Work in Progress: Generation of Algebraic Data Types using Evolutionary Algorithms

Ignacio Ballesteros[1,2], Clara Benac-Earle[2], Lars-Ake Fredlund[2], Ángel Herranz[2], and Julio Mariño[2]

[1] IMDEA Software Institute
[2] Universidad Politécnica de Madrid, Spain

Abstract. Automatic data generation is a key component of automated software testing. Random generation of test input data can uncover some bugs in software, but its effectiveness decreases when those inputs must satisfy complex properties in order to be meaningful. In this work, we study an evolutionary approach to generate values that can be encoded as algebraic data types plus additional properties. More specifically, we have conducted an experiment on the automated generation of red-black trees using evolutionary algorithms. Although relatively simple, this example will allow us to introduce the main principles of evolutionary algorithms and how these principles can be applied to obtain valid, nontrivial samples of a given data structure. While the preliminary results show some potential, further experimentation is needed.

Keywords: Evolutionary algorithms · Genetic Programming · Software Testing · Property-Based Testing · Red-Black Tree

1 Introduction

Random generation of data is a commonly used approach in automatic software testing, e.g., Property-based Testing or the QuickCheck tool [3]. However, the effectiveness of random generation of test input data decreases when the input data must satisfy complex properties in order to be meaningful. Search-based strategies are used when there is a particular objective to achieve. In addition to defined objectives, search-based techniques usually include knowledge about the domain to be explored. For example, in software testing, knowledge about the code coverage typically plays an important role in data generation.

Evolutionary algorithms (EAs) [14] are particularly useful for finding solutions to problems which: 1. have a large search space; 2. where the search space is not uni-modal, that is, multiple optimizations are possible; and 3. a global optimum is not required. Test data generation meets these requirements, as the possible values might be infinite. Additionally, the required values could require multiple constraints or be valid on different optimizations, for example: size and a biased distribution of internal attributes. Finally, there is not a global optimum for generated values, as the goal is to find or approaching a particular search

area. In testing, EAs can be successfully used to generate test cases with better quality than random approaches [15,17]. Usually, these techniques focus on properties as control-flow coverage or non-functional requirements, for example, performance.

In this work, we study the application of evolutionary algorithms as a way to generate values of a given algebraic data type (ADT) with additional properties. We have focused on the case study of *red-black trees* [9]. A red-black tree is a data structure easy to represent as an ADT, but requiring some non-trivial properties that make very hard to randomly generate valid instances from just its bare algebraic definition. The main goal of this work is to design and evaluate an evolutionary algorithm capable of generating red-black trees. Hopefully, the principles used in this algorithm may be generalized to other complex algebraic data types, which could help in the automated testing of Haskell programs.

The rest of the paper is organized as follows. We start by discussing some related work in Section 2. Section 3 and 4 respectively introduce in detail red-black trees and the main concepts of evolutionary algorithms. Our approach to generate red-black trees is explained in Section 5, followed by an evaluation in Section 6. We conclude with some discussion and future work in Section 7.

2 Related Work

The Haskell QuickCheck's ecosystem has many proposals to improve the generation of test input data, for example, in [16], the authors introduce a way of automatically deriving random QuickCheck [3] generators with additional structure information. Some tools use enumeration, like SmallCheck [18] and the proposal of Duregård et al. [6,2], which includes constraints on the generated values. Finally, there is one search based approach using code coverage as a metric [8]. In the Erlang/Elixir property-based testing ecosystem, PropEr implements a search based generation using Hill-Climbing and Simulated Annealing methods [12]. None of these works apply evolutionary algorithms to generate test input data.

An evolutionary algorithm used to generate trees for testing in Haskell is GödelTest [7]. The goal is to optimize the size and height of the generated trees. Our work, in contrast, focuses on the generation of input data that fulfills certain properties, in particular, in this paper we study the generation of valid red-black trees.

The use of heuristics to find good test input data has been widely studied in the area of Search Based Software Testing (SBST). Usually, these approaches optimize test suites with analyzing code properties as coverage goals or performance metrics [15,17,10,13,21]. Generating test cases with constraints has been studied as an approach to guide the generation to useful input spaces. Sakti [19,20] introduce an enhancement of SBST including constraints on the input generation, but also based on branch coverage criteria. Dinh [5] uses an external Z3 solver to generate higher path-coverage test inputs.

3 Red-Black Trees

Red-black trees can be defined as a (polymorphic) Haskell algebraic data type as follows:[3]

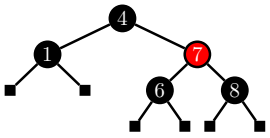
```
data Tree a
  = Node Color (Tree a) a (Tree a)
  | Nil

data Color = Red | Black
```

This definition does not enforce the properties of *valid* red-black trees. In order for a tree to be a valid red-black tree it has to satisfy the properties of a *binary search* tree, and also the following ones [4]:

1. A red node cannot have a red child.
2. Every path from a given node to any of its descendant leaves goes through the same number of black nodes.

An example of a valid red-black tree is shown in Figure 1a. The following definition will be useful when defining transformations on red-black trees. A *unitary* red-black tree is one that only has one node and is the direct parent of two leaves. All unitary red-black trees are valid. Figure 1b shows two unitary red-black trees.



(a) A valid red-black tree.

(b) Example of unitary red-black trees.

Fig. 1: Red-black trees.

Definition 1 (Unitary Red-Black Tree). *An unitary red-black tree is a valid red-black tree that only has one node and is a direct parent of two leaves.*

Definition 2 (Subtree of a tree). *A subtree of a given tree is either one of its children or any subtree of its children.*

Definition 3 (Black Depth). *Given a tree T and the set of nodes and leaves N_T and L_T, the black-depth of an element $e \in N_T \cup L_T$, is the number of black-colored parents from e to the root of T.*

[3] In practice, red-black trees do not hold any type of value. Instead, the type parameter `a` must be an instance of the `Ord` class in Haskell. We could have expressed this with GADTs [1], but we have chosen this definition for the sake of simplicity.

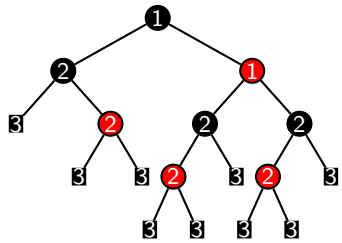


Fig. 2: Black-depth of a valid red-black tree
Numbers in nodes and leaves represent the black-depth of the element.

4 Evolutionary Algorithms

An evolutionary algorithm (EA) [14] is a bio-inspired algorithm that imitates natural pressure on a set of individuals to make them evolve into an adapted population fitted for a desired environment. The implementation of an evolutionary algorithm requires, on one hand, defining the following steps in the evolution of the individuals: *selection*, *crossover* and *mutation*. On the other hand, a fitness function determines the quality of the individuals, i.e., how well adapted they are to the desired environment. Let us explain each of these components in more detail:

Selection. In this step, the fittest – according to the fitness function – individuals from the population are selected. For example, individuals can be ordered based on how "close" they are to meet a given requirement or some additional property. Only the best – sometimes a fixed number of them – individuals are selected. The non-selected individuals are discarded and will not be considered in the next stage.

Selection 1 (Elite) *The population P is sorted based on the fitness of each individual. Only the best N individuals are selected to crossover.*

Selection 2 (Pareto Elite) *On multi-objective optimizations, every individual that is Pareto-dominant is selected to crossover. Given a Fitness function F, with multiple objectives $\{f_1, f_2, ..., f_n\}$, an individual $i \in P$ is Pareto-dominant if and only if*

$\forall i' \in P$ such that $i \neq i'$, then $\forall f_n \in F$, $f_n(i) \geq f_n(i')$ and $\exists f_{n'} \in F$ such that $f_{n'}(i) > f_{n'}(i')$

Crossover. In this step, the selected individuals are combined. For example, two random individuals from the previous selection stage could be combined by interchanging some of their internal parts. Different crossover operations can be considered for this phase.

Mutation. Once a new population made of the crossovers of the selected individuals appears, random mutation operations are performed on some individuals. These mutations modify some internal attribute of the individual, i.e., they introduce some variability in the population. Hopefully, beneficial changes will prevail in the next generations.

5 Evolutionary Generation of Red-Black Trees

In this work we have implemented, in Haskell, an evolutionary algorithm for the generation of red-black trees. The code and the experiments can be found at the following repository:

– `https://gitlab.com/babel-upm/evolutionary-generation-of-adts`

In the rest of the section, the EA operations (crossover and mutation), and the fitness function used are described.

5.1 Operations

Crossover Operations. We have defined a crossover operation as an exchange between two trees, A and B, where a subtree of A replaces a subtree of B and the subtree from B replaces the one from A, resulting in two new trees. Figure 3 illustrates this transformation.

Operation 1 (Exchange) *Given two trees, T_1 and T_2, and two arbitrary subtrees $t_1 \in T_1$ and $t_2 \in T2$. The operation Exchange replaces t_2 in the position of t_1 in T_1 and replaces t_1 in the position of t_2 in T_2.*

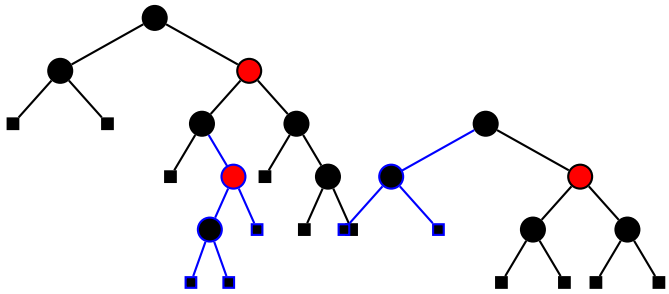
Mutation Operations. We have defined the following mutation operations, illustrated in Figure 4, for red-black trees:

1. Unitary insertion. A unitary tree replaces a leaf of the given tree.
2. Trimming a node. A leaf replaces a unitary tree of the given tree.
3. Change the color of a node in a given tree.
4. Change the value of a node in a given tree.

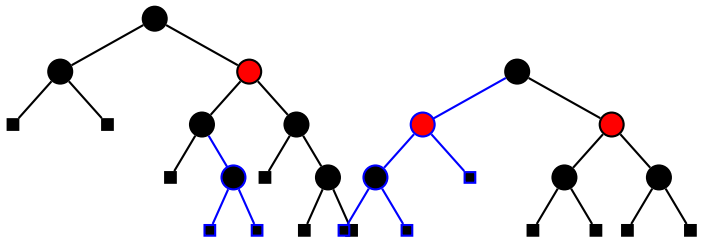
Operation 2 (Unitary insertion) *Given a tree T, applying the unitary insertion operation replaces one of the leaves of T with a random unitary red-black tree.*

Operation 3 (Unitary trimming) *Given a tree T, applying the unitary trimming operation replaces a unitary tree contained in T with a leaf.*

Operation 4 (Change Color) *Given a tree T and its set of nodes N_T, applying a color change operation replaces the color of some node $n \in N_T$ with a random new one.*



(a) Parent A (subtree in blue). (b) Parent B (subtree in blue).



(c) Child A (subtree from B). (d) Child B (subtree from A).

Fig. 3: Crossover between two trees.

Operation 5 (Change Value) *Given a tree T and its set of nodes N_T, applying a value change operation replaces the value of some node $n \in N_T$ with a random new one.*

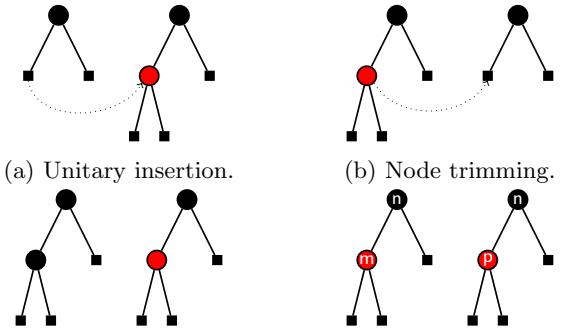
Definition 4 (Completeness of mutation operations for trees). *A set tree mutation operations is complete if, for any two trees, there exists a sequence of mutations that translates the first tree into the second one.*

For example, using the operations *insert*, *trim*, *color* and *value* we could potentially transform any tree into any other one.

5.2 Fitness functions

For selecting the best individuals we define several fitness functions that approximate the notion of how "close" a tree is to being a valid red-black tree, or to complying with some validity requirements, e.g. being black-balanced, etc. So, this kind of fitness function should reach a maximal (or minimal) value whenever an individual meets that property.

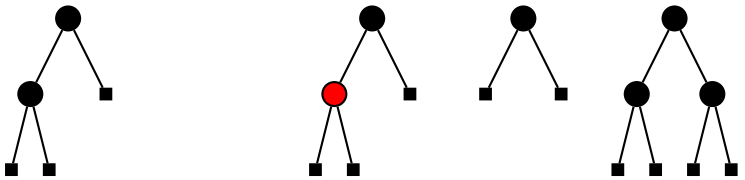
An ideal fitness function is the measure in the proximity of a given tree to the closest valid red-black tree. This fitness function represents the minimum number of changes that we need to apply to a tree to obtain a valid red-black tree. We define the proximity to a tree with respect to a set of operations, for example, the ones in Section 5.1.



(a) Unitary insertion. (b) Node trimming.

(c) Color mutation. (d) Value mutation.

Fig. 4: Red-black tree mutations.



(a) Invalid Red-Black Tree. (b) Valid red-black trees with Proximity(a)=1.

Fig. 5: Example: proximity to valid red-black trees.

Definition 5 (Proximity to a Red-Black Tree). *For any given tree and a set of operations, its proximity to a red-black tree is the minimum number of operations required to transform the given tree into a valid red-black tree.*

All valid red-black trees have a proximity of 0. Moreover, a tree could be equally-proximal to multiple valid red-black trees (Figure 5). As we do not directly know the closest valid red-black tree of any given tree, we approximate it with the definition of fitness functions.

By counting the parts of a red-black tree that are valid, the intuition of the *Validity fitness* arises. This fitness is used because, for any valid red-black tree, both left and right subtrees from the root are also valid red-black trees. Figure 6 represents the validity fitness of an invalid red-black tree and a valid red-black tree.

Fitness 1 (Validity of a red-black tree) *The red-black tree validity fitness of a given tree T is the number of valid subtrees of T and T itself, divided by the total number of subtrees of T plus the given tree T.*

$$F_{\text{validity}}(T) = \frac{|\text{valid}(T \cup \text{subtrees}(T))|}{|T \cup \text{subtrees}(T)|}$$

(a) Validity Fitness = 4 / 5. (b) Validity Fitness = 5 / 5.

Fig. 6: Red-black Tree Validity Fitness of an invalid tree.

Fitness 2 (Standard deviation of black-depth of the leaves) *The fitness based on the black-balance of the leaves of a given tree T is the standard deviation of the black-depth for all the leaves in a tree.*

$$F_{\text{black\_depth}}(T) = \sigma(\{\ \text{black\_depth}(\text{leaf}) \mid \text{leaf} \in L_T\ \})$$

Fitness 3 (Red-red nodes) *Given a tree T, the fitness based in the amount of red nodes of T which are parent of a red node.*

Fitness 4 (Sorting) *Given a tree and its pre-order traversal, the sorting fitness is the amount of pair of neighbor values where the first value is greater than the second value.*

$$\text{Let } p = \text{preorder}(T),\ F_{\text{sorting}} = |\{p_i > p_{i+i} \mid i \in [0..|p|)\}|$$

The validity fitness (1) can classify if a given tree is a valid red-black tree, while the rest of the fitness functions only do it partially. For example, ensuring the sorting of the values is necessary but not sufficient to be a red-black tree. These individual fitness functions on the properties of red-black trees can be combined as a unified fitness.

Fitness 5 (Properties of a red-black tree) *The mean value between the standard deviation of the black-depth of leaves (Fitness 2), the amount of red nodes parent of a red-rooted tree (Fitness 3) and the pre-order sorting of the values in a tree (Fitness 4).*

6 Experiments

In this section, we explain the experiments conducted with the goal of evaluating the effectiveness of generating valid red-black trees with the fitness functions and operations defined in Section 5. Experiments are run with a parameterized configuration of the evolutionary algorithm. First, we define an evaluation case, what attributes can be customized for each run and the measurements that we take on the experiments.

Evaluation Case. An evaluation case is a single run of the evolutionary algorithm with values for a set of attributes in the evolutionary algorithm.

Evaluation Case Attributes. An evaluation case can be run under the following evolutionary algorithm attributes:

P	Maximum population size.
N	Maximum number of generations.
Sel	Selection Method.
C_{Ops}	Crossover Operations.
M_{Ops}	Mutation Operations.
M_R	Mutation Ratio.
F	Fitness function.

Default values on Evaluation Case Attributes. Each evaluation case has these values for the following attributes:

P	100 *(Maximum population size)*
N	1000 *(Maximum number of generations)*
Sel	Elite of best 20 terms (Selection 1).
C_{Ops}	Single point crossover. (Operation 1)
M_{Ops}	*Trim, Insert, Color change* and *Value change*. (Operations 2, 3, 4 and 5).
M_R	Each individual of a population has a 5% probability of suffering a mutation. *(Mutation Ratio)*

Measurements. Our objective is evaluating the generation of valid red-black trees with the goal of using these generated values as test inputs. After each run of an evaluation case, we collect the following statistics to compare the quality and performance of each evaluation case.

1. Distinct valid red-black trees.
2. Maximum size of generated trees.
3. Time required to generate all the trees.
4. Size of the generated trees.

6.1 Evaluation Cases

In the evaluation cases we assess the quality of the fitness functions defined in Section 5.2. Additionally, along with these fitness functions, we include another optimization based on the size of the generated term. That is, the closer a term is to a given objective size, the better. The objectives using the fitness functions are:

- Maximizing the validity of a red-black tree.
- Minimizing the standard deviation on the black-depth balance of the leaves.
- Minimizing the number of red-red nodes repetitions.
- Minimizing the number of unsorted values.
- Approaching to a given size of the tree.

These fitness functions and objectives are grouped in 4 evaluation cases:

1. Validity of a tree (Fitness 1).
2. Properties of a red-black tree (Fitness 5).
3. Validity of a tree (Fitness 1) and size.
4. Properties of a red-black tree (Fitness 5) and size.

The selection of these fitness functions, "Validity" (1) and "Properties" (5) allows us to compare the effectiveness of two fitness functions that are able to determine if a tree is a valid red-black tree. Additionally, the inclusion of the size objective allows us to study the distribution of tree sizes while preserving the properties of a red-black tree.

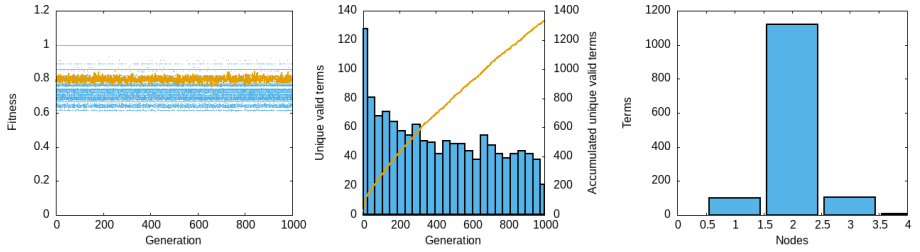
In Table 1, we present the results of generated red-black trees for each evaluation case. The table contains the total amount of valid and distinct red-black trees generated in a single run of the evaluation case for each fitness function. The "time" column measures the required time to generate all the trees in the evaluation case, that is, $100,000$ trees with the default parameters. The experiment is run in a single Dell Inc. XPS 15 9510 laptop with 16 GiB of RAM and an 11th Gen Intel® Core™ i7-11800H processor.

Table 1: Evaluation of Red-Black Trees.

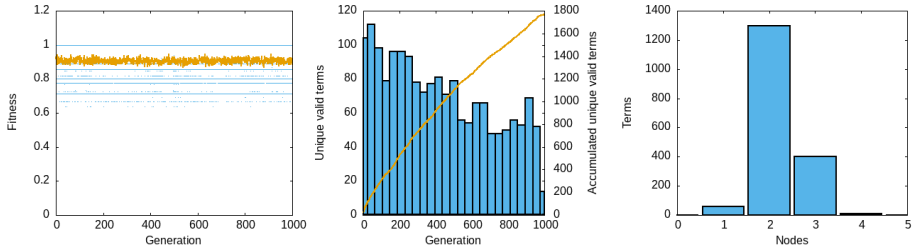
Fitness	**Distinct valid trees**	**Time (s)**
No fitness (Random generation)	1313	3.45
Validity	1768	1.28
Red-black properties	1721	1.60
Validity + Size	893	8.58
Red-black properties + Size	2836	4.28

Figure 7 represents a comparison between the evaluation cases and the generated trees. Each evaluation case is represented with three plots. In the first one (left), each blue dot is a generated tree, placed on the generation created and with the value of fitness function. The yellow line represents the mean fitness of the generation. The second plot (center) represents the amount of distinct valid red-black trees. In blue, the amount (left axis) in groups of 40 generations (25 bars). In yellow (right axis) the accumulated number of unique valid red-black trees. The third and final plot (right) represents the size of a tree using the distribution in the number of nodes for the distinct valid red-black trees across all generations.

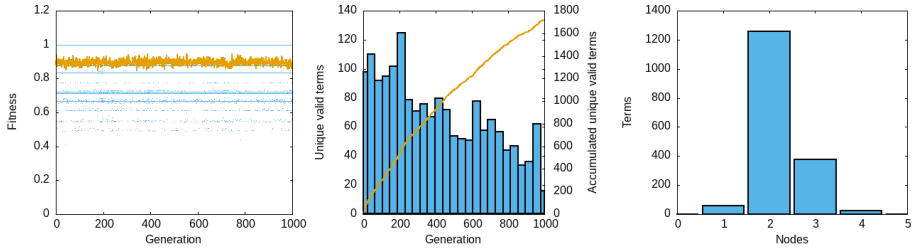
As a baseline evaluation case, no evolutionary algorithm is used in the random generation run. Under the evolutionary algorithm, the selected fitness functions are capable of guiding the generation to obtain valid red-black trees. The use of evolutionary algorithms is able to generate terms at least in an equivalent way as random generation, because it is able to generate valid values in at least the



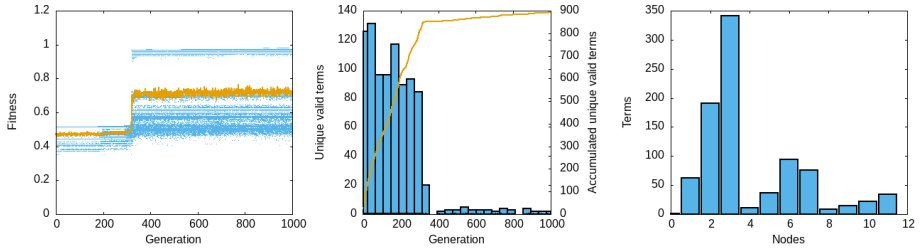
(a) Random generation (fitness measured with validity fitness).



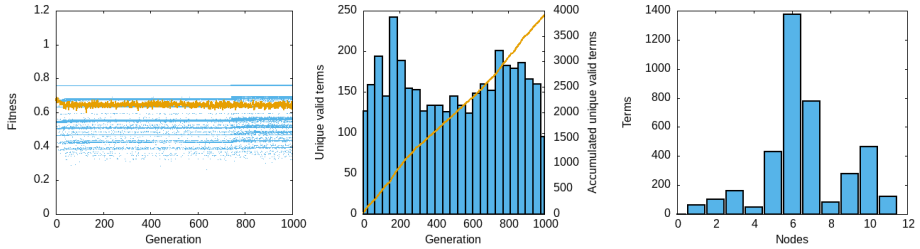
(b) Fitness: red-black tree validity.



(c) Fitness: red-black tree properties.



(d) Fitness: red-black validity + size.



(e) Fitness: red-black tree properties + size.

Fig. 7: Experimental results.

same quantity in less time. In the case of fitness functions with the additional size as a property, the execution shows promising results on the generation of valid red-black trees.

Random generation presents a lower mean fitness of the generated values, and the majority of randomly generated red-black trees have two nodes. This is an expected result as the combination of a black node and a red node always satisfies at least two of the properties of a red-black tree, that is, the black-depth of the leaves and not having two consecutive red nodes.

Without the pressure on the size, the evolutionary algorithm converges and generates small sized red-black trees, but with a consistent increase in the amount of three-node trees compared to random generation and with a greater amount of distinct valid red-black trees. Moreover, it is able to generate them in less time. Without the size property and for this evaluation cases runs, there is no appreciable difference between the two fitness functions selected for the experiment.

Once the size property is introduced, the results of these evaluation cases change. First, the mean fitness is lower compared to the previous experiments. The mean fitness is lower because now there is a *push* towards increasing the size of the generated trees. Second, the size of generated trees has a wider distribution: from 1 node to 11 nodes in both fitness functions. Finally, we observe different behaviors in the number of generated trees depending on the fitness function. Evaluation case *(d)* has a smaller amount of distinct valid red-black trees, abruptly decreasing the through-output near generation 400. This abrupt change corresponds to a moment of high increase in the mean fitness value. A fast increase in the fitness function is not usual in the experiments run, but expected to sometimes happen in evolutionary algorithms. With respect to the evaluation case (e), the amount of valid red-black trees is superior to any other execution. These are some initial promising results not only because of the increase in the amount of valid generated trees but also because of the greater size they have.

7 Conclusions and Future Work

We have developed an evolutionary algorithm in Haskell for test input generation of values conforming to algebraic data types. In particular, with the case study of red-black tree, in which generated values require fulfilling some additional properties to be valid. We have defined a set of operations for red-black trees and the fitness functions to be used in the evolutionary algorithm.

The results in the Section 6 show some promising results. The evolutionary algorithm is able to generate valid red-black trees either in less time or with a greater size. Random generation was not able to create a valid red-black tree with a size near to the ones generated with the evolutionary algorithm.

This is a work still in progress, and our plan for future work concerns three components:

Evaluation. We plan to improve the evaluation to answer if the improvements over random generation are statistically significant.

Evolutionary Algorithm. The current implementation of the evolutionary algorithm is naive with respect to the optimization and selection methods. We plan to perform a comparison between evolutionary algorithms on multi-objective optimization to find the best suited for the generation of algebraic data type with additional properties and constraints inspired in state of the art evolutionary algorithms [11].

Generalization. This work has been done with the objective of generalizing the generation to algebraic data types with additional constraints. Currently, as a work-in-progress paper, we have not addressed the capabilities of generalizing these concepts.

Acknowledgments

This work has been partially supported by PROCODE Project.
(PID2019-108528RB-C21/MCIN/AEI/10.13039/501100011033)
This work has been partially supported by PRODIGY Project (TED2021-132464B-I00) funded by MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU/PRTR.
Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European
Union. The European Union can not be held responsible for them (GA No. 101061343)

References

1. Haskell/GADT - Wikibooks, open books for an open world. https://en.wikibooks.org/wiki/Haskell/GADT
2. Claessen, K., Duregård, J., Pałka, M.H.: Generating Constrained Random Data with Uniform Distribution. In: Codish, M., Sumii, E. (eds.) Functional and Logic Programming. pp. 18–34. Lecture Notes in Computer Science, Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-07151-0_2
3. Claessen, K., Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs
4. Cormen, T.H., Leiserson, C.E.: Introduction to Algorithms, Fourth Edition. MIT Press, London, England (Apr 2022)
5. Dinh, N.T., Vo, H.D., Vu, T.D., Nguyen, V.H.: Generation of Test Data Using Genetic Algorithm and Constraint Solver. In: Król, D., Nguyen, N.T., Shirai, K. (eds.) Advanced Topics in Intelligent Information and Database Systems, pp. 499–513. Studies in Computational Intelligence, Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-56660-3_43

6. Duregård, J., Jansson, P., Wang, M.: Feat: Functional enumeration of algebraic types. In: Proceedings of the 2012 Haskell Symposium. pp. 61–72. Haskell '12, Association for Computing Machinery, New York, NY, USA (Sep 2012). https://doi.org/10.1145/2364506.2364515
7. Feldt, R., Poulding, S.: Finding test data with specific properties via metaheuristic search. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE). pp. 350–359. IEEE, Pasadena, CA, USA (Nov 2013). https://doi.org/10.1109/ISSRE.2013.6698888
8. Goldstein, H., Hughes, J., Lampropoulos, L., Lampropoulos, L., Pierce, B.C.: Do Judge a Test by its Cover pp. 264–291 (2021). https://doi.org/10.1007/978-3-030-72019-3_10
9. Guibas, L.J., Sedgewick, R.: A dichromatic framework for balanced trees. In: 19th Annual Symposium on Foundations of Computer Science (Sfcs 1978). pp. 8–21 (Oct 1978). https://doi.org/10.1109/SFCS.1978.3
10. Jain, N., Porwal, R.: Automated Test Data Generation Applying Heuristic Approaches—A Survey. In: Hoda, M.N., Chauhan, N., Quadri, S.M.K., Srivastava, P.R. (eds.) Software Engineering. pp. 699–708. Advances in Intelligent Systems and Computing, Springer, Singapore (2019). https://doi.org/10.1007/978-981-10-8848-3_68
11. Liang, J., Ban, X., Yu, K., Qu, B., Qiao, K., Yue, C., Chen, K., Tan, K.C.: A Survey on Evolutionary Constrained Multiobjective Optimization. IEEE Transactions on Evolutionary Computation **27**(2), 201–221 (Apr 2023). https://doi.org/10.1109/TEVC.2022.3155533
12. Löscher, A., Sagonas, K.: Automating Targeted Property-Based Testing. In: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST). pp. 70–80 (Apr 2018). https://doi.org/10.1109/ICST.2018.00017
13. Malhotra, R., Khari, M.: Heuristic search-based approach for automated test data generation: A survey. International Journal of Bio-inspired Computation **5**(1), 1–18 (Apr 2013). https://doi.org/10.1504/ijbic.2013.053045
14. Melanie, M.: An Introduction to Genetic Algorithms (1998)
15. Michael, C., McGraw, G., Schatz, M., Walton, C.: Genetic algorithms for dynamic test data generation. In: Proceedings 12th IEEE International Conference Automated Software Engineering. pp. 307–308 (Nov 1997). https://doi.org/10.1109/ASE.1997.632858
16. Mista, A., Russo, A.: Generating Random Structurally Rich Algebraic Data Type Values. In: 2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST). pp. 48–54. IEEE, Montreal, QC, Canada (May 2019). https://doi.org/10.1109/AST.2019.00013
17. Nogueira, A.F., Ribeiro, J.C.B., Fernández de Vega, F., Zenha-Rela, M.A.: Object-Oriented Evolutionary Testing: A Review of Evolutionary Approaches to the Generation of Test Data for Object-Oriented Software. International Journal of Natural Computing Research **4**(4), 15–35 (Oct 2014). https://doi.org/10.4018/ijncr.2014100102
18. Runciman, C., Naylor, M., Lindblad, F.: Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values. In: Gill, A. (ed.) Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008. pp. 37–48. ACM (2008). https://doi.org/10.1145/1411286.1411292
19. Sakti, A., Guéhéneuc, Y.G., Pesant, G.: Boosting Search Based Testing by Using Constraint Based Testing. In: Fraser, G., Teixeira de Souza, J. (eds.) Search Based

,

Software Engineering. pp. 213–227. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33119-0_16
20. Sakti, A., Guéhéneuc, Y.G., Pesant, G.: Constraint-Based Fitness Function for Search-Based Software Testing. In: Gomes, C., Sellmann, M. (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. pp. 378–385. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38171-3_29
21. Varshney, S., Mehrotra, M.: Search based software test data generation for structural testing: A perspective. ACM SIGSOFT Software Engineering Notes **38**(4), 1–6 (Jul 2013). https://doi.org/10.1145/2492248.2492277