A Scalable Symbolic Expression Tree Interpreter for the HeuristicLab Optimization Framework

Simone Cirillo
cirillo@almontcapital.com

Stefan Lloyd
lloyd@almontcapital.com

Almont Capital LLC
12800 Hillcrest Road
Dallas, Texas 75230

ABSTRACT

In this paper we describe a novel implementation of the Interpreter class for the metaheuristic optimization framework HeuristicLab, comparing it with the three existing interpreters provided with the framework. The Interpreter class is an internal software component utilized by HeuristicLab for the evaluation of the symbolic expression trees on which its Genetic Programming (GP) implementation relies. The proposed implementation is based on the creation and compilation of a .NET Expression Tree. We also analyze the Interpreters' performance, evaluating the algorithm execution times on GP Symbolic Regression problems for different run settings. Our implementation results to be the fastest on all evaluations, with comparatively better performance the larger the run population size, dataset length and tree size are, increasing HeuristicLab's computational efficiency for large problem setups.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Interpreters*; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program synthesis*

Keywords

Genetic Programming; HeuristicLab; Interpreter; .NET Expression Trees; Performance

1. INTRODUCTION

HeuristicLab is an open-source metaheuristic optimization framework developed and maintained by the Heuristic and Evolutionary Algorithms Laboratory of the Upper Austria University of Applied Sciences [14, 13]. HeuristicLab supports a variety of heuristic optimization techniques, among which Genetic Programming. Algorithms are implemented using an Operator Graph: a series of loosely-coupled `Operator` objects executed in sequence exposing values and other objects to each other, representing the successive steps in the algorithm workflow as well as the solution analysis process.

Thanks to an entirely modular, plugin-based architecture, HeuristicLab decouples the heuristic algorithm from the problem to be solved, provided the two are compatible. Examples of supported problems are, as of version 3.3.9, Symbolic Regression, Symbolic Classification, Artificial Ant, Traveling Salesman, and Vehicle Routing. HeuristicLab is open-source, and developers are encouraged to write their own plugins for use-cases not supported out of the box.

HeuristicLab supports communication to and from other software [7]; one of the intended uses for this feature is the optimization of simulation-based problems for which the simulation environment is external, as described by Beham et al. [2, 3]; in these documented implementations HeuristicLab was used to optimize simulation parameters and settings. This contrasts with algorithms evolving agents or models acting in a simulation environment, as with Genetic Programming. External evaluation, additionally, is not a feasible approach when the simulation software cannot be interfaced to other programs, or where doing so would be structurally inconvenient and computationally costly, or finally where the optimization algorithm requires inputs representing simulation state variables.

Considering there are no clear guidelines or previous examples regarding the implementation of such a scenario and that HeuristicLab provides all the necessary source files and tools, encouraging the development of additional functionality, our final goal is the development of a series of HeuristicLab plugins implementing a general framework for the internal evaluation of Genetic Programming solutions for simulation-based problems, providing a clear execution flow and domain-specific abstractions and features.

In order to make HeuristicLab capable of running Genetic Programming simulation-based evaluations internally, several components of its execution flow have to be reworked.

1.1 GP Interpreters

The Genetic Programming (GP) implementation present in HeuristicLab is tree-based, an approach popularized by Koza [8]. In tree-based GP individuals consist of expression trees: data structures used to represent and evaluate mathematical (algebraic, boolean) expressions. In an expression tree leaf nodes contain operands, while internal nodes contain the operators. To evaluate the expression, the repre-

GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2881-4/14/07 ...$15.00.
http://dx.doi.org/10.1145/2598394.2605692.

sentation tree has to be traversed. In HeuristicLab, this functionality is implemented by the `Interpreter` class [7].

HeuristicLab provides three alternative implementations of the `Interpreter` class, they implement a common interface and they are therefore interchangeable to other `Operator` objects requiring their functionality; however their internal structure and logic are substantially different.

The `Interpreter`'s public method returns, as its output, the raw numerical values resulting from the evaluation of the individual on the dataset; these are then passed on to the `Evaluator` operator, where the fitness score is computed. This is an acceptable solution for regression and classification problems, both of which are prominent HeuristicLab use-cases, however it is not optimal for simulation problems.

Therefore, as the first step towards the implementation of a HeuristicLab plugin framework able to evolve symbolic Genetic Programming solutions for simulation problems, in this work we will describe an `Interpreter` with a modified public interface as well as a novel internal implementation and we will compare its performance against the `Interpreter` implementations provided together with HeuristicLab.

In Section 2 we will review the three `Interpreter` implementations provided with HeuristicLab and identify their weaknesses on the prospective use on simulation problems. In Section 3 we will describe our novel `Interpreter` implementation. In Section 4 we will evaluate the `Interpreters`' performance in terms of run execution times. In Section 5 we will draw conclusions and outline future steps.

2. GP INTERPRETERS IN HEURISTICLAB

HeuristicLab, as of version 3.3.9, provides three different `Interpreter` implementations able to process GP trees encoding mathematical expressions: `SymbolicDataAnalysisExpressionTreeInterpreter`, `SymbolicDataAnalysisExpressionTreeILEmittingInterpreter`, and `SymbolicDataAnalysisExpressionTreeLinearInterpreter`; these can be interchangeably used in the Symbolic Regression, Symbolic Classification, Time Series Evaluation and Trading problem templates.

The three symbolic tree interpreters all implement the `ISymbolicDataAnalysisExpressionTreeInterpreter` interface, exposing the `IEnumerable<double> GetSymbolicExpressionTreeValues(ISymbolicExpressionTree tree, Dataset dataset, IEnumerable<int> rows)` method; from the method's signature, it is evident how the `Interpreter` objects are not only performing tree interpretation and evaluation, but they are also responsible for computing and returning the output values given a dataset.

For the sake of completeness, we mention that HeuristicLab also features other `Interpreter` classes; these are however specific to the type of optimization problem they are used for and do not evaluate mathematical expressions, therefore we will briefly describe them in Section 2.5 but we will not be analyzing their performance.

2.1 Common Logic

Part of the three `Interpreter`'s internal workflow is common. In computing, an interpreter can be defined as a program that implements a virtual machine using a programming language as its own assembly language. As such, the three HeuristicLab interpreters all start by traversing the expression tree, parsing it into a linear, one-dimensional array of `Instruction` objects. `Instruction` is a data structure comprising an `OpCode` that specifies an operation and appropriate operands (data, number of arguments, addressing), as described by Kommenda et al. [7]. After the `Instruction[]` corresponding to the program has been generated, it is iterated a second time to specify the jump locations for flow control instructions, if any. Finally, it is iterated a third time to finalize input variable referencing: `Instruction` objects referring to variables are assigned, as their data field, the entire set of values for that variable contained in the problem `Dataset`.

`Instruction` objects therefore represent assembly-level instructions for the virtual machine the HeuristicLab GP individuals run on. The `Instruction[]` is produced and then processed differently for the three `Interpreter` implementations.

2.2 SymbolicDataAnalysisExpressionTreeInterpreter

Here to create the `Instruction[]` the tree is traversed in pre-order fashion, then the list is passed to the `Evaluate (Dataset dataset, ref int row, InterpreterState state)` method. `Evaluate` is recursive, calling itself on the following `Instruction` number of arguments times. Therefore its operations are analogous to a post-order evaluation of the original expression tree. `InterpreterState` is a class containing the `Instruction[]` and a simple call-stack implementation; this allows `SymbolicDataAnalysisExpressionTreeInterpreter` to support GP individuals featuring Automatically Defined Functions (ADFs).

`Evaluate` is called at the root level, and therefore the entire `Instruction[]` list is interpreted, once per evaluated row. This results in computational overhead proportional to the number of evaluated rows length, `SymbolicDataAnalysisExpressionTreeInterpreter` is therefore best suited for the evaluation of short datasets.

Throughout the rest of this paper this interpreter will be referred to as `Standard`.

2.3 SymbolicDataAnalysisExpressionTreeILEmittingInterpreter

`SymbolicDataAnalysisExpressionTreeILEmittingInterpreter`, from here on referred to as `IL`, strictly speaking, is an interpreter only partially. The class, before performing evaluation, first traverses the tree in pre-order, like `Standard`; however later it performs a recursive post-order parsing of the `Instruction[]`, translating it directly into Common Intermediate Language (CIL) instructions, making use of the `Reflection.Emit` capabilities of .NET.

CIL is is the lowest-level human-readable language defined by the Common Language Infrastructure specification and used by the .NET Framework; CIL is an object-oriented assembly language, and it is entirely stack-based [5].

Once all the CIL instructions for the program have been obtained, the .NET Just-In-Time compiler is called, returning an executable handle for the function in the form of a `delegate` method with input signature matching the `Dataset` variables. Therefore, `IL` is a compiler: it produces an executable function implementing the operations encoded in the original expression tree.

Successively, to evaluate the function, the delegate is invoked once per row to obtain the output values. In terms of computational overhead this approach results in an initial one for every GP individual, due to the invocation of the .NET compiler; however, as there is no re-interpretation or re-compilation for every row, such overhead is completely independent from the number of rows to be evaluated. For this reason `IL` is much more suited for the evaluation of longer datasets than `Standard`, the creators state 10,000 rows as the point where this interpreter starts performing better [7]. Currently, `IL` does not support ADFs.

2.4 SymbolicDataAnalysisExpressionTree-LinearInterpreter

`SymbolicDataAnalysisExpressionTreeLinearInterpreter`, from now `Linear`, is a refinement of `Standard`: they share the general concept as well as the interpreted approach. However, in this implementation `Instruction` is extended to `LinearInstruction`, adding a `value` and a `childIndex` field.

The initial tree traversal is here performed breadth-first, therefore the obtained `LinearInstruction[]` is in different order than with the other two interpreters. `childindex` is initialized to contain the instruction number for the first argument to the instruction while the others, given the traversal, will be found in the adjacent successive positions; `value` will be used at evaluation time to store partial results.

This implementation results in behavior analogous to that of a linear register machine: the `value` fields of `LinearInstruction` act as the linear registers. At evaluation time, the `LinearInstruction[]` is iterated starting from the bottom. This resolves the `value` fields in a leaves-to-root direction, ensuring by design that the value of arguments to a `LinearInstruction` have already been computed when the instruction is executed.

As a consequence of this, the `Evaluate(Dataset dataset, int row, LinearInstruction[] code)` method of this interpreter is recursion-free and executes the `LinearInstruction[]` entirely linearly. Therefore, the computational overhead due to row-by-row interpretation is substantially reduced compared to `Standard`. Currently, `Linear` does not support ADFs.

It is worth specifying that even though the individual evaluation is performed using a linear machine abstraction, the type of Genetic Programming in use is still Tree Genetic Programming, as opposed to Linear Genetic Programming (LGP) [1, 4]. That is because the GP Selection, Crossover and Mutation operators still act on the original tree representations.

2.5 Problem Specific Interpreters

In addition to the three interpreters described so far, HeuristicLab relies on this architectural pattern for other types of optimization problems as well. Examples of these are `Lawnmower.Interpreter` and `ArtificialAnt.AntInterpreter`. The interpreters for such other problems are, however, far less generic and generalizable. These do not compute mathematical expressions nor they output a list of numerical values: they operate instead on tree structures featuring problem-specific types of nodes and directly output an action to be performed in response to the inputs.

2.6 Limitations

As mentioned in Section 2, the symbolic expression interpreters' `GetSymbolicExpressionTreeValues(ISymbolicExpressionTree tree, Dataset dataset, IEnumerable <int> rows)` method returns the raw output numerical values resulting from the evaluation of the GP individual on the dataset; the `Evaluator` computing the fitness score assumes this behavior, as specified by the `ISymbolicExpressionTreeInterpreter` interface. Therefore, in order to comply to this specification, for a simulation-based evaluation the running of the simulation would have to be implemented within the `Interpreter`, providing the `Evaluator` with the values required to compute the individual's fitness.

For the Santa Fe Trail problem implementation [9], the `ArtificialAnt.Interpreter` mentioned in Section 2.5 is realized exactly in such a way: `ArtificialAnt.Interpreter` does not implement the `ISymbolicExpressionTree` interface but does instead expose a `Run()` method inside which the ant world simulation takes place, once `Run()` completes execution the fitness score is retrieved. The design works well if the simulation to be run is relatively simple, with few evaluation and performance measures. If the simulation problem or the evaluations and analyses to be performed are more complex instead, such design choice becomes constraining and intensive from a developmental and computational point of view:

- HeuristicLab does not provide a structured class and interface framework to generalize tree interpretation and simulation running within an `Interpreter`, so for every problem type `Interpreter`, `Evaluator`, and `Analyzer` classes would have to be written ad-hoc, in addition to the code for the simulation logic itself.

- Interpreting a symbolic expression and evaluating its fitness are two distinct and separate tasks, implementing them inside the same logical object implicitly promotes the emergence of unnecessary dependencies and greatly reduces the maintainability and reusability of the code base.

- Another drawback is the computational impact of having an interpreted approach, even if partial, as opposed to a compiled one: for very large population and dataset sizes, as it is common with simulation problems, the once-per-row overhead of interpreted GP individuals becomes dominant with respect to the initial, once-per-individual, overhead of compiled individuals; resulting in substantially longer execution times for algorithm runs.

- Finally, in the case GP individuals are not only to have access to external, provided inputs, but to simulation state variables depending on their own actions on previous datapoints as well, it follows that it is impossible to obtain these internal state values without running the simulation itself, which is also what ultimately produces the fitness score, de facto coinciding with the intended functionality of `Evaluator`.

Given that the architecture of HeuristicLab strongly relies on interface and inheritance mechanisms to provide as much generalization as possible, we think that a framework of classes providing the necessary abstractions enabling HeuristicLab to internally run GP encoding mathematical expressions on simulation problems fits well with HeuristicLab's vision and purpose, as it would add a very general and useful problem type to its capabilities.

To achieve such goal, partial reimplementation of `Interpreter`s, `Evaluator`s, and `Analyzer`s is required. We will begin from the `Interpreter` class because it is the first one to be employed in HeuristicLab's execution workflow.

3. .NETEXPRESSIONINTERPRETER

3.1 Design Overview

As stated in Section 2.6, our goal is to have compiled GP individuals as well as separating interpretation/compilation from evaluation, therefore our interpreter does not implement the `ISymbolicDataAnalysisExpressionTreeInterpreter` interface and the `GetSymbolicExpressionTreeValues` method described in Section 2.

Our interpreter instead is designed to interface with its `Evaluator` via the `Func<int, double[][], double[], double> GetCompiledFunction(ISymbolicExpressionTree tree)` method. In C#, `Func` is an anonymous, generic function type [11]; the `GetCompiledFunction` method therefore outputs an executable function with `int`, `double[][]`, and `double[]` arguments and with `double` return value.

It will be then the `Evaluator` to be responsible for evaluating the individual on the problem data, running the simulation.

3.2 .NET Expressions

.NET Expression Trees are data structures used to represent, construct, compile and run code itself [10]. In an Expression Tree, nodes are constituted by `Expression` objects. `Expression` objects can be used to represent many code constructs including constant values, variables, mathematical operations, method calls, control flow statements, scope blocks, etc... [12].

`Expression` trees can be constructed programmatically by combining single `Expression` objects, passing them as constructor parameters to others, resulting in a tree structure. Finally, an `Expression` tree can be converted into a lambda expression, specifying one or more `ParameterExpression` as its argument(s), and then compiled into an executable delegate which, invoked, executes the function encoded by the original tree.

The code in Listing 1 shows a simple example of this process for an expression tree representing the sum of two elements of an array.

.NET Expression Trees allow the conversion of a tree data structure into executable code. Given how closely this behavior matches with the concept of Tree GP, we chose to rely on .NET Expression Trees for the implementation of our `Interpreter` class: `DotNETExpressionInterpreter`.

Compared with emitting CIL instructions and then calling the compiler on them, as in the case of `IL`, the use of `Expression` trees entails a slightly larger initial computational overhead: in the former case assembly instructions are already specified so the compiler is invoked directly, while in the latter the `Expression` tree has to first be parsed into CIL and compiled subsequently. However, the use of `Expression` trees give several advantages over explicit CIL emission:

```
ParameterExpression inputs =
    Expression.Parameter
      (typeof(double[]), "inputs");

BinaryExpression i1 =
    Expression.ArrayIndex
      (inputs, Expression.Constant(0));

BinaryExpression i2 =
    Expression.ArrayIndex
      (inputs, Expression.Constant(1));

BinaryExpression sum =
    Expression.Add(i1, i2);

Expression<Func<double[], double>> lambda =
  Expression.Lambda<Func<double[], double>>
    (sum, new ParameterExpression[] {inputs});

Func<double[], double> method = lambda.Compile();

double[] values = {1, 2};

method(values);
```

Listing 1: Construction, parameterization, compilation and invocation of a .NET Expression Tree implementing the sum of two numbers

- The `Interpreter`'s own code is much more readable and understandable
- Expressions can be visualized and printed in linear, parenthesized, infix form via their `ToString()` method, therefore the correctness of the tree building process can be easily verified
- Considering how `Expression` objects can represent arbitrary method calls, it is in principle possible to rely on such technique to support ADFs
- Relieving the developer from directly writing CIL instructions makes it easier to implement functionality for tree nodes representing more elaborate instructions

3.3 Workflow

1. Our `DotNETExpressionInterpreter` has a pre-processing step, shown in Listing 2, executing only when the `Dataset` object for the problem setup is modified. Here the names of the `Dataset` variables are mapped to `ParameterExpression` objects containing their column index, later allowing to build the `Expression` tree using parameterized input arguments whose value will only be specified when the already compiled tree is evaluated.

2. `DotNETExpressionInterpreter` parses the HeuristicLab `SymbolicExpressionTree` recursively, in post-order; each recursive call on a node returns the compound `Expression` representing that node's subtree including itself. No additional tree traversal is needed to produce the final `Expression` tree, as the input variables have been mapped in pre-processing.

   The code in Listing 3 shows parts of the tree parsing logic exemplifying the processing for different types of nodes.

```
...

private Dictionary<string, Expression> paramVarMap;
private Dictionary<string, Expression> inputsVarMap;

...

private void UpdateInputsMap(IEnumerable<string> vars)
{

  paramVarMap = new Dictionary<string, Expression>();
  inputsVarMap = new Dictionary<string, Expression>();

  ParameterExpression varParam =
      Expression.Parameter
          (typeof(double[][]), "variables");

  ParameterExpression rowIndexParam =
      Expression.Parameter
          (typeof(int), "rowIndex");

  paramVarMap.Add("variables", varParam);
  paramVarMap.Add("rowIndex", rowIndexParam);

  List<string> varNames = vars.ToList();
  for (int i = 0; i < vars.Count(); i++)
      inputsVarMap.Add
          (varNames[i], Expression.Constant(i));

}
```

Listing 2: DotNETExpressionInterpreter pre-processing stage. Dataset variable names are mapped to ParameterExpression objects, later enabling the compiled Expression tree to be evaluated on arbitrary inputs

3. The tree, as described earlier, is then wrapped into a LambdaExpression.

4. The .NET compiler is invoked on the LambdaExpression.

5. DotNETExpressionInterpreter outputs a compiled `Func<int, double[][], double[], double>`, mapped to a `double CompiledFunction(int rowindex, double[][] data, double[] states)` for better typed access.

   `double[][] data` is the entire problem dataset in matrix form (HeuristicLab's Dataset was modified to provide such accessor), `int rowindex` represents the row to be evaluated in the call, `double[] states` is the input vector representing the internal simulation state variables for the problem.

6. At Evaluation time a compatible Evaluator will be responsible for running the simulation, making use of the now compiled individual.

4. PERFORMANCE ANALYSIS

In order to comparatively evaluate the performance of the three interpreters described in Section 2 as well as DotNETExpressionInterpreter (.NET), we wrapped the latter in logic implementing the
ISymbolicDataAnalysisExpressionTreeInterpreter
interface, enabling it to run on HeuristicLab's problem templates without the need for developing additional code.

```
private Expression RecursiveParser
    (ISymbolicExpressionTreeNode node)
{
    ISymbol s = node.Symbol;
    Expression expr = null;

    if (s is Addition)
    {
      Expression left =
          RecursiveParser(node.GetSubtree(0));
      Expression right =
          RecursiveParser(node.GetSubtree(1));
      expr = Expression.Add(left, right);
    }
    ...
    else if (s is Sine)
    {
      Expression sub =
          RecursiveParser(node.GetSubtree(0));
      expr = Expression.Call
          (null, typeof(Math).GetMethod("Sin"), sub);
    }
    ...
    else if (s is Variable)
    {
      VariableTreeNode vn = node as VariableTreeNode;
      Expression ve = Expression.ArrayIndex
          (Expression.ArrayIndex
              (paramVarMap["variables"],
               paramVarMap["rowIndex"]),
          inputsVarMap[vn.VariableName]);

      if (vn.Weight != 1)
      {
        Expression we = Expression.Constant(vn.Weight);
        expr = Expression.Multiply(ve, we);
      }
      else
        expr = ve;
    }

   return expr;
}
```

Listing 3: DotNETExpressionInterpreter parsing logic for tree nodes representing a binary operator (Addition), a unary operator (Sine), and a terminal node (Variable)

The fraction of the total GP algorithm run time taken up by tree interpretation and evaluation is proportional to the number of dataset rows, while the execution time for other algorithm steps such as tree creation, selection, crossover, mutation, etc... is not dependant on it. In the case of HeuristicLab Kommenda et al. note that tree interpretation and evaluation dominate the total runtime for datasets exceeding 1,000 rows [7]. Since we will be analyzing cases with no less than 5,000 dataset rows, such considerations allow us to approximate the evaluation of the interpreters' performance to the simple measuring of the total algorithm execution time.

All the presented evaluations were performed on an Intel Core i7-2600K machine set at a frequency of 3.8 GHz with 16GB of RAM.

4.1 Population Size

The performance of all four interpreters was tested by running the Genetic Programming - Symbolic Regression problem template on the Real World Benchmark Problems - Tower dataset, both provided with HeuristicLab.

Common settings for the runs are 20 calculated generations, a dataset length of 5,000, a parallelization setting of

Population	Interpreter			
	Standard	Linear	IL	.NET
1,000	34.2 (1.2)	20.2 (1.3)	20.3 (1.2)	19.6 (0.8)
2,000	79.8 (0.6)	43.7 (0.4)	43.9 (0.5)	42.4 (0.6)
5,000	253.1 (0.4)	125.2 (0.5)	128.3 (0.5)	121.3 (0.1)
10,000	421.5 (0.4)	225.1 (0.2)	233.5 (0.7)	221.3 (0.7)

Table 1: Average (s) and relative standard deviation (%) values of the algorithm run times for different population sizes

Dataset Length	Interpreter		
	Linear	IL	.NET
5,000	103.4 (2.7)	103.0 (0.2)	99.1 (0.5)
10,000	166.9 (1.0)	144.7 (0.3)	136.7 (0.5)
15,000	234.3 (0.4)	192.4 (0.4)	176.4 (0.5)

Table 2: Average (s) and relative standard deviation (%) values of the algorithm run times for different dataset lengths

4, probabilistic tree creator with maximum tree size 150, and a fixed, constant seed value to ensure the interpreters evaluate the very same expression trees. For each of the four interpreters we performed 5 runs for each of the following population sizes: 1,000, 2,000, 5,000, 10,000. Table 1 shows the average and relative standard deviation of the obtained execution times.

Confirming the observation by Kommenda et al. in [7], for such dataset lengths the run times increase linearly with respect to the total number of tree evaluations performed, this results to be true regardless of the interpreter.

It is evident how `Standard` is consistently the slowest by a considerable margin: on average `Standard` times are 1.8 longer. The overhead with respect to the others also appears to be increasing with growing population sizes. This first evaluation demonstrates how the `Standard` interpreter is not suited for large population runs and therefore we will not be evaluating it further.

`.NET` is the best performing across all population sizes, even if by a small margin compared to `IL` and `Linear`.

4.2 Dataset Length

The Tower dataset used previously only provides 5,000 datapoints, so to investigate the relation between dataset length and execution times we used the Bike Sharing dataset [6]. We evaluated the `Linear`, `IL`, and `.NET` interpreters on the on the first 5,000, 10,000, and 15,000 rows of the dataset, performing 3 runs per setting. Population size is set at 5,000; other settings are the the same as the ones in Section 4.1. The results for this evaluation are reported in Table 2.

Kommenda et al. mention 10,000 as the number of dataset rows where the use of `IL` is recommended over `Standard` and `Linear` [7]; on our results such threshold appears to be at a lower row count: at 10,000 rows `Linear` is already 15% slower than `IL`.

The values clearly show how `Linear` goes from having a comparable performance to `IL` and `.NET` for 5,000 datapoints (4% slower than `.NET`, 0.4% than `IL`), to being considerably worse at 15,000 (33% slower than `.NET`, 22% slower than `IL`). This growing gap is explained by the inherent per-datapoint overhead of having an individual interpreted, as opposed to compiled; therefore, `Linear` too would not be a good choice for large population, long dataset runs.

To explain the increasing performance difference between `IL` and `.NET` on this test: a 3% difference for 5,000 rows and a 9% one for 15,000, one possibility could be the way the variables are accessed at evaluation time. `.NET`'s input is a two-dimensional array; while `IL` partially relies on a list, bound to have a slight performance overhead over the former when accessing elements.

Tree Length	Interpreter		
	Linear	IL	.NET
100	76.9 (0.2)	81.1 (0.3)	77.5 (0.7)
150	159.3 (1.1)	157.8 (0.3)	148.5 (0.3)
200	435.3 (0.3)	386.1 (0.4)	341.7 (0.3)

Table 3: Average and relative standard deviation (%) of the algorithm run times for different tree sizes

4.3 Tree Length

Another factor affecting interpreter performance is length of the symbolic trees; we analyzed the execution times of `Linear`, `IL` and `.NET` changing only the maximum allowed tree size: 100, 150, 200. This evaluation was performed on the 5,000 rows Tower dataset, with a population size of 5,000. In order to magnify the effect of tree length on performance for this run we utilized the full tree creator rather than the probabilistic one. Table 3 displays the results.

`Linear` exhibits a behavior analogous to the one it has in the evaluation on different dataset lengths described in Section 4.2: it goes from having a very slight, sub-second, performance advantage for length 100, to a very slight overhead at 150, to lastly obtaining significantly slower times than the other two for tree length 200. The increasingly worse performance is still because of a per-datapoint overhead, this time due to longer, deeper, trees requiring more time to be parsed rather than having more rows to evaluate.

`.NET` results to be the best performer on this evaluation as well, the performance gains with respect to both `Linear` and `IL` positively correlate with tree size.

The growing difference between `IL` and `.NET` could be explained by the fact that for the former the starting `SymbolicExpressionTree` is parsed three times to arrive to its compiled form; in the case of the latter instead it is explicitly parsed once in the interpreter and another time by the compiler, for a total of two. Furthermore, considering that in `IL` the logic for CIL translation was coded manually, another possibility could be the presence of suboptimalities in the generated CIL instructions, leading to an increasing overhead over a compiler-optimized CIL the more instructions, and correspondingly tree nodes, there are.

5. CONCLUSION

The high sensitivity of `Standard` to population size and of `Linear` to dataset length demonstrate how a compiled execution of GP individuals is crucial to have usable performance in large population, long dataset scenarios.

Considering how our `DotNETExpressionInterpreter` performs better than `IL` in all three performed evaluations and furthermore that the difference in performance is also proportional to increasing population size, dataset length, and tree length, we consider our implementation a success.

The shown good performance scaling is not only of key importance in the prospected context of simulation-based problem GP evaluations within HeuristicLab; but it can also be useful in large population, long dataset setup for more conventional problem types such as symbolic regression and symbolic classification.

An issue not addressed in this work that could still pose a challenge for GP runs entailing a very large number of evaluations is that dynamically emitted code, as both `IL` and `.NET` do, is not removed by the .NET Garbage Collector; potentially leading to system memory exhaustion before the optimization process is complete. Therefore, further examination of this phenomenon is required to assess its quantitative impact.

The following steps for the implementation of the entire simulation-problem framework consist in the establishing of common Interfaces and reimplementation of Evaluators and Analyzers.

6. REFERENCES

[1] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic programming: an introduction*, volume 1. Morgan Kaufmann San Francisco, 1998.

[2] A. Beham et al. Simulation optimization with HeuristicLab, 2008.

[3] A. Beham, M. Kofler, S. Wagner, and M. Affenzeller. Coupling simulation with HeuristicLab to solve facility layout problems. In *Winter Simulation Conference*, WSC '09, pages 2205–2217. Winter Simulation Conference, 2009.

[4] M. F. Brameier and W. Banzhaf. *Linear genetic programming*. Springer, 2007.

[5] ECMA. *Common Language Infrastructure (CLI) Partitions I to VI*, chapter Partition III: CIL Instruction Set, pages 290–432. ECMA International, 6th edition, June 2012.

[6] H. Fanaee-T. Uci machine learning repository: Bike sharing dataset data set. `https://archive.ics.uci.edu/ml/datasets/Bike+Sharing+Dataset`, March 2014.

[7] M. Kommenda, G. Kronberger, S. Wagner, S. Winkler, and M. Affenzeller. On the architecture and implementation of tree-based genetic programming in HeuristicLab. In *Proceedings of the fourteenth international conference on genetic and evolutionary computation conference companion*, pages 101–108. ACM, 2012.

[8] J. R. Koza. *Genetic Programming: vol. 1, On the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[9] J. R. Koza. *Genetic Programming: vol. 1, On the programming of computers by means of natural selection*, volume 1, pages 147–155. MIT press, 1992.

[10] MSDN. Expression Trees (C# and Visual Basic). `http://msdn.microsoft.com/en-us/library/bb397951.aspx`, March 2014.

[11] MSDN. Func(T1, T2, T3, TResult) Delegate. `http://http://msdn.microsoft.com/en-us/library/bb549430%28v=vs.110%29.aspx`, March 2014.

[12] MSDN. System.Linq.Expressions Namespace. `http://msdn.microsoft.com/en-us/library/system.linq.expressions.aspx`, March 2014.

[13] S. Wagner and M. Affenzeller. HeuristicLab: A generic and extensible optimization environment. In *Adaptive and Natural Computing Algorithms*, pages 538–541. Springer, 2005.

[14] S. Wagner, G. Kronberger, A. Beham, M. Kommenda, A. Scheibenpflug, E. Pitzer, S. Vonolfen, M. Kofler, S. Winkler, V. Dorfer, and M. Affenzeller. *Advanced Methods and Applications in Computational Intelligence*, volume 6 of *Topics in Intelligent Engineering and Informatics*, chapter Architecture and Design of the HeuristicLab Optimization Environment, pages 197–261. Springer, 2014.