Genetic Operators in Evolutionary Music Composition

Sulyok Csaba
Faculty of Mathematics and Computer Science
Babeș-Bolyai University
RO-400084 Cluj-Napoca, Romania
Email: sulyok.csaba@cs.ubbcluj.ro

***Abstract*—Genetic operators represent the alterations applied to entities within an evolutionary algorithm; they help create a new generation from an existing one, ensuring genetic diversity while also preserving the emergent overall strengths of a population.**

In this paper, we investigate different approaches to hyperparameter configuration of genetic operators within a linear genetic programming framework. We analyze the benefits of adaptively setting operator distributions and rates using hill climbing. A comparison is drawn between the constant and adaptive methodologies.

This research is part of our ongoing work on evolutionary music composition, where we cast the actions of a virtual composer as instructions on a Turing-complete virtual register machine. The created music is assessed by statistical similarity to a given corpus. The frailty to change of our genotype dictates fine-tuning of the genetic operators to help convergence.

Our results show that adaptive methods only provide a marginal improvement over constant settings and only in select cases, such as globally altering operator hyperparameters without changing the distribution. In other cases, they prove detrimental to the final grades.

***Keywords*-genetic operators; linear genetic programming; evolutionary algorithms; algorithmic music composition**

I. Introduction

Within the life cycle of any evolutionary algorithm, *genetic operators* [1], [2] are the set of procedures creating subsequent generations from existing ones. They ensure the diversity of the population while maintaining an ongoing stochastic process. By De Jong's [1] definition, they "determine the genetic makeup of offspring from the genetic material of the parents". For classical genetic algorithms, three genetic operators are commonly defined [3]: reproduction, crossover and mutation (each discussed at length in Section II). In classical examples, each individual in a generation is built using exactly one operator. It is possible however to chain multiple operators to create a single individual, thus potentially exploring the search space faster. The most common pairing is crossover with mutation [4], [5] or with simulated annealing [6].

The problem addressed in the current ongoing project is that of algorithmic corpus-taught music composition [7], [8], [9] using linear genetic programming (LGP) [10]. This ongoing project uses the conceptual separation of genotype and phenotype to evolve what might be considered to be the thought process of a virtual music composer instead of the music itself. We cast the actions of the composer as atomic instructions performed by a virtual register machine whose output is the "observable" phenotype: a piece of music. Our genotype is represented by a genetic string: a byte array fully representing a state of the virtual machine (VM). The underlying code is interpreted sequentially, making this a linear genetic programming problem. The pieces of music are qualitatively assessed by comparing statistical likeness to a corpus of real music input in MIDI format.

The current work performs a comparative study on the adaptive settings of genetic operators. Although online hyperparameter optimization has been researched extensively for genetic algorithms and programs, it has seldom been attempted in an LGP setting. Our explored hyperparameters include the distribution of the different operators as well as operator-specific settings: the number of cut points in crossover and the number of altered bytes in mutation.

The remainder of this document is structured as follows: Section II describes the background concepts of genetic operators used in different types of evolutionary algorithms, together with a history of adapting their parameters. Section III outlines the linear genetic programming problem and the associated framework, while Section IV describes how we incorporate adaptive genetic operators into it. Section V describes our experiments, with the results detailed in Section VI. Section VII draws the appropriate conclusions and presents future possibilities of exploration.

II. Background & related work

The suggested operator distribution for genetic programming [3], [11] is 8% reproduction, 90% crossover and 2% mutation. Ideal percentages may vary based on data representation and problem complexity [12] and may even change during any single run of an algorithm, when elite units start to emerge [13].

In the following subsections, classical genetic operators and their adaptability are discussed together with their usage in (linear) genetic programming environments.

A. Reproduction

Reproduction represents carrying an entity over from one generation to the next without change, ensuring the "survival of the fittest"; it mitigates the ever-present probability that fit individuals may be omitted during the breeding process.

Survivors may be chosen by taking the best units directly or by randomly selecting them based on their fitness, using a process such as roulette-wheel selection [14]. As discussed in our previous research [8], guaranteeing the survival of the highest-scoring entities in our setting leads to elitism: the best individuals may be overly vulnerable to mutation and prohibit other units with better potential from emerging. Therefore a preferred evolution strategy is probabilistic survival: selecting survivors based on chance, where strength only increases the possibility of survival, it does not guarantee it. This allows vulnerable albeit decent entities to die out and make way for more stable ones.

Entities may be selected solely based on their fitness or a combination of fitness and age. Incorporating age allows older units to have a smaller chance in roulette-wheel selection, adding a "ticking clock" element to their forwarding of helpful genes to their offspring.

B. Mutation

Mutation represents random alterations in the genetic code of an individual, either by reshuffling small segments of genetic material or by swapping/reorganizing existing segments. It is used to maintain population diversity and dissolve stuck positions in local optima.

In linear genetic programming, mutation may alter a genetic string in various ways: a single randomly chosen instruction may be changed to a random one, multiple randomly chosen bytes may be randomized or entire series of linear code may be moved, shuffled or randomized [10]. The number of mutated or swapped bytes constitutes a new hyperparameter whose best value may depend on context and may also change within a run; we label this value n_m.

Intuitively, mutating too few segments in a genetic string may slow down the exploration of the search space. On the other hand, mutating overly many bytes increases the likelihood of distorting or destroying the genetic material that made the original individual strong enough to be selected. This could reduce the explorative strength of mutation to a random search.

Piszcz and Soule [12] analyze the optimum mutation rate in correlation with the difficulty and complexity of the presented problem; the latter is based on the size of the genetic string. Their findings show that more complex problems narrow the interval of effective mutation rates, therefore fine-tuning is required.

C. Crossover

Crossover (or recombination) implies combining the genetic information of two selected individuals to produce one or two offspring, which hopefully combine the strengths of the parents. In the context of linear genetic programming, Hu et al. [5] demonstrate that recombination "significantly accelerates the evolutionary search process and particularly promotes robust phenotypes".

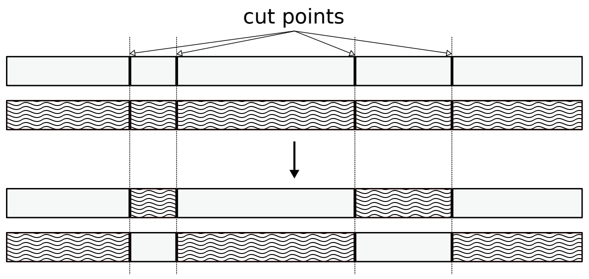

Figure 1. Applying homologous crossover to a pair of genetic strings to create two offspring. The number of cut points n_c is proportional to the genetic string size; their position is selected randomly.

The genotype is subject to recombination by selecting a cut point; one of the offspring inherits the information on one side of this point from the mother, and the rest from the father.

Classically, crossover only uses one or two cut points. However, because we use a large amount of simple sequential data to represent virtual composers' thought processes, increasing the number of cut points may help improve the search process [7]. The cut points count relative to the genetic string size therefore becomes a new hyperparameter which may benefit from live adaptation; we label it n_c.

In a similar vein, Esquivel [15] proposes multiple crossovers per couple (MCPC), creating multiple offspring reusing the same pair of parents. The selection for crossover, similarly to reproduction, can be deterministic or probabilistic; higher fitness parents have a better chance of producing more than one offspring.

Other variations of crossover have been proposed for genetic programming, such as uniform crossover [3] or randomly selecting the depth separately from the node [16]; the latter allows bias correction from the classical case where leaves are chosen often. However, these methods both hinge on the tree-based genotype representation abandoned in LGP, replaced by a sequential approach.

Crossover in LGP classically involves one or more randomly selected cut points serving as alteration points [17]. These do not necessarily appear in the same place in the two genetic strings, therefore the swapped code segments often differ in size. This unconstrained genotype size may lead to "code bloat": continuously growing code caused by only marginally better fitness. A simple solution is homologous crossover [18]: swapping aligned and equal-sized blocks from the mother and father genotypes, effectively disallowing code bloat (see Figure 1).

D. Adaptive genetic operators

Munroe [13] highlights the importance of the ratio between the number of units created using mutation and crossover; this ratio should decrease during an algorithm's run, since increasingly stable units should arise, reducing the necessity of random impact.

Dynamically altering operator parameters in genetic algorithms has been attempted as far back as the late 1980s [19], [20], with later research also extending to genetic programming problems [21]. Common operator adaptation methods include hill climbing [22], [23], [6], differential evolution [24] and using a second genetic algorithm within the original for parameter optimization [25]; Kazarlis et al. refer to the latter strategy as a "microgenetic algorithm (MGA)" [26].

Parameters which may benefit from adaptive strategies can be separated into two categories. The distribution of operators used to construct a new generation constitutes a global parameter of the algorithm which may be updated per generation. Adaptivity has been applied to global distributions [19] as well as the probabilities of crossover [21] and mutation [27], [28], [29]; better results are achieved in all cases compared to standard operators. On the other hand, mutation and crossover use the operator-specific parameters discussed in Sections II-B and II-C, respectively. We collectively name these parameters operator rates.

Adaptive strategies for operator rates are separated into two classes. The centralized approach [19], [30], [31] provides discrete operators applicable throughout a population, adapted once per generation based on how much overall fitness improves. Conversely, in the individualized strategy (also referred to as decentralized or self-adaptive) [28], [20], [21], the settings are part of each individual and are adapted and inherited independently.

III. System overview

This section introduces the linear genetic programming framework used in our experiments [1]. It is built as a holistic system capable on evolving general data, not necessarily just music.

We follow widely accepted genetic programming conventions (see Figure 2). We separate the concepts of genotype (the object of evolution) and phenotype (the object of quality assessment). The genotype represents a state of the virtual composer's brain, therefore the phenotype rendering may be viewed as the composition process itself. The actions of the composer are viewed as a sequential series of instructions interpreted by a Turing-complete virtual machine. The VM is constructed as a simple 8-bit general-purpose register machine with a complex instruction set. We define a genetic string as any condition of the VM; it holds all memory segments and register values in a fixed-size byte array.

The instruction set and general architecture of the virtual machine may take on many shapes [9]. The only condition for viability is the inclusion of output instructions, which

[1]Project hosted open-source at https://bitbucket.com/csabasulyok/emc

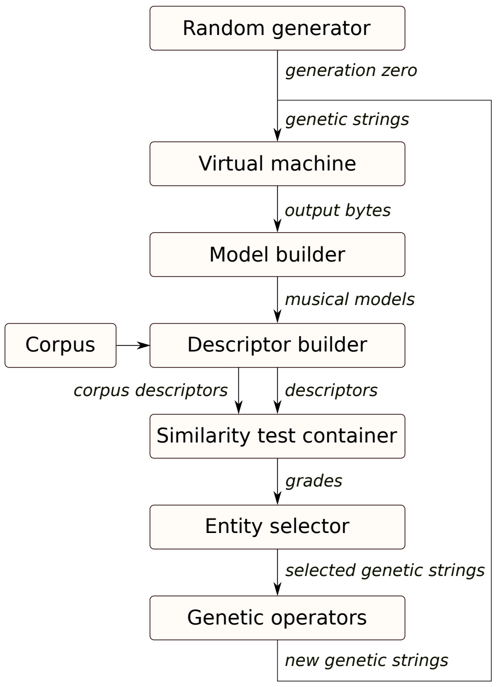

Figure 2. Our algorithm starts out with a seed of random genetic strings, interpreting them using a Turing-complete virtual machine and using the output to construct musical models (phenotypes). Fitness is represented by similarity to members of a corpus of real music. The grades are used to select and create a subsequent generation using genetic operators.

act as events firing to the virtual score. The output bytes are used to construct the musical phenotype: a MIDI-like representation of a score. This two-stage approach to rendering the model contrasts previous musical evolutionary systems in that the structure of the genetic string does not directly represent that of the phenotype. Decoupling the genetic program and the resulting score in this way allows for the emergence of complex structures not possible in more constrained models. For example, a conditional branch instruction executed on the VM could cause the repetition of a single note or section, or perhaps even the entire piece depending on where it occurs in the program and how far it jumps the instruction pointer. While the musical possibilities of this approach are effectively unconstrained, the resulting space of possible outcomes becomes correspondingly vast. Therefore any small changes in hyperparameters may result in a chain reaction, greatly changing the outcome.

The evaluation process employs fitness tests that judge the similarity between certain properties of a musical phenotype and those of a chosen corpus of existing music. We use Bach's *Inventions and Simfonias*[2] set of keyboard exercises as our corpus, chosen mainly because of the pieces' homogeneity in length, style and complexity. The corpus data is in no way copied or included into any rendered phenotype, it

[2]Works BWV 772-801 downloaded from www.midiworld.com/bach.htm

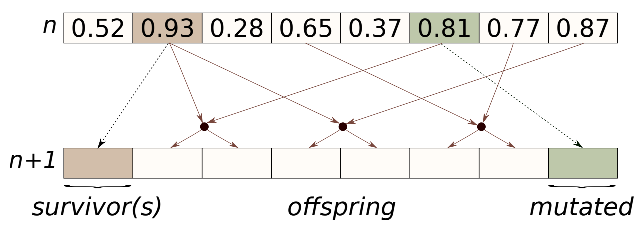

Figure 3. An example distribution of different operators when assembling generation $n+1$ from generation n. The numbers represent fitness values. In this small case, 12.5% (1 out of 8 individuals) of the population is created through reproduction, another 12.5% through mutation, while the rest is bred. All source units from generation n are selected using roulette-wheel selection.

is merely used to inform our fitness evaluation. The tests focus on the underlying statistical properties of a model rather than the sameness of the data itself; it is therefore possible for a model to achieve high grades without being identical to a member of the corpus as long as it shares certain traits captured by those statistics. To produce these statistics, we subject models to a series of transform methods such as histograms, histograms of differentials and Fourier transforms; together they produce a fixed-size descriptor. The final similarity of a musical model is determined by Pearson-correlation of the descriptor to that of the corpus members. Correlation of different properties may contribute to the final grade with different weights, making the assessment multi-objective by nature [32]. The array of fitness values contribute to the genetic operators which build the next generation.

Before the application of adaptive genetic operators, the system uses probabilistic reproduction of a fixed number of individuals in each generation, incorporating fitness and age into the roulette-wheel selection (see Figure 3). It uses a fixed number of altered bytes n_m in mutation, and homologous crossover with multiple cut points n_c.

IV. Adaptive operators

Our adaptive parameters all employ heuristic hill climbing [23]. We label the value of an adaptive variable at time t as x_t. Each such variable is assigned a set of adaptation vectors $\{\Delta_0, \Delta_1, \ldots\}$; these slightly alter the monitored value to search for a positive impact. An initial value x_0 is given and an adaptation vector index i is randomly selected upon launch; the value is altered by the appropriate vector Δ_i at every evaluation ($x_{t+1} := x_t + \Delta_i$). The vectors are represented as Markov states [33] whose transition probabilities at every step (represented by a change of the value i) depend on the positive impact of their usage. In the case of global adaptive parameters, positive impact is equivalent to a higher overall fitness of the population; in individual cases it is a higher fitness than the average of the two parents. Fitness growth gives higher probability

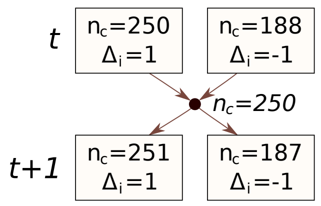

Figure 4. Usage and setting of the adaptive crossover rate n_c. A mother and father are selected, the cut point count of the first is applied and two offspring are born; their crossover rates are inherited and modified based on hill climbing. If the new units perform poorly, their adaptation vector Δn_c will change in the next iteration.

to remaining in the current state of i, continuing the hill climb in the same direction. If the fitness drops or remains unchanged, the probability of a state change grows, with uniform probability assigned to all other $j, j \neq i$ values.

Single operator parameters such as operator rates use a single discrete value for x_t, with Δ_i taking on values from $\{-1, 0, 1\}$. The values (and therefore their adaptation vectors) are integers, since they represent discrete byte counts in the genetic strings involved in operations, such as the number of randomized bytes or cut points.

In our implementation of individualized adaptive operators, the rate of crossover and mutation is hereditary. In crossover, each of the two offspring receives one parents' rates, possibly adapted (see Figure 4). Similarly, mutated units inherit the original rate.

Operator distribution is represented through the numbers of units created using each operator, yielding a 3-value vector $x_t = (r_t, c_t, m_t)$ whose values sum up to the population size: $r_t + c_t + m_t = N, \forall t$. Adapting x_t implies preserving the sum of the underlying values, therefore Δ_i exclusively takes on values whose elements sum to 0, such as $(0, 0, 0)$ and vectors containing 0, -2 and 2 in any order. The latter examples use a step size of 2 instead of 1 to preserve c_t as an even number, since pairs of parents are involved in crossover. For example, $\Delta_i = (-2, 2, 0)$ represents an alteration after which 2 fewer children of the subsequent generation are created through reproduction, replaced instead by a pair bred using crossover.

V. Experiments

The presented experiments explore the distribution of operators and the main parameters of crossover/mutation; in the latter case, the tests extend to both the centralized and individualized adaptive approaches. The following settings are tested for operator distribution:

1) the standard distribution of 8% reproduction, 90% crossover and 2% mutation, as proposed by [11];
2) adaptive distribution using the standard one as a starting point.

Comparing a standard distribution to an adaptive one allows us not only to measure if adaptivity accelerates or

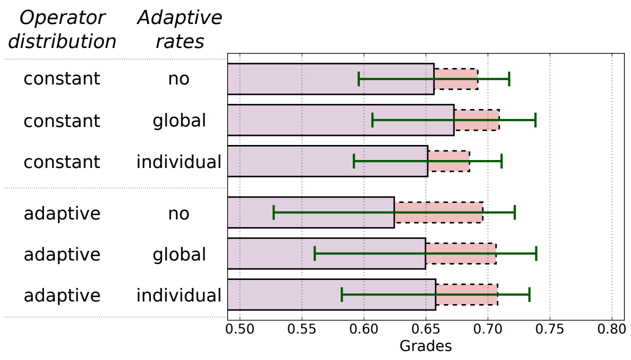

Figure 5. Final generation fitness scores grouped by operator distribution and rates. The wide bars show the average mean and standard deviation of the populations throughout the run, while the narrow bars show highest scoring entities.

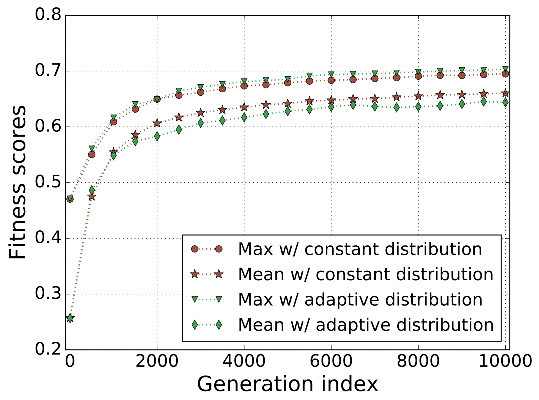

Figure 6. Fitness progression through the generations when using a constant vs. adaptive operator distribution; the results suggest adaptive operators provide better highest scoring individuals at the cost of a poorer overall population.

improves the algorithm, but also should highlight an optimal distribution which may be used in further experiments (if it differs from the standard).

The tested rates include the number of cut points n_c and the number of mutated bytes n_m, both represented proportionally to the size of the genetic string. Changing values include:

1) constant values used in [8]: $n_c = 0.1\%$, $n_m = 2\%$;
2) global adaptive values starting with the same values used in the first case, adapted once globally every generation;
3) individualized adaptive values starting with the same values used in the first case, inherited and adapted per individual every generation.

For each of the resulting 6 configurations we run a total of 20 iterations, allowing the algorithm to reach 10000 generations each time. Other settings are given as defined in [8]: a population size of 1024, we generate 30 second long pieces, the VM uses an instruction set with immediate addressing and a 64kB size RAM. Furthermore, survival through reproduction (aging) reduces further survival chances by 10%.

All highest scoring entities of the runs can be downloaded in MIDI format from https://csabasulyok.bitbucket.io/emc.

VI. Results & discussion

Figure 5 shows a snapshot of the last generation of each test run; it depicts the mean and maximum values averaged over the 20 test runs for all 6 configurations. The results suggest adaptive operator setting provides only marginal benefits and only in select cases.

For instance, adaptive operator distribution seems to increase the maximum fitness within a population, but at the cost of lower mean values in each case. Figure 6 provides further analysis of the result, averaging the different configurations together and visualizing the progression of mean and maximum grades through the generations. Indeed, the best entities using the adaptive operator distribution consistently outperform the ones using a constant distribution, but the mean fitness of the populations is consistently smaller. This result is surprising, since the hill climbing operation uses the population's mean fitness to adapt the distribution, without analyzing the maximum or the deviation.

Figure 7 shows the average change of the distribution values through the generations. The values are shown as percentages relative to the starting configuration of $8/90/2$. The results show that the marginal improvement of maximum grades in a generation may be achieved by increasing the percentage of entities created through mutation by 3-4%, taking away from those bred through crossover.

Individual rate adaptations provide no significant improvement over the baseline; this could be due to the overwhelming number of dimensions hill climbing is trying to explore.

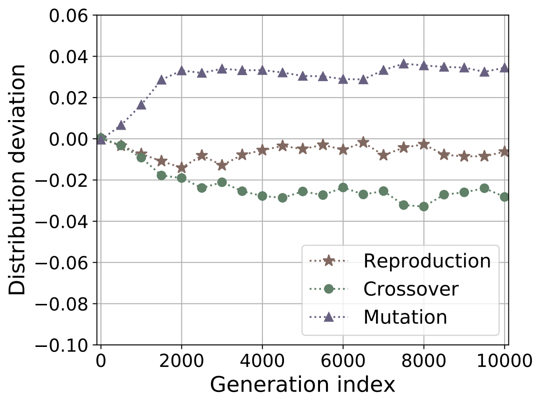

Figure 7. Average change of the genetic operator distribution relative to the starting configuration when using hill climbing. The results suggest better convergence when more units are created through mutation and slightly less by crossover.

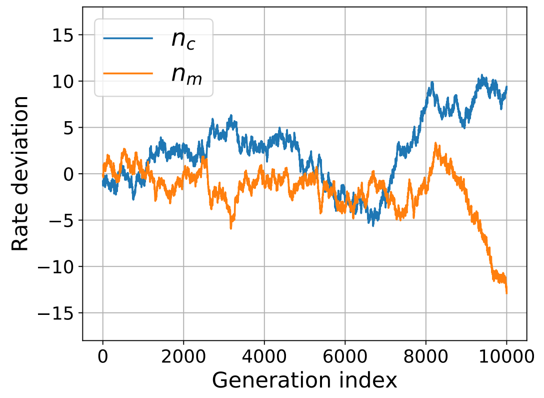

Figure 8. Average change of global operator rates when using a constant distribution. The number of cut points n_c show an increase in later generations, while the number of altered bytes n_m slightly decrease.

With our experiments using 1024 entities, the rates employ 2048 independent hill climbing operators.

Further analyzing the last generation leads us to conclude that the best results are achieved when globally adapting mutation and crossover rates, leaving the distribution constant. Analysis of how the rates change in this case (see Figure 8) reveals that the number of cut points show a slight increase in later generations, while the number of bytes involved in mutation slightly lower. However, because the number of mutated bytes has a larger starting value, the change of n_m has a much smaller impact than that of n_c.

VII. Conclusion & future work

In this paper, we have explored the application of adaptive genetic operators when faced with a complex problem: automatic music composition using linear genetic programming. Our approach to this problem differs from others in the literature by modelling the thought process of the composer as a sequential program running on a simple von Neumann register machine. Finding "good" pieces of music then becomes a linear genetic programming problem, where quality measurement is fully automated and is based on similarity to real world music.

The three basic operators used in evolutionary algorithms have been detailed: reproduction, crossover and mutation. We have tackled setting their distribution as well as their individual settings (number of mutated bytes and number of cut points in crossover) based on commonly accepted values, but also employing an adaptive strategy with hill climbing.

The results demonstrate that hill climbing does not radically impact the overall quality or conversion speed of the presented system. It gives only marginally better maximum fitness values at the cost of worse overall performance when applied to operator distributions. Global operator rate adaptation proves to be efficient when not combined with distribution changes, reducing the dimensionality of the problem hill climbing must tackle.

In the current state of our framework, it uses genetic operators exclusively: one individual is built using exactly one operator. However, it is common in evolutionary algorithms to combine crossover and mutation to create new units [5]; we propose further algorithms exploring adaptive methods in such a context.

We further propose experiments using the adapted operator distribution values from Section VI as a starting point. Perhaps an increase in mutation at the cost of less crossover already implemented in generation zero may help these values reach their full potential.

Hill climbing is only one of many adaptation possibilities for operators. Alternative strategies such as a second genetic algorithm [26], simulated annealing [34] or supervised learning methods may prove useful in such a linear genetic programming problem.

The other building blocks of our evolutionary music composition system also dictate possible expansions. For example, the currently used corpus contains many key changes and complex passages, possibly making it overly complex for a system beginning its learning from pure randomness.

Further experiments involving the virtual machine as a phenotype renderer may also aid the current research. To this end, we propose the exploration of more complex VM architectures and instruction sets [9] as well as Turing universality in spiking neural P systems [35].

References

[1] K. De Jong, "Learning with genetic algorithms: An overview," *Machine Learning*, vol. 3, no. 2-3, pp. 121–138, 1988.

[2] D. J. Cavicchio Jr., "Reproductive adaptive plans," in *Proceedings of the ACM annual conference*, vol. 1. New York, New York, USA: ACM Press, 1972, pp. 60–70.

[3] R. Poli and W. B. Langdon, "On the search properties of different crossover operators in genetic programming," *Genetic Programming*, pp. 293–301, 1998.

[4] D. E. Goldberg and R. Lingle, "Alleles, loci, and the traveling salesman problem," in *Proceedings of an international conference on genetic algorithms and their applications*, vol. 154. Lawrence Erlbaum, Hillsdale, NJ, 1985, pp. 154–159.

[5] T. Hu, W. Banzhaf, and J. H. Moore, "Robustness and evolvability of recombination in linear genetic programming," in *European Conference on Genetic Programming*. Springer, 2013, pp. 97–108.

[6] U.-M. O'Reilly and F. Oppacher, "Hybridized crossover-based search techniques for program discovery," in *Proceedings of 1995 IEEE International Conference on Evolutionary Computation*, vol. 2. IEEE, 1995, pp. 573–578.

[7] C. Sulyok, A. McPherson, and C. Harte, “Corpus-taught Evolutionary Music Composition,” in *Proceedings of the 13th European Conference on Artificial Life*. York, UK: The MIT Press, 2015, pp. 587–594.

[8] ——, “Evolving the process of a virtual composer,” *Natural Computing*, pp. 1–14, 2016.

[9] C. Sulyok and C. Harte, “On virtual machine architectures for evolutionary music composition,” in *Proceedings of the 14th European Conference on Artificial Life*. Lyon: MIT Press, 2017, pp. 577–584.

[10] M. F. Brameier and W. Banzhaf, *Linear genetic programming*. Springer Science & Business Media, 2007.

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

[12] A. Piszcz and T. Soule, “Genetic Programming: Analysis of Optimal Mutation Rates in a Problem with Varying Difficulty.” in *FLAIRS Conference*, 2006, pp. 451–456.

[13] D. R. Munroe, “Genetic Programming: The ratio of Crossover to Mutation as a function of time,” *Research Letters in the Information and Mathematical Sciences*, no. 6, pp. 83–96, 2004.

[14] A. Lipowski and D. Lipowska, “Roulette-wheel selection via stochastic acceptance,” *Physica A: Statistical Mechanics and its Applications*, vol. 391, no. 6, pp. 2193–2196, 2012.

[15] S. C. Esquivel, A. Leiva, and R. H. Gallard, “Multiple crossover per couple in genetic algorithms,” in *Evolutionary Computation, 1997., IEEE International Conference on*. IEEE, 1997, pp. 103–106.

[16] K. Harries and P. Smith, “Exploring alternative operators and search strategies in genetic programming,” *Genetic Programming*, vol. 97, pp. 147–155, 1997.

[17] R. Poli, W. B. Langdon, N. F. McPhee, and J. R. Koza, *A field guide to genetic programming*. Lulu. com, 2008.

[18] P. Nordin, W. Banzhaf, and F. D. Francone, “Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover,” *Advances in genetic programming*, vol. 3, pp. 275–299, 1999.

[19] L. Davis, “Adapting operator probabilities in genetic algorithms,” in *proc. 3rd International conference on genetic algorithms*, 1989, pp. 61–69.

[20] T. Bäck, “Self-adaptation in genetic algorithms,” in *Proceedings of the First European Conference on Artificial Life*. MIT Press, 1992, pp. 263–271.

[21] P. J. Angeline, “Two self-adaptive crossover operators for genetic programming,” in *Advances in genetic programming*. MIT Press, 1996, pp. 89–109.

[22] J.-M. Renders and H. Bersini, “Hybridizing genetic algorithms with hill-climbing methods for global optimization: two possible ways,” in *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence.,*. IEEE, 1994, pp. 312–317.

[23] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Malaysia; Pearson Education Limited, 2016.

[24] F. Vafaee and P. C. Nelson, “Self-adaptation of genetic operator probabilities using differential evolution,” in *Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems. SASO’09*. IEEE, 2009, pp. 274–275.

[25] J. Grefenstette, “Optimization of Control Parameters for Genetic Algorithms,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 16, no. 1, pp. 122–128, 1986.

[26] S. Kazarlis, S. Papadakis, J. Theocharis, and V. Petridis, “Microgenetic algorithms as generalized hill-climbing operators for GA optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 5, no. 3, pp. 204–217, 2001.

[27] A. Tuson and P. Ross, “Adapting operator settings in genetic algorithms,” *Evolutionary computation*, vol. 6, no. 2, pp. 161–184, 1998.

[28] J. Gomez, “Self adaptation of operator rates in evolutionary algorithms,” in *Genetic and Evolutionary Computation Conference*. Springer, 2004, pp. 1162–1173.

[29] F. Vafaee, P. C. Nelson, C. Zhou, and W. Xiao, “Dynamic adaptation of genetic operators’ probabilities,” *Studies in Computational Intelligence*, 2008.

[30] M. Srinivas and L. M. Patnaik, “Adaptive probabilities of crossover and mutation in genetic algorithms,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 24, no. 4, pp. 656–667, 1994.

[31] B. A. Julstrom, “Adaptive operator probabilities in a genetic algorithm that applies three operators,” in *Proceedings of the 1997 ACM symposium on Applied computing*. ACM, 1997, pp. 233–238.

[32] C. M. Fonseca and P. J. Fleming, “Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization,” in *Proceedings of the 5th International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, pp. 416–423.

[33] C. L. López, “Heuristic Hill-Climbing as a Markov Process,” in *International Conference on Artificial Intelligence: Methodology, Systems, and Applications*. Springer Berlin Heidelberg, 2008, pp. 274–284.

[34] D. Adler, “Genetic algorithms and simulated annealing: a marriage proposal,” in *IEEE International Conference on Neural Networks*. IEEE, 1993, pp. 1104–1109.

[35] G. Zhang, H. Rong, F. Neri, and M. J. Pérez-Jiménez, “An optimization spiking neural P system for approximately solving combinatorial optimization problems,” *International Journal of Neural Systems*, vol. 24, no. 05, p. 1440006, 2014.