols must be included in a grammar which defines the set of valid and well-formed trees. We have chosen to implement this by defining which `Symbols` are allowed as child symbol of other symbols, and at which position they are allowed. For example, the first child of a conditional symbol must either be a comparison symbol or a boolean function symbol.

A base class for grammars is provided in the encoding. Problem-specific grammars should derive from this base class and configure the grammar rules for the desired tree structures. This initialization of a arithmetic expression grammar is shown in Figure 4. Code given in lines 1–10 creates the symbols and adds them to lists for easier handling. Afterwards the created symbols are added to the grammar (lines 12-13) and the number of allowed subtrees is set to two for all function symbols (lines 14–15). Terminal symbols do not have to be configured, because the number of allowed subtrees is automatically set to zero. The last lines define which symbols are allowed at which position in the tree. Below the `StartSymbol` all symbols are allowed (lines 16,17)

[1] `http://tech.groups.yahoo.com/group/genetic_programming/message/5410`

```
<expr> := <expr> <op> <expr> | <terminal>
<op>   := + | - | / | *
<terminal>  := variable | constant
```

Figure 3: Backus-Naur Form of an arithmetic grammar defining symbolic expression trees to solve a regression problem.

```
1 var add = new Addition();
2 var sub = new Subtraction();
3 var mul = new Multiplication();
4 var div = new Division();
5 var constant = new Constant();
6 var variableSymbol = new Variable();
7 var allSymbols = new List<Symbol>()
8  {add,sub, mul,div,constant,variableSymbol};
9 var funSymbols = new List<Symbol>()
10  {add,sub,mul,div};
11
12 foreach (var symb in allSymbols)
13   AddSymbol(symb);
14 foreach (var funSymb in funSymbols)
15   SetSubtreeCount(funSymb, 2, 2);
16 foreach (var symb in allSymbols)
17   AddAllowedChildSymbol(StartSymbol, symb);
18 foreach (var parent in funSymbols) {
19   foreach (var child in allSymbols)
20     AddAllowedChildSymbol(parent, child);
21 }
```

Figure 4: Source code for the configuration of the `ArithmeticGrammar` formally defined in Figure 3.

and in addition, every symbol is allowed under a function symbol (Lines 18–21).

Default grammars are implemented and pre-configured for every problem which can be solved by GP. These grammars can be modified within the GUI to change the arity of symbols or to enable and disable them.

A typical operator for tree creation first adds nodes with the `RootSymbol` and the `StartSymbol`. Afterwards the `Grammar` is asked to return a list of allowed symbols, from which one is randomly chosen and an according tree node is created and added. This procedure is recursively applied until the desired tree size is reached. In addition, the crossover and mutation operators also adhere to the rules defined by the grammar resulting in valid and well-formed trees during the whole algorithm run.

3.3 Automatically Defined Functions

As previously mentioned, the described GP system supports automatically defined functions (ADFs). ADFs are program subroutines that provide code encapsulation and reuse. They are not shared between individuals but have to be evolved separately in individuals, either by crossover or mutation events and are numbered according to their position below the tree root. The `Defun` tree node and symbol define a new subroutine and are used next to the `StartSymbol` directly below the `ProgramRootSymbol`. ADFs can be called from any part of the tree by inserting an `InvokeFunction` tree node, except from ADFs with a lower index to prevent infinite recursions and non-stopping programs.

ADFs are created during algorithm execution by the subroutine creator which moves a subtree of an individual into a subroutine and inserts an `InvokeFunctionTreeNode` instead of the original subtree. Furthermore, ADFs can have an arbitrary number of arguments that are used to parameterize the subroutines. An example for creating an ADF with one argument is shown in Figure 5. On the left hand side the original tree describing an artificial ant program is displayed. Additionally, two cut points for the ADF extraction are indicated by dashed lines. The subtree between the cut points is added beneath a `DefunTreeNode` displayed as the newly defined ADF *ADF0* and as a replacement an `InvokeFunctionTreeNode` is inserted. The subtree below the second cut point is left unmodified and during interpretation its result is passed to *ADF0* and replaces all occurrences of *ARG0*.

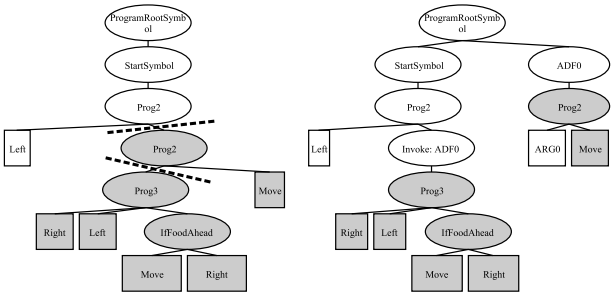

Figure 5: Creation of an ADF in an artificial ant program. The dashed lines indicate the cut points in the tree from which the ADF is created.

Altering the structure of a tree to create ADFs is performed by so-called architecture manipulating operators that can be called in the mutation step. Architecture manipulating operators for subroutine and argument creation, duplication, and deletion are provided by the framework. All of these work by moving parts of the tree to another location, either in the standard program execution part (below the `StartTreeNode`) or into an ADF (below the `DefunTreeNode`). For example, the subroutine deletion operator replaces all tree nodes invoking the affected subroutine by the body of the subroutine itself and afterwards deletes the subroutine from the tree by removing the `DefunTreeNode`.

The combination of architecture altering operators and tree structure restrictions with grammars is non-trivial as grammars must be dynamically adapted over time. Newly defined ADFs must be added to the grammar; however, the grammar of each single tree must be updated independently because ADFs are specific to trees. This has led to a design where tree-specific grammars contain dynamically extended rules and extend the initially defined static grammar. The combination of the tree-specific grammar and the static grammar defines the valid tree structures for this solution and also for child solutions because grammars are inherited by child solutions. If no ADFs are allowed in the GP algorithm the tree-specific grammar is always empty since no symbols are dynamically added during the run.

Architecture manipulating operators automatically update the tree-specific grammar correctly by altering the allowed symbols and their restrictions. This mechanism allows the execution of crossover and mutation, without even knowing if ADFs are present in the current tree and only valid trees according to the grammar are created.

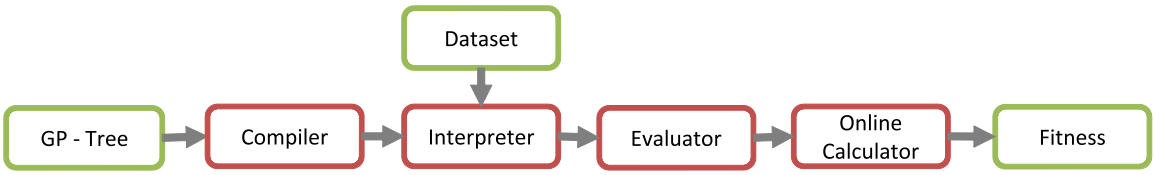

Figure 6: Workflow for calculating the fitness of symbolic regression models.

4. SYMBOLIC REGRESSION MODEL EVALUATION

In this section we describe the implementation for evaluating symbolic regression models represented as symbolic expression trees. Symbolic regression is frequently used as a benchmark task for testing new algorithmic concepts and ideas. If symbolic regression is applied on large real-world datasets containing several thousand data rows, performance as well as memory efficiency becomes an important issue.

The main concepts for the symbolic regression evaluation in HeuristicLab are streaming and lazy evaluation provided by constructs of the .NET framework. Figure 6 depicts how the symbolic expression trees are passed through different operators for fitness value calculation, which is explained in the following sections.

4.1 Interpretation of Trees

Evaluators for symbolic regression must calculate the error of the predicted values produced by the model and the actual target values. To prevent the allocation of large double arrays we implemented an interpreter for symbolic regression models that yields a lazy sequence of predicted values for a given model, a dataset and a lazy sequence of row indexes. As a preparatory step the interpreter first compiles the model represented as a symbolic expression tree down to an array of instructions. This preparation can be done in a single pass over all nodes of the tree so the costs are rather small and the linear instruction sequence can be evaluated much faster. First of all the nodes of the original tree are scattered on the heap, while the instruction array is stored in a continuous block of memory. Additionally, the instructions have a small memory footprint and consist of a single byte for the operation code (opcode), a byte for the number of arguments of the function and an object reference which can hold additional data for the instruction. As a result the instruction array is much more cache friendly and the number of cache misses of the tree interpretation can be reduced. Another benefit of the compilation step is that simple static optimizations can be applied (e.g. constant folding).

The interpretation of the instruction array is implemented with a simple recursive `Evaluate` method containing a large switch statement with handlers for each opcode. Figure 7 shows an excerpt of the evaluation method with the handlers for the opcodes for addition, division and variable symbols. The recursive evaluation method is rather monolithic and contains all the code for symbol evaluation. However, the implementation as a single monolithic switch loop with recursive calls is very efficient as no virtual calls are necessary, the switch statement can be compiled down to a relative jump instruction, and the arguments are passed on the runtime stack which again reduces cache misses.

An alternative design would be to implement a specific evaluation method in each symbol class. This would be the preferable way regarding readability and maintainability. However, with this alternative design a costly indirect virtual call would be necessary for each node of the tree and for each evaluated row of the dataset.

```
1  double Evaluate(Dataset ds, State state) {
2    var curInstr = state.NextInstruction();
3    switch (curInstr.opCode) {
4      case OpCodes.Add: {
5        double s = Evaluate(dataset, state);
6        for (int i = 1; i < curInstr.nArgs; i++) {
7          s += Evaluate(dataset, state);
8        }
9        return s;
10     }
11     // [...]
12     case OpCodes.Div: {
13       double p = Evaluate(dataset, state);
14       for (int i = 1; i < curInstr.nArgs; i++) {
15         p /= Evaluate(dataset, state);
16       }
17       if (curInstr.nArgs == 1) p = 1.0 / p;
18       return p;
19     }
20     // [...]
21     case OpCodes.Variable: {
22       if (state.row < 0 ||
23           state.row >= dataset.Rows)
24         return double.NaN;
25       var varNode =
26        (VariableTreeNode)curInstr.dynamicNode;
27       var values = ((IList<double>)curInstr.iArg0)
28       return
29        values[state.row];
30     }
31   }
32 }
```

Figure 7: Excerpt of the evaluation method for symbolic regression models showing handlers for the addition, division and variable opcodes.

In addition to the recursive interpreter HeuristicLab also provides an interpreter implementation that compiles symbolic expression trees to linear code in intermediate language (IL) which can be directly executed by the .NET CLR using the `Reflection.Emit` framework. This interpreter is useful for large datasets with more than $10,000$ rows as the generated IL code is further optimized and subsequently compiled to native code by the framework JIT-compiler. The drawback is that the JIT-compiler is invoked for each evaluated tree and these costs can be amortized only when the dataset has a large number of rows.

```
// [...]
case OpCodes.Call: {
  // evaluate subtrees
  var argValues = new double[curInstr.nArgs];
  for (int i = 0; i < curInstr.nArgs; i++) {
    argValues[i] = Evaluate(dataset, state);
  }
  // push on argument values on stack
  state.CreateStackFrame(argValues);

  // save the pc
  int savedPc = state.ProgramCounter;
  // set pc to start of function
  state.PC = (ushort)curInstr.iArg0;
  // evaluate the function
  double v = Evaluate(dataset, state);

  // delete the stack frame
  state.RemoveStackFrame();

  // restore the pc => evaluation will
  // continue at point after my subtrees
  state.PC = savedPc;
  return v;
}
case OpCodes.Arg: {
  return
    state.GetStackFrameValue(curInstr.iArg0);
}
// [...]
```

Figure 8: Code fragment for the interpretation of ADFs and function arguments.

4.2 Interpretation of ADFs

The interpreter must be able to evaluate trees with ADFs with a variable number of arguments. The instruction for calling ADF uses the `Call` opcode and contains the index of the ADF to call and the number of arguments of the ADF. The code fragment for the interpretation of ADFs and function arguments is shown in Figure 8. First the interpreter evaluates the subtrees of the `Call` opcode and stores the results. Next the interpreter creates a stackframe which stores the current program counter and the argument values. The interpreter internally manages a stack of stackframes to support ADFs that subsequently call other ADFs. After the stackframe has been created the interpreter jumps to the first instruction of the ADF. When argument are encountered while interpreting the ADF instructions the interpreter accesses the previously calculated argument values which are stored in the top-most stackframe. Precalculation of argument values is only possible because symbolic regression expressions do not have side effects. Otherwise the interpreter would have to jump back to the subtrees of the call symbol for each encountered `Arg` opcode. At the end of the ADF definition the interpreter deletes the top-most stackframe with `RemoveStackFrame` and continues interpretation at the point after the subtrees of the just evaluated `Call` opcode.

4.3 Online Evaluation of Programs

The first step for the evaluation of programs is to obtain the dataset on which the trees have to be evaluated on and to calculate the rows that should be used for fitness evaluation. If all samples are to be used, the rows are streamed as an `Enumerable` beginning with the start of the training partition until its end. Otherwise, the row indices to evaluate the tree on, are calculated and yielded by the selection sampling technique [3], which is shown in Figure 9.

```
public IEnumerable<int> SampleRandomNumbers
       (int start, int end, int count) {
  int remaining = end - start;
  var random = new Random();
  for (int i = 0; i < end && count > 0; i++) {
    double rand = random.NextDouble();
    if (rand < ((double)count) / remaining) {
      count--;
      yield return i;
    }
    remaining--;
  }
}
```

Figure 9: Selection sampling technique to generate a stream of numbers containing exactly a specified number of elements between a given minimum and maximum.

The row indices, together with the dataset and the individual are passed to the interpreter that in fact returns a sequence of numbers. Until now no memory is allocated (except the space required for the iterators) due to the streaming capabilities of the interpreter and the way of calculating row indices. But the whole streaming approach would by pointless if the estimated values of the interpreter were stored in a data structure for fitness calculation. Therefore, all fitness values must be calculated on the fly which is done by `OnlineCalculators`. Such calculators are provided for the mean and the variance of a sequence of numbers and for calculation metrics between two sequences such as the covariance and the Pearson's R^2 coefficient. Further error measures are the mean absolute and squared error, as well as scaled ones, the mean absolute relative error and the normalized mean squared error. `OnlineCalculators` can be nested; for example the `MeanSquaredErrorOnlineCalculator` just calculates the squared error between the original and estimated values and then passes the result to the `MeanAndVarianceOnlineCalculator`. The code of the `MeanAndVarianceOnlineCalculator` is presented in Figure 10 and in the `Add` method it can be seen how the mean and variance are updated, when new values are added.

The source for calculating the mean squared error of an individual is shown in Figure 11, where all the parts described are combined. First the row indices for fitness calculation are generated and the estimated and original values obtained (lines 1-3). Afterwards these values are enumerated and passed to the `OnlineMeanSquaredErrorEvaluator` that in turn calculates the actual fitness.

In addition, all operations performed for evaluation are stateless, which implies no changes are necessary for parallel solution evaluation. HeuristicLab uses the task parallel library (TPL) provided by the .NET Framework, that automatically handles work partitioning, thread scheduling, and cancellation of threads. As the evaluation must be done for multiple individuals in a population-based algorithm like GP, the framework simple spawns an evaluation task for every individual. A comparison of different achieved speed ups using parallelization is shown in the next section.

```
public class OnlineMeanAndVarianceCalculator {
  private double oldM, newM, oldS, newS;
  private int n;

  public int Count { get { return n; } }
  public double Mean {
    get { return (n > 0) ? newM : 0.0; }
  }
  public double Variance {
    get { return (n > 1) ? newS / (n-1) : 0.0; }
  }

  public void Reset() { n = 0; }
  public void Add(double x) {
    n++;
    if(n == 1) {
      oldM = newM = x;
      oldS = newS = 0.0;
    } else {
      newM = oldM + (x - oldM) / n;
      newS = oldS + (x - oldM) * (x - newM);

      oldM = newM;
      oldS = newS;
    }
  }
}
```

Figure 10: Source code of the `MeanAndVarianceOnlineCalculator`.

```
var rows = Enumerable.Range(0,trainingEnd);
var estimated = interpreter.GetExpressionValues
  (tree, dataset, rows).GetEnumerator();
var original = dataset.GetDoubleValues
  (targetVariable, rows).GetEnumerator();
var calculator = new OnlineMSECalculator();

while(original.MoveNext() & estimated.MoveNext()) {
  double o = original.Current;
  double e = estimated.Current;
  calculator.Add(o.Current, e.Current);
}
double MSE = calculator.MeanSquaredError;
```

Figure 11: Source code for calculating the mean squared error between the original values and the estimated values of an individual.

4.4 Performance

A goal of the presented GP implementation is to make it as efficient as possible, without sacrificing the benefits of real-time algorithm analysis in the GUI or overly complicated source code. Therefore, the performance of most GP related operators is measured and tracked by unit tests. In addition to performance unit tests, unit tests which execute a predefined algorithm with a fixed random seed have been implemented. This allows the detection of algorithmic changes and therefore the reproducibility of results is guaranteed.

Table 1: Operator performance for tree creation, crossover and tree interpretation on a symbolic regression problem.

Symbolic expression tree	Trees / second
Full tree creator	1,261
Grow tree creator	1,695
Probabilistic tree creator	502
Subtree crossover	2,960
Symbolic regression problem	**Nodes / second**
Normal interpreter	26,664,077
IL emitting interpreter	14,233,213

Operator performance

The performance of operators is measured by executing the operator multiple times to get an estimate for its execution time. The operators are tested without any parallelization support on different symbolic expression trees, as the structure and size of the trees may also affect the execution time.

Table 1 shows in the top the performance of encoding specific operators as number of processed trees per second. The reasons for the differences in the number of created trees is that the probabilistic tree creator (PTC2 [8]) tries to create trees of a specific size and its logic is therefore more complex. Furthermore, the performance of tree creators is not that important because these are only executed in the beginning of an evolutionary algorithm.

A more performance critical comparison specific for symbolic regression problems is shown in the lower part. The performance of two different interpreters is tested on a dataset containing 1,000 rows with 1,000 randomly created trees consisting of a maximum of 100 nodes which are evaluated three times. The numbers are stated as nodes / second and thus can only be compared with the number of rows in the dataset times the population size times the average tree size. The IL emitting interpreter is slower because it contains the overhead of compiling the trees into .NETs intermediate language (IL) before the interpretation can be performed (cf. Section 4.1).

Parallelization benefits

HeuristicLab facilitates thread-based parallelization on multi-core processors for algorithm execution. Algorithms are modeled as a graph of operators that perform separate task like selection, crossover, or evaluation. If an operator is applied multiple times and does not depend on previous execution results, it can be executed in parallel. By default, individual creation and evaluation are performed in parallel.

The benefits of using multiple cores have been tested on a multi-processor machine with eight cores and 32 gigabytes of RAM. A genetic algorithm with a symbolic regression problem was executed three times whereby the number of cores and the number of training samples have been varied. All other parameters have been fixed of which the most important concerning performance are a population size of 500, 100 calculated generations after which the algorithm stops and a maximum tree size of 100.

Table 2 shows the average and the standard deviation of the execution times in seconds for different configurations. The more samples are used the better is the achieved speed up with multiple cores as the evaluation speed highly depends on this figure. The table also shows that using eight cores does not really pay off on smaller problem sizes. However, it is remarkable that if 1,000 rows are exceeded the

Table 2: Average and standard deviation of the execution times of a genetic algorithm with a symbolic regression problem in seconds. The number of cores and samples for learning has been varied to show the parallelization speed up.

Samples	1 Core	2 Cores	4 Cores	8 Cores
100	46.7 (0.7)	35.5 (3.3)	27.6 (0.4)	24.4 (0.2)
500	92.7 (0.4)	57.9 (1.1)	40.8 (0.7)	31.4 (0.3)
1,000	125.4 (0.8)	74.9 (1.1)	46.8 (1.5)	33.6 (0.1)
4,000	413.8 (1.0)	218.1 (1.0)	119.4 (1.0)	70.4 (0.1)

runtime of the algorithm is almost solely dependent on the tree evaluation. This is indicated, as the execution time drops linearly if more cores are added.

5. CONCLUSION

The implementation of GP frameworks is not as easy as it looks in the beginning if efficiency, extensibility and scalability are demanded. Especially support for ADFs and strongly typed GP is complex if provided in a generic way. HeuristicLab achieves these goals by following strict design principles and utilizing state of the art programming concepts supplied by the .NET framework. Contrary to most other GP frameworks, HeuristicLab further provides a comprehensive GUI that allows algorithm configuration, adaption and results analysis. The usefulness of HeuristicLab for implementing and testing new research approach has been demonstrated in several publication and the GP functionality has been used for data mining in various domains [7, 4, 14].

6. ACKNOWLEDGEMENTS

A substantial part of the work described in this paper was done in the Josef-Ressel-Centre *Heureka!* for heuristic optimization, which is supported by the Austrian Research Promotion Agency (FFG).

7. REFERENCES

[1] C. Gagne and M. Parizeau. Open BEAGLE: A new versatile C++ framework for evolutionary computations. In *Late-Breaking Papers of the 2002 Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 161–168, 2002.

[2] M. Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer. Evolving Objects: A general purpose evolutionary computation library. *Artificial Evolution*, 2310:829–888, 2002.

[3] D. E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[4] M. Kommenda, G. K. Kronberger, C. Feilmayr, L. Schickmair, M. Affenzeller, S. M. Winkler, and S. Wagner. Application of symbolic regression on blast furnace and temper mill datasets. In *Proceedings of International Conference on Computer Aided Systems Theory EUROCAST 2011*, pages 305–307, Las Palmas, Spain, 2011.

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

[6] J. R. Koza. *Genetic programming II: automatic discovery of reusable programs*. MIT Press, Cambridge, MA, USA, 1994.

[7] G. Kronberger, S. Fink, M. Kommenda, and M. Affenzeller. Macro-economic time series modeling and interaction networks. In *Proceedings of the 2011 international conference on applications of evolutionary computation - Part II*, EvoApplications'11, pages 101–110, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] S. Luke. Two fast tree-creation algorithms for genetic programming. *IEEE Transactions on Evolutionary Computation*, 4(3):274–283, Sept. 2000.

[9] S. Luke. ECJ: A java-based evolutionary computation research system, 2002. http://cs.gmu.edu/ eclab/projects/ecj/.

[10] D. Montana. Strongly typed genetic programming. *Evolutionary computation*, 3(2):199–230, 1995.

[11] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008.

[12] S. Wagner. *Heuristic Optimization Software Systems - Modeling of Heuristic Optimization Algorithms in the HeuristicLab Software Environment*. PhD thesis, Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria, 2009.

[13] S. Wagner, S. Winkler, E. Pitzer, G. Kronberger, A. Beham, R. Braune, and M. Affenzeller. Benefits of plugin-based heuristic optimization software systems. In *Computer Aided Systems Theory EUROCAST 2007*, volume 4739 of *Lecture Notes in Computer Science*, pages 747–754. Springer Berlin / Heidelberg, 2007.

[14] S. Winkler, M. Affenzeller, W. Jacak, and H. Stekel. Identification of cancer diagnosis estimation models using evolutionary algorithms: a case study for breast cancer, melanoma, and cancer in the respiratory system. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, pages 503–510. ACM, 2011.