The Programming Game: Evaluating MCTS as an Alternative to GP for Symbolic Regression

Author details removed
for blind review.

ABSTRACT

Monte Carlo Tree Search (MCTS) is an effective and relatively new technique for searching game trees and making decisions in decision processes. Previous work by Cazenave has demonstrated that MCTS can be applied to symbolic regression. In this paper, we implement and apply two MCTS algorithms to the problem of symbolic regression, and empirically evaluate their performance on four standard regression benchmarks. We compare them to standard Genetic Programming (GP), and find that MCTS algorithms can achieve results competitive with GP for some benchmarks. We conclude by outlining future work we consider promising for the application of MCTS to symbolic regression and program search in general.

Categories and Subject Descriptors

SD I.2.8 [**Artificial Intelligence**]: Heuristic methods

General Terms

Algorithms, Experimentation, Performance

Keywords

Genetic Programming, Monte Carlo Tree Search, UCT, Nested Search, Symbolic Regression

1. INTRODUCTION

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm designed to explore game trees, which has achieved impressive results in a number of applications, most notably in playing the game of Go [6, 11]. It is ideally suited to solving problems where it is difficult to assess the value of intermediate states, such that deciding "what to do next" cannot be solely made by evaluating the states that would immediately result as a consequence of taking a given action.

Instead, like other Monte Carlo methods, MCTS algorithms rely on a combination of heuristics and stochastic playouts to assess the likely result of taking an action from the current state: beyond a certain distance of lookahead, an estimate of the quality of the decision to be taken is gathered by randomly playing the remaining moves and assessing the final state, such as whether it is a win or loss state. This outcome is then used to attribute rewards to the decisions taken prior to this random playout.

Although MCTS algorithms were originally designed to play games, they can be generalised beyond this domain [7]. Cazenave [9, 10] demonstrates that MCTS can be used for symbolic regression, and this paper is based on his work.

We follow and extend Cazenave's approach and make comparisons with Genetic Programming (GP) on four symbolic regression problems. We propose that MCTS methods can be used for program search in general, and suggest extensions to Cazenave's work that enable this generalisation.

GP maintains a population of solutions, making it a global search algorithm in a way that MCTS is not, and it has a track record of success in areas beyond the scope of previous MCTS applications [1]. However, MCTS offers some advantages over GP: it is inherently resistant to bloat, it has a sound mathematical underpinning, is relatively easy to understand and implement, and the exploration – exploitation trade-off can be specified and controlled explicitly.

2. MCTS

Monte Carlo methods comprise a wide collection of computational algorithms that depend on repeated random sampling [16]. The primary goal of sampling is to approximate a probability distribution, the direct computation of which is either intractable or nondeterministic. The obtained probability distribution can, in turn, be used to make various decisions. Intuitively, the reward of a single action can be approximated by sampling random playouts that follow the particular action, and evaluating their final outcomes. If an action A is *better* than B, than samples of random playouts that follow the action A will, on average, end more favourably than those following B.

Whilst individual variants differ in detail, intuitively all MCTS methods seek to *grow* a tree by repeatedly deciding which node to add, based on the approximated reward.

In order to apply MCTS to symbolic regression, and indeed programming problems, we consider the incremental development of a program. At each step, a function (terminal or nonterminal) is pushed onto a stack representation of the program as a symbolic expression. Thus the act of programming becomes a single-player game. Each node in the game tree branches according to the possible functions that may be used at that point in the abstract syntax tree. This incremental nature of MCTS makes it resistant to bloat.

The best survey of MCTS is [7], and we direct the reader to that paper for an excellent overview of the field. We borrow heavily from the survey's approach in structuring our pseudocode. MCTS is actually a family of algorithms, and in this paper we discuss two main variants: the first is the Upper Confidence Bound for Trees (UCT) algorithm [13], perhaps the most well-known MCTS algorithm, and the second is the Nested Tree Search algorithm proposed by Cazenave [8]. We introduce UCT first, because it provides a strong intuition of how MCTS operates in general.

In both algorithms, expressions (solutions, programs) are represented as a stack consisting of operators immediately followed by their operands (i.e. Polish notation). A stack representation is helpful in ensuring straightforward pseudocode, and also reduces the confusion between the search tree — the tree of all possible expressions — and the expression tree, that is an individual program or solution.

In traditional MCTS, game trees describe the possible actions and rewards from the current state, and here we refer to such a tree as the search tree. The process of completing a program by adding stochastically chosen functions from the function set is a playout. An example of a playout is given by the code in Algorithm 1.

Algorithm 1 Pseudocode for a typical MCTS playout
```
function PLAYOUT($stack$)
    while $stack.leaves > 0$ do
        if $stack$.reached_max_size() then
            $symbol$ = random.choice($terms$)
        else
            $symbol$ = random.choice($terms + nonterms$)
        end if
        $stack$.push($f$)
    end while
    return eval($stack$)
end function
```

In general, MCTS works by incrementally constructing the game tree through a combination of enumeration and sampling, and then updating the tree based on the outcome of games sampled. The explanation below, in combination with Figure 1, will clarify this description.

2.1 Upper Confidence Tree

We provide a new implementation of UCT [13] for expression search, given in Algorithm 2. This is consistent with the general structure from [7]. We omit the playout function, similar to that in Figure 1, and also the best child function, which simply chooses the best child according to the Upper Confidence Bound (UCB) heuristic discussed below.

The UCT algorithm works by repeatedly updating the search tree that is held in memory. Each iteration of tree growth works as follows:

1. The search tree is descended from the root to a leaf, making choices of which child to follow (and hence which function to push onto the expression stack) based on a formula that specifies the balance between exploration and exploitation.

2. When a leaf node is found that has yet to be fully expanded, i.e. not all possible functions have been considered, a new node is added to the search tree that represents one of the possible functions to push onto the stack.

3. A playout is then made from the new node by stochastically choosing functions from the function set to complete the expression, and the resulting stack is evaluated to ascertain its score.

4. Finally, the score is used to update the search tree: each node that was navigated when creating the final expression is updated, to keep a record of the number

Algorithm 2 UCT Algorithm
```
function UCT
    $root$ = new Tree_Node()
    repeat
        if $root$.fully_explored then
            Break
        end if
        $leaf$ ← tree_policy($root$)
        $score$ ← playout($leaf.stack$)
        backup($leaf$, $score$)
        if $leaf$.is_terminal() then
            $leaf.fully\_explored ← True$
        end if
    until exhausted evaluation budget
    return $root$
end function
function TREE_POLICY($root$, $stack$)
    while $stack.leaves$>0 do
        if not $node.all\_functions\_tried$ then
            $new\_child$ ← expand($node$)
            $stack$.push($new\_child.node\_function$)
            if $stack.leaves$ == 0 then
                $new\_child.explored ← True$
            end if
            return $new\_child$
        else
            $node$ ← best_child($node$)
            $stack$.push($node.node\_function$)
        end if
        return $node$
    end while
end function
function EXPAND($node$)
    $new\_node$ ← new Tree_Node()
    $new\_node.function$ ← $node$.next_function()
    $new\_node.parent$ ← $new\_node$
    $node$.add_child($new\_node$)
    return $new\_node$
end function
function BACKUP($node$, $score$)
    while $node$ is not null do
        $node.visits ← node.visits + 1$
        $node.sum\_scores ← node.sum\_scores + score$
        if $node$.fully_expanded() and
            children fully explored then
            $node.explored ← True$
        end if
        $node ← node.parent$
    end while
end function
```

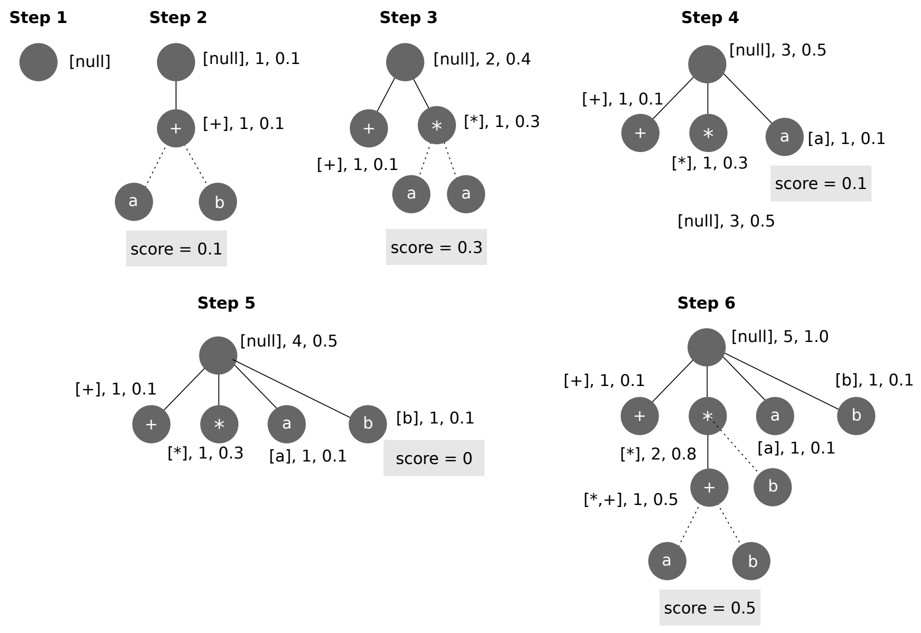

Figure 1: Six steps of a UCT Search for the language $\{+, *, a, b\}$. Dotted lines indicate playouts. Each node is accompanied by the expression stack, number of visits, and sum of scores for that decision.

of visits and the total score received. This information is later used in the UCB formula.

In addition, we modify the algorithm to keep track of fully expanded sections of the search tree, where all possible subexpressions have been tried. It then avoids re-sampling them, as their true reward is known.

Figure 1 illustrates the first six steps of the search for a very simple expression language using the function set $\{+, *, a, b\}$. At Step 1 the search tree is initialised with an empty root. At Step 2, the first possible function is considered: this is pushed onto the stack and a new search node is added as a child of the root. The rest of the expression is filled with randomly chosen functions, evaluated, and the score of 0.1 is propagated up the tree, along with the number of visits at each node. In Step 3, the second possibility of multiplication is considered instead, and the expression completed by adding two functions to the stack (both terminals, for brevity, but they could be any sequence of functions that results in a complete expression).

At Step 4, the only possible option is a terminal and hence no playout is required, only an evaluation. The same situation is found in Step 5. Having now fully expanded the root node, at Step 6 the best child is selected according to the UCB heuristic. In this case it is the multiply child, which is then expanded by an addition child, and the stack is completed with a random playout that adds terminals (again, for brevity). Note that the information held in the tree is complete, in that it can be used to restart the algorithm.

The UCB heuristic used at Step 6 selects the child that maximises the following formula:

$$\frac{S_c}{n_c} + K\sqrt{\frac{2\ ln\ n_c}{n_p}} \tag{1}$$

Where S_c is the sum of scores for the child, n_c and n_p the number of times the child and its parent have been visited, and K is the UCB constant that tunes the balance between exploration and exploitation. The first term encourages exploitation of nodes that have previously led to good scores, whereas the second term encourages exploration of infrequently visited nodes.

2.2 Nested Tree Search

Pseudocode for Nested Tree Search [8] is given in Algorithm 3. Nested tree search does not explicitly build a search tree. Instead, it works by continually building complete expressions. At each iteration, an empty stack is created, and functions are added to that stack based on the performance of a Nested Search that begins with a stack containing the function being considered. Having completed one recursive Nested Search for each possible function, the function that resulted in the best score is chosen and pushed permanently onto the stack. The process is repeated until the stack contains a complete expression.

The algorithm is a combination of selective enumeration up to a certain expression length, followed by random sampling using a Monte Carlo playout below level one. The functions considered for insertion to the stack at each step are restricted to those that will not make the expression too large. The parameter `level` determines how deeply the expression space will be enumerated before invoking playouts. In contrast, UCT incrementally constructs the game tree, by relying on random playouts to guide its construction.

3. IMPLEMENTATION

We implemented UCT and Nested Tree Search in Java 8. We provide all our code and an extensive junit test suite on-

Algorithm 3 Nested Algorithm

```
function NESTED($maxLen, levels$)
    $stacks \leftarrow [[]]$                                    ▷ List of empty stacks
    nested($levels, stacks$)
    return $stacks[levels]$
end function
function NESTED_INNER($level, stacks$)
    $best \leftarrow []$
    $bestScore \leftarrow 0$
    while $stacks[level].leaves > 0$ do
        for each $f$ in function set do     ▷ Consider functions that won't make our expression too long
            if $stacks[level].size + stacks[level].leaves < maxLen$ or $f.arity == 0$ then
                $stack.push(f)$
            end if
            if $level == 1$ then
                $score \leftarrow$playout($stacks[level]$)          ▷ Below a certain level, playouts only
            else
                $score \leftarrow$nested_inner($level - 1, stacks$)   ▷ Otherwise, recursive search at the next level
            end if
            if $score > bestScore$ then   ▷ Store best expression found, as may be from a playout and otherwise discarded
                $best \leftarrow stacks[level - 1]$
                $bestScore \leftarrow score$
            end if
            $stacks[level].pop()$
        end for
        $stacks[level].push(best.get(stacks[level].size))$   ▷ Select the function that resulted in the best nested search result
    end while
    return $score(stacks[level])$
end function
```

line [5]. We also evaluate GP; we use the version of ECJ [2] provided as part of the GPBenchmarks.org project [14, 20], which is available via that project's website [3]. This code includes an implementation of the benchmarks we employ, which we use as a reference implementation for our MCTS code. For example, the way we calculated raw and normalised error, the test and training sets for the benchmarks, and the target expressions themselves, were all based on the code in the ECJ implementation. We also reuse the implementation of the `MersenneTwisterFast` pseudorandom number generator from ECJ, so that all algorithms employ the same method of random number generation.

In order to apply MCTS to symbolic regression, we must support ephemeral random constants (ERCs). In ECJ, these constants are assigned random values when the initial population is created, and also when new genetic material is created as part of a mutation operation. In MCTS, a decision to push an ERC onto the stack may occur during systematic expansion of the game tree, or during a random playout. In both cases, at the point that an ERC is pushed onto the stack (`stack.push`) in our pseudocode, we initialise it in such a way as the value is frozen into the game tree. Thus, revisiting a node in a game tree that utilises an ERC is guarantee to have the same result. ERCs used in playouts are discarded, adding to the uncertainty to the algorithm's estimate of the value of an intermediate state.

To compare the three algorithms, we needed to be able to control the number of fitness evaluations each algorithm executed. We consider this more significant than wall-clock time, as all our implementations could have been optimised in many ways, and it is not yet clear how best to adapt MCTS for expression search. ECJ was the slowest of the implementations, but it is not optimised for speed.

UCT takes as input a maximum number of evaluations. ECJ parameters are set to ensure $p * g = E$, where p was the population size, g the number of generations, and E the total number of evaluations we allocated to each algorithm in its budget. In Nested Search, however, the amount of work performed is determined by a combination of factors such as the number of levels to be systematically explored, the maximum length of the expression, and the composition of the function set. We therefore modified the implementation to throw an exception on exhausting its allocation budget. In all cases, the test set fitness of the best individual found throughout the search is used to quantify the success of the algorithm in solving each benchmark.

4. EXPERIMENTAL METHOD

We now apply UCT, Nested Search, and a straightforward Genetic Programming implementation in ECJ to four benchmarks taken from GPBenchmarks.org's recommendations. Due to the large number of parameter settings available within GP, and the many different ways MCTS could be adapted to this problem, we caution against generalisation, but we do think that it illustrates the potential of MCTS in contributing to the solution of problems usually tackled using GP alone.

4.1 Parameter Settings

Parameters for ECJ are summarised in Table 1. The population size is set to the ECJ default, but is included for clarity. The other parameters were selected based on a review of the papers in which our benchmarks originally appeared,

Parameter	Value
Population Size (p)	1024
Generations	max_evals / p
Probability of Crossover	0.9
Probability of Mutation	0.1
Tournament Size	7
Initial Population Tree Depth	5

Table 1: ECJ Parameter Settings. All other parameters were left at their defaults values.

such that these settings can be considered representative; all values were either explicitly used in the papers or lie within the range of values that the papers used overall.

One of the advantages of MCTS compared to GP is the relatively small number of parameters that must be configured. For Nested Search, all that is required is a maximum expression length (set to 35 as per Cazenave's work) and the variable "level", which determines the depth to which the expression space will be enumerated. For "level", we simply chose the value 4, the smallest such value so that the search would not terminate before completing a million evaluations on any benchmark.

Similarly, for UCT there are two parameters: the maximum length of an expression (again 35), and the constant K, which determines the balance between exploration and exploitation. For this constant, we use the popular value $1/\sqrt{2}$ as per Kocsis and Szepesvári [13].

We must also specify how many evaluations the search is allowed to perform: replicating Cazenave's approach, we limit each algorithm to a maximum of 2^{20} i.e. just over a million evaluations. The wallclock times of ECJ runs were the longest, although all algorithm execution times were within a magnitude of each other. Given that ECJ is not optimised for speed, and a stack representation is both more space-efficient and easier to evaluate, we consider comparisons based on evaluations to be more principled.

When employing GP, we must limit our budget of evaluations to a maximum of 2^{20}, and hence divide those evaluations between the population size and the number of generations. We kept the population size of GP to the default of 1024, and adjusted the number of generations accordingly.

We ran with evaluation limits of $2, 4, 8 \ldots 2^{20}$, and each measurement was taken on a *distinct* search run. Thirty repetitions were made at each evaluation budget.

We measured the success of each algorithm by recording the best expression found. We re-tested the expression on the test set (where applicable), and the total error across test cases was taken as raw fitness R. To normalise this, we set normalised fitness to $\frac{1}{1+R}$. This transforms the problem into a maximisation one, suitable for MCTS, which expects evaluation functions to return rewards.

All the seeds used to initialise the RNGs in our experiments were taken from Random.org [4].

4.2 Benchmarks

The details of our benchmarks are given in Table 2. The ERC function in the Keijzer6 benchmark is a random number sampled from $N(0.0, 5.0)$; variations of ERCs in Vladislavleva4 are three unary functions involving a single ERC, sampled from $[0.0, 5.0]$ with uniform distribution, respectively.

Benchmark	Target Function	Function Set
Keijzer6 [12]	$\sum_{i=1}^{x} 1/i$	`ADD`, `MULTIPLY`, `INV`, `NEG`, `SQRT`, `ERC`
Nguyen7 [17]	$ln(x+1) + ln(x^2+1)$	`ADD`, `MULTIPLY`, `DIVIDE`, `SUBTRACT`, `SIN`, `COS`, `LOG`, `EXP`
Pagie1 [15]	$\frac{1}{1+x^{-4}} + \frac{1}{1+y^{-4}}$	`ADD`, `MULTIPLY`, `DIVIDE`, `SUBTRACT`, `SIN`, `COS`, `LOG`, `EXP`
Vladislavleva4 [18]	$\frac{10}{5+\sum_{i=1}^{5}(x_i-3)^2}$	`ADD`, `MULTIPLY`, `DIVIDE`, `SUBTRACT`, `SQUARE`, $\texttt{ERC}_A$, $\texttt{ERC}_B$, $\texttt{ERC}_C$

Table 2: Benchmark symbolic regression problems from White et al. [20], and the available function sets.

Benchmark	Test Set	Training Set
Keijzer6 [12]	E[1, 50, 1]	E[1,120,1]
Nguyen7 [17]	U[0, 2, 20]	Same
Pagie1 [15]	E[-5, 5, 0.4]	Same
Vladislavleva4 [18]	U[0.05, 6.05, 1024]	U[-0.25, 6.35, 5000]

Table 3: Benchmarks. Table based on Table 5 in [20]. U[a,b,c] indicates c uniform random samples in $[a, b]$. E[a,b,c] is the grid of points between a and b interval c.

Table 3 shows how training and test data sets are configured. Nguyen's test set is separate from its training set; the other two use the same data for both training and testing.

5. RESULTS

Our results are composed of the 30 best individuals from each algorithm for each number of evaluations. The GP results begin at 1024 evaluations, as that was the population size we used and hence the minimum number of evaluations possible. We tested the distributions of fitnesses for each algorithm-benchmark pair using a Shapiro-Wilk test and found them to be non-normal, hence we plot the median rather than the mean in our graphs. We also show the interquartile range as a indicator of dispersion.

Our results are given in Figures 2, 3, 4 and 5, where we plot median normalised fitness at each number of evaluations. We performed a Mann-Whitney-Wilcoxon nonparametric significance test on the results of the runs with 2^{20} evaluations. The results are given in Table 4, with values significant at the 0.05 level in bold.

UCT performs poorly on all benchmarks. We expect that this is because UCT assumes normally distributed rewards in the game tree, i.e. that the set of expressions beginning with the same function should have normally distributed fitnesses. We expect that this is not the case for expression spaces. This is self-evident for very small subtrees, but would require further effort to investigate more generally.

Nested Search performs as well as GP for two benchmarks, Keijzer and Nguyen, but it is outperformed by GP on the Pagie and Vladislavleva benchmarks. We conjecture that the reason for GP's superior performance on two of the benchmarks is related to the process of random playouts. As we have here intentionally implemented a naive playout algorithm, the length of an expression generated by a playout is determined by the ratio of terminals to non-terminals

Benchmark	$p_{E>U}$	$p_{E<U}$	$p_{E>N}$	$p_{E<N}$	$p_{U>N}$	$p_{U<N}$
KEIJZER6	**< 1e-5**	1.00000	0.31738	0.68566	1.00000	**< 1e-5**
VLADISLAVLEVA4	**< 1e-5**	1.00000	**< 1e-5**	1.00000	1.00000	**< 1e-5**
PAGIE1	**< 1e-5**	1.00000	**< 1e-5**	1.00000	1.00000	**< 1e-5**
NGUYEN7	**< 1e-5**	1.00000	0.15559	0.84791	1.00000	**< 1e-5**

Table 4: Wilcoxon rank-sum test p-values, comparing best normalised fitness values from 30 runs of each algorithm with 2^{20} evaluations. $p_{A>B}$ is the P-Value for the alternative hypothesis that Algorithm A outperforms Algorithm B. E is ECJ, U is UCT, N is Nested Search. Values significant at the 0.05 level are in bold.

in the function set. The benchmarks with a smaller number of terminals are the ones where MCTS is competitive with GP. It is well-know [**?**] that size distributions play an important role in GP search, and the playouts here are an unprincipled approach.

Another factor is the size of the target function; for the Vladislavleva benchmark, the function takes around 30 symbols to express concisely. As Nested Search's expression size limit is 35 it is likely to be difficult for Nested Search to find similar expressions. This limit ensures only that the *depth limit* of expressions search by Nested Search and GP are the same. At first we though this may be a major factor in performance, but preliminary experiments at a maximum expression size of 50 yielded little sign of improvement. It appears that the issue with playouts dominates.

The variance of the GP results is greater than the other algorithms. In fact, for the Keijzer benchmark the median score even decreases at higher numbers of evaluations, an observation that can only be explained by stochastic variation. Given that Nested Search contains a large element of systematic recursive exploration as well as randomised playouts, it is likely that the difference in variance is due to its more systematic approach.

6. CONCLUSIONS

We draw the following conclusions:

- Our implementation of UCT performs poorly in comparison to GP and Nested Search, for expression search.
- A naive version of Nested Search is competitive with GP on some benchmarks.

It is not possible to generalise about the performance of GP and Nested Search further from these benchmark results. However, they do suggest that MCTS is a promising search method, give the performance of a very naive implementation on two benchmarks. Furthermore, there is little difference in principle from our applications here and most GP applications.

As with most experimentation involving GP, the usual caveats apply: tuning GP parameters may improve performance, although the same assertion could be made of the MCTS algorithms, in particular regarding the exploration-exploitation formula used in UCT, and the expression size limits chosen by Cazenave.

7. FUTURE WORK

The playouts shape the distribution of tree sizes sampled, and thus an improved playout mechanism that incorporates heuristics to guide the playout, an approach typically used in MCTS when tuning to a given domain, may produce superior results.

We recommend a further comparison of the algorithms on more benchmarks to investigate the effect of maximum expression size and the UCB heuristic used in UCT. Analysis of the distributions of fitnesses within the game tree for expression search could be used to inform these settings.

As our implementations explore the *expression* in a depth-first fashion, they are susceptible to "freezing" the earlier functions in the expression first, which is reminiscent of the ripple effect in grammar-based GP [19]. It is possible to apply MCTS in a breadth-first fashion instead, so that the functions highest in the implicit expression tree would be determined first, rather than a depth-first approach.

The most obvious element of automated programming missing from the algorithms in this paper is typing. A straightforward alteration to our pseudocode, such that the choice of the next function to push onto the expression stack is limited by its type, will enable the search for programs directly in programming languages.

Finally, there is much research on the specialisation of MCTS for particular applications. The literature on single-player MCTS may be relevant to the searching of program spaces: we suggest that *programming is a single-player game.*

8. ACKNOWLEDGEMENTS

Removed for anonymous review.

9. REFERENCES

[1] GECCO Human-Competitive Results Awards 2014. http://www.sigevo.org/gecco-2014/humies.html, 2014.

[2] ECJ, Luke et al., http://cs.gmu.edu/˜eclab/projects/ecj/, 2015.

[3] GPBenchmarks.org, McDermott et al., http://www.gpbenchmarks.org, 2015.

[4] Random.org, http://www.random.org, 2015.

[5] Removed for anonymous review, 2015.

[6] B. Bouzy and B. Helmstetter. Developments On Monte Carlo Go. In *Advances in Computer Games 10*, 2003.

[7] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search Methods. *Computational Intelligence and AI in Games, IEEE Trans.*, 4(1):1 –43, 2012.

[8] T. Cazenave. Nested Monte-Carlo Search. *IJCAI*, 9:456–461, 2009.

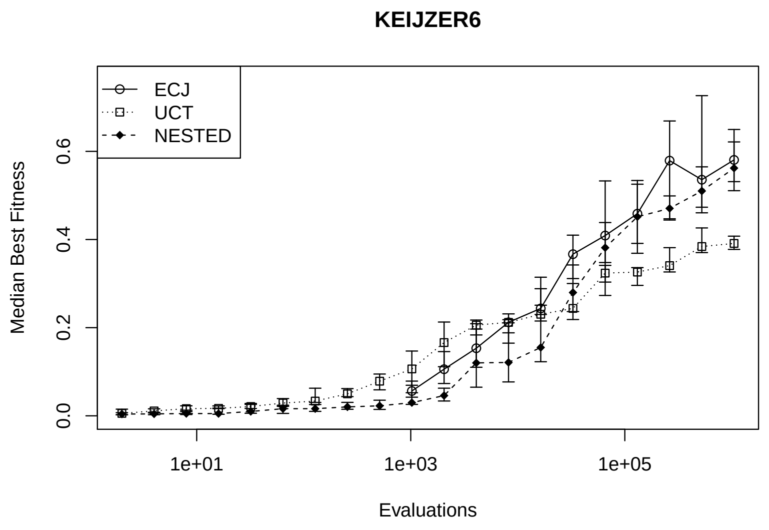

Figure 2: Results for the Keijzer6 Benchmark

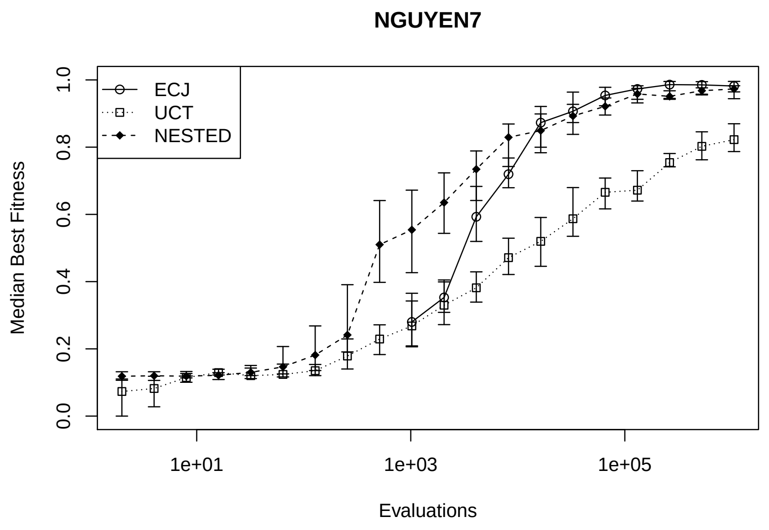

Figure 3: Results for the Nguyen7 Benchmark

[9] T. Cazenave. Nested Monte-Carlo Expression Discovery. In *ECAI 2010*, 2010.

[10] T. Cazenave. Monte-Carlo Expression Discovery. *International Journal on Artificial Intelligence Tools*, 22(1), 2013.

[11] R. Coulom. The Monte-Carlo Revolution in Go. In *The Japanese-French Frontiers of Science Symposium (JFFoS 2008), Roscoff, France*, 2009.

[12] M. Keijzer. Improving Symbolic Regression with Interval Arithmetic and Linear Scaling. In *Genetic Programming*, pages 70–82. 2003.

[13] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo Planning. In *ECML-06*, 2006.

[14] J. McDermott, D. R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaskowski, K. Krawiec, R. Harper, K. De Jong, and U.-M. O'Reilly. Genetic Programming needs better Benchmarks. In *GECCO 2012*, 2012.

[15] L. Pagie and P. Hogeweg. Evolutionary Consequences of Coevolving Targets. *Evolutionary Computation*, 5(4):401–418, 1998.

[16] R. Y. Rubinstein and D. P. Kroese. *Simulation and the Monte Carlo Method*. John Wiley & Sons, 2011.

[17] N. Uy, N. Hoai, M. O'Neill, R. McKay, and E. Galván-López. Semantically-based Crossover in Genetic Programming: Application to Real-valued

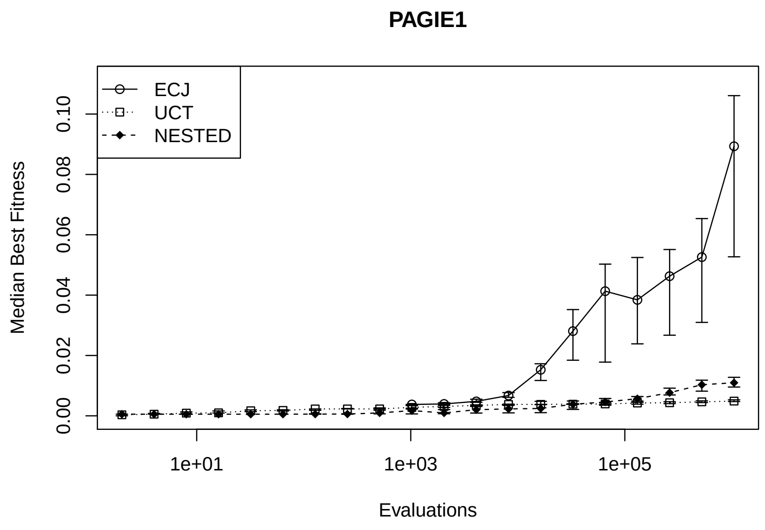

Figure 4: Results for the Pagie1 Benchmark

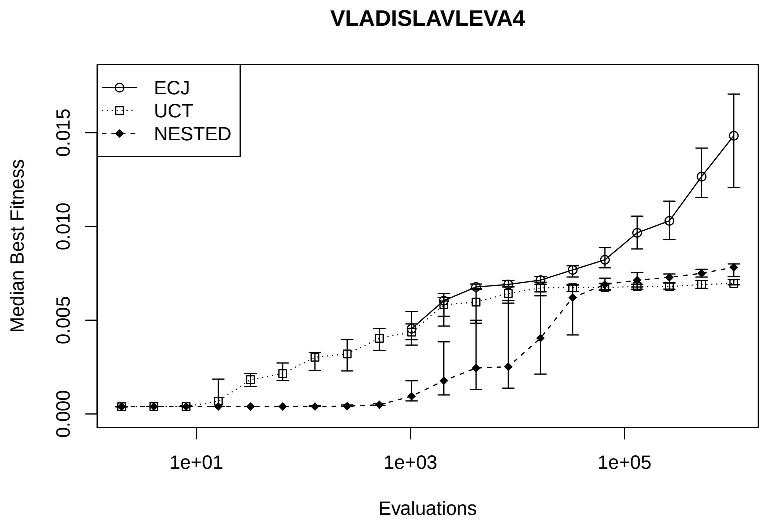

Figure 5: Results for the Vladislavleva4 Benchmark

Symbolic Regression. *Genetic Programming and Evolvable Machines*, 12(2):91–119, 2011.

[18] E. J. Vladislavleva, G. F. Smits, and D. Den Hertog. Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. *Trans. Evol. Comp*, 13(2):333–349, 2009.

[19] P. A. Whigham. Grammatically-based Genetic Programming. In *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, 1995.

[20] D. R. White, J. McDermott, M. Castelli, L. Manzoni, B. W. Goldman, G. Kronberger, W. Jaśkowski, U.-M. O'Reilly, and S. Luke. Better GP Benchmarks: Community Survey Results and Proposals. *Genetic Programming and Evolvable Machines*, 14(1):3–29, 2013.