Grammar-based Tree Swarm Optimization

David Griñán[1], Alfredo Ibias[2] and Manuel Núñez[2]

***Abstract*— Particle Swarm Optimization (PSO) has been successfully applied to find good solutions through a guided search. This optimization technique usually works with vectors as individuals of the population conforming the search space. Nevertheless, there exist problems such that the search space cannot be transformed into a vector search space. In this paper we propose a novel technique based on the intuition behind PSO but overcoming its limitations concerning search spaces. Specifically, we present a PSO framework where the individuals conforming the search space are tree-like structures. In particular, our framework naturally includes classical PSO but also search spaces where elements are structures that can be represented as trees (in addition to usual trees, linear structures such as lists, queues and stacks).**

I. Introduction

Search problems are common in the fields of Mathematics and Computer Science. One of the most popular groups of techniques to solve this kind of problems is the *metaheuristics family*. This is a very big area encompassing different paradigms that distinguish from each other in the way they compute the solutions. For example, if the solution is built from an initial solution after constantly improving it, then we have *global search algorithms* [6]. Another approach considers a population of candidate solutions and evolve them. In this case we are considering the subfamily of *evolutionary techniques* [1]. Finally, if the algorithm starts from an empty solution and builds upon it, then we are considering a *constructive technique* such as the well-known Ant Colony Optimization paradigm [8].

Particle Swarm Optimization (PSO) [5], [11] is a population-based trial and error search algorithm that falls into the evolutive techniques subfamily, which relies on the communication among the particles of the population to find the best solution for a given search problem. Particles in this algorithm move all over the search space and have a memory to record the best solution that they have seen, as well as the location of the best solution that the population has found. The next location of each particle is obtained using this information. As it has been described, this algorithm relies on the concepts of location and direction, which rules out spaces where there is no similar notion of either of them.

In this paper we are going to tackle the specific problem of the search space, more expressly the constraints that classical PSO imposes in the individuals of the search space. A search space can be defined as the set of possible solutions of a given problem, where the algorithm tries to find the best one (or at least, a very good one). The most common search spaces are subsets of sets such as $\{0,1\}^n$, $\mathbb{N}^n$, $\mathbb{Z}^n$ and $\mathbb{R}^n$. These sets can encode most candidate solutions, not limited to simple numbers but including CPU assignments [16], card combinations [10], the weights of a given neural network [2] or the number of neurons per layer for a deep neural network [3]. In addition, there are other variations of the PSO algorithm that work with discrete search spaces [17]. However, there are problems that need more complex structures to define the search space. Specifically, we are interested in *tree search spaces*. They can be used to solve, in a natural way, most of the problems that used the previously described search spaces. In addition, they can be used in problems with more complex solutions. For example, problems like finding the maximum spanning tree of a given graph, the shortest path in a graph or the best Bayesian network structure [9] can be solved using this search space. Notably, and this is the main contribution of this paper, tree search spaces have not been used before to conform the individuals of the search space in a PSO. There are two main differences between tree search spaces and other search spaces mentioned earlier. The first one is the absence of a size limit. Solutions in standard search spaces are fixed-length vectors, which can be interpreted in order to simulate solutions of different sizes but will always have an upper bound on their size. However in tree search spaces there is no such size limit.[1] The second difference is that there is no notion of *direction* or *location*. Standard search spaces are, essentially, subsets of $\mathbb{R}^n$ and they have a clear notion of direction defined as a vector. In contrast, tree search spaces are not equipped with a similar notion what would facilitate, actually, the translation between both search spaces.

There is a family of trees that hold certain particularly interesting conditions when defining search spaces. We are referring to *grammar derivation trees*. Grammar-guided techniques are common in the field of evolutionary programming, specifically in *genetic programming* [7], [12], [14]. Usually, trees are used in genetic programming to encode the solution in their leaves, traversing them in post-order and solving the *closure problem*. This problem appears when we work with a tree search space that has not been generated by applying a

Research partially supported by the Spanish MINECO/FEDER projects DArDOS (TIN2015-65845-C3-1-R) and FAME (RTI2018-093608-B-C31) and the Comunidad de Madrid project FORTE-CM (S2018/TCS-4314) co-funded by EIE Funds of the European Union.

[1]Polytechnic University of Madrid, Madrid 28223, Spain david.grinanm@alumnos.upm.es

[2]Complutense University of Madrid, Madrid 28040, Spain {aibias,manuelnu}@ucm.es

[1]In practice, there must be a bound on the size of the trees depending on the memory of the computer where we run the tests and on how the chosen programming language manages memory usage. Since this is variable, and generally speaking we are going to have high memory capacity, from this point on we will assume that there is no bound on a tree size.

grammar and, therefore, the population can include trees that represent either invalid solutions or elements that cannot be a viable solution. In order to fix this problem, context-free grammars are used to ensure that every tree in the population encodes a valid candidate solution. This approach needs specific crossover methods that do not violate the closure property, such as the grammar based crossover [13] and the Wickham crossover [7].

In this paper we modify the original PSO algorithm to deal with tree search spaces. The main novelty of our approach, in addition to the differences already explained, is the capability to effectively use an oriented search method in a tree search space. In order to show the usefulness of our approach, we have applied it to solve a classical problem where a tree search space is needed: the generation of balanced equations problem. We have compared our PSO solution with one where a classical genetic programming algorithm was used.

The rest of the paper is structured as follows. In Section II we introduce the basic concepts we will use along the paper. In Section III we introduce our adaptation of the PSO algorithm. In Section IV we perform experiments to compare our algorithm with currently established algorithms. In Section V we discuss the decisions that we made along our work. In Section VI we discuss the threats to the validity of our results. Finally, in Section VII we summarize the conclusions of our work.

II. Preliminaries

Particle Swarm Optimization (PSO) is a collaborative technique that searches for solutions of a given problem in a vector space. This methodology is based on the behaviour of bird flocks and fish schooling, where communication among individuals in the population is key. Using its specific terminology, elements in the population are referred to as *particles*, which move around the multidimensional vector space to find the optimal solution by trial and error. In order to update their position, particles rely both on their own experience and on the collective experience of the population, which is induced by the memory each particle has. Any given particle in the population remembers (has stored) the best position in the search space that it has visited: this is its *inner memory*. Every particle also knows the best out of all the positions the population remembers, that is, the *global memory*. These two pieces of information are adjusted by tuning some user-specified parameters and by considering two random parameters, between 0 and 1, which not only choose the relative importance of each type of memory, but also establish how far a particle will travel towards its new position. Specifically, the update of the position of each particle in the i-th epoch follows the following formula:

$$p_i = p_{i-1} + v_i$$

where p_i is the particle position in the i-th epoch and v_i is the particle *velocity* in the i-th epoch. In turn, velocity is given by the following formula:

$$v_i = v_{i-1} + \alpha\psi \cdot best_p + \beta\phi \cdot best$$

where $best_p$ is the inner memory of the particle, *best* is the global memory, α and β are the user-specified parameters for each memory, and ψ and ϕ are the random parameters for each memory.

The concept of velocity resembles the direction a particle moves in order to reach the best possible solution. It can be seen as moving each particle to a midpoint between their actual position and the position of the best solutions found. This concept is the basis of the translation from the concept of distance and direction in sets like $\mathbb{R}^n$ to a similar concept in our derivation trees space. Building upon this idea, we can say that particles *move* along the search space in a sort of directed manner instead of performing a random update of their state. This concept will be explored further in the following section.

A *grammar* G is a tuple $\{S, P, N, \Sigma\}$ where

- Σ is a finite set of elements that make up the words accepted by the grammar, which are called *terminal elements*.
- N is a finite set of elements, disjoint from Σ, called the *non-terminal elements*.
- $S \in N$ is the *start symbol* or axiom.
- P is a finite set of rules, called *production rules*, that allow the generation of words. Rules in this set must have at least one non-terminal element in the head of the rule, but have no restrictions for the tail.

In order to generate derivation trees, a specific family of grammars is used: *context-free grammars*. A production rule is said to be context free if the head of the rule is only one non-terminal element. For a grammar to be context free, all production rules have to be context free. The branch in evolutionary computing that uses this sort of grammars is called *grammar-guided genetic programming* [14]. Actually, genetic programming accumulates the majority of algorithms designed to deal with tree search spaces, both standard trees and derivation trees. Evolution of the individuals of the population in this algorithm is based on the selection of candidates for the mating pool based on their fitness value and a crossover operator that, in most cases, randomly selects a node of each tree and performs a subtree swap. The main goal of the work reported in this paper is to propose a collective intelligence algorithm that performs a similar process and is able to search for candidate solutions in tree search spaces. In order to define this algorithm, we will start with the standard PSO and will adapt some components in a similar way as the changes applied to genetic algorithms to create genetic programming.

III. Grammar-Based Tree Swarm Optimization

Our approach takes PSO as initial point and adapts it to perform a search in a tree search space. Therefore, we have to update and adapt the encoding, the initial population and the concept of velocity to this new scenario.

A. Encoding

The encoding that we will use for our algorithm will be a *tree shape encoding* because we will search in a tree search

space. In order to ensure that we are generating valid results, we will require that our solutions can be encoded using a context-free grammar, although our algorithm can be used with trees not generated by a grammar. Summarizing, we will encode each particle as a tree-shape element generated by a context-free grammar that encodes the solution space.

B. Initial Population

The initial population will be a set of trees generated by the context-free grammar. It is important that the initial population tries to cover all the possible solutions without bias. In order to do that, we will proceed as in grammar-guided genetic programming and avoid code bloat by using the maximum and minimum length that each production adds to the tree depth.

C. Velocity

Velocity, as known in the original PSO algorithm, does not have an obvious equivalent in a tree search space. Although there are proposals for a tree distance [4] that play the role of the distance in the original PSO, they are not useful in our framework because we cannot use them to transform one tree in a midpoint between itself and another tree. Therefore, we had to completely rethink the concept of velocity.

Our proposal uses a notion of velocity where we consider a function that moves each particle to a midpoint between its current position and the position of the best solution found. In this case, the positions will be trees and, therefore, we will transform each tree into a *midtree* somehow situated between the current tree and the best tree found. In order to perform this transformation, we will define an operation that plays the role of crossover in genetic algorithms. Our *tree quasi-intersection* (or simply tree intersection) takes two trees, keeps their common structure and performs an update of the part of the structure that it is different in each tree. The idea of the update is that it produces a new branch of the tree that is in between the characteristics of the two branches we had before. Specifically, the operation performs as follows. It traverses in level order both trees and compares each pair of nodes. If they correspond to the same non-terminal or terminal element, then the crossover continues with the following node. Otherwise, we have found a difference and, therefore, we have to perform an update. In order to implement the update, we move backwards to the father of the node and delete all his children. Then, we select a branch among all the possible branches that we can generate with our grammar. In addition, the depth of the new branch must be between the minimum and maximum depths of the former branches. That is, we select a *middle point* in between the structure of both trees. Note that we should select all potential new branches with the same probability in such a way that it represents the random factor of the original velocity. However, the computation of all the possible branches is computationally expensive. That is the reason why we generate a random branch with a depth between the desired boundaries.

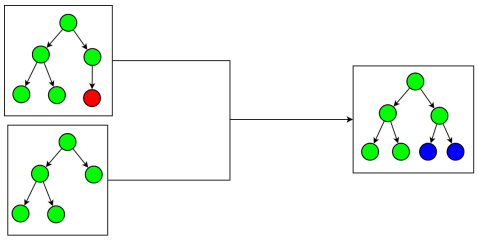

Fig. 1. Cross in the adapted PSO algorithm.

We can see a graphical representation of a tree quasi-intersection in Figure 1. Here, given two trees, the green nodes are the common structure and the red nodes are the differences in the structure. Then, we perform the operation and obtain a tree such that the green nodes are kept and a new branch is selected for the different nodes.

Next we give further details about some of the design decisions behind our notion of velocity. We delete all the children of the father of the node were we found the difference in order to avoid the problem of generating a combination of nodes that does not correspond to any production of the grammar. If we would replace only the different node (and its associated subtree), then we can generate a combination that does not correspond to any production of the grammar because we are modifying a part of the production of the grammar instead of choosing another production. Therefore, by removing all these children we avoid the generation of invalid solutions.

It is important to note that update only modifies the first tree, not the second one because the latter contains the best observed solution and we would like to keep it.

Finally, in order to more precisely adapt the concept of velocity to our setting, we have to perform this type of crossover two times: first, between the particle's inner memory and the global memory and, second, between the particle and the result of the previous crossover.

IV. Experimental Results

In this section we present a case study to show how our algorithm works and a comparison with an alternative solution using a classical algorithm. Specifically, we compare our algorithm with the performance of the grammar-guided genetic programming algorithm [14] because this is the most similar algorithm that works with trees (as we already said, most algorithms are not well-suited to work with tree search spaces). All the experiments were executed in a GNU/Linux machine with an Intel Core i7-7700HQ at 2.80GHz $\times$ 8 cores and with 16GB of RAM (although only one core was running at a time), and we have used python to code the algorithms.

We have considered a classical benchmark in grammar-guided approaches: the generation of balanced equations. Equations are conformed by an arithmetic expression of additions and subtracts, an equal operator, and a number at the other side that should be equivalent to the evaluation of

the arithmetic expression. The grammar that we have chosen to generate these equations is the following one [7]:

- $S \mapsto E = N$
- $E \mapsto E + E \mid E - E \mid N + N \mid N - N \mid N$
- $N \mapsto 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

where S is the start symbol.

We will compare the solution returned by our algorithm with a grammar-guided genetic programming algorithm having a selection probability for the mating pool directly proportional to the fitness value, a mating pool of a third of the population size, the Whigham crossover, a simple randomly replacement mutation with 3% of probability and an elitist replacement policy. These parameters have been chosen to mimic the standard values used in genetic programming [15].

In our experiments, we used the same grammar for both algorithms, bounding the depth of the generated tress between 2 and 10 levels. We also set an initial population of 20 individuals. As the common stop criterion, we used that the equation is balanced (that is, the difference between the evaluation of the arithmetic expression of both sides is 0) and that we perform a maximum of 10 epochs (that is, we compute at most 10 generations). Consequently, our fitness function for each individual is minus the absolute value of the difference between the left hand side of the equation and the right hand side, since we are trying to maximize its value. These values have been chosen to design an initial scenario with few hard constraints on the population size so that the algorithm has more flexibility. We bounded the depth of the generated trees because the grammar is intended for trees with a small depth. Finally, the bound on the maximum number of epochs has the goal of working within a scenario where obtaining optimal results is a complicated task. In other words, a small number of epochs will make more difficult to find the optimal solution.

In order to get meaningful results, we performed this whole experiment 10000 times for each algorithm, measuring the time elapsed and the number of times the algorithms returned as its best individual one that it is not balanced (that is, the number of times we got a sub-optimal solution). Due to its size, we cannot display the full results. Therefore, we will work with the average results of all the runs.

The experiment reveals that our proposal took 165.84 seconds to compute the result while the grammar-guided genetic programming algorithm needed 121.5 seconds. Neither of them got sub-optimal solutions. It is important to note that, as we are using the same method to generate the initial population, the number of times that the algorithms could avoid the evolutive loop because an optimal solution could be found in the initial population was almost the same. The results show that we got a similar performance, with the difference that our proposed algorithm lasted slightly longer. After analyzing the results of this first experiment we realized that if we have big populations, then our proposal will not work better, concerning execution time, than previous work. However, if the information that we need to store for each individual of the population is very big (for example, huge trees representing a part of a road map), then we realistically cannot afford to evolve populations with many individuals.

The previous observation gave rise to our second experiment. The population that we were evolving was decreased to 10 individuals, thus reducing the amount of information already stored in the population. Therefore, we wanted to check the performance of both algorithms if they need to strongly rely on their exploratory capabilities to find an optimal solution. The results of this experiment can be found in Table I. This time, instead of running the whole experiment only 10000 times, as we did in the previous experiment, we wanted to analyze how both algorithms evolve when we run them from 1000 times to 10000 times, which implies that a different number of epochs were performed on each scenario. It has to be taken into consideration that both algorithms have a limit of 10 epochs to find the optimum in each run, but along all the runs the number of aggregated performed epochs is higher, allowing us to analyze the performance in terms of epochs also. The two first columns of Table I indicate the scenario: the first one shows the number of runs of the algorithm while the second one presents the selected algorithm. To account for variability in the results, this procedure has been repeated 10 times and the averages of those 10 repetitions are stored in the following columns of the table. The third and fourth columns indicate, respectively, the average value of the accumulated fitness obtained and the average number of times that the algorithm has not reached an optimal solution in this scenario. The fifth column indicates the average number of epochs performed while the sixth column shows the number of runs that did not found an optimal solution in the initial population. Finally, the last column indicates the average elapsed time in this scenario. For instance, the information coded in the first row of Table I would describe the results after running the TSO algorithm a total of 1000 times. This scenario is repeated 10 times. For each of the 10 repetitions, we have added up the fitness of the best individual on each of the 1000 runs. The average of these results, as shown in the third column, is equal to $-1,8$. Moreover, the average of the number of runs that have not been able to produce an optimal solution was equal to $1,8$. Similarly, the remaining columns show information, on average, about epochs and the elapsed time for the selected scenario.

The results in Table I indicate that our algorithm consistently makes less mistakes than the genetic algorithm, even though the operation is more complex and thus takes almost double the time. It is worth mentioning that our algorithm requires less epochs to find an optimal solution, which represents roughly a reduction of 25% in the number of steps required. Also, it is important to note that when our algorithm fails to find an optimal solution, it fails by the minimum possible, what we can conclude by the fact that the difference between the absolute values of the accumulate fitness error and the number of runs with fitness error is 0. This situation did not happen with the genetic programming algorithm and this might show that our proposed algorithm truly moves towards the optimal solution (although it does

# runs	Algorithm	Accumulated fitness error	# runs with fitness error	Accumulated epochs	# runs with epochs	Elapsed time (s)
1000	*TSO*	−1,8	1,8	1338	564,8	14,85
1000	*GGGP*	−15,2	14,7	1776	578,5	8,12
2000	*TSO*	−5	5	2671,3	1136,4	29,41
2000	*GGGP*	−28,9	28,4	3471,8	1140,6	15,96
3000	*TSO*	−6,7	6,7	3979,8	1703,4	45,67
3000	*GGGP*	−43,4	43,3	5228,4	1701,8	25,07
4000	*TSO*	−8,7	8,7	5297,9	2281,7	59,59
4000	*GGGP*	−59,4	58,7	6912,4	2266,8	32,98
5000	*TSO*	−12,1	12,1	6695,2	2848,5	74,87
5000	*GGGP*	−68,6	67,5	8612,7	2849,4	40,95
6000	*TSO*	−13,5	13,5	8030	3422,8	89,78
6000	*GGGP*	−96,6	95	10455,2	3421,6	49,01
7000	*TSO*	−18,1	18,1	9329,6	3982,2	104,86
7000	*GGGP*	−103	101,8	12132,6	3991,7	58,08
8000	*TSO*	−20,1	20,1	10734,4	4565,7	120,98
8000	*GGGP*	−118,7	117	13854	4555,6	66,91
9000	*TSO*	−19,9	19,9	11921,9	5110,7	133,83
9000	*GGGP*	−128,2	127,6	15525,2	5109,3	74,69
10000	*TSO*	−24,8	24,8	13375,3	5720,2	147,21
10000	*GGGP*	−139,6	138,1	17199	5692,3	82,73

TABLE I

RESULTS OF THE EXPERIMENT.

# runs	Accumulated fitness error	# runs with fitness error	Accumulated epochs	# runs with epochs	Elapsed time (%)
1000	88,16%	87,76%	24,66%	2,37%	−82,83%
2000	82,70%	82,39%	23,06%	0,37%	−84,24%
3000	84,56%	84,53%	23,88%	−0,09%	−82,16%
4000	85,35%	85,18%	23,36%	−0,66%	−80,67%
5000	82,36%	82,07%	22,26%	0,03%	−82,86%
6000	86,02%	85,79%	23,20%	−0,04%	−83,20%
7000	82,43%	82,22%	23,10%	0,24%	−80,54%
8000	83,07%	82,82%	22,52%	−0,22%	−80,82%
9000	84,48%	84,40%	23,21%	−0,03%	−79,19%
10000	82,23%	82,04%	22,23%	−0,49%	−77,93%

TABLE II

PERCENTAGES OF IMPROVEMENT (TSO VS GGGP).

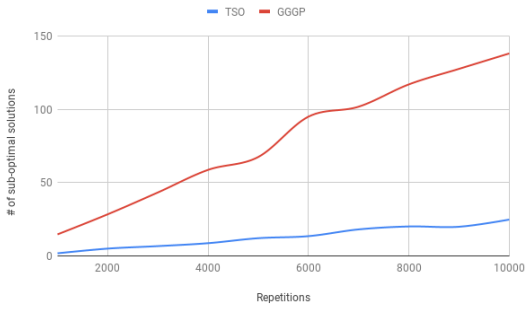

Fig. 2. Sub-optimal solutions obtained by each method.

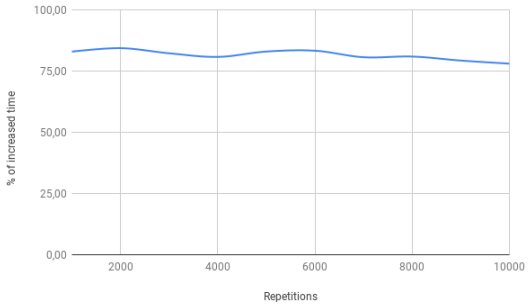

Fig. 3. Performance times comparison (in percentage of increase of TSO with respect to GGGP).

not reach it before the bound of epochs is reached). Figure 2 shows the differences in the number of sub-optimal solutions.

Table II shows the improvement percentages of our proposed algorithm with respect to the genetic programming algorithm. Our proposal shows an improvement on the results between 80% and 90%, although it takes between 75% to 85% more time to obtain them (see Figure 3). As a summarizing remark, if our algorithm reaches a *difficult to find* optimal solution (for example, if we have a low bound on the number of epochs, then it is *difficult* that we reach the exact optimal solution), then it is unlikely that the genetic algorithm reaches it under the same conditions. This can be clearly seen in Table II because our algorithm always beats the genetic one. However, our algorithm will need, in general, more time to compute this solutions.

V. DISCUSSION

It is necessary to discuss some topics that arose during the development of our algorithm. Specially, we would like to return to the idea behind the proposed notion of *velocity*, and mention alternative proposals that could work well, and discuss about the nature of our algorithm because it combines constructive and evolutive ideas.

The idea behind our proposed notion of *velocity* is to be as similar as possible to the original velocity. Therefore, the development of a concept of middle point between the actual particle and the best particle has to be kept by the proposed operation. Using this concept allows the algorithm to be somehow directed, so that it requires a lower amount of operations (as the experiments show). This is similar to the original idea behind the PSO algorithm where we will perform a *not so random* search.

Other proposals for *velocity* arise when thinking about solutions. An easy and direct approach could be classical crossover from genetic programming algorithms. This option has the advantage that we do not need to introduce a new concept because we can use the already existing crossovers and benefit from its characteristics based on the problem we want to solve. In fact, our preliminary work shows that they are indeed a feasible option and they should not to be discarded lightly. The downside of this option is that they are purely random crossover of trees, where there is no notion of aim as in a directed search. Also, it is difficult to find a feasible translation from the standard tree crossover to the calculation of a middle point between two trees. This is why we developed a different operator as otherwise we could be losing part of the spirit of the PSO algorithm.

Remember that our notion is such that the midpoint between two trees is a tree that starts with the intersection of the two original trees and develops from there using the grammar devised for the problem. In this regard, it has a high exploration capability. For example, if the intersection is applied in the root of the tree, then we generate a completely new individual. Moreover, if two trees are similar, then their descendant will build upon the common structure of both trees to improve their results. In this regard, our operator can balance between exploration and exploitation, being clear when the algorithm converges towards a good solution because the common structure will grow and be common

for most of the individuals of the population. If one was going to deeply study our concept, it would be possible to argue that this is a mixed heuristic, since the main approach comes from the evolutive family but the development in the population is based on the notion of a *best seen particle*, which evolves over time and is slowly being built. Following that line of thought, this heuristic could also be catalogued as a constructive algorithm.

VI. Threats to Validity

The conclusions obtained after performing our experiments have possible threats to their validity. In this section we will discuss some of them.

Concerning threats to *internal validity*, which consider uncontrolled factors that might be responsible for the obtained results, the main threat is associated with the possible faults in the developed experiment because they could lead to misleading results. In order to reduce the impact of this threat we tested our code with carefully constructed examples for which we could manually check the results. In addition, we repeated the experiment many times in order to get a mean so that the randomization impact is reduced.

The main threat to *external validity*, which concerns conditions that allow us to generalize our findings to other situations, is the different possible problems to which we could apply our algorithm. Such a threat cannot be entirely addressed since the population of possible problems is unknown and it is not possible to sample from this (unknown) population. A proper consideration of this problem will be matter of future work.

Finally, we considered threats to *construct validity*, which are related to the *reality* of our experiments, that is, whether they reflect real-world situations or not. In our work, the main construct threat is what would happen if we used our algorithm with much more complex problems. We do not expect very different results but, certainly, more experiments are needed and this should be part of our future work.

VII. Conclusions and future work

Evolutionary programming encompasses interesting algorithms to solve relatively complex problems. In this paper, we contributed to this field, in particular concerning computational collective intelligence, with a new revision of a well-known algorithm, expanding its possibilities by increasing the variety of search spaces that it can handle. Specifically, we have adapted PSO to work with tree-like structures. This new algorithm will allow users to solve a huge new brand of problems that could not be solved before with classical PSO. It is worth to mention that we keep the spirit of the original PSO. In particular, our algorithm is able to perform more guided searches. We have compared our algorithm with the current standard in tree-like search spaces: the genetic programming algorithm. We have shown that our algorithm, thanks to its more guided nature, improves the results of the existing ones when they are not able to obtain a valid solution, although our algorithm needs more time to do so. Lastly, we have discussed other possible approaches that we could have chosen and consistently explained why we made those decisions.

This is only the first step on the application of a new algorithm. Future work should expand in the possible uses of this new algorithm, as well as to apply it to new problems. These are priority lines because progress in them will allow us to dispel the threats to validity that we have identified in this paper. Also, it is important to analyze if the elapsed time of our algorithm always stays around 75% higher than the one corresponding to the genetic programming algorithm or whether the percentage gap shrinks. We will need further experimentation to have a plausible conjecture. Finally, as suggested by one of the anonymous reviewers, we will compare our method against a fine-tuned Genetic Programming approach. In this case, we have to be especially careful with potential overfitting problems.

References

[1] T. Bäck, D. B. Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computation*. CRC Press, 1997.

[2] D. Barrios, A. Carrascal, D. Manrique, and J. Rios. Optimisation with real-coded genetic algorithms based on mathematical morphology. *International Journal of Computer Mathematics*, 80(3):275–293, 2003.

[3] D. Barrios, G. Delgado, and D. Manrique. Multilayered neural architectures evolution for computing sequences of orthogonal polynomials. *Annals of Mathematics and Artificial Intelligence*, 84(3-4):161–184, 2018.

[4] P. Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337(1-3):217–239, 2005.

[5] C. Blum and D. Merkle, editors. *Swarm Intelligence: Introduction and Applications*. Springer, 2008.

[6] E. K. P. Chong and S. H. Zak. Global search algorithms. In *An Introduction to Optimization*, chapter 14. John Wiley & Sons, Inc., 3rd edition, 2008.

[7] J. Couchet, D. Manrique, J. Rios, and A. Rodríguez-Patón. Crossover and mutation operators for grammar-guided genetic programming. *Soft Computing*, 11(10):943–955, 2007.

[8] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man and Cybernetics B*, 26(1):29–41, 1996.

[9] J. M. Font, D. Manrique, and E. Pascua. Grammar-guided evolutionary construction of bayesian networks. In *4th Int. Work-Conference on the Interplay Between Natural and Artificial Computation, IWINAC'11, LNCS 6686*. Springer, 2011.

[10] P. García-Sánchez, A. Tonda, A. M. Mora, G. Squillero, and J. J. Merelo. Automated playtesting in collectible card games using evolutionary algorithms: A case study in hearthstone. *Knowledge-Based Systems*, 153:133 – 146, 2018.

[11] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Int. Conf. on Neural Networks, ICNN'95*, pages 1942–1948, 1995.

[12] J. M. Luna, J. R. Romero, and S. Ventura. Design and behavior study of a grammar-guided genetic programming algorithm for mining association rules. *Knowledge and Information Systems*, 32(1):53–76, 2012.

[13] D. Manrique, F. Márquez, J. Ríos, and A. Rodríguez-Patón. Grammar based crossover operator in genetic programming. In *1st Int. Work-Conference on the Interplay Between Natural and Artificial Computation, IWINAC'05, LNCS 3562*, pages 252–261. Springer, 2005.

[14] R. I. McKay, N. X. Hoai, P. A. Whigham, Y. S., and M. O'Neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3-4):365–396, 2010.

[15] M. Mitchell. *An introduction to genetic algorithms*. MIT Press, 1998.

[16] A. A. Salman, I. Ahmad, and S. Al-Madani. Particle swarm optimization for task assignment problem. *Microprocessors and Microsystems*, 26(8):363–371, 2002.

[17] S. Strasser, R. Goodman, J. Sheppard, and S. Butcher. A new discrete particle swarm optimization algorithm. In *18th Annual Conf. on Genetic and Evolutionary Computation, GECCO'16*, pages 53–60. ACM Press, 2016.