A Dynamic Programming Approach to Individual Initialization in Genetic Programming

Tomáš Křen
Charles University in Prague
Faculty of Mathematics and Physics
Malostranské náměstí 25
Prague, Czech Republic
tom.kren@gmail.com

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

***Abstract*—In this paper we present a new initialization method for genetic programming based on randomized exhaustive enumeration. It naturally enables complete sharing of subtrees among individuals which in turn allows an efficient reuse of computations. Moreover, it can be implemented as a random one-pass initialization. We present experimental results on different instances of simple symbolic regression exploring the landscape of possible initializations based on our approach and confirming the usability of these initializations.**

***Index Terms*—Genetic programming, Initialization, Dynamic programming.**

I. Introduction

Genetic programming (GP) represents an efficient method for automatic generating of programs by means of evolutionary techniques [1], [2]. It is a population based search heuristics that operates on a tree structures representing computer programs. The algorithm consists of several operators – the initialization operator provides starting solutions for the search, that is in turn realized by the selection operator mimicking natural selection, and mutation and crossover operators recombining and improving existing solutions. The initialization operator is especially important in GP, as it should not only provide suitable initial population but it is also utilized in mutation operators. Moreover, in contrast to evolutionary techniques working on linearly encoded individuals, there exist several approaches how to initialize trees in GP.

A wide spectrum of initialization methods for GP can be arranged by the amount of included randomness – from random one-pass initializations (e.g. the standard ramped half-and-half [1]) to systematic approaches (e.g. exhaustive enumeration). The exhaustive enumeration can be relaxed by adding a random step, and it can be straightforwardly used to faithfully simulate random one-pass initializations as will be demonstrated further in this paper.

Our previous work with exhaustive-inspired initializations in typed GP [3] have indicated that such approach can be useful for standard GP, thus the motivation was to explore this approach and examine its usefulness in standard GP. The main advantage of the approach described here is the reduction of time complexity of the initialization algorithm on the account of space requirements that are still very reasonable and negligible in practice.

Among the approaches to initialization procedures, the by far most frequently used one is the so-called ramped half-and-half, but several alternative methods are also used.

Authors of [4], [5] show how to perform *uniform initialization* which allows to sample trees uniformly based on Alonso's bijective algorithm [6]. Our method can be considered as a generalization of such uniform generating. Below, we introduce a *generating strategy* as a parameter guiding our generating method; the uniform generating method is equivalent to our generating with a simple strategy $p(d) = 1$.

Our method can be also perceived as a generalization of exhaustive enumeration by adding randomness into it. An interesting relationship between GP and exhaustive enumeration – which comes from the typed functional GP context – is reported in [7]. The typed GP is also the original motivation for our proposed method. The below described AND and OR gadgets are designed so that they allow further generalization for a typed case.

The structure of this work is as follows. In section II we discuss the main principles of random and systematic initializations. The main result of the paper is presented in section III where our initialization method is described. The algorithms proposal is tested on a simple symbolic regression problem, the results are gathered in section IV. Finally, the discussion of the results is presented in section V.

II. Background

Genetic programming initialization methods fall into the broader category of generating algorithms. One can imagine a whole spectrum of such generating algorithms arranged by the amount of included randomness. The standard method, called *ramped half-and-half* [1], is on one end of the spectrum where the amount of randomness is very high. On the opposite end of this spectrum we can place the systematic enumeration of all possible trees from the smallest towards the bigger trees.

The ramped half-and-half generating process produces one complete individual, after which it restarts and starts again. It iterates in this fashion until the desired number of individuals is produced. We call this kind of initialization an *one-pass* generating (cf. Fig. 1). This is in contrast with systematic methods which may share the substantial portion of the generating process and substructures among many individuals.

978-1-4799-8697-2/15 $31.00 © 2015 IEEE
DOI 10.1109/SMC.2015.307

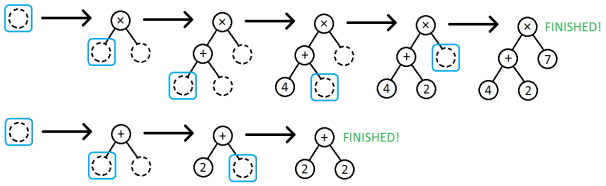

Fig. 1. Examples of one-pass generation

For the sake of easier explanation, it is convenient to describe the generating process by means of the *unfinished node expansions*. By a node expansion we mean the replacement of the unfinished node by a more specific subtree; this new subtree may be a terminal or a function with unfinished nodes as its children (respecting the arity of the function).[1]

A straightforward approach to exhaustive enumeration can be based on the *priority queue* data structure. At the beginning, the queue contains only one unfinished tree that consists of a single unfinished node. As described on Fig. 2, in every step we retrieve the smallest unfinished tree[2] from the queue, and expand one of its unfinished nodes (e.g. the depth-first search one) in all possible ways. Doing so, we obtain a set of more specific trees – the *successors*. Some successors may be finished trees (i.e. they do not contain any unfinished node), while the rest are unfinished trees. We put the finished successors to output and the unfinished successors back to the queue. This way of generating trees produces them in the order from the smallest to the largest.

The field of evolutionary computing relies on randomness, thus it seems inappropriate to use fully systematic generating for population initialization. On the other hand we believe that it is worth exploring whether we can gain some benefit from using higher level of systematic nature in the initialization. So we can ask the following question: How to modify the systematic generating in order to make it more random?

The fully systematic generating procedure uses all the successor trees, i.e. it puts all the successors back to the queue or it outputs them. One way to weaken the systematic nature is to discard some of them. Formally, we can generalize the systematic approach into this randomized version by adding a parameter in the form of filter procedure, which decides for each successor whether to keep it or discard it.

Since the generating is now randomized, the queue may go empty before the wanted number of individuals is generated. If this happens, a single unfinished node is added into the queue and generating may continue. We call this event a *restart*.

[1]An unfinished node in its simple untyped form does not contain any additional information, however for the typed version it is a convenient place for storing useful information such as the type of the subtree to be generated (or local variables information if anonymous functions are involved).

[2]The algorithm for exhaustive enumeration described here is an instance of well-known A* algorithm [8]. The A* heuristic corresponds here to the choice of how we deal with unfinished nodes. We count unfinished nodes as one node in the simple untyped version (since we can always replace it with a terminal). For the typed GP, the situation is more complicated, because there may not be an applicable terminal.

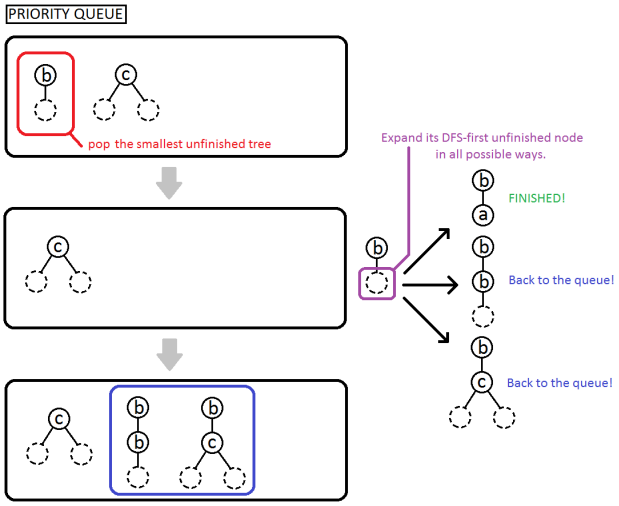

Fig. 2. Systematic generating based on priority queue

It is possible to emulate one-pass generating by using a filtration procedure that discards all but one successors. Such filtration causes that the queue contains at most one unfinished tree. Therefore the randomized systematic generating can be seen as a generalization of both systematic and one-pass generating.

In this paper we use a more specific form of the filtration procedure defined by a function $p : \mathbb{N} \to [0, 1]$ that assigns probability of keeping a successor based on the depth of the expanded unfinished node – let d be the depth of the expanded node that produced the successor, then $p(d)$ is the probability of keeping the successor. We call such a function p a *generating strategy*.

Example of such a strategy is our *geometric strategy* [3] that is given by a simple formula $p(d) = q^d$ where q is a fixed constant (we used $q = 0.75$).

The intuition behind the geometric strategy is that we want the probability of keeping the successors produced by expanding nodes in small depths to be high and we want it to descend with increasing depth. We can imagine the generating process as unsupervised exploration of the tree space. We want to systematically explore all the possible roots (i.e. nodes with $d = 0$), but with increasing depth we are more willing to drop more successors. By dropping a successor we cut off a portion of searched space; with increasing depth the portion of the cut-off space is getting smaller. Experiments on three well known benchmarks presented in [3] showed that the geometric strategy have the same or better performance than the standard ramped half-and-half with respect to success rate, best fitness value, time consumption and average individual size.

III. Our Approach

Our goal is to simulate the queue-based generating procedure by one-pass generating capable of complete subtree sharing. We assume generating procedure is parameterized by a

Algorithm 1: Simplified initialization procedure.

```
function generate(Int popSize)
    population ← ∅
    n ← 1
    while |population| < popSize do
        limit ← min(|population| + ⌈e(n)⌉, popSize)
        while |population| < limit do
            tree ← generateOne(n)
            population ← population ∪ {tree}
        n ← n + 1
    return population
```

search strategy, i.e. the filtration function defined by probability function depending on depth of a node.

A search strategy function assigns probability to each node in a tree. It is a probability of keeping a successor produced by expanding that node. When a complete tree is produced it means that each of its nodes survived the expansion. The probability of this event is equal to the product of node probabilities, provided that the generating procedure manages to go through all of the trees with size less than or equal to n.

The queue-based approach generates trees in the order given by the number of nodes of a tree. Assuming we have a method for generating one tree of a given size (respecting the whole-tree probability) and with the knowledge of the expected number of generated trees for a given size, we can simulate the systematic queue-based generating by a probabilistic one-pass generating in the following way.

We proceed iteratively starting with the tree size $n = 1$. In each iteration we compute the expected number $e(n)$ of generated trees of size n; and generate $\lceil e(n) \rceil$ trees by the one-pass generating. This is repeated until the required number of trees is generated. Algorithm 1 describes this process in greater detail. This is the core idea of our approach that will be elaborated in this section. First we describe how to compute the $e(n)$ value.

In order to build a tree (of size n) we must first select a root symbol (a terminal for $n = 1$, or a function for $n > 1$), and after that we construct a subtree for each input argument recursively. We call the symbol selection an *OR-step* (since we choose one from many choices), and the subtree generation an *AND-step* (since we must generate subtrees for all arguments). Since we need to control the exact size of the constructed trees, our OR-step is enriched with additional choice; in an OR-step we also choose specific sizes for the subtrees (a so-called *size profile*).

For a specific symbol with k arguments there is $sp(n, k)$ size profiles. For a terminal symbol (i.e. $n = 1, k = 0$) we have $sp(1, 0) = 1$, for a function symbol (i.e. $n > 1, k > 0$) it is:

$$sp(n, k) = \binom{n-2}{k-1}.$$

The total number of choices $ch(n)$ for OR-step for size n is:

$$ch(n) = \sum_{s} sp(ar(s), n),$$

where the sum ranges over all applicable symbols s (for $n = 1$, s ranges over terminal set; for $n > 1$, s ranges over function set) and $ar(s)$ is the number of arguments of the symbol s. The set of trees with size n is divided into $ch(n)$ disjoint subsets, each corresponding to one OR-choice (i.e. the choice of a root symbol together with a size profile). Fig. 3 summarizes these notions.

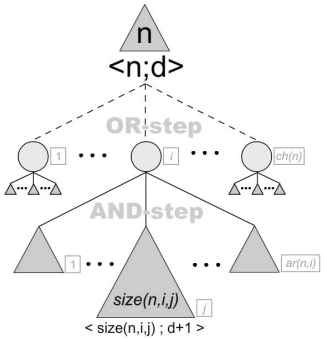

Fig. 3. Summary of OR and AND step.

In order to compute $e(n)$ (the expected number of trees of size n) we use auxiliary function $e(n, d)$ which is the expected number of (sub)trees of size n with root in depth d. Analogically we can compute $\#(n)$, the total number of trees of size n.

$$e(n) = e(n, 0), \qquad \#(n) = \#(n, 0).$$

In order to compute $e(n, d)$ we use another auxiliary function $e(n, i, d)$ which is the expected number of (sub)trees of size n with root symbol and size profile corresponding to the i-th OR-choice with root in depth d. Since the OR-choice subsets are disjoint we can compute the $e(n, d)$ value as the following sum:

$$e(n, d) = \sum_{i=1}^{ch(n)} e(n, i, d), \qquad \#(n, d) = \sum_{i=1}^{ch(n)} \#(n, i, d).$$

It remains to show how to compute $e(n, i, d)$. We know the root depth d, and since we know (n, i) we also know the root symbol and the size profile. The AND-step is computed as the following product:

$$e(n, i, d) = p(d) \cdot \prod_{j=1}^{ar(n,i)} e(size(n, i, j), d + 1),$$

$$\#(n, i, d) = \prod_{j=1}^{ar(n,i)} \#(size(n, i, j), d + 1).$$

where $ar(n, i)$ is the number of arguments of the root symbol, and $size(n, i, j)$ is the size of the j-th subtree in the i-th OR-choice for tree of the size n. Fig. 4 show a simple example of computing e and $\#$.

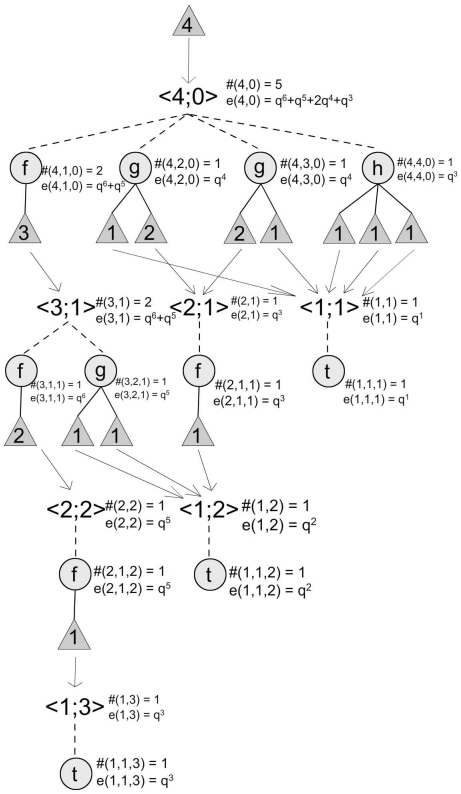

Fig. 4. Example for simple $T \cup F$ with one terminal t, one unary function f, one binary function g and one ternary function h.

Now we show how to generate one individual of size n in a one-pass fashion. The $generateOne(n)$ procedure generates one tree of size n. The $generateOne(n, d)$ procedure generates one (sub)tree of size n with root at depth d. Again, $generateOne(n) = generateOne(n, 0)$. Let us describe the $generateOne(n, d)$ procedure recursively. In order to generate a root node for tree with n nodes at depth d, select one index i from $\{1, \ldots, ch(n)\}$ (i.e. do an OR-choice) where the probability of selecting an index i is:

$$p(n, i, d) = \frac{e(n, i, d)}{e(n, d)}$$

The index i determines a choice of a node symbol together with subtree sizes given by $size(n, i, j)$ for $j = 1, \ldots, ar(n, i)$. The j-th subtree is generated by $generateOne(size(n, i, j), d + 1)$. Algorithm 2 shows simplified pseudocode for this procedure.

A nice property of this approach to tree generating is that it allows for reusable generating, i.e. generating that shares common subtrees among generated trees. Let us dive more into the implementation details. During the generating process, many computed values (e.g. $e(n, i)$, $e(n, i, j)$) are used repeatedly, so it is efficient to compute them only once and store them for later use. To do so we introduce two data structures associated with the OR-step and the AND-step; we call them *OR-gadget* and *AND-gadget*. There is an OR-gadget

Algorithm 2: Simplified generation of one individual.

function ***generateOne****(Int n, Int d)*
  $i \leftarrow$ **select** $\{1, \ldots, ch(n)\}$ **with probability** $\frac{e(n,i,d)}{e(n,d)}$
  $tree \leftarrow$ new node with $symbol(n, i)$
  for $j \in \{1, \ldots, ar(n, i)\}$ **do**
    $tree.addSon($**generateOne**$(size(n, i, j), d + 1))$
  return $tree$

G^n for each tree of size n. And there is an AND-gadget G^n_i for each tree of size n and for each OR-choice $i \in \{1, \ldots, ch(n)\}$. So we have $G^n.e[d] = e(n, d)$ and $G^n_i.e[d] = e(n, i, d)$. An AND-gadget G^n_i can be seen as a collection of trees of size n with a root symbol and a size profile corresponding to the i-th OR-choice (together with those useful values). So it is a natural place where to store those already generated trees. Analogically, an OR-gadget G^n can be seen as a collection of AND-gadgets G^n_i.

We believe that the most important benefit of generating trees that share identical subtrees is that it allows a natural way for sharing not only the structure but also the computations. For the sake of simplicity, suppose that we have a problem with solutions of the type $A \to A$ and that fitness evaluates them for N fitness cases. We can say that fitness has a type $(A \to A) \to \mathbb{R}$. Also, suppose that we have a terminal x standing for an input variable (i.e. $x : A$), and functions from the F set are unary or binary, i.e. $A \to A$ or $A \times A \to A$. An individual is called N times during the evaluation by the fitness function, once for each fitness case. We can now reformulate such a problem to be suitable for computation reuse. Alternative understanding is that an individual tree represents a vector A^N, the terminal $x : A^N$ is a constant vector containing inputs for fitness cases, and functions from F are $A^N \to A^N$ or $A^N \times A^N \to A^N$. A fitness function now has type $A^N \to \mathbb{R}$. Each individual subtree represents a A^N vector – and this value can be stored in the root object of the subtree. In order to compute a value of a tree we use the values of its subtrees. In order to evaluate fitness of a tree we compute fitness of its value. This is further discussed in the example in the next section.

It is also important to state that this one-pass approach to generating is not a completely faithful simulation of the queue-based approach; the difference lies in the *restarts* of the queue-based approach, and in the fact that the one-pass approach may generate the same tree multiple times – so we perform uniqueness check to avoid that. In our practice this uniqueness checking is not an efficiency issue. The queue-based generating never outputs the same tree several times, unless it restarts; then there is possibility of generating one tree several times. It would be interesting to investigate the exact relationship between those two phenomena.

Let us conclude this section with remarks concerning the space consumption of our method. Suppose we generate trees up to size n. Then, there is n OR-gadgets and $O(n^2)$ values for computing the e function. The number of AND-gadgets

G_i^n for OR-gadget G^n depends on maximal function arity k, which is a small fixed number. We have $O(\binom{n-2}{k-1}) = O(n^{k-1})$ AND-gadgets for one OR-gadget, so $O(n^k)$ AND-gadgets in total, therefore there is $O(n^{k+1})$ values for e. Such space consumption is fine for reasonable values of k since n is usually a small number. These data may be generated once and further shared for problems with the same "shape" of $T \cup F$.

IV. Experiments

We made two experiments comparing the performance of the standard *ramped half-and-half* method with our *geometric strategy* $p(d) = q^d$ (with various values of parameter q).

In our previous experiments with geometric strategy [3] implemented by the queue-based approach we showed that our method outperforms the standard method on two benchmarks – the (*Artificial Ant* and the *Even Parity*) – and performs the same on the *Simple Symbolic Regression* (SSR) problem (described in [1]), so we chose this benchmark to test our new approach in greater detail.

The objective of the SSR problem is to find a function $f(x)$ that fits a sample of N given points ($N = 20$). The standard target function is a polynomial function defined as $f_t(x) = x^4 + x^3 + x^2 + x$. However, in our experiments we use a wide range of automatically generated target functions. The terminal set T consists of a single input variable x. The function set $F = \{+, -, *, rdiv, sin, cos, exp, rlog\}$ where:

$$rdiv(p,q) = \begin{cases} 1 \text{ if } q = 0 \\ p/q \text{ if } q \neq 0 \end{cases} \qquad rlog(x) = \begin{cases} 0 \text{ if } x = 0 \\ log(|x|) \text{ if } x \neq 0 \end{cases}$$

The fitness function is computed as:

$$fitness(f) = \frac{1}{1 + \sum_{i=1}^{N} |f(x_i) - f_t(x_i)|}$$

where $(x_i, f_t(x_i))$ are the input points.

Individual trees share common subtrees. In the case of SSR, the same subtrees correspond to the same computation. We can reformulate the SSR problem to be suitable for computation reuse. An SSR individual tree represents a $\mathbb{R} \to \mathbb{R}$ function, the terminal x is an input variable, and functions from F are $\mathbb{R} \to \mathbb{R}$ or $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$. An individual is called N times during evaluation by the fitness function, once for each input point. Alternative understanding is that a SSR individual tree represents a $\mathbb{R}^N$ vector, the terminal $x = (x_1, \ldots, x_N)$ is a constant vector containing x-components of input points, and functions from F are $\mathbb{R}^N \to \mathbb{R}^N$ or $\mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}^N$. Fitness function is then $\mathbb{R}^N \to \mathbb{R}$. Each individual subtree represents a $\mathbb{R}^N$ vector – and this value can be stored in the root object of the subtree. In order to compute a value of a tree we use the values of its subtrees. In order to evaluate the fitness of a tree we compute the fitness of its value, i.e. distance from target vector $(f_t(x_1), \ldots, f_t(x_N))$.

In order to see how reusable computation speeds up the evaluation, it is illustrative to observe how it works for the simple case of exhaustive enumeration. Suppose we have generated all the trees smaller than n, then we need to perform just a single computation, whereas with no reuse we need to perform n computations. With nonexhaustive generating the speedup is smaller depending on the $e(n)$ values. Note that it is also important to implement computation reuse for trees created by cross-over, but that is beyond the scope of this paper.

Each of the two experiments presented here is divided into several *middle-experiments* which are further divided into several *steps*.

Each *step* tests a generating strategy on one generated SSR problem. It consist of $k = 10$ runs of GP algorithm. Each run has 51 generations with 500 individuals. We call the average best individual fitness from those k runs a *step-result*.

A *middle-experiment* compares performances of various generating methods on various SSR problems. The compared strategies are standard method and our geometric strategy $p(d) = q^d$ for $q \in Q$ where $Q = \{0.01\} \cup \{0.05 \cdot i \mid i \in \{1, \ldots, 20\}\} \cup \{0.5 + 0.01 \cdot j \mid j \in \{1, \ldots, 30\}\}$, i.e. the promising interval $[0.5, 0.8]$ is explored more extensively than the rest of $[0, 1]$. Each generating method is tested on a set of $m = 50$ different generated SSR problems. So there is m steps for each tested generating method. A *middle-result* for a generating method is an average from those m steps.

The whole experiment consists of $n = 10$ middle-experiments, each for a different set of generated SSR problems. We repeat the middle-experiments with different problem sets because random problem sets tend to have fluctuating difficulty, so this makes the result more general. The total number of GP runs for the whole experiment is $n*m*|Q|*k = 10 * 50 * 45 * 10 = 225000$.

The two experiments differ in the method used for generating the target functions for the SSR problems. The graphs (Figs. 5 and 6) show mean values of the middle-results for various q values and contain error bars representing the standard error of the mean (SEM). For easy comparison the result of the ramped half-and-half is depicted as a horizontal line. From each experiment a q value with maximal mean value is selected and its performance is compared with the standard method by statistically analyzing the difference by the paired t-test.

A. Experiment 1

Target functions for this experiment are randomly generated from SSR's $T \cup F$ by the standard ramped half-and-half generating method. The best performing method is the geometric strategy for $q = 0.66$. It performs slightly better then the standard method, but this difference is not statistically significant (cf. Table I). Fig. 5 shows the results.

B. Experiment 2

In the first experiment, the standard method has an advantage in that it has to find the target functions generated by itself. In the second experiment we turn the situation around, and let the winner geometric strategy $q = 0.66$ generate the target functions, again from from SSR's $T \cup F$. But now we have the opportunity to better balance the difficulty of generated problems by using $generateOne(n)$ for sizes $n \in \{1, \ldots, m\}$.

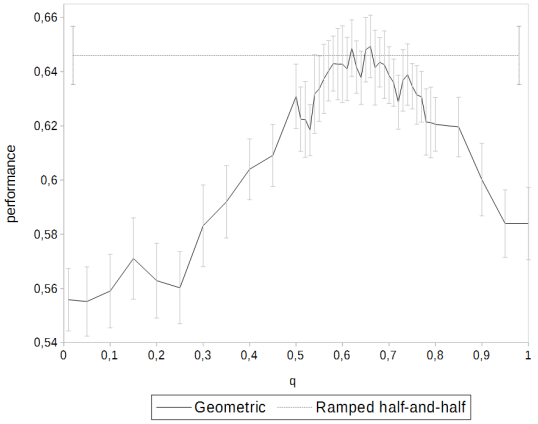

Fig. 5. Results of the experiment 1.

TABLE I
STATISTICAL ANALYSIS - EXPERIMENT 1

	Ramped half-and-half	Geometric(0.66)
Mean	0.646009157360	0.649373590030
SD	0.033828586830	0.036479053197
SEM	0.010697538441	0.011535689499
N	10	10
t-value	0.3013	$\alpha = 0.05$
p-value	0.7701	**Not statistically significant.**

The best performing method for this experiment was the geometric strategy for $q = 0.62$. Fig. 6 shows that this problem is significantly harder but the performance shape is similar. Again the performance is better then the standard method, now the difference is statistically significant according to the t-test (cf. Table II).

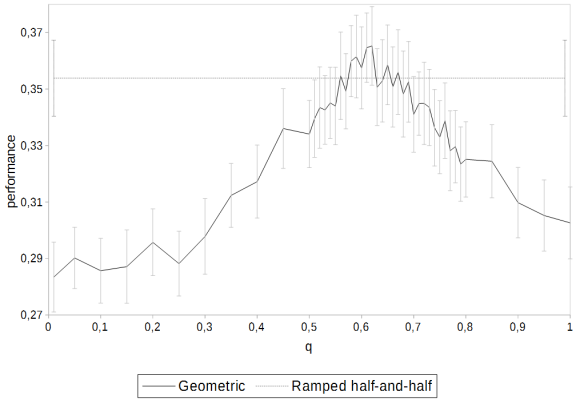

Fig. 6. Results of the experiment 2.

V. CONCLUSIONS

In our previous work [3] we have shown that a one-pass initialization can be simulated by an exhaustive approach with a priority queue reduced to one element only. Here we demonstrate how to simulate the queue behavior by a one-pass initialization with pre-computed values of augmenting variables by means of dynamic programming. In a sense, this problem represents the harder implication of the equivalence we are trying to demonstrate. Thus, although the approach is of direct use for the typed version of GP, here we deal with the standard GP only. The important advantage of our approach is that a large amount of subtrees can be shared, and thus the fitness computations can be reused. This speedup is achieved by sacrificing a reasonable space needed by the algorithm.

TABLE II
STATISTICAL ANALYSIS - EXPERIMENT 2

	Ramped half-and-half	Geometric(0.62)
Mean	0.353883524810	0.365299367810
SD	0.042564161852	0.044041303628
SEM	0.013459969815	0.013927083059
N	10	10
t-value	2.6191	$\alpha = 0.05$
p-value	0.0279	**Statistically significant.**

Our two experiments demonstrated that this generation procedure achieved comparable or better results than the ramped half-and-half strategy for the symbolic regression benchmark. A more thorough evaluations have been used to assess the optimal value of the q parameter.

For the future work, we are currently developing the counterpart of the generating procedure for the case of typed GP. It seems that the AND-gadgets and OR-gadgets can be easily augmented by all the necessary information for the more complex typed trees.

It is interesting to note that the initialization procedure is not used solely during the initialization part of the evolutionary algorithm. The mutation operator makes use of generating a random tree and replacing a certain subtree of the individual by it. This context can hint about different kinds of systematic approach which will be tailored for mutations.

ACKNOWLEDGMENTS

Tomáš Křen has been partially supported by the project GA UK 187115 and by the SVV project number 260 224. Roman Neruda has been partially supported by the Czech Grant Agency grant GA15-18108S.

REFERENCES

[1] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. Cambridge, MA, USA: MIT Press, 1992.
[2] ——, *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Norwell, MA, USA: Kluwer Academic Publishers, 2003.
[3] T. Křen and R. Neruda, "Generating lambda term individuals in typed genetic programming using forgetful A*," in *Evolutionary Computation (CEC), 2014 IEEE Congress on*, July 2014, pp. 1847–1854.
[4] H. Iba, "Random tree generation for genetic programming," in *Parallel Problem Solving from Nature — PPSN IV*, ser. Lecture Notes in Computer Science, H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, Eds. Springer Berlin Heidelberg, 1996, vol. 1141, pp. 144–153.
[5] W. Langdon, "Size fair and homologous tree crossovers for tree genetic programming," *Genetic Programming and Evolvable Machines*, vol. 1, no. 1-2, pp. 95–119, 2000.
[6] L. Alonso and R. Schott, *Random generation of trees: random generators in computer science*. Springer Science & Business Media, 1994.
[7] F. Briggs and M. O'Neill, "Functional genetic programming and exhaustive program search with combinator expressions," *International Journal of Knowledge-Based and Intelligent Engineering Systems*, vol. 12, no. 1, pp. 47–68, 2008.
[8] S. J. Russell and P. Norvig, *Artificial Intelligence - A Modern Approach (3rd Ed.)*. Prentice Hall, 2010.