Algorithm Discovery with Monte-Carlo Search: Controlling the Size

Josef Moudřík
Charles University
Faculty of Mathematics and Physics
Malostranské náměstí 25
Prague, Czech Republic
email: j.moudrik@gmail.com

Tomáš Křen
Charles University
Faculty of Mathematics and Physics
Malostranské náměstí 25
Prague, Czech Republic
email: tomkren@gmail.com

Roman Neruda
Institute of Computer Science
Academy of Sciences of the Czech Republic
Pod Vodárenskou věží 2
Prague, Czech Republic
email: roman@cs.cas.cz

***Abstract*—The problem of automated algorithm discovery has been mainly approached by means of Genetic programming. Recently, Monte-Carlo tree search methods—well known from games—have been used for program discovery, using stack-based program representations. In this paper, we analyze the behavior of the stack-based representations and describe an approach that provides finer control over generated program sizes and fast uniform playouts. Our approach utilizes type system with parametric polymorphism to generate typed programs. We evaluate the proposed solution with two Monte-Carlo tree search algorithms, and conclude that it is a good alternative which has a better control of exploration.**

***Index Terms*—Monte-Carlo Tree Search, Parametric Polymorphism, Genetic Programming, Nested Monte-Carlo Search, UCT.**

I. Introduction

Genetic programming (*GP*) represents a successful method for automated program discovery, using evolutionary techniques to develop tree program structures in the form of Lisp S-expressions [1]. Several alternative representations for programs have been proposed in the field of GP, ranging from linear structures to computational graphs, but the trees are most prevalent. Genetic programming has been enriched by type information in [2]; in [3], a grammar-based approach to GP has been proposed, dealing with the problem of large program search spaces.

On the other hand, Monte-Carlo Tree Search (*MCTS*) methods [4] have been used as a very efficient way of searching state spaces such as game trees [5], [6]. Recently, an approach that uses MCTS in a combination with stack-based program representation has been proposed and tested for the problem of program discovery in [7], [8].

In this paper, we firstly analyze the behavior of the stack-based program representation during MCTS playouts. We find that its exploration capabilities are highly domain dependent, producing uneven program sizes and bloating (creating large programs) easily. Based on our analysis, we conclude that a better approach is needed in order to have a finer control over program size and exploration of the state space of programs.

In our previous work [9], [10], we have presented a way of efficiently representing and generating typed programs using a type system with parametric polymorphism. We argue that the generation procedure has a very fine-grained control of program size and performs fast uniform sampling among programs of equal program sizes, which is a desirable property for MCTS playouts.

This paper is organized as follows. In the following section, the basics of Monte-Carlo tree search methods are sketched, with the emphasis on currently used variants, namely upper confidence bound for trees, and nested Monte-Carlo search. Details of the possible program representations are presented in section III. Section IV deals with details of program generation procedures, while the analysis of Monte-Carlo playouts for stack based representations is reported in section V. Our experiments on the untyped problem of parity and the typed problem of physics formulas are presented in section VI. We compare our approach with two versions of MTCS algorithm, as well as with the stack-based representations. Finally, the discussion of the obtained results, and conclusions are in sections VII and VIII, respectively.

II. Monte-Carlo tree search

Monte Carlo tree search (*MCTS*) is a family of algorithms for finding actions maximizing an objective by randomly sampling problem's state space and building a search tree from the results. The random sampling (playout) tackles the combinatorial explosion of large state spaces in exchange for completeness. See [4] for a detailed survey.

A. Upper Confidence Bound for Trees

Upper Confidence bound for Trees (*UCT*) is a popular MCTS algorithm based on a UCT formula. The search tree is repeatedly descended from the root and expanded by a tree policy. The descend is performed by repeatedly (until a leaf is reached) choosing the child c maximizing:

$$\overline{S_c} + C * \sqrt{\frac{2 \ln n_c}{n_p}}$$

where $\overline{S_c}$ is c's mean score, n_c and n_p is a number of visits in the child (resp. parent). UCT has a nice property of balancing exploration (the first term) and exploitation (the second term) asymptotically optimally, see [11]; the balance is manipulated by a constant C, which is often fine-tuned for

2375-0197/17/31.00 ©2017 IEEE
DOI 10.1109/ICTAI.2017.00067

the particular domain. In case of single player games (such as in the application of searching for programs), it is possible to use the subtree's best score for S_c instead of the mean.

The leaf is then expanded (if possible) by adding new leaf nodes according to available actions. Finally a random playout is run from the new node(s) and information about the score is backed up the tree.

B. Nested Monte-Carlo Search

Nested Monte-Carlo Search (*NMCS*) is a recursive algorithm operating on levels (given by a parameter n). Globally, the algorithm remembers the best sequence of actions found so far. On level 0, the game is finished by a random playout. On level n, each of possible actions is evaluated using a level $n-1$ search, the best one is chosen, and this procedure is repeated until the game is finished.

NMCS can be thought of in terms of spending a fixed budget (given by n) on estimating the best action a on the first move, then committing to a and continuing with the second move, and so forth. See [6] for details.

III. The State Space of Programs

To search the state space of programs well, one needs a useful program representation with the ability to generate successor programs easily. This is important in both the GP approaches (to seed the initial population) and in the MCTS approaches (to generate a random program for a playout).

A. Expressions

Naturally, an expression is represented as a tree, where the inner nodes of the tree represent non-terminals (functions; denoted by NT), and leaves represent terminals (T) — variables or constants. The tree is evaluated by recursively replacing inner nodes which have all of their children already evaluated by the result of applying node's function to children values. A popular example of this representation are Lisp's S-expressions, where, for instance, `(- x y)` represents mathematical function `x - y`. Moreover, we can allow the expressions to have missing leaves denoted by `?`, representing that we do not yet know how to realize a given sub-tree. Clearly, an expression tree must have no `?` if we want to evaluate it.

With one program being represented by an expression, the whole search space of programs can be viewed as an ordered tree of (partially specified) expressions. In this tree, for instance `(- ? ?)` is a parent of `(- x ?)`, which in turn is a parent of `(- x y)`. Naturally, the `?` can be seen as the root node of the ordered search tree (and a parent of all programs).

B. Stack-based Representation

For functions with a fixed number of arguments, the expression tree can be represented in a computer memory as a stack (instead of a tree structure with pointers) corresponding to serializing the tree by prefix order (the so-called Polish notation). This has the advantage of quick evaluation and a simple playout in case of untyped domain (see Section V).

C. Types

Not all finished trees (with no `?`) represent a valid computation. For instance, string manipulation functions fail when called with numerical operand. This does not matter much for domains such as symbolic regression where all functions operate on floats.

In simple domains, the problem of correctness can be solved by penalizing or discarding invalid programs. In complex domains, a better approach is needed, because otherwise vast majority of candidate programs might be incorrect. Strength of a type system varies from simple heuristic approaches, that do not guarantee correctness (such as the one used in [12]), to type systems involving parametric polymorphism such as the Hindley-Milner type system [13].

Types deeply relate to the problem of generating correct successors and thus, ultimately, to random playouts for MCTS.

IV. Generation Procedure and Type System

This section presents a brief overview of program generation procedure and the type system we use. See [10] for a fully rigorous treatment. The key feature is that for a given partially specified program p and size k, we can quickly count the number N of correctly typed programs of size k without having to actually generate them. Using the generation procedure, we can then realize the MCTS playouts by throwing an N-sided dice and generating the program of the corresponding number. The system is a combination of top-down and bottom-up approaches.

The top-down approach is a way of finding successors for a given partially specified program. These successors are gradually constructed by iteratively running the top-down successor operation from the root node `?`. The MCTS playouts are ran from these partially specified programs in order to find fully specified descendants. For this, the bottom-up approach is used. It combines information about ever larger programs in a dynamic programming manner. Information about bigger programs is built by combining counts and type information of smaller sub-programs.

A. Type System

The type system adds another layer of abstraction to the programs. Each terminal and non-terminal has its type. The types are constructed from a type language, we use a type language compatible with that of the Hindley-Milner [13] type system. Type is either a type symbol (`Bool`, `Int`, ...); type term (`List<Int>`, `Int -> Bool`, ...); or type variable (a, b, ...), which stands for an unspecific type. Type variables are used in *polymorphic* types, such as `List<`a`>`.

B. Physics Formulas

Having a parametric polymorphic type-system allows us to considerably lower the size of the state space of programs. Consider the following symbolic regression problem of physics formulas:

We are given the following terminals: {`x:`m, `v:`ms^{-1}, `t:`s, `a:`ms^{-2}, `0.5`} and functions { `+`, $\star$}, and the goal

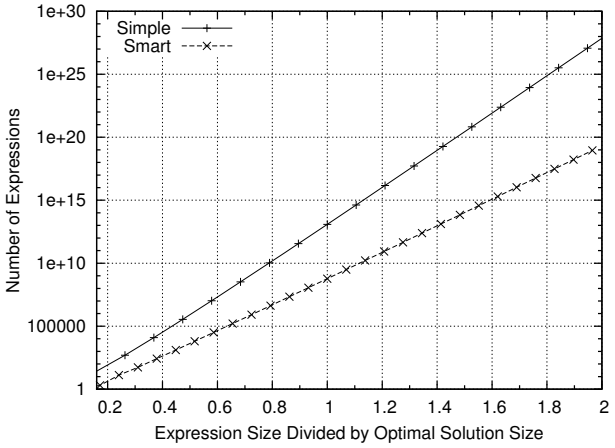


Fig. 1: Physics Formulas problem: precise number of expressions for the given expression size expressed in multiples of optimal solution size (19 for the Simple variant, 29 for the Smart).

is to find a pair of formulas describing the position and speed of an object in space:

$$x(x_0, v_0, a, t) = x_0 + v_0 \cdot t + \frac{1}{2} \cdot a \cdot t^2$$
$$v(v_0, a, t) = v_0 + a \cdot t$$

Note that each terminal has a physical unit in the SI base $(m, kg, s, A, K, mol, cd)$. It is perfectly reasonable to know the unit of the target formulas as well, it is equivalent of asking: "Can we find a physical formula for position and speed of X?" Fitness in this problem is the average error of predicted positions and speeds, as measured on 9^4 points evenly sampled from 4-d grid given by input variables.

The type system allows us to formulate the problem in such a way that only formulas with correct dimension are considered. For example, it is only possible to add two variables with the same dimension (essentially forbidding adding apples and oranges). This drastically reduces the search space, as can be seen from Figure 1. Since our generation procedure can count all correct programs of a given size, we can compare the precise numbers of correct programs for a variant with dimensionality check (marked as Smart in the figure), or without it (Simple). This leads to a big improvement in speed of the regression. See [10] for details on implementing the dimensionality check and further experiments.

V. Analysis of Stack-based Playouts

Cazenave in [14] describes the following simple (untyped) playout procedure. Let T be a set of terminal atoms, NT a set of non-terminal atoms and L a fixed limit of stack size (expression size). The playout proceeds in steps. At first, atoms are added randomly from $T \cup NT$. Once adding another non-terminal atom would make it impossible to finish the stack within the size limit—s.t. all non-terminals have all arguments—remaining atoms are added randomly from T.

We have studied the behavior of this playout implementation with regard to expression length. There are two factors involved. Firstly, it is the size of T and NT and the distribution of arity of the non-terminals (domain), and secondly, the number of missing leaves when the playout is called. For empty stack, this number is one, as the smallest expression has at least one terminal. During the search however, this number gets bigger because programs tend to have functions at the topmost nodes, and the playout thus usually finishes an expression which has (many) more than one missing leaf.

A. Simulation

We ran a statistical simulation with the aim to analyze the average number of nodes that a playout adds, given the domain (sets of T and NT) and a number of missing leaves at the playout's start. The simulation is a simple random walk—each run starts at M (M missing leaves) and takes random steps up (non-terminals, arity is added), or down (terminals, number of missing leaves is reduced by 1). We notice when each run reaches zero for the first time (the program is finished), or hits the limit (set to 50 steps in this experiment). All experiments have been run for M from 1 to 10.

B. Results

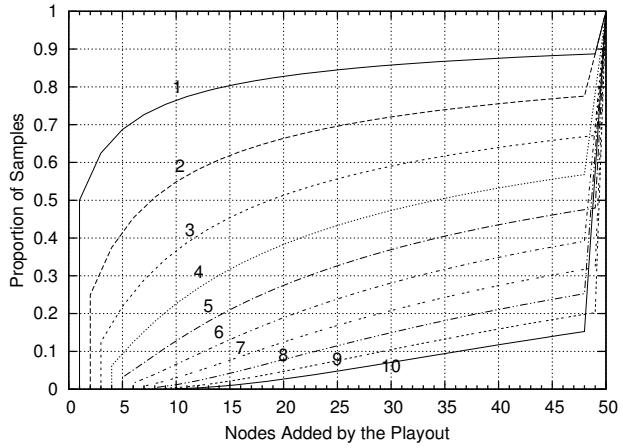
Figure 2 shows the results for three domains. A basic domain with $|T| = |NT|$ and non-terminals with arity 2 is treated in Fig 2a. Fig 2b shows behavior of a domain with many terminals, $|T| = 10 * |NT|$. Fig 2c shows behavior of a domain with more non-terminals, $2 * |T| = |NT|$; this corresponds to the MAX problem [15], as used e.g. in [14].

The results suggest that the behavior is in no way consistent. For domains with many non-terminals (or with high arity) or for a bigger number of missing leaves at playout's start, the playout tends to reach the expression size limit for a large proportion of playouts, which can be interpreted as an equivalent of a bloat phenomena in genetic programming (Fig 2c, Fig 2a). On the other hand, a domain with a large number of terminal atoms—e.g. symbolic regression with many constants—closes the playout very quickly, limiting the space searched (Fig 2b).

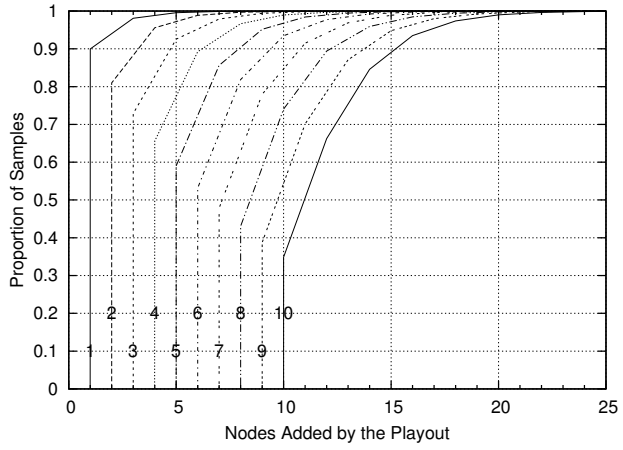
The conclusion is that stack-based playout's behavior is heavily domain dependent and that playout can easily mimic GP's bloat for big trees with many missing leaves. Ideally, thus, one should tweak the domain definition (number of terminals and non-terminals, arity) with regard to this fact to control the exploration of the space of expressions, which is often neither desirable, nor possible.

VI. Evaluation and Comparison

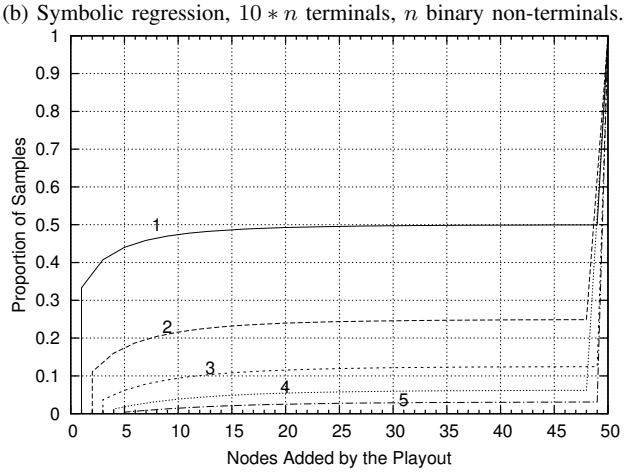
In this section, the results of our experiments are presented. We evaluated the generation procedure to see which MCTS variant works best on a non-trivially typed problem of Physics Formulas (see Section IV-B). Also, we wanted to make sure that our approach is no worse than the stack-based representation on a real untyped problem; we have thus performed another comparison on the 6-parity task.



(a) Basic model, n terminals, n binary non-terminals.



(b) Symbolic regression, $10 * n$ terminals, n binary non-terminals.



(c) MAX problem [15], 1 terminal, 2 binary non-terminals. Plots for 6 and more missing leaves at start are not shown, because they effectively diverge (playout reaches the size limit for virtually all the samples).

Fig. 2: Cumulative distribution of average number of nodes added by a stack-based playout with a size limit of 50 nodes, for three different domains. Plots are shown for 1 to 10 missing leaves (curve labels) on playout's start. Each curve was made by taking $5 * 10^6$ samples.

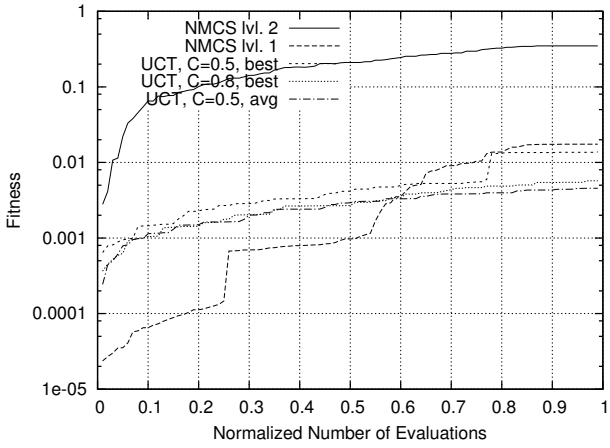


Fig. 3: Comparison of different MCTS variants (see section II) on the typed Physics Formulas domain.

A. Generation and MCTS

We have compared our generation approach using two different MCTS algorithms, Nested Monte Carlo Tree Search (*NMCS*, see II-B) and a standard Upper Confidence Bound for Trees algorithm (*UCT*, see II-A). The results (Figure 3) suggest, that NMCS of level 2 has the superior performance when compared to UCT, which is very similar to NMCS of level 1. The relatively bad performance of UCT is in line with [8]. The most likely cause is the fact that UCT does not—unlike NMCS—guide the search by the best solution found.

B. Even 6 Parity

To check that our approach works well with an untyped problem suitable for stack-based representation, we have chosen the Parity problem [1], which has been used in [14].

The goal of the Parity problem is to find a function that computes the parity of a given number (we use 6) of bits. Four logical functions and, $nand$, or, nor are used, the input is loaded via 6 variables b_1 to b_6. Fitness is simply a number of correctly classified inputs, ranging from 0 to 2^6.

Results are shown in Figure 4. The figure shows dependency of fitness on the number of evaluations made by one run of NMCS (see II-B) of a given level, results were obtained as an average of 100 runs and clearly show that on this untyped problem, our method is no worse than stack-based playouts.

VII. Discussion: Program Sizes, Quality and Playouts

In general, the space of all possible programs solving a given algorithmic task is likely infinite. This raises a natural question of compromising quality vs. solution size. There are many factors involved. Firstly, it has been shown that for some problems, increasing the possible solution size makes it actually harder to find a good solution at all [16]. In other words, some problems have Occam razor's property in the sense that it is easier to find a smaller good solution

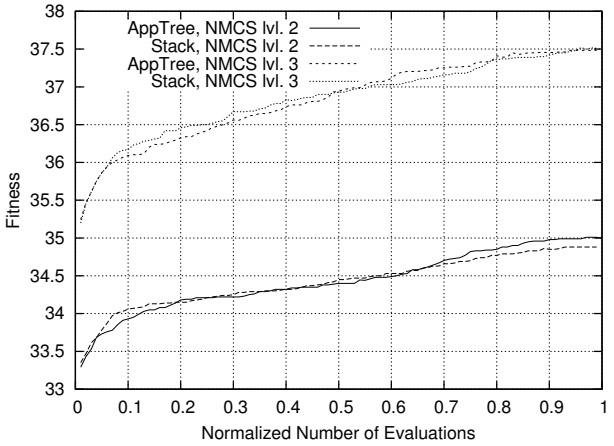


Fig. 4: The Even 6 Parity problem, comparing a stack-based representation with our Applicative Tree representation and generation. Nested Monte Carlo Tree search of level 3 and level 2 was used in both cases. The number of evaluations was normalized by dividing by the maximal number, 1e6 for level 3 and 1e5 for level 2.

than a big one. Secondly, sometimes the program size itself can be an objective—in contrast to using a single objective of maximizing solution performance. Motivation might be a speed of execution, a simplicity of design (e.g. for circuit design optimization), an easy interpretability, etc. Furthermore, this question has a strong link with bloat in GP; when fitness is the only criterion, program size tends to grow a lot without improving the solutions much.

A straightforward approach to limit the solution size is to introduce a fixed threshold. Of course, the limit must be chosen big enough to contain the good solutions. Moreover, it has been shown that in genetic programming, the solution size actually tends to converge to the explicit threshold itself [17]. MCTS approaches [7], [8] with stack-based representations do not have a good behavior with regard to program size either. The plain random playouts (see III-B) produce programs of limiting size with high probability for domains with more non-terminals than terminals, as we have shown in Sec. V.

Our generation procedure allows to precisely count a number of valid programs for a given program size. This gives us a strong way to control how the program state space is searched—for instance, we can approach problems with Occam's razor property (above) by giving smaller program sizes a bigger probability to be chosen during a MCTS playout. Moreover, we can sample uniformly among all programs of a given size (as described in IV). This is a strong and useful property.

VIII. Conclusion

This paper deals with Monte-Carlo tree search for program discovery. We have analyzed two program representations, the stack-based representation which was originally proposed in the seminal work of [7] and our original typed approach which allows to finely control the program sizes [10].

The experiments suggest that on untyped problems, our approach achieves similar performance as its stack-based counterpart, which is very efficient for simple untyped domains. However, our analysis shows that the stack-based approach suffers from bloat for some domains. With our typed generation procedure, we have a very good control of the size of the generated programs, controlling bloat. Our approach generates programs of given size uniformly, which is an interesting property from MCTS point of view.

The implementation of the proposed methods is available under an open license at [18].

Acknowledgment

J. Moudřík has been supported by the Charles University Grant Agency project no. 364015 and by SVV project no. 260 453. Tomáš Křen has been supported by the Grant Agency of the Charles University project no. 187115 and by SVV project no. 260 453. Roman Neruda has been supported by the Czech Science Foundation project no. P103-15-19877S.

References

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

[2] D. J. Montana, "Strongly typed genetic programming," *Evolutionary computation*, vol. 3, no. 2, pp. 199–230, 1995.

[3] A. Ratle and M. Sebag, "Genetic programming and domain knowledge: Beyond the limitations of grammar-guided machine discovery," in *Parallel Problem Solving from Nature PPSN VI*. Springer, 2000, pp. 211–220.

[4] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[5] S. Gelly and D. Silver, "Combining online and offline knowledge in uct," in *Proceedings of the 24th international conference on Machine learning*. ACM, 2007, pp. 273–280.

[6] T. Cazenave, "Nested monte-carlo search." in *IJCAI*, vol. 9, 2009, pp. 456–461.

[7] ——, "Nested monte-carlo expression discovery." in *ECAI*, 2010, pp. 1057–1058.

[8] D. R. White, S. Yoo, and J. Singer, "The programming game: evaluating mcts as an alternative to gp for symbolic regression," in *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, 2015, pp. 1521–1522.

[9] T. Křen, M. Pilát, and R. Neruda, "Automatic creation of machine-learning workflows with strongly typed genetic programming," *International Journal on Artificial Intelligence Tools*, vol. 26, no. 5, 2017.

[10] T. Křen, J. Moudřík, and R. Neruda, "Combining top-down and bottom-up approaches for automated discovery of typed programs," in *Computational Intelligence, 2017 IEEE Symposium Series on*. IEEE, 2017.

[11] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *ECML*, vol. 6. Springer, 2006, pp. 282–293.

[12] J. Lim and S. Yoo, "Field report: Applying monte carlo tree search for program synthesis." in *SSBSE*, 2016, pp. 304–310.

[13] R. Milner, "A theory of type polymorphism in programming," *Journal of computer and system sciences*, vol. 17, no. 3, pp. 348–375, 1978.

[14] T. Cazenave, "Monte-carlo expression discovery," *International Journal on Artificial Intelligence Tools*, vol. 22, no. 01, p. 1250035, 2013.

[15] W. B. Langdon and R. Poli, "An analysis of the max problem in genetic programming," *Genetic Programming*, vol. 1, no. 997, pp. 222–230, 1997.

[16] W. B. Langdon, R. Poli *et al.*, "Why ants are hard," 1998.

[17] C. Gathercole and P. Ross, “An adverse interaction between crossover and restricted tree depth in genetic programming,” in *Proceedings of the 1st annual conference on genetic programming*. MIT Press, 1996, pp. 291–296.
[18] T. Křen and J. Moudřík. (2017) Tfgpy. [Online]. Available: http://github.com/tomkren/tfgpy