

Received: 27 December, 2021; Accepted: 5 April, 2022; Publish: 24 April, 2022

Automatic Programming As An Open-Ended Evolutionary System

Sebastian Fix, Thomas Probst, Oliver Ruggli, Thomas Hanne and Patrik Christen

FHNW University of Applied Sciences and Arts Northwestern Switzerland, Institute for Information Systems,
Riggenbachstrasse 16, 4600 Olten, Switzerland
patrik.christen@fhnw.ch

Abstract: Combinatorial evolution – the creation of new things through the combination of existing things – can be a powerful way to evolve rather than design technical objects such as electronic circuits. Intriguingly, this seems to be an ongoing and thus open-ended process creating novelty with increasing complexity. Here, we employ combinatorial evolution in software development. While current approaches such as genetic programming are efficient in solving particular problems, they all converge towards a solution and do not create anything new anymore afterwards. Combinatorial evolution of complex systems such as languages and technology are considered open-ended. Therefore, open-ended automatic programming might be possible through combinatorial evolution. We implemented a computer program simulating combinatorial evolution of code blocks stored in a database to make them available for combining. Automatic programming in the sense of algorithm-based code generation is achieved by evaluating regular expressions. We found that reserved keywords of a programming language are suitable for defining the basic code blocks at the beginning of the simulation. We also found that placeholders can be used to combine code blocks and that code complexity can be described in terms of the importance to the programming language. As in a previous combinatorial evolution simulation of electronic circuits, complexity increased from simple keywords and special characters to more complex variable declarations, class definitions, methods, and classes containing methods and variable declarations. Combinatorial evolution, therefore, seems to be a promising approach for open-ended automatic programming.

Keywords: Automatic programming, combinatorial evolution, open-endedness

I. Introduction

Genetic algorithms and evolutionary computation in general are widely used for solving optimisation problems [1, 2, 3, 4]. Such algorithms [5, 6, 7] follow the paradigm of biological evolution. They consist of a collection of virtual organisms, where every organism represents a possible solution to a given problem. Some fitness measure is then calculated for each organism in an iterative process and it tries to find improved solutions by forming random mutations and

crossovers on them.

In contrast to such evolutionary computation, *combinatorial evolution* as proposed by W. Brian Arthur [8, 9], makes no modifications to the organisms themselves. New solutions are formed through the combination of existing components which then form new solutions in later iterations with the goal of satisfying certain needs. The more useful a combination is, the higher is its need rating. Combining existing components to construct new components can be observed in the evolution of technology [8, 9]. For instance, the invention of radar was only possible through combining simpler electronic parts fulfilling functions like amplification and wave generation [10]. In order to investigate combinatorial evolution, Arthur and Polak [10] created a simple computer simulation, where electronic circuits were evolved in a combinatorial manner. Their simulation started by randomly combining primitive elementary logic gates and then used these simpler combinations for more complicated combinations in later iterations. Over time, a small number of simple building blocks was transformed into many complicated ones, where some of them might be useful for future applications. It was concluded that combinatorial evolution allows building some kind of library of building blocks for the creation of future and more complicated building blocks.

Intriguingly, combinatorial evolution is a key ingredient to achieve open-ended evolution [11, 12], that is the ongoing creation of novelty [13, 14]. This contrasts classical computational approaches where the aim is to converge towards a solution as fast as possible. Computational approaches according to open-ended evolution are therefore not more efficient but they are more creative since they generate ongoing novelty.

Here we want to explore whether combinatorial evolution could be also applied to software development, more specifically to automatic programming to eventually make it open-ended [15]. An early idea of automatic programming was to implement high-level programming languages that are more human readable resulting in compilers, which produce low-level programs – down to machine code – from human readable syntax [16]. However, human input in some form was still needed and the programming task was simply transferred to a higher level. Furthermore, the software solution is lim-

ited by the programmer's capabilities and creativity. Language therefore remains a barrier between programmers and computers. A way around this barrier would be to let the computer do the programming (also occasionally denoted as metaprogramming [17]), which might even lead to better programs. Koza [18] addressed this issue through genetic programming, where populations of computer programs are generated by a computer using genetic algorithms. The problem space consists of programs that try to solve (or approximately solve) problems. It has been demonstrated that random mutations and crossovers in source code can effectively contribute in creating new sophisticated programs [19]. Therefore, it seems possible to define a programming task and let a computer do the programming. However, looking at the process of software development, programming seems more comparable to technological rather than biological evolution. Existing libraries or algorithms are often integrated into new software without the necessity of modifying them. Therefore, an automatic programming approach that creates new computer programs by means of combinatorial evolution might be an interesting alternative to genetic programming. Also, due to open-endedness, combinatorial evolution holds the promise to be more creative generating ongoing novelty. In the present study we investigate ways to define a programming task for automatic programming through combinatorial evolution including the evaluation of the generated code with a need rating. Our research question is whether it is possible to generate computer programs of increasing complexity using automatic programming through combinatorial evolution. Specifically, we ask what kind of basic code blocks are needed at the beginning? How are these code blocks implemented to allow them to combine? How can code complexity be measured?

II. Automatic Programming

Since the development of computers, it has been a challenge to optimise and adapt program code to access the potential performance of a computer. While the computational power of computers has been steadily increasing in recent years, program code is still limited by the ability of programmers to create efficient and functioning code. Programming languages have also evolved over the past decades. The development of programming languages has sought to provide programmers with abstractions at higher levels. However, this also led to limitations, especially regarding performance and creativity. It is thus intriguing to shift the programming to the computer itself. Most of the programming is currently done by human programmers, which often leads to a time-intensive and error-prone process of software development. The idea that computers automatically create software programs has been a long-standing goal [20] with the potential to streamline and improve software development. Automatic programming was first considered in the 1940s describing the automation of a manual process in general and with the goal to maximise efficiency [21]. Later, automatic programming was considered a type of computer programming in which code is generated using tools that allow developers to write code at a higher level of abstraction [21]. There are two main types of automatic programming: *application generators* and *generative programming*. Cleaveland [22]

describes the development of application generators as the use of high-level programming models or templates to translate certain components into low-level source code. Generative programming, on the other hand, assists developers in writing programs. This can be achieved, e.g. by providing standard libraries as a form of reusable code [17]. In generative programming it is crucial to have a domain model, which consists of three main parts: a problem space, a solution space, and a configuration knowledge mapping that connects them [23]. The problem space includes the features and concepts used by application engineers to express their needs. These can be textual or graphical programming languages, interactive wizards, or graphical user interfaces. The solution space consists of elementary components with a maximum of combinability and a minimum of redundancy. The configuration knowledge mapping presents a form of generator that translates the objects from the problem space to build components in the solution space [17]. Most recently, automatic programming shifted towards higher level programming languages and incorporating even more abstraction [24].

While these kinds of automatic programming heavily depend on human interaction and thus the capabilities and creativity of programmers, genetic programming can be regarded an attempts to reduce this dependency and shift the focus to automation done by the computer itself. Koza [18] describes genetic programming as a type of programming in which programs are regarded as genes that can be evolved using genetic algorithms [25, 26]. It aims to improve the performance of a program to perform a predefined task. According to Becker et al. [20], a genetic algorithm takes, as an input, a set of instructions or actions that are regarded as genes. A random set of these instructions is then selected to form an initial sequence of DNA. The whole genome is then executed as a program and the results are scored in terms of how well the program solves a predefined task. Afterwards, the top scorers are used to create offspring, which are rated again until the desired program is produced. To find new solutions, evolutionary techniques such as crossover, mutation, and replication are used [27]. Crossover children are created by picking two parents and switching certain components. Another technique is mutation, which uses only one individual parent and randomly modifies its parts to create a new child. Sometimes parents with great fitness will be transferred to the next iteration without any mutation or crossover because they might do well in later steps as well. The specific choice or design of the variation operators (mutations and crossover) are also a significant limitation of the techniques as usually infeasible solutions might result from them. Whereas some applications of simple genetic algorithms may be unaffected by this problem, search problems with various kinds of restrictions might suffer from frequently infeasible offspring solutions. This is especially true for genetic programming as the evolutionary approach is applied to programs. Without specific consideration, random changes such as mutations may already lead to syntactically invalid programs or source code suffering from compile-time errors. To avoid this, programming languages such as Lisp are preferred which are based on a tree representation of the program. Such a representation makes it easier to define variation operators which keep at least the syntactic correctness of a program. For in-

stance, nodes of a tree (together with their subtrees) could be exchanged between different solutions (crossover) or replaced by random new nodes or subtrees. However, occasionally genetic programming approaches have also been developed and tested for more traditional imperative programming languages such as Java [28].

III. Combinatorial Evolution

With combinatorial evolution, new solutions build on combinations of previously discovered solutions. Every evolution starts with some primitive, existing building blocks and uses them to build combinations. Those combinations are then stored in an active repertoire. If the output satisfies a need better than an earlier solution, it replaces the old one and will be used as the building block in later iterations. Building blocks are thus not modified, they are combined together creating new building blocks. The result is a library of functionalities that may be useful for a solution in the future [8, 9].

As Ogburn [29] suggested, the more equipment there is within a material culture, the greater the number of inventions are. This is known as the Ogburn's Claim. It can therefore be inferred that the number and diversity of developed components as well as their technological developments matters because next generation components build upon the technological level of the previous, existing components. To investigate this, Arthur and Polak [10] created a simple computer simulation to 'discover' new electronic circuits. In their simulation, they used a predefined list of truth tables of basic logic functions such as full adders or n-bit adders. Every randomly created combination represented a potential satisfaction of a need, which was then tested against this list. If the truth table of a newly created circuit matched one from the predefined list, it is added to the active repertoire as it fulfils the pre-specified functionality. Sometimes, it also replaced one that was found earlier, if it used fewer parts and therefore would cost less. New technologies in the real world are not usually found by randomly combining existing ones nor do they exist in a pre-specified list to be compared against. Nevertheless, their needs are generally clearly visible in economics and current technologies [10].

Combinatorial evolution is in general an important element of evolutionary systems. Stefan Thurner and his colleagues developed a general model of evolutionary dynamics in which the combination of existing entities to create new entities plays a central role [11, 30, 12]. They were able to validate this model using world trade data [31], therefore underlining the importance of evolutionary dynamics in economic modelling in general and combinatorial interactions in particular. The model shows punctuated equilibria that are typical for open-ended evolutionary systems [11, 30, 12].

IV. Code Complexity

Genetic algorithms have been used for automatic programming already, however, a large number of iterations are required to significantly increase code complexity in order to solve more complex problems [32]. It therefore seems beneficial to use combinatorial evolution in which complexity seems to increase in fewer steps and thus less time.

Code complexity has been measured in this context with dif-

ferent approaches. The cyclomatic complexity of a code is the number of linearly independent paths within it [33]. For instance, if the code contains no control flow elements (conditionals), the complexity would be 1, since there would be only a single path through the code [34]. If the code has one single-condition IF statement, the complexity would be 2 because there would be two paths through the code – one where the IF statement evaluates to TRUE and another one where it evaluates to FALSE [34]. Two nested single-condition IFs (or one IF with two conditions) would produce a complexity of 3 [35, 34]. According to Garg [36], cyclomatic complexity is one of the most used and renowned software metrics together with other proposed and researched metrics, such as the number of lines of code and the Halstead measure. Although cyclomatic complexity is very popular, it is difficult to calculate for object-oriented code [37].

V. Methods

A. Development Setup and Environment

We used the programming language Java though other programming languages would have been feasible as well. The development environment was installed on VirtualBox – an open source virtualisation environment from Oracle. Oracle Java SE Development Kit 11 was used with Apache Maven as build automation tool. To map the existing code with a database, Hibernate ORM was used. It allows mapping object-oriented Java code to a relational database. Furthermore, code versioning with GitHub was used.

B. Simulation

Simulations are initialised by adding some basic code building blocks into a repository. The first simulation iteration then starts by randomly selecting code blocks from this repository. Selected blocks are then combined into a new code block, which subsequently gets analysed for its usefulness and complexity. Based on this analysis, the code block is assigned a value. Nonsense code, which is the most common result when randomly combining keywords of a programming language, are assigned a value of 0 and not used any further. Only code blocks with a value greater than 0 are added to the repository and consequently have a chance of being selected in a later iteration.

C. Code Building Blocks

Defining the initial code building blocks of the repository is a challenging task since they should not contain too much predefined logic and on the other hand have a minimal complexity in order to allow creative combinations without limiting or predefining too much. One way would be to define code snippets with placeholders, e.g. a code snippet of a method definition where the body is a placeholder. Placeholders are important for the combination of code blocks – they define where a certain code block can be linked to another code block in the repository. However, some preliminary experiments in which code snippets with placeholders were predefined showed that this approach would limit the creativity and complexity of the automatic programming solution by the predefined snippets. The simulation would only create

program logic that is already given by the basic set of code blocks.

To overcome this limitation, we defined basic code building blocks according to keywords and special characters of the Java programming language, e.g. the keywords **int**, **for**, **class**, and `String` as well as the special characters `&`, `=`, `;`, and `{`. Additionally, we defined three more extra code blocks: First, `PLACEHOLDER` to define where blocks allow other code blocks to be combined and integrated. This is particularly important for nesting certain code elements, such as methods that must be nested into a class construct to be valid Java code. Second, `NAME` to name something, e.g. classes, methods, and variables. And third, the special keyword `main` in the main method definition. A complete list of the defined basic code blocks is provided below in figs. 1- 3.

D. Selecting and Combining Code Blocks

During the selection process, new source code is generated based on combinations of existing code blocks from the repository. The chance that a particular code block is selected depends on its classification value (see next section). In a first step, a helper function defines a random value of how many code blocks are taken into consideration in the current iteration. There is a minimum of two code blocks required to generate a new code block. The maximum number can be predefined in the program. Arthur and Polak [10] combined up to 12 building blocks. To reduce the number of iterations needed for receiving valid Java code, a maximum of eight blocks turned out to be a good limit. After randomly defining the number of code blocks to be combined, the weighted random selection of code blocks based on their classification value follows. Instead of simply chaining all selected code blocks together, there is also the possibility to nest them into a placeholder if available. A random function decides whether a code block is nested into the placeholder, or simply added to the whole code block. This procedure is important because program code usually exhibits such nested structures.

E. Code Analysis and Building Block Classification

After the selection and combination process, the newly generated source code is passed into the classification function where it gets analysed. The classification process is required to weight the different code blocks according to their relevance in the Java programming language and to see whether the code evolved with respect to complexity. This is achieved with regular expression patterns, which allow identifying relevant Java code structures such as classes and methods that can be weighted with predefined classification values for these code structures. Basic structures such as variable declarations are assigned a value of 1. More elaborate structures such as classes have a value of 2 and even more complicated structures such as methods have a value of 3. If a structure contains several of these substructures, their classification values is added. An important structure in many programming languages is the declaration of a variable. With the following regular expression, any declaration of the value types **boolean**, **byte**, **char**, **double**, **float**, **int**, **long**, and **short** are detected:

```
(PLACEHOLDER(?!PLACEHOLDER)) ?
(boolean|byte|char|double|float |
int|long|short) NAME;
(PLACEHOLDER(?!PLACEHOLDER)) ?
```

Other important elements are brackets. E.g. they are used in methods and classes specifying the body. The syntax is given by the programming language. Placeholders inside brackets are important, they allow new code to be injected into existing code blocks in future combinations. We therefore created the following regular expression:

```
^( \{PLACEHOLDER\} | \ (PLACEHOLDER\)) $
```

As already shown in the simple simulation with electronic circuits [10], one needs a minimal complexity of the initial building blocks to be able to generate useful and more complex future combinations. Classes and methods are essential to build anything complex in Java. Therefore, regular expressions were implemented to identify valid classes and methods. Valid means, the element is closed and it successfully compiles. Variable declarations and methods are allowed to be nested in the class structure. The following regular expression to detect classes was developed:

```
(protected|private|public) class NAME \{
( (boolean|byte|char|double|float |
int|long|short) NAME;
| (protected|private|public) void NAME\ (
( (boolean|byte|char|double|float |
int|long|short) NAME)?\ ) \{
( (boolean|byte|char|double|float |
int|long|short) NAME;
| PLACEHOLDER(?!PLACEHOLDER) *\ }
| PLACEHOLDER(?!PLACEHOLDER) *\ } $
```

A valid method needs to be correctly closed and can contain either a placeholder or a variable declaration. We wanted to influence the generated code as little as possible. Therefore, many attempts had to be done with formulating regular expression patterns and weighting the detected code structure accordingly. The following regular expression to detect methods was developed:

```
(PLACEHOLDER(?!PLACEHOLDER)) ?
(protected|private|public) void NAME\ (
( (boolean|byte|char|double|float |
int|long|short) NAME)?\ ) \{
( (boolean|byte|char|double|float |
int|long|short) NAME;
| PLACEHOLDER(?!PLACEHOLDER) *\ }
(PLACEHOLDER(?!PLACEHOLDER)) ?
```

F. Regular Expression Validation

In some preliminary experiments, we automatically compiled source code files of newly combined code blocks to check whether they are valid. However, this process is too time consuming to allow large numbers of iterations. An iteration required one to three seconds compilation time. As combinatorial evolution relies on rather large numbers of iterations, we instead used regular expressions to check whether newly combined code blocks compile and are thus valid. Java allows compiling regular expression into a pattern

abstract	default	goto	package	this
assert	do	if	private	throw
boolean	double	implements	protected	throws
break	else	import	public	transient
byte	enum	instanceof	return	true
case	extends	int	short	try
catch	false	interface	static	void
char	final	long	strictfp	volatile
class	finally	native	super	while
const	float	new	switch	
continue	for	null	synchronized	

Figure. 1: Complete list of keywords in Java.

'	&	.	^	=
-	(/	{	>
!)	:		
"	*	;	}	
%	,	[+	
]	<	

Figure. 2: Complete list of special characters in Java.

PLACEHOLDER
NAME
main

Figure. 3: Additionally defined basic code blocks in the present combinatorial evolution simulation.

Table 1: Examples of newly generated code blocks within the first 100'000 iterations of a combinatorial evolution simulation. Class refers to the classification value representing how useful the code block is in programming.

Iteration	Block	New Code Block	Class
4'647	25	short NAME ;	1
16'394	30	public void NAME	1
22'729	34	boolean NAME ;	1
50'419	42	protected class NAME	1
58'595	44	{ PLACEHOLDER }	1
93'722	55	public class NAME	1

Table 2: Examples of newly generated code blocks within a wide range of iterations of a combinatorial evolution simulation of 1.6 billion iterations. Class refers to the classification value representing how useful the code block in programming.

Iteration	Block	New Code Block	Class
58'903	45	protected class NAME { PLACEHOLDER }	2
112'609	61	public class NAME { PLACEHOLDER }	2
> 1 · 10 ⁹	168	public void NAME () { PLACEHOLDER }	3
> 1 · 10 ⁹	169	public class NAME { public void NAME () { PLACEHOLDER } short NAME ; } protected class NAME { boolean NAME ; public void NAME () { PLACEHOLDER } }	6
> 1 · 10 ⁹	170	protected class NAME { boolean NAME ; public void NAME () { PLACEHOLDER } }	6

object, which can then be used to match it with a string containing the code to be tested. It turned out to be a much faster alternative to the actual compilation of source code files.

VI. Results

Using Java keywords for the initial basic code blocks, we found the first useful combinations of code blocks within 100'000 iterations in a simulation of 1.6 billion iterations, which took approximately 5 hours on a desktop computer. These code blocks mainly consisted of combinations of three basic building blocks classified with a value of 1. Table 1 shows some examples that were found in the simulation. Such combinations are typically assigned a small classification value due to their simplicity, keeping in mind that only code blocks that are assigned values greater than 0 are added to the code block repository for later combinations. It did not take long for the combinatorial evolution simulation to find the first combinations that consisted of previously found code blocks as illustrated in Table 2. E.g. code block 45 – which consists of block 42 and block 44 – was found only 308 iterations later. Though it took some time to find a Java method in code block 168, only a small number of iterations later, many subsequent code blocks followed with

higher classification values. Code blocks 169 and 170 characterise Java classes that contain methods and declarations of variables.

Only after 308 iterations after **protected class** NAME and { PLACEHOLDER } were found, the simulation combined these code blocks to form a class with a placeholder in it leading to a higher classification of value 2. Shortly after this code block for classes with a placeholder in it was found, around 100 combinations followed that represented classes containing variable declarations. This can be observed by looking at the dots between 100k and 10m in Fig. 4. The larger the repository grew, the smaller became the probability of a Java method being found. The dot with value 3 at around 1 billion iterations is the method code block 168 in Table 2. All the following code blocks built on this one.

It took considerably longer to jump to the next higher classification value of 3 as compared to the jump from value 1 to 2. More than $1 \cdot 10^9$ iterations were required to evolve a method with a placeholder in it, classified with a value of 3. From there it only took a few iterations to jump to classification values of 4, 5, and even 6. Combinations of a method with a variable declaration were assigned a classification value of 4, combinations with a class were assigned a classification value of 5, and combining all three resulted in the assignment of a classification value of 6. E.g., a class with a variable and a method declaration in it: **protected class** NAME { **boolean** NAME ; **public void** NAME () { PLACEHOLDER } }.

VII. Discussion and Conclusion

In the present paper, we investigated whether it is possible to generate computer programs of increasing complexity using automatic programming through combinatorial evolution since this would make it an open-ended process. Specifically, we wanted to know what kind of basic code blocks are needed at the beginning of a simulation, how are these code blocks implemented to allow them to combine, and how can code complexity be measured. To start the first iteration of the combinatorial evolution simulation we needed to define code blocks that existed in the programming framework Java. As initial code blocks we defined reserved keywords of the Java programming language that are used to define classes, methods, initialise variables, and so on. This also includes some special characters used in the programming language that we also added. Placeholders within code blocks are used to allow combining code blocks and thus source code. Newly generated code blocks are assigned a classification value according to their structure, which represents code complexity. The combinatorial evolution simulation generated code blocks including classes, methods, variables, and combinations thereof. It therefore generated code of increasing complexity.

Regarding measuring complexity, different approaches to do so, e.g. determining the number of lines of code and McCabe's cyclomatic complexity [35], have been taken into consideration but the code blocks from the outcomes after nearly 2 billion iterations were in our opinion still too short to implement these complexity measures. Two factors were important why we did not use McCabe's cyclomatic complexity [35]. First, it did not generate the required main method

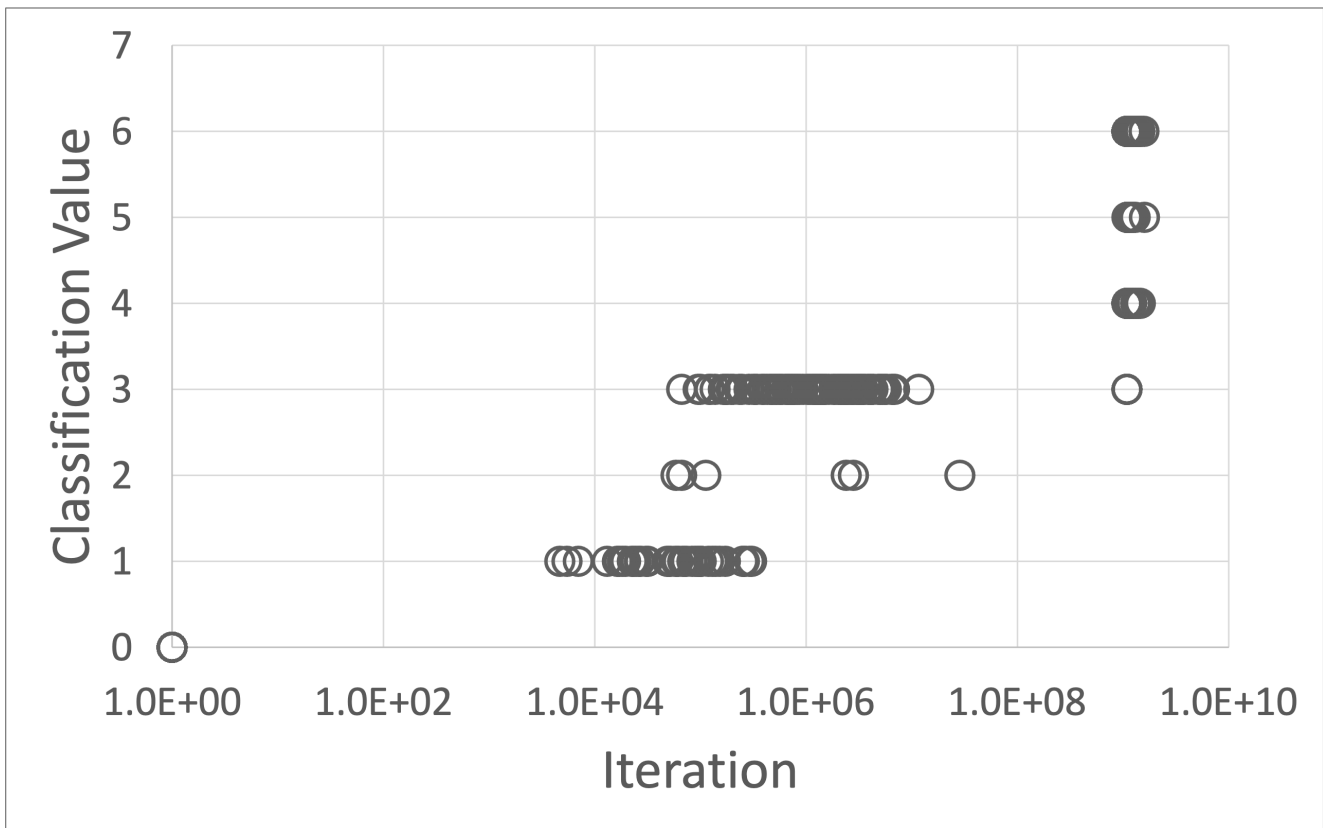


Figure 4: Classification values of newly generated code blocks over combinatorial evolution iterations.

within a reasonable number of iterations, so there was no starting point. Second, we decided to not have the decision code block assigned a value greater than 0 in the initial code blocks. Without any of these code blocks, the complexity would always be evaluated as 1.

We conclude that the combinatorial evolution simulation clearly shows how Java code can be automatically created using combinatorial evolution. Simple keywords and special characters were successfully combined into more complex and different structures like variable declarations or methods and in later iterations they even got combined into more sophisticated results such as classes consisting of methods and variable declarations. We also conclude that due to combinatorial evolution, open-ended automatic programming could be achieved, indicating an intriguing approach if creativity is important.

The reached limitations of complexity show that further research is required. Similar observations for genetic programming [32] suggest that more advanced evolutionary operators could be useful. However, already when starting with further elaborated code blocks or when reaching them during previous combinatorial evolution, the goal of automatic programming might come much closer. Therefore, forthcoming research may also study the concept with much increased computational power and distributed computing.

References

- [1] Said Tkatek, Otman Abdoun, Jaafar Abouchabaka, and Najat Rafalia. A Parallel Genetic Algorithm to Optimize the Massive Recruitment Process. *International Journal of Computer Information Systems and Industrial Management Applications*, 13:364–371, 2021.
- [2] Paulo Salgado and Paulo Afonso. Evolutionary Genes Algorithm To Path Planning Problems. *International Journal of Computer Information Systems and Industrial Management Applications*, 10:65–72, 2019.
- [3] Chandra Naik and D. Pushparaj Shetty. Differential Evolution Meta-Heuristic Scheme for k-Coverage and m-Connected Optimal Node Placement in Wireless Sensor Networks. *International Journal of Computer Information Systems and Industrial Management Applications*, 11:132–141, 2019.
- [4] Cláudio A. D. Silva, Carlos Grilo, and Catarina Silva. Model Optimisation for Server Loading Forecasting with Genetic Algorithms. *International Journal of Computer Information Systems and Industrial Management Applications*, 11:178–191, 2019.
- [5] Altaf Q. H. Badar. *Evolutionary Optimization Algorithms*. CRC Press, Boca Raton, 2022.
- [6] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York, 1996.

- [7] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.
- [8] W. Brian Arthur. *The Nature of Technology: What it Is and How it Evolves*. Free Press, New York, 2009.
- [9] W. Brian Arthur. How We Became Modern. In Shuzhen Sim and Benjamin Seet, editors, *Sydney Brenner's 10-on-10: The Chronicles of Evolution*. Wildtype Books, 2018.
- [10] W. Brian Arthur and Wolfgang Polak. The evolution of technology within a simple computer model. *Complexity*, 11(5):23–31, 2006.
- [11] Stefan Thurner. A Simple General Model of Evolutionary Dynamics. In Hildegard Meyer-Ortmanns and Stefan Thurner, editors, *Principles of Evolution: From the Planck Epoch to Complex Multicellular Life*, The Frontiers Collection, pages 119–144. Springer, Berlin Heidelberg, 2011.
- [12] Stefan Thurner, Rudolf Hanel, and Peter Klimek. *Introduction to the Theory of Complex Systems*. Oxford University Press, New York, 2018.
- [13] Wolfgang Banzhaf, Bert Baumgaertner, Guillaume Beslon, René Doursat, James A. Foster, Barry McMullin, Vinicius Veloso de Melo, Thomas Miconi, Lee Spector, Susan Stepney, and Roger White. Defining and simulating open-ended novelty: requirements, guidelines, and challenges. *Theory in Biosciences*, 135(3):131–161, 2016.
- [14] Tim Taylor. Evolutionary Innovations and Where to Find Them: Routes to Open-Ended Evolution in Natural and Artificial Systems. *Artificial Life*, 25(2):207–224, 2019.
- [15] Patrik Christen. Modelling and implementing open-ended evolutionary systems. *The Fourth Workshop on Open-Ended Evolution (OEE4), The 2021 Conference on Artificial Life (ALife)*, 2021. arXiv:2201.06858v1 [cs.NE].
- [16] Wendy Hui Kyong Chun. On Software, or the Persistence of Visual Knowledge. *Grey Room*, 18:26–51, 2005.
- [17] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, 2000.
- [18] John R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112, 1994.
- [19] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. Genetic programming: An introductory tutorial and a survey of techniques and applications. *University of Essex, UK, Tech. Rep. CES-475*, pages 927–1028, 2007.
- [20] Kory Becker and Justin Gottschlich. AI Programmer: Autonomously Creating Software Programs Using Genetic Algorithms. *arXiv*, 2017.
- [21] David Lorge Parnas. Software aspects of strategic defense systems. *ACM SIGSOFT Software Engineering Notes*, 10(5):15–23, 1985.
- [22] James Craig Cleaveland. Building application generators. *IEEE Software*, 5(4):25–33, 1988.
- [23] Krzysztof Czarnecki. Perspectives on generative programming. In *SFB 501 "Development of Large Systems with Generic Methods"*, 2003.
- [24] Michael O'Neill and Lee Spector. Automatic programming: The open issue? *Genetic Programming and Evolvable Machines*, 21(1-2):251–262, 2020.
- [25] John H. Holland. Genetic Algorithms. *Scientific American*, 267(1):66–72, 1992.
- [26] John H. Holland. *Signals and Boundaries: Building Blocks for Complex Adaptive Systems*. The MIT Press, Cambridge, 2012.
- [27] Nelishia Pillay and Caryl K. A. Chalmers. A hybrid approach to automatic programming for the object-oriented programming paradigm. In *Proceedings of the 2007 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries*, SAICSIT '07, page 116–124, New York, NY, USA, 2007. Association for Computing Machinery.
- [28] Brendan Cody-Kenny, Edgar Galván-López, and Stephen Barrett. locoGP: improving performance by genetic programming java source code. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 811–818, 2015.
- [29] William Fielding Ogburn. *Social Change: With Respect to Culture and Original Nature*. B. W. Huebsch, New York, 1922.
- [30] Stefan Thurner. The Creative Destruction Of Evolution. In Shuzhen Sim and Benjamin Seet, editors, *Sydney Brenner's 10-on-10: The Chronicles of Evolution*. Wildtype Books, 2018.
- [31] Peter Klimek, Ricardo Hausmann, and Stefan Thurner. Empirical Confirmation of Creative Destruction from World Trade Data. *PLoS ONE*, 7(6):e38924, 2012.
- [32] Adam Tyler Harter. *Advanced techniques for improving canonical genetic programming*. Missouri University of Science and Technology, 2019.
- [33] Christof Ebert, James Cain, Giuliano Antoniol, Steve Counsell, and Phillip Laplante. Cyclomatic complexity. *IEEE Software*, 33(6):27–29, 2016.

- [34] Wikipedia contributors. Cyclomatic complexity — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Cyclomatic_complexity&oldid=1054490449, 2021. [Online; accessed 19-November-2021].
- [35] Tom J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.
- [36] Ankita Garg. An approach for improving the concept of Cyclomatic Complexity for Object-Oriented Programming. *arXiv*, 2014.
- [37] Mir Muhammd Suleman Sarwar, Sara Shahzad, and Ibrar Ahmad. Cyclomatic complexity: The nesting problem. In *Eighth International Conference on Digital Information Management (ICDIM 2013)*, pages 274–279. IEEE, 2013.