Field Report: Applying Monte Carlo Tree Search for Program Synthesis

Jinsuk Lim, Shin Yoo

Korea Advanced Institute of Science and Technology
291 Daehak Ro, Yuseong Gu, Daejeon, Republic of Korea

Abstract. Program synthesis aims to automatically generate an executable segment of code that satisfies a given set of criteria. Genetic programming has been widely studied for program synthesis. However, it has drawbacks such as code bloats and the difficulty in finer control over the growth of programs. This paper explores the possibility of applying Monte Carlo Tree Search (MCTS) technique to general purpose program synthesis. The exploratory study applies MCTS to synthesis of six small benchmarks using Java Bytecode instructions, and compares the results to those of genetic programming. The paper discusses the major challenges and outlines the future work.

1 Introduction

Program synthesis aims to automatically generate an executable segment of code that satisfies a given specification. A number of different approaches have been studied, including logical reasoning [11], similarity-based gradient descent [4], as well as the widely studied genetic programming [1,9]. While genetic programming has been used for many successful applications of program synthesis, such as coevolution of programs and tests [1] as well as automated patching [5], it has drawbacks such as code bloats [10] and parameter tuning [8].

This paper considers Monte Carlo Tree Search (MCTS) [7] for general purpose program synthesis. MCTS is a search heuristic that jas achieved impressive results in a number of applications, most notably in computer Go [2]. It has a number of advantages over GP. First, it is more robust against bloats as it is a constructive algorithm. Second, it is mathematically well-established, with a provable guarantee for convergence. Moreover, it has fewer hyperparameters to tune, making it amenable to experimentations and analyses.

MCTS has been recently studied in the context of symbolic regression [12]. This paper extends the application area with an exploratory study of MCTS based synthesis of six small benchmark programs using Java Bytecode instructions. We report initial findings, which suggests that the performance of MCTS is comparable to that of genetic programming. The paper aims to serve as a launchpad for future research on applications of MCTS in SBSE with discussions of practical issues in MCTS based program synthesis.

2 MCTS for Program Synthesis

Although MCTS is typically applied to playing games [2], it has recently been applied [3] and evaluated removed: [12] in the context of symbolic regression. In case of symbolic regression, MCTS iteratively builds a stack-based representation of an expression tree, in which consuming a subsequent symbol is equivalent to finding the next optimal move in a game state.

This paper extends the same core idea to program synthesis by replacing expression trees with program trees. Both symbolic regression and program synthesis are based on the same intuition that sequences of nodes (symbols or instructions) can be interpreted as (expression or program) trees. However, unlike pure functions in symbolic regression, a general purpose program presents a few additional challenges, such as program control flow structure and typing.

2.1 Control Flow Structure

Since we rely on the stack representation of program trees, concatanation of an arbitrary number of program statements raises an issue. If each statement can be represented as a subtree in the program tree, these subtrees should be concatanated using a fixed-arity node type. Our solution is to introduce a binary node `concat`, whose semantic is equal to `nop`: it simply acts as a placeholder so that two subtrees can be concatenated. Concatenation of multiple lines require a successive use of `concat` nodes.

Similarly, branching instructions such as `if` are represented as tertiary nodes: they take three child subtrees, each representing the Boolean predicate, the `true` body, and the `false` body. When generating code from `if` subtrees, we insert `goto` instructions immediately following a comparison operator (e.g `icmplt`), which points to the beginning of the else block, and immediately following the then block, which points to the instruction following the else block.

2.2 Typing

Use of typing system is either absolutely necessary, because the synthesis task or the actual instruction specifically requires statically typed elements, or strongly encouraged, because it greatly reduces the search space by restricting the set of instructions to consider at each phase of the search.

Our typing system consists of seven types: `int`, `float`, `boolean`, `string`, `void`, `control`, and `conditional`. The first five are natural consequences of choosing Jave Bytecode as our code generation tool. The `control` is a special type reserved for instructions that affect control flow: `if` and `concat`. The `conditional` is used as the return type of comparison operators - `icmplt` ($<$) and `icmple` ($\leq$). In the expansion step, MCTS considers only those instructions that have compatible types as its next instruction, i.e., instructions whose return types are compatible with the type of the required arguments. The `true` or `false` body of the `if` instruction, as well as the (empty) program root, may start with instructions of any type.

3 Experimental Setup

3.1 Implementation

We implemented our MCTS based program synthesis tool using Java and Byte Code Engineering Library (BCEL)[1]. Not all of the instructions used by MCTS are Java bytecode. Some of them are lightweight Intermediate Representations (IRs) that provide shortcuts and type specific instructions. For example, instead of preparing appropriate method invocation of `System.out.println`, we provide `iprint` and `fprint` for integers and floats respectively. Others are directly from bytecode instructions (such as `iadd`, `fadd`, etc). The IR program is translated into actual Java bytecode for fitness evaluation.

For comparison, we have also implemented a genetic programming based synthesis tool that generates Java bytecode instructions. Since no existing tool fit our exact purpose, we constructed a bytecode generation tool that takes node sequence as input and writes corresponding Java classfiles as output; the actual search has been driven by `pyevolve`[2], with all typing restrictions added.

Table 1. Subject Benchmarks from Helmuth et al. [6]

Name	Instructions	Expected Behaviour
ADD INTEGER AND FLOAT	`iload_1 iadd isub imul idiv fload_2 fadd fsub fmul fdiv concat return iprint fprint f2i i2f`	Given an integer (`iload_1`) and a float (`fload_2`), print their sum
COMPARE STRING LENGTHS	`sload_1 sload_2 sload_3 true false nop breturn strlen if icmplt icmple concat`	Given three strings s_1, s_2 and s_3, return `true` if $\text{len}(s_1) < \text{len}(s_2) < \text{len}(s_3)$ and `false` otherwise.
GRADE	`iload_1 iload_2 iload_3 iload_4 iload_5 sload_1 sload_2 sload_3 sload_4 sload_5 sprint return if icmplt icmple concat`	Given five integers, the first four represent the lower numeric thresholds for achieving an A, B, C, and D, distinct and in descending order, and the fifth represents the student's grade. Print the grade.
MEDIAN	`iload_1 iload_2 iload_3 nop iprint return if icmplt icmple concat`	Given three integers, print their median.
SMALL OR LARGE	`iload_1 iload_2 iload_3 sload_4 sload_5 nop iprint sprint return if icmplt icmple concat`	Given an integer i, print "small" if $i < 1000$ and "large" if $i \leq 2000$.
SMALLEST	`iload_1 iload_2 iload_3 iload_4 nop iprint return if icmplt icmple concat`	Given four integers, print the smallest of them.

3.2 Benchmarks

We evaluated our method on six benchmarks from Helmuth et al. [6], whose descriptions are given in Table 1. Each benchmark is given a distinct set of terminals and non-terminals which is sufficient to output a correct program. For the test cases, we follow the prescriptions outlined by Helmuth et al. [6].

[1] http://commons.apache.org/bcel/
[2] http://pyevolve.sourceforge.net

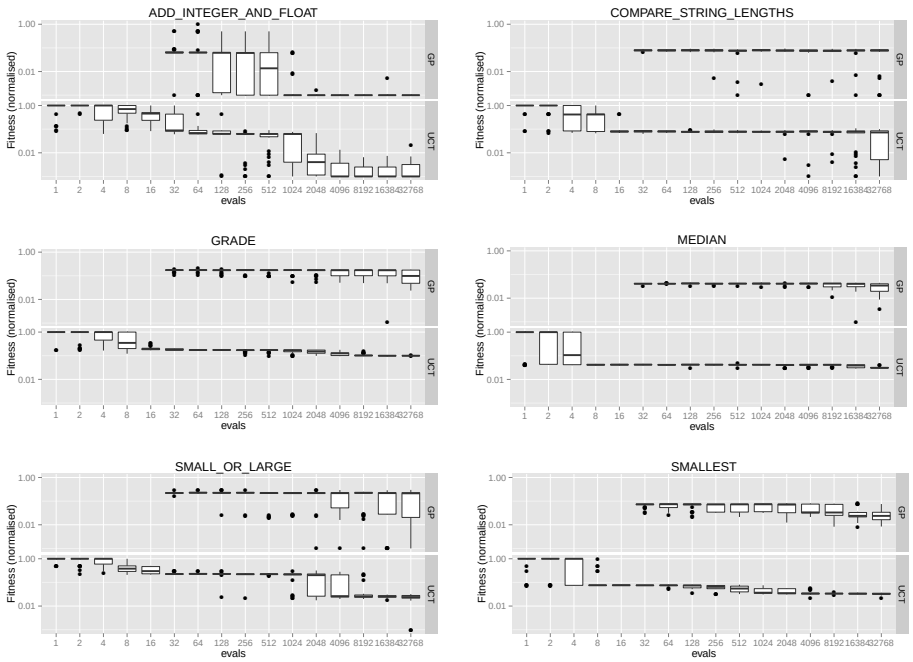

Fig. 1. Boxplots of test data fitness for UCT and GP across different number of evaluation budgets

3.3 Configurations

UCT has two hyperparameters: exploration constant e_c and maximum program length l_p. In particular, l_p has to be sufficiently large enough for a candidate program to be able to encode the correct behavior. We set l_p to be 100 and e_c to be 10 in our experiments. GP is configured with population size 32, rank selection, mutation rate 0.1, crossover rate 0.9 and maximum tree depth 7[3].

Both UCT and GP were run for 30 times to cater for the stochastic nature of each algorithm. Each run was given a maximum of $2^{15} = 32,768$ evaluations. Experiments have been run on machines with Core i7 6700 with 8GB RAM running Ubuntu 14.04, Java version 1.7.0_80, and Python runtime version 2.7.11.

3.4 Fitness Function

Each benchmark either prints or returns an output.Our fitness function considers three aspects of a candidate program: whether it is executable, whether it prints the correct output, and whether it returns the correct output. Given a program P and a test suite T, the fitness of P with respect to T is defined as follows for minimisation:

[3] GP should generate programs of lengths similar to l_p. As most non-terminals have one or two leaves, maximum depth of 7 achieves this.

$$f(P,T) = \begin{cases} 1.0 & \text{if } P \text{ is non-executable} \\ w \cdot f_p(P,T) + (1-w) \cdot f_r(P,T) & \text{otherwise } (w = 0.5) \end{cases}$$

where f_p and f_r measures fitness for printed and returned output respectively. For each output type, we adopt a widely-used distance measure between two instanced: absolute distance for `int`, `float`, and `character`, Levenshtein distance for `strings`, and `NAND` for `boolean`. Both f_p and f_r return the worst fitness when something is printed or returned when it should not be.

4 Results

Results for the six benchmarks are shown in Figure 1[4]. Both UCT and GP show clear trends of improvement as the number of evaluations increase. Both perform well on the relatively easy benchmarks, Add Integer And Float, Small Or Large and Compare String Lengths: several runs produce correct programs. Grade, Median and Smallest are harder because correct solutions require non-trivial control flow structures. Both algorithms fail to output correct programs, although the fitnesses continue to improve.

5 Discussion and Future Work

Both UCT and GP shows inferior performance compared to those reported in Helmuth et al. [6]. This may be due to much smaller budget, but it may also be relevant that Helmuth et al. use a language specifically designed for GP.

We observe that typing is critical. A vast majority of samples by MCTS is non-executable when types are not considered. However, implementing a full type system on top of a tree search can make the algorithm bulky. We plan to investigate the feasibility of implementing a type system as a skewed sampling probability distribution.

Second, being a constructive algorithm, MCTS is prone to early suboptimal commitment. This tendency is shown in difficult benchmarks such as Median and Smallest: the fitness hardly improves past a certain number of evaluations. It appears that MCTS commits to an instruction that yields moderate rewards and keeps exploiting it, when in fact its rewards are suboptimal. Tuning the exploration constant and favoring longer samples may improve this behaviour.

The choice of code generation layer can have a significant impact on performance. While Java bytecode achieves good expressiveness with a relatively small set of instructions, the low level nature of the instructions introduces challenges such as having to deal with explicit jumps to implement branching. We plan to compare different levels of abstractions for program synthesis.

Finally, it should be noted that MCTS is only concerned with a sequence of choices (i.e. selection of nodes); there may be alternatives ways to translate this into programs other than the stack-based representation of trees. We plan to investigate other forms of program construction.

[4] Detailed statistics, as well as the output program instructions, are available from http://coinse.kaist.ac.kr/projects/mcts-program-synthesis

6 Conclusion

This paper presents an early exploration on how to apply Monte Carlo Tree Search for general purpose program synthesis. Java bytecode based implementations of MCTS shows comparable performance to genetic programming. There are many challenges that are specific to different aspects of program synthesis, such as control flow structure, typing, and the choice of code generation layer.

Acknowledgments

Authors would like to thank David White and Kee-eung Kim for many thoughtful discussions about Monte Carlo Tree Search. This research has been supported by Undergraduate Research Program (URP) at KAIST.

References

1. Arcuri, A., Yao, X.: Co-evolutionary automatic programming for software development. Information Sciences 259, 412–432 (2014)
2. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of monte carlo tree search methods. IEEE Transactions on 4(1), 1–43 (2012)
3. Cazenave, T.: Monte Carlo Expression Discovery. International Journal on Artificial Intelligence Tools 22(1) (2013)
4. Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A., Marron, M., R, S., Roy, S.: Program synthesis using natural language. CoRR abs/1509.00413 (2015), http://arxiv.org/abs/1509.00413
5. Forrest, S., Nguyen, T., Weimer, W., Le Goues, C.: A genetic programming approach to automated software repair. In: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation. pp. 947–954 (2009)
6. Helmuth, T., Spector, L.: General program synthesis benchmark suite. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation. pp. 1039–1046. GECCO '15, ACM, New York, NY, USA (2015)
7. Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: Machine Learning: ECML 2006, pp. 282–293. Springer (2006)
8. de Lima, E.B., Pappa, G.L., de Almeida, J.M., Gonçalves, M.A., Meira, W.: Tuning genetic programming parameters with factorial designs. In: IEEE Congress on Evolutionary Computation. pp. 1–8 (July 2010)
9. Orlov, M., Sipper, M.: Flight of the finch through the java wilderness. Evolutionary Computation, IEEE Transactions on 15(2), 166–182 (2011)
10. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. Published via `http://lulu.com` (2008), (With contributions by J. R. Koza)
11. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. SIGPLAN Not. 45(1), 313–326 (Jan 2010)
12. White, D.R., Yoo, S., Singer, J.: The programming game: Evaluating mcts as an alternative to gp for symbolic regression. In: Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference. pp. 1521–1522. GECCO Companion '15, ACM, New York, NY, USA (2015)