Leap mapping: Improving Grammatical Evolution for Modularity Problems

Allan de Lima
University of Limerick
Limerick, Ireland
Allan.DeLima@ul.ie

Samuel Carvalho
Technological University of the Shannon: Midlands Midwest
Limerick, Ireland
samuel.carvalho@tus.ie

Douglas Mota Dias*
University of Limerick
Limerick, Ireland
Douglas.MotaDias@ul.ie

Joseph P. Sullivan
Technological University of the Shannon: Midlands Midwest
Limerick, Ireland
joe.sullivan@tus.ie

Conor Ryan
University of Limerick
Limerick, Ireland
Conor.Ryan@ul.ie

ABSTRACT

We introduce *Leap* mapping, a new mapping process for Grammatical Evolution (GE), which spreads introns within the effective length of the genome (the part of the genome consumed while mapping), preserving information for future generations and performing less disruptive crossover and mutation operations than standard GE. Using the exact same genotypic representation as GE, Leap mapping reads the genome in separate parts named 'frames', where the size of each is the number of production rules in the grammar. Each codon inside a frame is responsible for mapping a different production rule of the grammar. The process keeps consuming codons from the frame until it needs to map again a production rule already mapped with that frame. At this point, the mapping starts consuming codons from the next frame. We assessed the performance of this new mapping in some benchmark problems, which require modular solutions: four Boolean problems and three versions of the Lawnmower problem. Moreover, we compared the results with the standard mapping procedure and a multi-genome version.

CCS CONCEPTS

• **Computing methodologies** → **Genetic programming**.

KEYWORDS

Grammatical Evolution, mapping, introns

ACM Reference Format:
Allan de Lima, Samuel Carvalho, Douglas Mota Dias, Joseph P. Sullivan, and Conor Ryan. 2023. Leap mapping: Improving Grammatical Evolution for Modularity Problems. In *Genetic and Evolutionary Computation Conference Companion (GECCO '23 Companion), July 15–19, 2023, Lisbon, Portugal.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3583133.3590680

*Also with Rio de Janeiro State University.

GECCO '23 Companion, July 15–19, 2023, Lisbon, Portugal

ACM ISBN 979-8-4007-0120-7/23/07.
https://doi.org/10.1145/3583133.3590680

1 INTRODUCTION

Genetic Programming (GP) [8] and Grammatical Evolution (GE) [10, 14, 16] are methods inspired by Darwin's theory of biological evolution, in which we evolve a population of solutions. Following the principles of natural selection, the fittest solutions produce offspring by using genetic operators like crossover and mutation to pass on their genetic material throughout generations.

In GE, the genotype is represented by a list of integers and mapped into a phenotype using the *modulo* operator. This mapping happens from left to right in the genome, consuming codons as it does so. Once the process is terminated, the remaining unused part of the genome consists of *introns*. Despite being the non-effective part of the genome, introns have important tasks to perform during evolution, such as, for example, avoiding bias towards some production rules [11]. In fact, after crossover or mutation, introns can turn into effective codons, helping to generate an offspring completely different from its respective parents, and therefore preserving diversity in the population [1] and expanding the search space for better solutions [15]. When using the standard mapping of GE, these introns are clustered at the end of the genotype, and spreading them throughout the genome aiming to preserve more of the characteristics of the parents across generations is the main purpose of this work.

Considering its mapping process, GE has attracted some criticism [12] due to its low locality. For example, a mutation in a single codon can change all the choices made in the mapping process after that codon, which can hinder the search's exploitation of close-to-optimal solutions. The closer the codon is to the beginning of the genome, the stronger this *ripple effect* can be.

Although the Leap mapping still shows low locality, its codons map specific production rules, even after crossover and mutation operations, regardless of whether it was an effective codon or not. Even when the ripple effect makes significant changes to a phenotype, the previous respective genotypic information is in the same position in the frames, making it more likely to be rediscovered in future generations. In addition, Leap mapping does not need a different representation of the genome, making it simpler to implement.

2 BACKGROUND

GE is an evolutionary algorithm used to build programs in an arbitrary language. The genome in a GE individual is a variable-length sequence of codons, where crossover and mutation are performed to generate offspring for the next generation. This sequence can be mapped into a phenotype, which is usually used for assessing the fitness score of the individual. The mapping process is made through a grammar, a set of production rules consisting of *terminals* and *non-terminals*. This process starts from the start rule, and the choices are made one after another using the modulo operator between a codon and the number of possible choices of the production rule regarding the non-terminal currently being mapped. If all codons are consumed, and there are still non-terminals to be mapped, the individual is considered invalid.

Multi-chromosomal GE (MCGE) [6] represents the genome with multiple chromosomes, where the number of chromosomes is equal to the number of production rules, and each chromosome maps a different production rule. The mapping process is similar to standard GE, and it is made from left to right. Then, the introns are concentrated at the end of each chromosome. The crossover between two parents on MCGE is performed in chromosomes related to the same production rule.

3 LEAP MAPPING

Before starting the Leap mapping process, the frame size is defined as the number of production rules with more than one choice in the grammar being used. We consider only the production rules with at least two choices because, like in the original approach of GE, Leap mapping does not consume a codon when there is a single choice.

Similarly to the original approach of GE, we start the mapping process with the first rule of the grammar, and we consume the first codon of the genome to map according to it. Secondly and so on, if the next non-terminal to be mapped has its respective production rule in the i-th position of the grammar, we consume the codon in the i-th position of the current frame being used. Once we need to consume a codon in a position which was already used in the current frame, we "leap" to the following frame – hence the name *Leap mapping*. Note that we never move backwards to consume unused codons from previous frames, but in the current frame, we can consume codons in preceding positions to those ones already consumed. For example, if we start to consume a frame in the fifth position, and in the next step, we need to consume a codon from the first position of the same frame, that is allowed.

Considering the proposed structure for Leap mapping, an individual mapped using a grammar with a single production rule has the same genome length as a standard GE individual with the same phenotype. On the other hand, the larger the number of production rules, the larger a genome using Leap mapping will be compared to a GE genome with the same phenotype. In the worst scenario, the effective length of a Leap genome is the number of production rules in the grammar times the effective length of the equivalent GE genome. However, except in the case when using grammars with a single production rule, this situation is unlikely to happen since the same production rule would need to be used throughout the entire mapping process.

Figure 1 shows an example of an individual being mapped using Leap mapping. The given grammar has three production rules: the first has two choices, the second has four, and the third has three. Thus, each frame has a length of three, one for each production rule. We start the mapping process from the rule `<e>` consuming the codon 220, which maps to the choice `<op>`. Next, we map `<op>` to `nor(<e>,<e>)` by consuming the codon in the second position of the first frame; as we are mapping from the second production rule, that codon is currently unused. In the next step, we need to map a choice from the rule `<e>`, but the first position of the frame has already been consumed, so we move the mapping process to the next frame and consume the codon 15. After that, we need to map a choice from the third rule, so we consume a codon in the third position of the current frame. We continue the process until the individual is fully mapped. In the end, we consumed ten codons to map the genome, and its effective length is 15, which is 50% longer than an equivalent effective length in a GE individual.

When using Leap mapping, we apply a simple one-point crossover adapted to the Leap procedure to keep the codons in their original position inside the frames. Thus, with two parents, we randomly select a frame in each parent, and then we randomly select a position inside the frame, which will be used to cut both parents.

The mutation operator is performed similarly to GE. According to the mutation probability, we flip a codon to a random value inside the codon size. We also perform mutation only within the effective length of the genome and only apply it to codons that are actually being mapped to ensure that each mutation event at least has a chance of creating a different individual.

4 EXPERIMENTAL SETUP

We used GRAPE[1] [2], an implementation of GE built on top of the DEAP framework [4]. We implemented our proposed mapper and also the specific one-point crossover and mutation operators presented in the previous section. In addition, we implemented MCGE to run experiments and compare the results with our approach. For both MCGE and Leap, we implemented sensible initialisation [13], a method based on the ramped half and half employed by GP. This method generates an initial population of phenotypes within a pre-defined range of depths and then generates their respective genotypes. In our implementation, we follow the same ideas of its original proposal to generate phenotypes, and we generate their equivalent genotypes for Leap mapping and MCGE.

Table 1 summarises the parameters used in GE, Leap and MCGE experiments. The population size is 1,000 for the Lawnmower problems and 16,000 for the Boolean problems. We set the maximum depth to 35 for most of the problems to permit individuals to grow freely. However, since this value was too small to solve the two most complex versions of the Lawnmower problem, we set up the maximum depth to 70 for the 12x12 problem and 90 for the 14x14 problem.

Listing 1 shows some of the grammars used in our experiments. From the 4-bit Parity grammar, we just add more inputs to address the remaining parity problems. We adopted the Mean Classification Error (MCE) as the fitness function.

[1] https://github.com/bdsul/grape

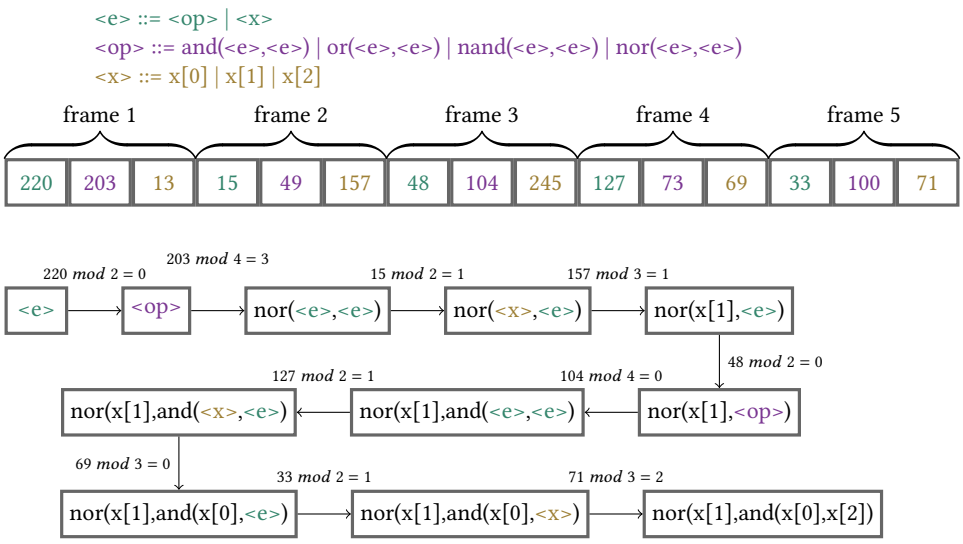

Figure 1: Example of an individual being mapped with the Leap process.

Table 1: Experimental parameters

Parameter type	Parameter value
Number of runs	30
Number of generations	200
Population size	1,000/16,000
Maximum depth	35/70/90
Elitism ratio	0
Selection method	tournament of 6
Mutation method	Codon-based integer flip [5]
Mutation probability	0.01
Crossover method	Variable one-point [5]
Crossover probability	0.8
Initialisation method	Sensible [13]
Maximum wraps	0
Codon size	255

```
<e>  ::=  <op> | <x>
<op> ::=  and(<e>,<e>) | or(<e>,<e>) | nand(<e>,<e>)
          | nor(<e>,<e>)
<x>  ::=  x[0] | x[1] | x[2] | x[3]
```

(a) 4-bit Parity

```
<expr> ::= o3 = <e>; o2 = <e>; o1 = <e>; o0 = <e>
<e>  ::=  <op> | <x>
<op> ::=  and(<e>,<e>) | or(<e>,<e>) | nand(<e>,<e>)
          | nor(<e>,<e>)
<x>  ::=  x[0] | x[1] | x[2] | x[3]
```

(b) 2-bit Multiplier

```
<e>  ::=  <op> | <x>
<op> ::=  v8a(<e>,<e>) | frog(<e>) | progn(<e>,<e>)
<x> ::=   left() | mow() | rv8()
```

(c) 8x8 Lawnmower

Listing 1: Grammars

Since the 2-bit Multiplier problem has an output of 4 bits, we evolve separate Boolean expressions, each regarding a different bit, although we map from a single genotype. We also use MCE as the fitness function in this case, but we calculate the fitness score comparing bit by bit instead of the whole output.

We implemented the Lawnmower problems following Koza's approach [9]. Listing 1c shows the grammar used for the 8x8 Lawnmower problem. We use equivalent functions for the 12x12 and 14x14 problems. As the fitness function for these problems, we use the number of squares which still contain grass.

5 RESULTS AND DISCUSSION

Table 2 presents the fitness analysis between Leap, standard GE and MCGE for all the problems. The average minimum fitness is reported. Standard deviations are also shown for all fitness values. To investigate the statistical significance of the results, first, the Shapiro-Wilk normality test was performed on all resulting fitness data sets, in which the data was found to be predominantly non-normal. Therefore, the non-parametric two-sided Wilcoxon test was used to compare the resulting fitness scores from the different methods at a given problem. The "p-value" column in the table shows the resulting p-values of these tests, and given the multiple comparisons performed here, a Bonferroni correction factor of 2 was used. Therefore, p-values below the 0.025 threshold are underlined, meaning that Leap mapping has significantly outperformed the method compared to it. It can be seen that Leap mapping consistently outperforms GE and MCGE for the Boolean problems. Meanwhile, on the Lawnmower problems, Leap mapping was still capable of outperforming GE, while MCGE was significantly better at the 8x8 problem (denoted by an asterisk), and no significant difference was found between the two methods on the 12x12 and 14x14 problems.

Table 2: Fitness scores and Wilcoxon Test p-values.

Problem	leap	GE	p-value	MCGE	p-value
2-bit Multiplier	0.0047 ± 0.0072	0.0354 ± 0.0213	3.23E-06	0.0547 ± 0.0251	1.70E-06
4-bit Parity	0.0187 ± 0.0366	0.0646 ± 0.0675	7.83E-03	0.0563 ± 0.0611	1.56E-02
5-bit Parity	0.1437 ± 0.0580	0.2219 ± 0.0595	3.32E-04	0.2313 ± 0.0237	5.55E-06
6-bit Parity	0.2453 ± 0.0509	0.3010 ± 0.0440	2.51E-04	0.3266 ± 0.0250	2.80E-06
8x8 Lawnmower	0.3333 ± 0.4714	1.9667 ± 1.1397	1.79E-05	0.0000 ± 0.0000	6.28E-03*
12x12 Lawnmower	1.2667 ± 0.9638	4.9667 ± 1.7792	1.82E-06	0.9667 ± 1.4020	2.28E-01
14x14 Lawnmower	0.9667 ± 0.7520	5.0000 ± 2.3238	2.44E-06	1.4333 ± 1.4302	1.23E-01

We can see in Table 3 the average remainders diversity [3] of the population in the last generation. This is a phenotypic diversity metric based on the remainders of the grammar mapping modulo operations, and it is calculated over valid individuals; this is the same as structural diversity [7], which returns the percentage of different valid phenotypes in the population, but since it is GE-specific, it has a lower cost to calculate. In this table, we can highlight that Leap has a high diversity of over 90% in all problems (i.e. over 90% of the phenotypes are different at the last generation), while GE and MCGE struggle to maintain a wide variety of phenotypes in the Boolean problems.

Table 3: Average remainders diversity in the last generation.

Problem	GE	MCGE	Leap
2-bit Multiplier	0.61 ± 0.19	0.12 ± 0.09	0.94 ± 0.03
4-bit Parity	0.11 ± 0.03	0.13 ± 0.01	0.90 ± 0.03
5-bit Parity	0.11 ± 0.03	0.13 ± 0.01	0.91 ± 0.03
6-bit Parity	0.11 ± 0.03	0.13 ± 0.01	0.92 ± 0.02
8x8 Lawnmower	0.99 ± 0.00	0.83 ± 0.04	0.98 ± 0.01
12x12 Lawnmower	0.99 ± 0.00	0.90 ± 0.05	0.99 ± 0.00
14x14 Lawnmower	0.99 ± 0.00	0.91 ± 0.04	0.99 ± 0.00

6 CONCLUSION

In this work, we presented a new mapping process for GE, which was able to statistically outperform standard GE on the analysed problems and MCGE on the Boolean problems. However, no statistical differences were observed between Leap and MCGE on the 12x12 and 14x14 Lawnmower problems, while MCGE statistically outperformed Leap on the 8x8 problem.

In future work, we intend to analyse the modularity of the evolved programs and also assess the performance of Leap mapping on more complex problems, such as real-world symbolic regression and program synthesis.

7 ACKNOWLEDGMENTS

This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant number 16/IA/4605. The third author is also financed by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brazil (CAPES), Finance Code 001, and the Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ).

REFERENCES

[1] Markus Brameier and Wolfgang Banzhaf. 2001. *Effective Linear Genetic Programming.* Technical Report Reihe CI 108/01, SFB 531. Department of Computer Science, University of Dortmund, 44221 Dortmund, Germany.

[2] Allan de Lima, Samuel Carvalho, Douglas Mota Dias, Enrique Naredo, Joseph P. Sullivan, and Conor Ryan. 2022. GRAPE: Grammatical Algorithms in Python for Evolution. *Signals* 3, 3 (2022), 642–663.

[3] Allan de Lima, Samuel Carvalho, Douglas Mota Dias, Enrique Naredo, Joseph P. Sullivan, and Conor Ryan. 2022. Lexi2: Lexicase Selection with Lexicographic Parsimony Pressure. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Boston, Massachusetts) *(GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 929–937.

[4] Francois-Michel De Rainville, Felix-Antoine Fortin, Marc-Andre Gardner, Marc Parizeau, and Christian Gagne. 2012. DEAP: a Python framework for evolutionary algorithms. In *GECCO 2012 Evolutionary Computation Software Systems (EvoSoft)*, Stefan Wagner and Michael Affenzeller (Eds.). ACM, Philadelphia, Pennsylvania, USA, 85–92.

[5] Michael Fenton, James McDermott, David Fagan, Stefan Forstenlechner, Erik Hemberg, and Michael O'Neill. 2017. PonyGE2: Grammatical Evolution in Python. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '17)*. ACM, Berlin, Germany, 1194–1201.

[6] Akira Hara, Tomohisa Yamaguchi, Takumi Ichimura, and Tetsuyuki Takahama. 2008. Multi-chromosomal Grammatical Evolution. In *Fourth International Workshop on Computational Intelligence & Applications.*

[7] David Jackson. 2010. Promoting Phenotypic Diversity in Genetic Programming. In *PPSN 2010 11th International Conference on Parallel Problem Solving From Nature (Lecture Notes in Computer Science, Vol. 6239)*, Robert Schaefer, Carlos Cotta, Joanna Kolodziej, and Guenter Rudolph (Eds.). Springer, Krakow, Poland, 472–481.

[8] John R. Koza. 1992. *Genetic Programming - On the Programming of Computers by Means of Natural Selection.* MIT Press. I–XVIII, 1–419 pages.

[9] John R. Koza. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts.

[10] Michael O'Neill and Conor Ryan. 2001. Grammatical evolution. *IEEE Trans. Evol. Comput.* 5 (2001), 349–358.

[11] Michael O'Neill, Conor Ryan, and Miguel Nicolau. 2001. Grammar Defined Introns: An Investigation Into Grammars, Introns, and Bias in Grammatical Evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke (Eds.). Morgan Kaufmann, San Francisco, California, USA, 97–103.

[12] Franz Rothlauf and Marie Oetzel. 2006. On the Locality of Grammatical Evolution. In *Genetic Programming*, Pierre Collet, Marco Tomassini, Marc Ebner, Steven Gustafson, and Anikó Ekárt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 320–330.

[13] Conor Ryan and R. Muhammad Atif Azad. 2003. Sensible Initialisation in Grammatical Evolution. In *GECCO 2003: Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference*, Alwyn M. Barry (Ed.). AAAI, Chigaco, 142–145.

[14] Conor Ryan, J. Collins, and Michael O'Neill. 1998. Grammatical Evolution: Evolving Programs for an Arbitrary Language.. In *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 83–96.

[15] Conor Ryan and Miguel Nicolau. 2001. Grammar Defined Introns: An Investigation Into Grammars, Introns, and Bias in Grammatical Evolution. (05 2001).

[16] Conor Ryan, Michael O'Neill, and J. J. Collins (Eds.). 2018. *Handbook of Grammatical Evolution.* Springer.