

# Making Hard(er) Benchmark Functions: Genetic Programming

Dante Niewenhuis<sup>1</sup> <sup>a</sup>, Abdellah Salhi<sup>2</sup> <sup>b</sup> and Daan van den Berg<sup>1,3</sup> <sup>c</sup>

<sup>1</sup>Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands

<sup>2</sup>Department of Mathematical Sciences, University of Essex, U.K.

<sup>3</sup>Universiteit van Amsterdam, The Netherlands

Keywords: Genetic Programming, Fitness Landscapes, Evolutionary Algorithms.

Abstract: TreeEvolver, a genetic programming algorithm, is used to make continuous mathematical functions that give rise to 3D landscapes. These are then empirically tested for hardness by a simple evolutionary algorithm, after which TreeEvolver mutates the functions in an effort to increase the hardness of the corresponding landscapes. Results are wildly diverse, but show that traditional continuous benchmark functions such as Branin, Easom and Martin-Gaddy might not be hard at all, and much harder objective landscapes exist.

## 1 INTRODUCTION


Branin, Easom, Martin-Gaddy and the Six-Hump Camel are household names in the evolutionary programming community. Being strictly mathematically formulated benchmark functions, they define continuous objective landscapes<sup>1</sup> with precisely known global minima in both values and locations, and a diverse range of hardness features such as many local minima, narrow basins or arctic plateaus on which metaheuristic navigation is almost impossible.


As household names do, these benchmark functions are well-known throughout the practitioner's field and are often used for testing and comparing optimization algorithms such as simulated annealing, genetic algorithms and ant colony optimization, but also machine learning (Fodorean et al., 2012; Joshi et al., 2021; Digalakis and Margaritis, 2001; Laguna and Marti, 2005; Socha and Dorigo, 2008; Orzechowski and Moore, 2022). Though one could easily mistake such reuse as intellectual poverty or just plain trendy, experimenting on a limited set of benchmark test functions actually has its upsides too, as it enables papers to be tied together, percolating our grains


of knowledge to larger clusters of more or less universally true principles. Benchmarking is important. So important even, that the study of 'best practice' in benchmarking has become a focus of scientific study in itself (Bartz-Beielstein et al., 2020).

Household names are individuals we are familiar with, that have known properties, and show predictable behaviour. This is no different for continuous or discrete optimization problem suites; the most famous example of which is undoubtedly TSPLib, Gerhard Reinelt's library of traveling salesman problem instances (Reinelt, 1991). The traveling salesman problem as a whole is NP-hard, meaning there is no known polynomial-time exact algorithm, thereby obstructing a view on the shortest tour, even for moderately sized instances. To make things worse, we also don't know its length. From this point, we have to fight from the numerical jungle, using guerilla algorithmics and tailor-made pattern hacks such as exploiting Euclideaness, bootstrapping upper bounds or exploit undiverse numerical values (Slegers and van den Berg, 2020a; Koppenhol et al., 2022; Applegate et al., 2009; Zhang and Korf, 1996). It is therefore not surprising that many TSP-instances only have a best *known* solution instead of a best solution - a phenomenon that occurs in many more combinatorial optimization problems (Rosenberg et al., 2021; Weise et al., 2021; Weise and Wu, 2018).

The situation is slightly different for continuous problems. Here, our household benchmark functions

<sup>a</sup>  <https://orcid.org/0000-0002-9114-1364>

<sup>b</sup>  <https://orcid.org/0000-0003-2433-2627>

<sup>c</sup>  <https://orcid.org/0000-0001-5060-3342>

<sup>1</sup>Please note that we deliberately omit the term "fitness landscapes" as objective values and fitness values are strictly spoken not interchangeable.

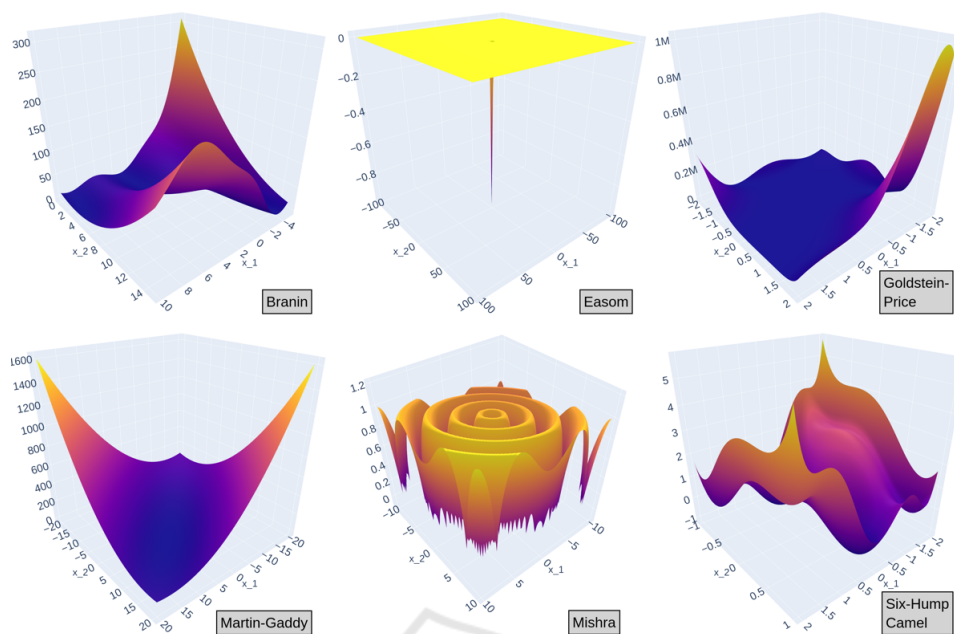


Figure 1: Some ‘household names’: continuous benchmark functions that are often used in testing evolutionary algorithm performance. But how hard are these objective landscapes really? Do harder instances exist?

usually not only have known global minimum values, but their exact location is also known, as are the landscape features. Sometimes these known properties even extend to parameterizable higher dimensions, so that even for a 257-dimensional Ackley function, the global minimum value - *and* its location - is known. Such scaleup is only dreamt of in TSP-communities, where the addition of just a few cities to a known instance can distort all its properties, except in the most structured of cases, such as circular or grid instances. Stop reading and contemplate this for a minute. Why is the scaleup in discrete benchmarking so different from the scaleup in continuous benchmarking? Why is it so much harder to find minima for larger search spaces in discrete problems as opposed their continuous counterparts?

<sup>2</sup>The answer, we will provisionally argue, is structure or the lack thereof, in the universalest sense of the word. By an argument of Kolmogorov complexity, easy traveling salesman instances are those whose cities are placed on a circle, on a grid, or on the vertices of a regular fractal (Fischer et al., 2005). These are easy, and have known global minima, both in value and location (or exact tour configuration) as they scale up. Nonetheless the computer programs needed to describe these instances remain relatively small, and hence their Kolmogorov complexity is low. In fact, an investigation by Fischer, Stützle, Hoos and Merz showed that these regular instances become

<sup>2</sup>Did you stop reading for a minute?

harder when the TSP instance’s city locations are distorted. The greater the distortion, the less structured the instance, the longer the algorithmic description, and the higher the Kolmogorov complexity. In other words: there is evidence that structured (or equivalent: shortly describable) instances of TSP are easier than unstructured (or not shortly describable) instances, which are harder to solve. And recently, similar phenomenon has also appeared in the NP-complete Hamiltonian cycle problem, where only the extremely hard and extremely easy instances appear to be highly structured (Slegers and van den Berg, 2020a; Slegers and van den Berg, 2020b; Slegers and Berg, 2021; Slegers and van den Berg, 2022). Moreover, these studies also suggests that the breakdown of an instance’s structure influences its hardness, even though this specific case pertains to a decision problem.

Now, the key question is of course: to what extent can the same argument apply to continuous optimization? Are continuous landscapes with structure, with short descriptions, with low Kolmogorov complexity also easier than those without structure, or long descriptions? They might. There is some evidence that the hardness of several continuous benchmark functions such as Schwefel, Rastrigin and Ackley *decreases* with increasing numbers of dimensions (De Jonge and Van den Berg, 2020; de Jonge and van den Berg, 2020). This shoehornily fits our provisional hypothesis because these functions usually increase their dimensionality (and thereby the search

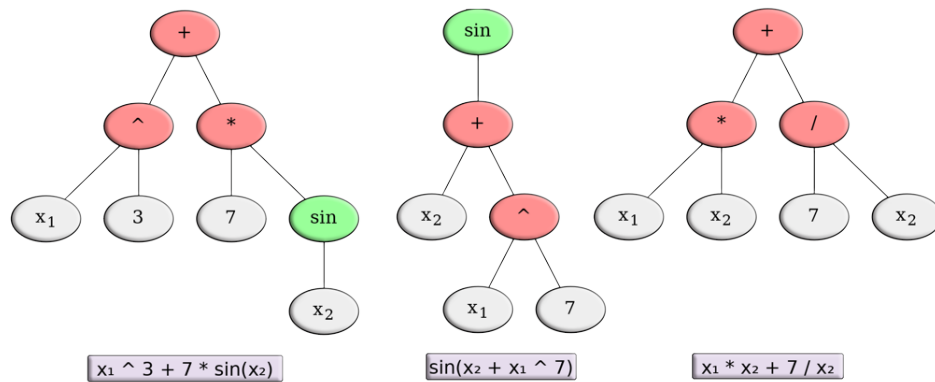


Figure 2: Example of different function trees. Functions are green, operators red and both variables and constants are grey.

space) by upping the  $D$ -parameter in its main term, which usually looks like  $\sum_{d=1}^D F(x_d)$ . In other words: the functional description (or Kolmogorov complexity) remains short, and does not increase along with the search space, much like an highly regular TSP-instance.

We'll skip the quite interesting question of *why* these benchmarks are the way they are for now, but rather ask: could they have been different? Could substantially different hard objective landscapes exist, and what do they look like? The answer is yes, and we can find out by evolving the tree structure of the objective functions (Fig. 2). This concept, the evolutionarily improvement of computer programs, is known as *genetic programming* and was introduced by John Koza in "Genetic Programming On the Programming of Computers by Means of Natural Selection" (Koza, 1992; Koza, 1994). Whereas genetic algorithms operate on strings, numbers or data structures, genetic programming evolves the source code *itself* (Sette and Boullart, 2001). It has been applied to a large number of problems such as robotics (Koza and Rice, 1992), financial modelling (Kordon, 2010), image processing (Lam and Ciesielski, 2004), and design (Koza, 2008). In artificial intelligence genetic programming has been used for tasks such associated discovery (Lyman and Lewandowski, 2005) and clustering (Alhajj and Kaya, 2008; Bezdek et al., 1994; De Falco et al., 2006; Jie et al., 2004; Liu et al., 2005). One survey notably provides an overview of more than 100 applications of genetic programming (Espejo et al., 2009).

In this work, we'll take a somewhat more conservative approach than abovementioned examples, "just" evolving the objective functions giving rise to the corresponding three-dimensional landscapes. As such, an easy-to-minimize function like  $x_1 + x_2$  might evolve to something like  $(-1.203 - 10) * 1.855 + \sin(x_2 + 6.363 - x_1) + \sin(-0.197 + x_2)$ , which is very

hard to minimize for an evolutionary algorithm. The task is therefore formulated as 'make a hard objective landscape', (for some evolutionary algorithm). More generally however, it is the beginning of an exploration into the combinatorial space of continuous objective landscapes. We, the community, are all too familiar with our household names like Mishra, Easom and Goldstein-Price ... but what *else* is out there?

## 2 OBJECTIVE LANDSCAPE HARDNESS

Evolving objective landscapes for hardness requires measuring that hardness. In evolutionary computing, a popular performance metric is the mean best fitness (MBF) (Eiben et al., 2003), which averages the best results from a number of runs on an objective landscape. While suitable for the comparison two algorithms, we rather require a method that compares the performance of two *landscapes*. In other words: is *algorithm*<sub>1</sub> more succesful on objective landscape  $OL_1$  or on objective landscape  $OL_2$ ? For such an assessment, the MBF is unsuitable because a better end result on  $OL_2$  means nothing if  $OL_2$ 's global minimum value is much lower, or the individuals get initialized much higher.

In a variation on MBF therefore, we will measure the performance of an evolutionary run relative to an objective landscape's global minimum, denoted as its *objective deficiency* (Niewenhuis and van den Berg, 2022; Niewenhuis and van den Berg, 2023):

$$\frac{v - OL_{min}}{OL_{max} - OL_{min}} * 100 \quad (1)$$

in which  $OL$  is an objective landscape and  $OL_{min}$  and  $OL_{max}$  are the values of the global minimum and maximum respectively. The range of a benchmark is de-

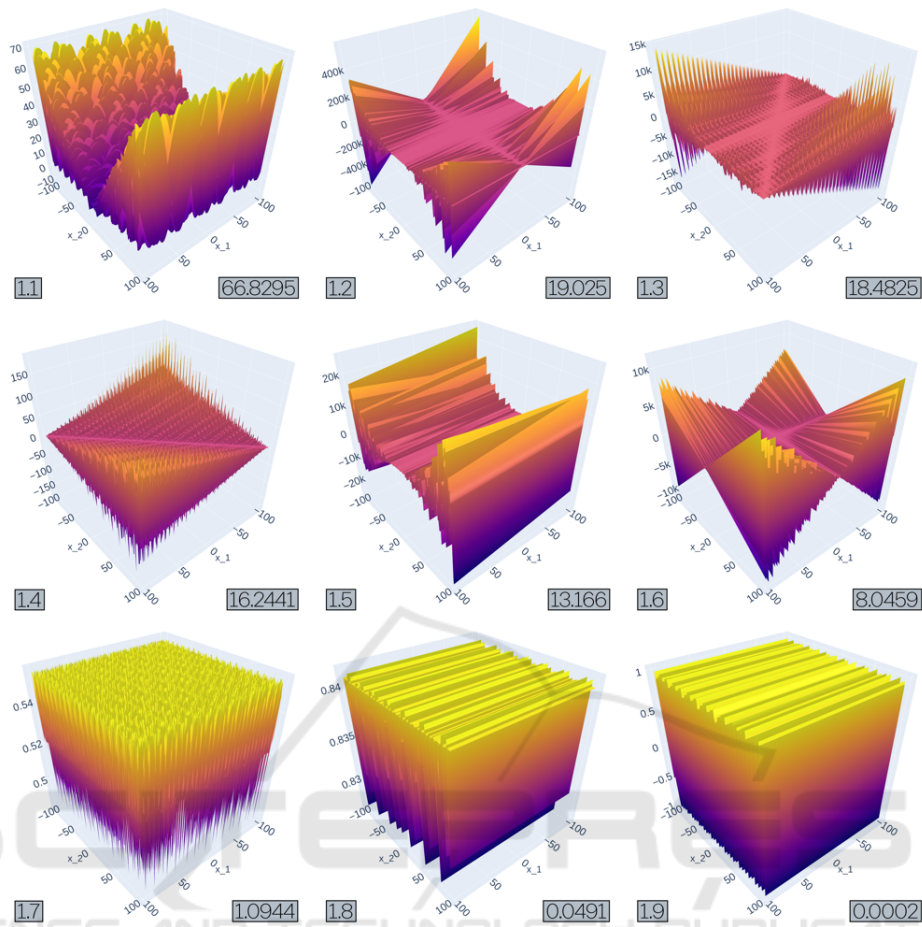


Figure 3: Objective landscapes from experiment 1 after 2000 generations in TreeEvolver, ordered by MOD. All runs had  $x_1 + x_2$  as their initial function. Operator nodes were limited to  $\{+, -, *\}$ , and Function nodes were limited to  $\{\sin\}$ .

defined as the difference between  $OL_{min}$  and  $OL_{max}$  and  $v$  is an assessable value, for instance ‘the best individual of an EA after 2000 evaluations’. When averaging over several (stochastic) runs, we obtain the *Mean Objective Deficiency* (MOD). In this work we consistently do 100 runs of the Plant Propagation Algorithm (PPA) (Section 3) for every MOD-assessment.

There is still a problem however. Whereas household benchmark functions such as Ackley, Goldstein-Price and Six-Hump Camel have precisely known global minima and can therefore easily be assessed, this is no longer the case for evolved objective landscapes. And finding them is also not easy; a preliminary run showed that calculating zero-derivatives of evolved objective landscapes, a conceptually ideal approach, is computationally infeasible, unreliable and in some cases even impossible from the incorporation of absolute signs, divisions by zero, square roots and logarithms. For that reason, after every mutation (‘generation’) we obtained a *surrogate* global maximum and minimum from  $10^6$  random objective sam-

ples from the evolved function’s domain. This method works well for getting the concept working, but more sophisticated future methods, or tighter restrictions on the mutation types are definitely not unthinkable.

For sakes of comparison, we also assessed the MOD of the household names from Figure 1. Sorted from hard to easy:  $MOD_{Easom} = 14.26$ ,  $MOD_{Mishra04} = 0.925$ ,  $MOD_{Branin} = 2.297e - 04$ ,  $MOD_{SixHump} = 7.149e - 04$ ,  $MOD_{MartinGaddy} = 3.077e - 04$  and  $MOD_{GoldsteinPrice} = 1.831e - 05$ . It seems safe to conclude that only Easom is slightly hard for PPA; the rest are easy.

### 3 THE PLANT PROPAGATION ALGORITHM

Before we come to the evolutionary algorithm of choice, let’s start with a few substantial disclaimers. We, the authors, are fully aware of the massive debate

Table 1: The 9 objective from experiment 1 ordered by MOD. All runs had  $x + y$  as their initial function. Operator nodes were limited to  $\{-, +, *\}$ , and Function nodes were limited to a value from  $\{\sin\}$ .

	MOD	Function
1.1	66.8295	$(\sin(9.483 * (\sin(x_2) + x_1)) * x_2 + x_2) * \sin(9.341 * 5.656 * (x_1 + \sin(x_1))) * x_2 * x_2 * x_2 * x_1 * (x_1 - x_2 * x_2 - 2.198 + x_2 - x_2 + \sin(x_2) * 2.775 * x_2))$
1.2	19.0250	$x_1 * ((\sin(-2.289) + 5.171) * 5.299 - x_2 * x_2 - x_2 + 5.102) * (\sin(\sin(\sin(\sin(-1.105)))) * \sin(-4.52 * (\sin(\sin(3.104)) - \sin(\sin(\sin(\sin(-6.796 - 3.117 * 8.209) + \sin(\sin(x_2 + x_2) + x_2)))))) - \sin(-3.105))$
1.3	18.4825	$(\sin(\sin(9.882)) + (\sin(x_1 + 10) + \sin(6.877)) * (x_2 - x_1) * \sin(\sin(-6.488) - 4.68 * \sin(x_2) + x_2) * \sin(x_2) * x_2) * \sin(x_2 + x_2 * (x_2 - \sin(x_2)))$
1.4	16.2441	$(x_2 + x_1) * (\sin(-1.689 + 1.36) + \sin(\sin(4.549) + \sin(\sin(\sin(x_2) - 1.991))) + \sin(-8.193 + 3.761 * (-8.649 - \sin(\sin(\sin(x_2 - \sin(2.983) - \sin(\sin(x_1 - 9.285 - \sin(\sin(\sin(x_1 + 9.01)))))) + x_2))) * \sin(x_2)$
1.5	13.1660	$(2.792 + (-8.546 + x_2 - \sin(\sin(\sin(\sin(\sin(\sin(-10 * x_2)))))) * \sin(0.383) * (x_2 + x_1) - 10) * \sin((10 + \sin(\sin(-5.099 + 1.037 + \sin(x_2) + 7.607)) * x_2)) * (x_2 + x_2)$
1.6	8.0459	$(-10 - x_2) * \sin(x_1 + x_1 - 7.896 - \sin(\sin(\sin(-8.594 - \sin(1.416 * x_1))) - 9.702)) * (x_1 + 6.539)$
1.7	1.0944	$\sin(\sin(\sin(\sin(\sin(\sin(7.465 + \sin(\sin(-3.419 - x_2) - 4.721 + \sin(x_1 + x_2)))))))))$
1.8	0.0491	$\sin(\sin(-5.657 + \sin(\sin(\sin(\sin(\sin(\sin(\sin(\sin(x_2 - 8.376) + 7.648))) * 2.174))) - 7.109)))$
1.9	0.0002	$\sin(x_1 + x_2 - 7.345 - x_1)$

that has been unfolding in the community pertaining to ‘metaphorical’ or ‘new’ evolutionary algorithms. We would therefore like to clearly state that: the plant propagation algorithm (PPA) is not newly introduced here, its name is just a name and nothing more, and it is not claimed to be competitive in any way. Reasons to choose this algorithm is that we find it *elegant*, especially in self-balancing itsexploration and exploitation, its easiness of implementation, availability of open source and finally, because a lot is known about its behaviour, also in different parameterizations, on continuous objective landscapes (Fraga, 2019; Rodman et al., 2018; Slegers and van den Berg, 2020a; Slegers and van den Berg, 2020b; Sulaiman et al., 2018; Vrieling and van den Berg, 2019; Vrieling and Van den Berg, 2021b; Vrieling and Van den Berg, 2021a; Geleijn et al., 2019; Haddadi, 2020; Paauw and Berg, 2019; Selamoğlu and Salhi, 2016; Sulaiman et al., 2016). However, this GP-method is not confined to PPA; other evolutionary algorithms or perhaps even exact methods could also do the assessment.

The central idea of PPA is idea that better individuals in its population produce *more* offspring with *smaller* mutations while worse individuals produce *fewer* offspring with *larger* mutations. Since fitness values are normalized  $\in [0, 1]$ , the algorithm constantly maintains a homeostatic balance between exploration and exploitation throughout its evolutionary run. In other words: no matter how near or far the population is from a global or local optimum, and no matter what its population looks like, it will always simultaneously keep exploiting and exploring. In this work we are using the seminal implementation (in-

cluding parameter settings) as introduced by Salhi & Fraga in 2011 (Salhi and Fraga, 2011), which goes through the following steps:

1. Initialize a population of  $popSize=30$  individuals by sampling from a uniform distribution within in the objective function’s domain.
2. Calculate the objective value  $f(x_i)$  for each individual in the current population.
3. Normalize the objective value  $f(x_i)$  between 0 and 1 using  $z(x_i) = \frac{f_{max} - f(x_i)}{f_{max} - f_{min}}$  wherein  $f_{min}$  and  $f_{max}$  are the lowest and highest objective values in the current population.
4. Determine the fitness of each individual using  $F(x_i) = \frac{1}{2}(\tanh(4 * z(x_i) - 2) + 1)$ .
5. Determine the number of offspring for each individual using  $n(x_i) = \lceil n_{max} F(x_i) r_1 \rceil$ , in which  $r_1$  is a random number in  $[0, 1)$  and  $n_{max} = 5$  is the parameter for the maximum number of offspring from any individual.
6. Create the offspring of each individual by mutating each dimension  $j$  using  $(b_j - a_j)d_j$ , in which  $b_j$  and  $a_j$  are the upper and lower of dimension  $j$ , and  $d_j = 2(r_2 - 0.5)(1 - F(x_i))$ ;  $r_2$  is a random number in  $[0, 1)$ .
7. Add all the offspring to the current population and select the  $popSize$  individuals with the highest fitness to become the next population.
8. If the number of function evaluations does not yet exceed 5000, return to step 2.

Table 2: TreeEvolver's available function nodes and operator nodes for all 9 experiments in this paper. A diverse set of initial conditions was used to explore the space of possibilities.

Exp	Initial	Functions	Operators
1	$x_1 + x_2$	sin	-, +, *
2	$x_1 * x_2 + 7$	sin	-, +, *
3	$\sin(x_1) + \sin(x_2)$	sin	-, +, *
4	$x_1 + x_2$	sin	-, +, *, /, ^
5	$(x_1/3) * (x_2/6)$	sin	-, +, *, /, ^
6	$\sin(x_1)/5 + x_2^2$	sin	-, +, *, /, ^
7	$x_1 + x_2$	sin, cos, sqrt, exp, abs	-, +, *
8	$\sqrt{x_1} *  \sin(x_2) $	sin, cos, sqrt, exp, abs	-, +, *
9	$\cos(x_1) * \exp(x_2)$	sin, cos, sqrt, exp, abs	-, +, *

## 4 OBJECTIVE LANDSCAPES FROM OBJECTIVE FUNCTION TREES

In genetic programming, the most common representation of an evolvable program is a tree data structure (Banzhaf et al., 1998). For continuous objective functions such as in this study, this need not be different. The binary trees representing the functions consist of maximally four node types:

1. The **single-child functions nodes** contain  $\sin(x)$ ,  $\cos(x)$ ,  $\exp(x)$ ,  $\text{abs}(x)$  or  $\text{sqrt}(\text{abs}(x))$ . Note that the  $\text{abs}$  inside the  $\text{sqrt}$  is to accommodate consistency in the real numbers, and avoid squarerooting a negative number.
2. The five **operator nodes**  $+$ ,  $-$ ,  $*$ ,  $\backslash$  and  $^$  all have two child nodes *left* and *right* which are processed *inorder* to accommodate for the three non-commutative (or 'asymmetric') operators. As such, subtraction is therefore understood as *left* - *right*, division as *left* \ *right* and power as *left*<sup>*right*</sup>.
3. **Variable nodes** contain all occurrences of  $x_1$  and  $x_2$ , given the fact that this study is limited to two-dimensional objective functions (with three-dimensional landscapes).
4. **Constant nodes** contain constant numeric values, such as 6, -4.22 or -0.00004. The value of a constant is kept within by the lowest power of 10 that encapsulates the initial value. For instance, if a constant node is instantiated with the value 4.6, its bounds will be  $[-10, 10]$ . If a constant node is instantiated with the value 3728 it will have the bounds  $[-10.000, 10.000]$ .

A few examples of function trees can be seen in Figure 2. Note that it is possible for functions to be represented by different trees; for example  $x_1 + x_2 + 5$

can have the first  $+$  or the second  $+$  as its root. In such cases the first operator is designated the root node. Furthermore, mathematical equivalents such as  $(x_1 * x_1)$  versus  $(x_1)^2$ , which could be regarded as symmetry in TreeEvolvers search space, is unaddressed.

## 5 EVOLVING OBJECTIVE FUNCTION TREES

The TreeEvolver is an implementation of genetic programming and follows a simple hillClimber paradigm: it makes a single mutation to the function tree, assesses its new MOD-hardness, keeps the new tree iff harder, and otherwise reverts the mutation. Abiding by the well-known KISS-doctrine<sup>3</sup>, it functions pretty well for getting the concept working, but more sophisticated methods, such as population-based tree evolvers, could certainly be incorporated in future experiments – most notably because the crossover operator can be readily implemented on this concept. In the current implementation, the algorithm begins each generation by creating a list of all eligible mutations on the current function tree.

The first and most straightforward mutation type, **change node**, is the only mutation type that does not change the number of nodes in a function tree. For trees to remain valid, the value of a selected node can only be changed into a value of the same type, e.g., a function node can only be changed into another function node, and not into an operator node. For example, a change from -4.27 to 5.5, a change from  $\sin(2 * x_1 + 4)$  to  $\exp(2 * x_1 + 4)$  or a change from  $\exp(2 * x_1 + 4)$  to  $\exp(\frac{2}{x_1} + 4)$  are all valid change

<sup>3</sup>Keep It Simple, Scientist!

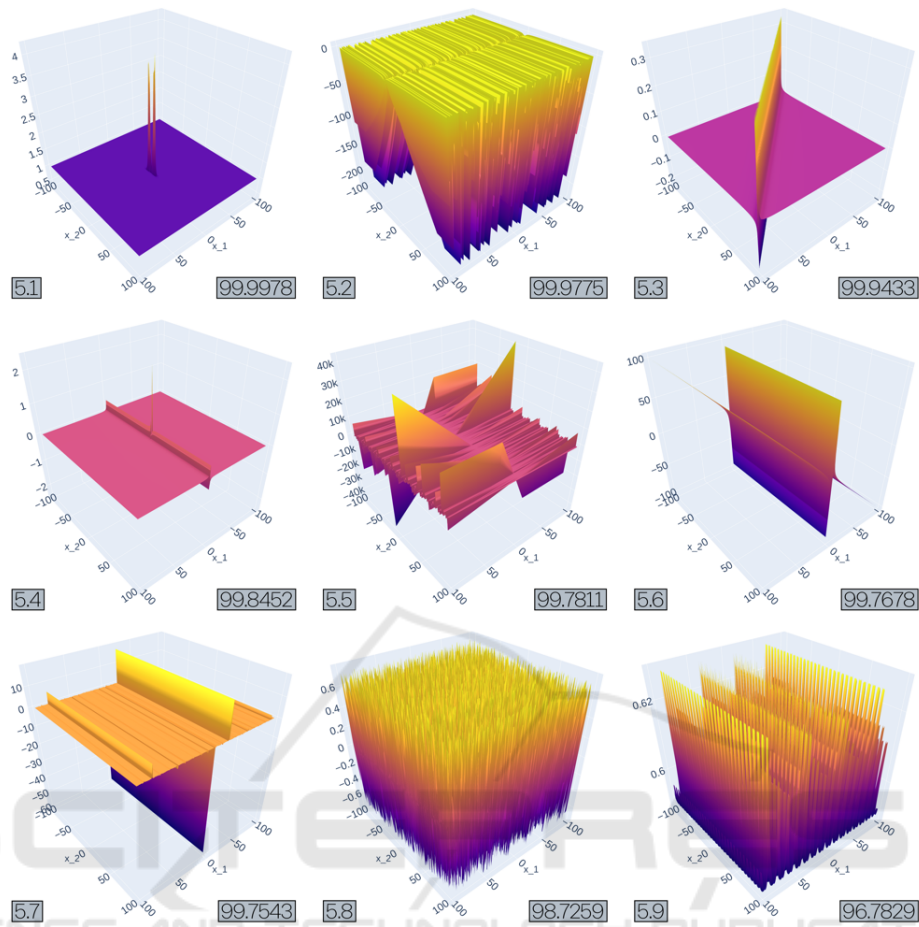


Figure 4: All 9 objective landscapes from experiment 5 after 2000 generations in TreeEvolver, ordered by MOD. All runs had  $x_1 + x_2$  as their initial function. Operator nodes were limited to a value from  $[+, -, *]$ , and function nodes were limited to a value from  $\{\sin\}$ .

node mutations. If there is only one possible value for a node type, and therefore the mutation would cause no real change, it is not added to the list.

Second, **insert function node** creates a new random functions node  $F$ , and then randomly chooses an existing node  $N$  from the tree above which  $F$  is inserted. Changing  $2 * x_1 - 4 * x_2$  to  $2 * x_1 - \cos(4 * x_2)$  is a valid insert function node mutation, and so is changing  $2 * x_1 - 4 * x_2$  to  $|2 * x_1 - 4 * x_2|$ .

The third mutation type, **remove function node**, randomly chooses a function node  $F$  from the current tree at random, removes it, and links its child to its parent. This mutation type will only be added to the list when the function tree has at least one function node.

Fourth, **insert operator node** first chooses a node  $N$  in the current tree above which to insert a newly randomly generated operator node  $O$ . Node  $N$  will become one of  $O$ 's children, the other will be a randomly generated variable node or constant node. Both

the choice variable-or-constant and the choice left-or-right are made at random.

Fifth, **remove operator node** selects an existing operator node from the tree, choosing one of its children at random to take its place. The other of the two children is deleted.

After listing all eligible mutations, a tree of  $n$  nodes of which  $l$  leave nodes, the list will contain  $4n - l$  mutations:  $n$  change node mutations,  $n$  insert function node mutations,  $n$  insert operator node mutations,  $n - l$  combined remove node mutations. A mutation is then randomly chosen from the list of possibilities and executed on the given tree. Finally, a check is performed whether the tree still holds at least one of the variables  $x_1$  or  $x_2$  in at least one place, which could have been accidentally deleted from a subtree. If this is not the case, the mutation is reverted and a different mutation is chosen.

Table 3: The 9 objective from experiment 5 ordered by MOD. All runs had  $(x_1/3) * (x_2/6)$  as their initial function. Operator nodes were limited to  $\{-, +, *, /, ^\}$ , and Function nodes were limited to a value from  $\{\sin\}$ .

	MOD	Function
5.1	99.9978	$(x_2 - \sin(\sin(4.572 * \sin(\sin(x_2 + \sin(\sin(x_1)) + 7.655/(3 + x_2))))))^\wedge \sin(\sin(\sin(x_1^\wedge(-2.454 * 2.968)))/x_2/8.654)^\wedge 9.814)$
5.2	99.9775	$-6.199/x_2^\wedge \sin(\sin(x_1/\sin(\sin(\sin(\sin(\sin(\sin(\sin(\sin(\sin(x_1)))/\sin(-5.354))))))) * \sin(-1.081))))))$
5.3	99.9433	$\sin(6.786/\sin(\sin(\sin(4.276))))/\sin(-1.864)/(x_1 - 4.944 + x_2) * 6$
5.4	99.8452	$x_2/(\sin(7.872) + 7.652 * x_2 * x_1)$
5.5	99.7811	$-1.811/x_2/x_2 * 1.817/7.001/(x_1 + 9.93/\sin(\sin(x_2)))/x_1)$
5.6	99.7678	$x_1 + 2.447/x_1/3 * 3.563/\sin(\sin(-1.598))$
5.7	99.7543	$-2.373^\wedge((-6.394 - x_1)/x_1/\sin((3.648/4.956)^\wedge 2.674))/10 * \sin(x_1 - 6.537)/\sin(\sin(6 - 0.648^\wedge - 7.139))$
5.8	98.7259	$\sin(\sin(\sin(\sin(-4.426) * x_2/7.683^\wedge(\sin(x_2)/\sin(x_1)/\sin(\sin(\sin(3))))))) x_1/x_1/\sin(x_1 + 10)$
5.9	96.7829	$\sin(\sin(\sin(\sin(\sin(\sin(\sin(-6.487 + x_1 * -0.244)^\wedge \sin(\sin(\sin(\sin(\sin(x_2)) + 5.185)))/10))))))$

## 6 EXPERIMENT & RESULTS

For this first exploration we conduct 9 experiments with widely different combinations of available function nodes, operator nodes and initial objective functions with a domain of  $[-100, 100]^2$ . Each initial objective function will undergo 9 independent evolutionary runs of 2000 generations in TreeEvolver, making for a total of 81 runs. Exact settings of the experiments can be found in Table 2, and full results are publicly available (Anonymous, 2022). For this exploration, we will be discussing experiments 1, 5 and 8 as a representative overview of the enormous diversity in results.

All experiments were run on a HP ZBook G5 with Intel Core i7-750H processor and 16 GB RAM running Ubuntu 20.04.2 LTS. All Python code was written in Python 3.10. All benchmark calculations are done using NumPy 1.21.4. Analysis is done using Pandas 1.3.4. Finally, Visualization of the benchmarks is done using Plotly 5.4. For consistency, all values are rounded to 10 decimals, and clipped at  $-1,000,000$  and  $1,000,000$ . All initial objective functions have  $MOD = 0$ , and are thus extremely easy.

### 6.1 Experiment 1

Experiment 1, the smallest in the ensemble, starts off initial objective function  $x_1 + x_2$ , with operator nodes  $\in \{+, -, *\}$  and function nodes  $\in \{\sin\}$  available to TreeEvolver. Table 1 holds the evolved functions after 2000 generations of the TreeEvolver in descending MOD, while Figure 3 shows the corresponding objective landscapes.

Roughly speaking, we can divide the functions into three groups. The first group consist solely of 1.1, a function that holds many multiplications and significantly harder than all others in this ensemble. Its landscape also looks quite a bit different. The second group consist of the five functions that are considerably less hard (1.2 to 1.6). Here, functions have grown substantially larger and have nested  $\sin$  functions, and have ‘butterflyish’ shapes, sometimes even ‘hairy’. These recurring patterns might tell us something about the hardness of these functions, or the preferred evolutionary paths of TreeEvolver. Finally, the existence of a group of easy functions (1.7 – 1.9) might signal that the initial conditions not optimally suited for making hard objective functions. Of these nine results, the first five functions are all substantially harder than our hardest household name, which is Easom with  $MOD_{Easom} = 14.26$ , which would come in fifth place here.

### 6.2 Experiment 5

Experiment 5, starting off from  $x_1/3 + x_2/6$ , having operators  $\in \{+, -, *, /, ^\}$  and function nodes  $\in \{\sin\}$ , resulted in some very hard objective landscapes after 2000 generations in TreeEvolver. The functional descriptions again show many nested sine functions (Table 3), but the landscapes are very different from experiment 1.1 (Figure 4). Instead of ‘hairy butterflies’, we now see largely flat stretches interrupted by very narrow extremities, possibly caused by the division operator. Of these, 5.6 could be argued to be a variation to the theme, as its tilt gives rise to a massive deceptive basin, covering half its domain.

Completely different are 5.2, 5.8, and 5.9 which



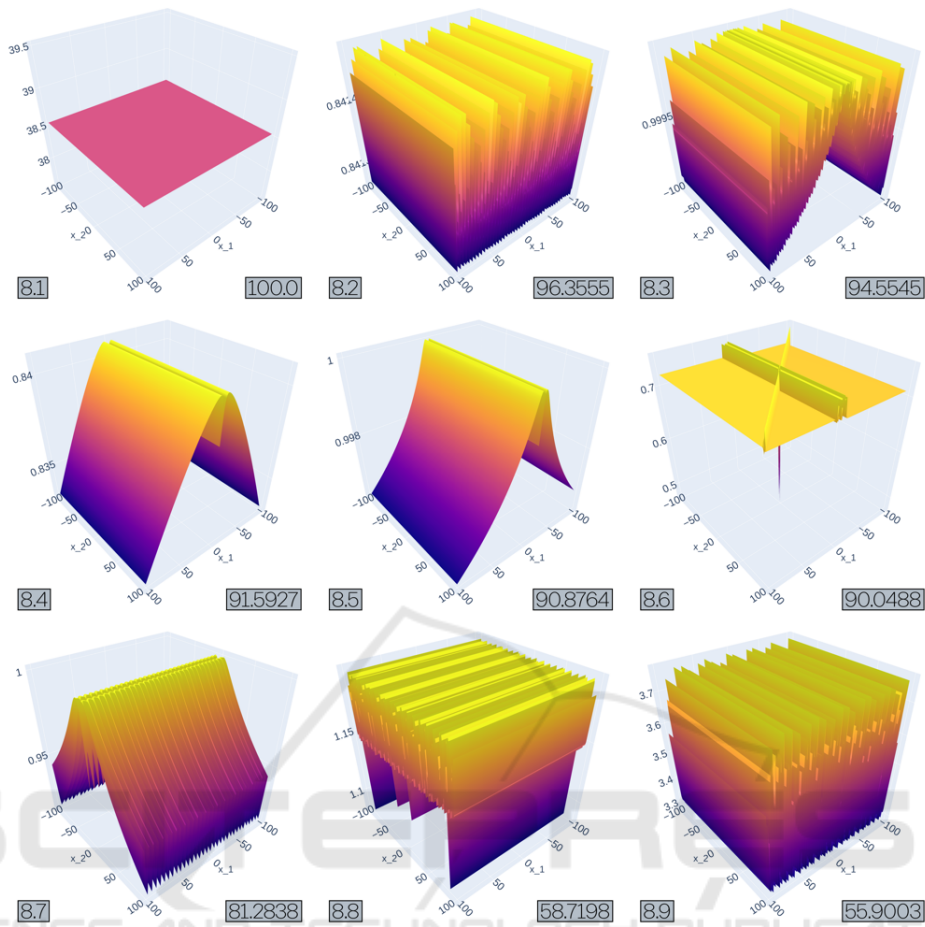


Figure 5: The 9 objective landscapes from experiment 9 after 2000 generations in TreeEvolver, ordered by MOD. All runs had  $\sqrt{x_1} * |\sin(x_2)|$  as their initial function, operator nodes were limited to  $\{-, +, *\}$ , and Function nodes were limited to a value from  $\{\sin, \cos, \text{sqrt}, \text{exp}, \text{abs}\}$ .

have large numbers of nested  $\sin$  functions and thereby have more of a ‘harmonicose structure’. Nonetheless, with the lowest MOD = 96.78, these functions are all very hard to minimize for PPA, and demonstrate that there is definitely more than one way of making hard objectives functions, and also that our rather unsophisticated TreeEvolver can readily find them in this experimental setting.

### 6.3 Experiment 8

TreeEvolver’s evolutionary trajectory from initial objective function  $\text{sqrt}(|(x_1)|) \cdot |\sin(x_2)|$ , with available operators  $\in \{+, -, *\}$  and function nodes  $\in \{\sin, \cos, \text{sqrt}, \text{abs}\}$  resulted in a yet a different species of very hard objective landscapes after 2000 generations of TreeEvolver, that can be roughly seen as three groups.

The first group (8.1 and 8.6) are flat surfaces, similar to those in Experiment 5, with extremely deep

and narrow minima; so narrow even they can hardly be rendered. The second group (8.2, 8.3, 8.8 and 8.9) shows another familiar pattern: a highly convoluted surface with harmonicose structure, also seen in Experiments 1 and 5. This time however, it is made up from mostly nested  $\text{abs}$  and  $\text{sqrt}$  functions, perhaps showing that similar hardness patterns can be achieved through different means. These harmonicities however, are of only mediocre hardness: harder than those in Experiment 1, but significantly easier than those in Experiment 5.

The third group (8.4, 8.5 and 8.7) is entirely new. These three tent-like surfaces have a very narrow central chasm and two sloping sides that basin off nearly the entire domain. These are highly deceptive functions, but look so simple that one cannot help asking if a smaller functional descriptions exist than the current 20+ invocations of  $\text{abs}$ ,  $\sin$  and  $\text{sqrt}$  functions. Another ready question is why the surfaces only slope over *one* dimension. Is it just a coincidence, or is



- <https://anonymous.4open.science/r/TreeEvolver-C005/README.md>.
- Applegate, D. L., Bixby, R. E., Chvátal, V., Cook, W., Espinoza, D. G., Goycoolea, M., and Helsgaun, K. (2009). Certification of an optimal tsp tour through 85,900 cities. *Operations Research Letters*, 37(1):11–15.
- Banzhaf, W., Nordin, P., Keller, R., and Francone, F. (1998). *Genetic Programming: An Introduction on the Automatic Evolution of computer programs and its Applications*.
- Bartz-Beielstein, T., Doerr, C., Berg, D. v. d., Bossek, J., Chandrasekaran, S., Eftimov, T., Fischbach, A., Kerschke, P., La Cava, W., Lopez-Ibanez, M., et al. (2020). Benchmarking in optimization: Best practice and open issues. *arXiv preprint arXiv:2007.03488*.
- Bezdek, J. C., Boggavarapu, S., Hall, L. O., and Bensaïd, A. (1994). Genetic algorithm guided clustering. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 34–39. IEEE.
- Dahmani, R., Boogmans, S., Meijs, A., and Van den Berg, D. (2020). Paintings-from-polygons: Simulated annealing. In *International Conference on Computational Creativity (ICCC'20)*.
- De Falco, I., Tarantino, E., Cioppa, A. D., and Fontanella, F. (2006). An innovative approach to genetic programming—based clustering. In *Applied Soft Computing Technologies: The Challenge of Complexity*, pages 55–64. Springer.
- de Jonge, M. and van den Berg, D. (2020). Parameter Sensitivity Patterns in the Plant Propagation Algorithm. Number April 2020. IJCCI 2020: Proceedings of the 12th International Joint Conference on Computational Intelligence.
- De Jonge, M. and Van den Berg, D. (2020). Plant Propagation Parameterization : Offspring & Population Size. volume 2, pages 1–4. *Evo\* LBA's 2020*, Springer.
- Digalakis, J. G. and Margaritis, K. G. (2001). On benchmarking functions for genetic algorithms. *International journal of computer mathematics*, 77(4):481–506.
- Dijkzeul, D., Brouwer, N., Pijning, I., Koppenhol, L., and van den Berg, D. (2022). Painting with evolutionary algorithms. In *International Conference on Computational Intelligence in Music, Sound, Art and Design (Part of EvoStar)*, pages 52–67. Springer.
- Eiben, A. E., Smith, J. E., et al. (2003). *Introduction to evolutionary computing*, volume 53. Springer.
- Espejo, P. G., Ventura, S., and Herrera, F. (2009). A survey on the application of genetic programming to classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 40(2):121–144.
- Fischer, T., Stützle, T., Hoos, H., and Merz, P. (2005). An analysis of the hardness of tsp instances for two high performance algorithms. In *Proceedings of the Sixth Metaheuristics International Conference*, pages 361–367.
- Fodorean, D., Idoumghar, L., N'diaye, A., Bouquain, D., and Miraoui, A. (2012). Simulated annealing algorithm for the optimisation of an electrical machine. *IET electric power applications*, 6(9):735–742.
- Fraga, E. S. (2019). An example of multi-objective optimization for dynamic processes. *Chemical Engineering Transactions*, 74:601–606.
- Geleijn, R., van der Meer, M., van der Post, Q., van den Berg, D., et al. (2019). The plant propagation algorithm on timetables: First results. *EVO\* LBA's*, page 2.
- Haddadi, S. (2020). Plant propagation algorithm for nurse rostering. *International Journal of Innovative Computing and Applications*, 11(4):204–215.
- Jie, L., Xinbo, G., and Li-Cheng, J. (2004). A csa-based clustering algorithm for large data sets with mixed numeric and categorical values. In *Fifth World Congress on Intelligent Control and Automation (IEEE Cat. No. 04EX788)*, volume 3, pages 2303–2307. IEEE.
- Joshi, M., Gyanchandani, M., and Wadhvani, R. (2021). Analysis of genetic algorithm, particle swarm optimization and simulated annealing on benchmark functions. In *2021 5th International Conference on Computing Methodologies and Communication (ICCMC)*, pages 1152–1157. IEEE.
- Koppenhol, L., Brouwer, N., Dijkzeul, D., Pijning, I., Slegers, J., and Van Den Berg, D. (2022). Exactly characterizable parameter settings in a crossoverless evolutionary algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1640–1649.
- Kordon, A. K. (2010). *Applying Computational Intelligence How to Create Value*. Springer.
- Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.
- Koza, J. R. (1994). Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112.
- Koza, J. R. (2008). Human-competitive machine invention by means of genetic programming. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 22(3):185–193.
- Koza, J. R. and Rice, J. P. (1992). Automatic programming of robots using genetic programming. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, AAAI'92, page 194–201. AAAI Press.
- Laguna, M. and Marti, R. (2005). Experimental testing of advanced scatter search designs for global optimization of multimodal functions. *Journal of Global Optimization*, 33(2):235–255.
- Lam, B. and Ciesielski, V. (2004). Discovery of human-competitive image texture feature extraction programs using genetic programming. In Deb, K., Poli, R., Banzhaf, W., Beyer, H.-G., Burke, E., Darwen, P., Dasgupta, D., Floreano, D., Foster, J., Harman, M., Holland, O., Lanzi, P. L., Spector, L., Tettamanzi, A., Thierens, D., and Tyrrell, A., editors, *Genetic and Evolutionary Computation – GECCO-2004, Part*

- II, volume 3103 of *Lecture Notes in Computer Science*, pages 1114–1125, Seattle, WA, USA. Springer-Verlag.
- Liu, Y., Özyer, T., Alhajj, R., and Barker, K. (2005). Cluster validity analysis of alternative results from multi-objective optimization. In *Proceedings of the 2005 SIAM International Conference on Data Mining*, pages 496–500. SIAM.
- Lyman, M. and Lewandowski, G. (2005). Genetic programming for association rules on card sorting data. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1551–1552.
- Niewenhuis, D. and van den Berg, D. (2022). Making hard(er) benchmark test functions. In *IJCCI*, pages 29–38.
- Niewenhuis, D. and van den Berg, D. (2023). Classical benchmark functions, but harder (submitted). Springer.
- Orzechowski, P. and Moore, J. H. (2022). Generative and reproducible benchmarks for comprehensive evaluation of machine learning classifiers. *Science Advances*, 8(47):eabl4747.
- Paauw, M. and Berg, D. v. d. (2019). Paintings, polygons and plant propagation. In *International Conference on Computational Intelligence in Music, Sound, Art and Design (Part of EvoStar)*, pages 84–97. Springer.
- Reinelt, G. (1991). Tsplib—a traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384.
- Rodman, A. D., Fraga, E. S., and Gerogiorgis, D. (2018). On the application of a nature-inspired stochastic evolutionary algorithm to constrained multi-objective beer fermentation optimisation. *Computers & Chemical Engineering*, 108:448–459.
- Rosenberg, M., French, T., Reynolds, M., and While, L. (2021). A genetic algorithm approach for the euclidean steiner tree problem with soft obstacles. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 618–626.
- Salhi, A. and Fraga, E. S. (2011). Nature-inspired optimisation approaches and the new plant propagation algorithm.
- Selamoğlu, B. İ. and Salhi, A. (2016). The plant propagation algorithm for discrete optimisation: The case of the travelling salesman problem. In *Nature-inspired computation in engineering*, pages 43–61. Springer.
- Sette, S. and Boullart, L. (2001). Genetic programming: principles and applications. *Engineering Applications of Artificial Intelligence*, 14(6):727–736.
- Sleegers, J. and Berg, D. v. d. (2021). Backtracking (the) algorithms on the hamiltonian cycle problem. *arXiv preprint arXiv:2107.00314*.
- Sleegers, J. and van den Berg, D. (2020a). Looking for the hardest hamiltonian cycle problem instances. In *IJCCI*, pages 40–48.
- Sleegers, J. and van den Berg, D. (2020b). Plant propagation & hard hamiltonian graphs. *Evo\* LBA's*, pages 10–13.
- Sleegers, J. and van den Berg, D. (2022). The hardest hamiltonian cycle problem instances: The plateau of yes and the cliff of no. *SN Computer Science*, 3(5):1–16.
- Socha, K. and Dorigo, M. (2008). Ant colony optimization for continuous domains. *European journal of operational research*, 185(3):1155–1173.
- Sulaiman, M., Salhi, A., Fraga, E. S., Mashwani, W. K., and Rashidi, M. M. (2016). A novel plant propagation algorithm: modifications and implementation. *Science International*, 28(1):201 – 209.
- Sulaiman, M., Salhi, A., Khan, A., Muhammad, S., and Khan, W. (2018). On the theoretical analysis of the plant propagation algorithms. *Mathematical Problems in Engineering*, 2018.
- Vrieling, W. and van den Berg, D. (2019). Fireworks algorithm versus plant propagation algorithm. In *IJCCI*, pages 101–112.
- Vrieling, W. and Van den Berg, D. (2021a). A Dynamic Parameter for the Plant Propagation Algorithm A Dynamic Parameter for the Plant Propagation Algorithm. Number March, pages 5–9. *Evo\* LBA's 2021*, Springer.
- Vrieling, W. and Van den Berg, D. (2021b). Parameter control for the Plant Propagation Algorithm Parameter control for the Plant Propagation Algorithm. Number March, pages 1–4. *Evo\* LBA's 2021*, Springer.
- Weise, T., Li, X., Chen, Y., and Wu, Z. (2021). Solving job shop scheduling problems without using a bias for good solutions. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1459–1466.
- Weise, T. and Wu, Z. (2018). The tunable w-model benchmark problem.
- Zhang, W. and Korf, R. E. (1996). A study of complexity transitions on the asymmetric traveling salesman problem. *Artificial Intelligence*, 81(1-2):223–239.